CS107, Lecture 12
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?
How can we use our knowledge of memory and data representation to write code that works with any data type?

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

• Disclosure and partiality
• **Overview**: Generics
• Generic Swap
• Generics Pitfalls
• Generic Array Swap

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

• Disclosure and partiality
• Overview: Generics
• Generic Swap
• Generics Pitfalls
• Generic Array Swap
What should someone do if they find a vulnerability? How can we incentivize responsible disclosure?
What’s the best way to disclose vulnerabilities?

• **Full disclosure?** Make vulnerabilities public as soon as they are found? Few people now endorse this approach due to its drawbacks.

• **Responsible disclosure?** Privately alert software maker to fix in reasonable amount of time before publicizing? Most common, and recommended by ACM code of ethics.
Disclosure

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

• The US Federal Government is one of the largest discoverers and purchasers of 0-day vulnerabilities. It follows a “Vulnerability Equities Process” (VEP) to determine which vulnerabilities to responsibly disclose and which to keep secret and use for espionage or intelligence gathering.
How do we weigh competing stakeholder interests here, such as country vs. individual?
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

self family friends state world
## Degrees of Partiality

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


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.
“[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 everywhere in the face of cyberattacks, regardless of their nationality.”
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.
Partiality: preference towards own family, friends, and state is morally acceptable or even required

Partial Cosmopolitanism: limited preference towards own state acceptable

Universal Care: preference towards family acceptable but not towards state

Impartial Benevolence: same moral responsibilities towards all people

Revisiting EternalBlue

Federal Government

Microsoft
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

• Disclosure and partiality

• 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

• Disclosure and partiality
• **Overview**: Generics
• **Generic Swap**
• Generics Pitfalls
• Generic Array Swap
You’re asked to write a function that swaps two numbers.

```c
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;
}
```
You're asked to write a function that swaps two numbers.

```c
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;
}
```
You’re asked to write a function that swaps two numbers.

```c
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;
}
```
You’re asked to write a function that swaps two numbers.

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

```c
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;
}
```
You’re asked to write a function that swaps two numbers.

```c
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;
}
```
You’re asked to write a function that swaps two numbers.

```c
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;
}
```
You’re asked to write a function that swaps two numbers.

```c
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.”
```c
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?” 😤
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;
}
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;
}
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;
}
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;
}
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;
}
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?”
What if we could write *one* function to swap two values of any single type?

```c
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) { ... }
...
```
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;
}
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;
}
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
}
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
}

Problem: each type may need a different size temp!
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!
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
}

**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?
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
}
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
}
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
}
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
}
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.
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.
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 \textbf{nbytes} of temp space?
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
}
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!
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?
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 \texttt{data1ptr} points to into \texttt{temp}?
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** is a function that copies a specified amount of bytes at one address to another address.

```c
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.

```c
int x = 5;
int y = 4;
memcpy(&x, &y, sizeof(x));  // like x = y
```
memmove is the same as memcpy, but supports overlapping regions of memory. (Unlike its name implies, it still “copies”).

```c
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).
When might memmove be useful?
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.
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? (Assume data to be swapped is not overlapping). **Respond with your thoughts on PollEv:**

pollev.com/cs107 or text CS107 to 22333 once to join.

```c
void *memcpy(void *dest, const void *src, size_t n);
void *memmove(void *dest, const void *src, size_t n);
```
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
}
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.
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
}
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?
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!
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?
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`!
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));
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));
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));
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);
}
• 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!

```c
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

• Disclosure and partiality

• **Overview:** Generics

• Generic Swap

• **Generics Pitfalls**

• Generic Array Swap

```
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!"

[Image: http://i.ytimg.com/vi/10gPoYjq3EA/hqdefault.jpg]
• Disclosure and partiality
• **Overview:** Generics
• Generic Swap
• Generics Pitfalls
• **Generic Array Swap**

```bash
cp -r /afs/ir/class/cs107/lecture-code/lect12 .
```
You’re asked to write a function that swaps the first and last elements in an array of numbers.

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

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

Wait – we just wrote a generic swap function. Let’s use that!
You’re asked to write a function that swaps the first and last elements in an array of numbers.

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

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

Wait – we just wrote a generic swap function. Let’s use that!
Let's write out what some other versions would look like (just in case).

```c
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!
Let’s write a version of `swap_ends` that works for any type of array.

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

Is this generic? Does this work?
Let’s write a version of `swap_ends` that works for any type of array.

```c
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!
Let’s write a version of `swap_ends` that works for any type of array.

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

```c
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.
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 $3 \times \text{sizeof(int)} = 12$ bytes
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?**
Let’s say \( \text{nelems} = 4 \). How many bytes beyond \( \text{arr} \) is this?

If it’s an array of...

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

**Short**: adds 3 `places` to \( \text{arr} \), and \( 3 \times \text{sizeof(short)} = 6 \) bytes
Let’s say \( \text{nelems} = 4 \). How many bytes beyond \( \text{arr} \) is this?

If it’s an array of...

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

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

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

In each case, we need to know the element size to do the arithmetic.
Let’s write a version of `swap_ends` that works for any type of array.

```c
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?

```plaintext
(nelems – 1) * elem_bytes
```
Let’s write a version of swap_ends that works for any type of array.

```c
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?

\[(\text{nelems} - 1) \times \text{elem_bytes}\]
Let’s write a version of swap_ends that works for any type of array.

```c
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?**
Let’s write a version of `swap_ends` that works for any type of array.

```c
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!
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!

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

```c
int nums[] = {5, 2, 3, 4, 1};
size_t nelems = sizeof(nums) / sizeof(nums[0]);
swap_ends(nums, nelems, sizeof(nums[0]));
```
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!

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

```c
short nums[] = {5, 2, 3, 4, 1};
size_t nelems = sizeof(nums) / sizeof(nums[0]);
swap_ends(nums, nelems, sizeof(nums[0]));
```
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!

```c
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]));
```
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!

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

```c
mystruct structs[] = ...;
size_t nelems = ...;
swap_ends(structs, nelems, sizeof(structs[0]));
```
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

- Disclosure and partiality
- **Overview**: Generics
- Generic Swap
- Generics Pitfalls
- Generic Array Swap

---

**Lecture 8 takeaway**: 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. We can cast `void *`s to perform pointer arithmetic. `void *`s have no type checking, so we must be vigilant!

**Next time**: More Generics, and Function Pointers
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?

  ```c
  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: You’re asked to provide an implementation for a function called `rotate` with the following prototype:

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

The expectation is that `front` is the base address of an array, `end` is the past-the-end address of the array, and `separator` is the address of some element in between. `rotate` moves all elements in between `front` and `separator` to the end of the array, and all elements between `separator` and `end` move to the front.
Exercise: Array Rotation

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

Before:

```
1 2 3 4 5 6 7 8 9 10
```

After:

```
4 5 6 7 8 9 10 1 2 3
```
Exercise: Array Rotation

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

```c
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:

front separator end

1 2 3 4 5 6 7 8 9 10

temp

Before last step:

front separator end

4 5 6 7 8 9 10 8 9 10
Exercise: A properly implemented `rotate` will prompt the following program to generate the provided output.

And here’s that properly implemented function!

```c
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);
}
```
Stacks

• C generics are particularly powerful in helping us create generic data structures.

• Let’s see how we might go about making a generic Stack in C.
A **Stack** is a data structure representing a stack of things.

Objects can be **pushed** on top of or **popped** from the top of the stack.

Only the top of the stack can be accessed; no other objects in the stack are visible.

Main operations:
- **push(value)**: add an element to the top of the stack
- **pop()**: remove and return the top element in the stack
- **peek()**: return (but do not remove) the top element in the stack
A stack is often implemented using a **linked list** internally.

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

```c
Stack<int> s;
s.push(42);
s.push(-3);
s.push(17);
```

**Problem:** C is not object-oriented! We can’t call methods on variables.
Demo: Int Stack

int_stack.c
What modifications are necessary to make a generic stack?
typedef struct int_node {
    struct int_node *next;
    int data;
} int_node;

typedef struct int_stack {
    int nelems;
    int_node *top;
} int_stack;

How might we modify the Stack data representation itself to be generic?
typedef struct int_node {
    struct int_node *next;
    int data;
} int_node;

typedef struct int_stack {
    int nelems;
    int_node *top;
} int_stack;

**Problem:** each node can no longer store the data itself, because it could be any size!
typedef struct int_node {
    struct int_node *next;
    void *data;
} int_node;

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

**Solution:** each node stores a pointer, which is always 8 bytes, to the data somewhere else. We must also store the data size in the Stack struct.
• **int_stack_create()**: creates a new stack on the heap and returns a pointer to it

• **int_stack_push(int_stack *s, int data)**: pushes data onto the stack

• **int_stack_pop(int_stack *s)**: pops and returns topmost stack element
int_stack_create

int_stack *int_stack_create() {
  int_stack *s = malloc(sizeof(int_stack));
  s->nelems = 0;
  s->top = NULL;
  return s;
}

How might we modify this function to be generic?

From previous slide:
typedef struct stack {
  int nelems;
  int elem_size_bytes;
  node *top;
} stack;
Generic stack_create

stack *stack_create(int elem_size_bytes) {
    stack *s = malloc(sizeof(stack));
    s->nelems = 0;
    s->top = NULL;
    s->elem_size_bytes = elem_size_bytes;
    return s;
}
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}

How might we modify this function to be generic?

From previous slide:

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

typedef struct node {
    struct node *next;
    void *data;
} node;
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}

**Problem 1:** we can no longer pass the data itself as a parameter, because it could be any size!
void int_stack_push(int_stack *s, const void *data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}

Solution 1: pass a pointer to the data as a parameter instead.
Generic stack_push

```c
void int_stack_push(int_stack *s, const void *data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

**Problem 2:** we cannot copy the existing data pointer into new_node. The data structure must manage its own copy that exists for its entire lifetime. The provided copy may go away!
int main() {
    stack *int_stack = stack_create(sizeof(int));
    add_one(int_stack);
    // now stack stores pointer to invalid memory for 7!
}

void add_one(stack *s) {
    int num = 7;
    stack_push(s, &num);
}
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->elem_size_bytes);
    memcpy(new_node->data, data, s->elem_size_bytes);
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}

**Solution 2:** make a heap-allocated copy of the data that the node points to.
```c
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;
    s->top = n->next;
    free(n);
    s->nelems--;
    return value;
}
```

How might we modify this function to be generic?

From previous slide:
```c
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```
```c
typedef struct node {
    struct node *next;
    void *data;
} node;
```
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
void *int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    void *value = n->data;

    s->top = n->next;
    free(n);
    s->nelems--;

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

    free(n->data);
    free(n);
    s->nelems--;
}

Solution: have the caller pass a memory location as a parameter and copy the data to that location.
int_stack *intstack = int_stack_create();
for (int i = 0; i < TEST_STACK_SIZE; i++) {
    int_stack_push(intstack, i);
}

We must now pass the address of an element to push onto the stack, rather than the element itself.
Using Generic Stack

```c
stack *intstack = stack_create(sizeof(int));
for (int i = 0; i < TEST_STACK_SIZE; i++) {
    stack_push(intstack, &i);
}
```

We must now pass the \textit{address} of an element to push onto the stack, rather than the element itself.
We must now pass the *address* of an element to push onto the stack, rather than the element itself.

```c
int_stack *intstack = int_stack_create();
int_stack_push(intstack, 7);
```
We must now pass the *address* of an element to push onto the stack, rather than the element itself.

```c
stack *intstack = stack_create(sizeof(int));
int num = 7;
stack_push(intstack, &num);
```
// Pop off all elements
while (intstack->nelems > 0) {
    printf("%d\n", int_stack_pop(intstack));
}

We must now pass the *address* of where we would like to store the popped element, rather than getting it directly as a return value.
Using Generic Stack

// Pop off all elements
int popped_int;
while (intstack->nelems > 0) {
    int_stack_pop(intstack, &popped_int);
    printf("%d\n", popped_int);
}

We must now pass the address of where we would like to store the popped element, rather than getting it directly as a return value.
Demo: Generic Stack
Generic stack_create

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

...

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

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

typedef struct node {
    struct node *next;
    void *data;
} node;
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->elem_size_bytes);
    memcpy(new_node->data, data, s->elem_size_bytes);
    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}

int x = 2;
stack_push(numStack, &2);

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

typedef struct node {
    struct node *next;
    void *data;
} node;
void stack_pop(stack *s, void *addr) {
    node *n = s->top;
    memcpy(addr, n->data, s->elem_size_bytes);
    s->top = n->next;
    free(n->data);
    free(n);
    s->nelems--;
}

... int num;
stack_pop(numStack, &num);
printf("%d\n", num);

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

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