CS107 Lecture 11

Heap Wrap, Generics – void *

Reading: None 😍

Based on slides created by Cynthia Lee, Chris Gregg, Nick Troccoli, Lisa Yan, Jerry Cain and others.

Other heap allocators: calloc, strdup

void *calloc(size_t count, size_t size);

calloc is a **heap allocation function** **just** like **malloc**, with one important guarantee—it **zeroes out the memory** before it returns.

Examples:

```
int *counts = calloc(26, sizeof(int)); // all zeroes
bool *answers = calloc(n, sizeof(bool)); // all falses
struct node ** = calloc(num_buckets, sizeof(struct node *)); // all NULLs
```

char *strdup(const char *str);

strdup is a convenience function for **duplicating C strings onto the heap**.

Example:

```
char *news = strdup("disinformation");
news[0] = 'm';
char *british = strdup("disorganization")
british[9] = 's';
```

Like anything created using **malloc**, anything created using **calloc** or **strdup** **should be freed** when it's no longer needed. Again, soon!

Cleaning Up with `free`

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

`free` accepts the address of a pointer previously returned by **`malloc`**, **`calloc`**, **`strdup`**, or (the soon to be discussed) **`realloc`** and donates the memory back to the heap.

Simple Example

```
char *earth = strdup("earth");
char *quake = strdup("quake");
char *earthquake = malloc(strlen(earth) + strlen(quake) + 1);
strcpy(earthquake, earth);
free(earth);
strcat(earthquake, quake);
free(quake);
printf("%s\n", earthquake);
free(earthquake); // three allocations reverted by three frees
```

When you **fail to free heap memory** you no longer need before letting go of it, you **leak** that memory.

free Etudes

For each the code snippet below, **identify whether the code visible to you contains an error or not**. If so, explain what that error is.

```
char *jennifer = strdup("garner");
char *alias = jennifer;
free(jennifer);
free(alias);
```

✗

Issue: both **`jennifer`** and **`alias`** are **linked to the same heap-based string `"garner"`**. This code suffers from a **double free**.

```
char *satword = strdup("grandiloquence");
free(satword + 5);
free(satword);
```

✗

Issue: **`satword + 5`**, whatever it is, is **not the base address** of a dynamically allocated figure. **`satword`**? Yes! But **`satword + 5`**? Nope!

```
char *heavy = strdup("bloated");
char *light = heavy;
free(heavy);
light[0] = 'f';
```

✗

Issue: **`heavy`** and **`light`** are each attached to the same heap-based string **`"bloated"`**. We **only free the string once**, but we **can't access the string through either variables after that**.

```
char superhero[] = "Batman";
char *sidekick = strdup(superhero);
strncpy(sidekick, "An", 2);
free(superhero);
```

✗

Issue: **`superhero`** is a stack-based **`"Batman"`**, and **`sidekick`** is a heap-based **`"Antman"`**. The first three lines, albeit contrived, are **correct** (yay!). But you can't free **`superhero`**, because **it has nothing to do with the heap**.

Resizing with `realloc`

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

`realloc` accepts an address previously returned by **`malloc`**, **`calloc`**, **`strdup`**, or **`realloc`**, resizes the memory attached to it as appropriate, and returns the (possibly new, possibly the same) address.

Example

```
char *serialize(const char *strings[], size_t count) {
   char *serialization = strdup("");
   assert(serialization != NULL);
   size_t sofar = 0;
   for (size_t i = 0; i < count; i++) {
      size_t len = strlen(strings[i]);
      serialization = realloc(serialization, sofar + len + 1);
      assert(serialization != NULL);
      strcpy(serialization + sofar, strings[i]);
      sofar += len;
   }
   return serialization;
}
```

If there's enough space **beyond the existing existing block** to accommodate the new, larger size, **`realloc`** simply adds that space to the allocation.

If there isn't enough space, **`realloc`** simply **`malloc`'s** a larger block, **copies all data** from old to new, **frees** the old block, and returns the address of the new memory.

In practice, the supplied **`size`** is **almost always larger**, though **`realloc`** will shrink the allocated size if asked to.

Heap Allocation Redux

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

Heap **memory allocation** guarantee:

- **`NULL`** on **failure**, so check with **`assert`**
- Memory is **contiguous** and **recycled only** when you call **`free`**
- **`realloc`** **preserves existing data**
- **`calloc`** **zeroes** out 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 you call **`free`** **twice** on same address.
- If you **`realloc`** or **`free`** an address **outside the heap**.

Heap vs Stack: Battle of the Segments

Stack (for local variables)

- **Fast**
  Fast to allocate and deallocate, okay to oversize
- **Convenient**
  Automatic allocation and deallocation, declare and initialize in one step
- **Reasonable type safety**
  Thanks to the compiler

⚠ **Not especially plentiful**
Total stack size fixed, default 8 – 32 MB

⚠ **Somewhat inflexible**
Cannot add or resize at runtime, scope dictated by control flow in and out of function calls

Heap (for dynamic memory)

- **Plentiful**
  Can generally provide more memory on demand
- **Exceptionally flexible**
  Runtime decisions about how much and when to allocate, can resize easily using **`realloc`**
- **Scope under programmer control**
  Can precisely determine lifetime

⚠ **Lots of opportunity for error**
Minimal type safety, forget to allocate/free before done, allocate wrong size, etc., memory leaks

CS107 Topic 4: C Generics

How can we **leverage our knowledge of memory and data representation** to write code that works **for all data types**?

Why is answering this question useful?

- Writing general purpose code that works on all types means **one implementation, not many**, thereby avoid cut-and-paste with type changes.
- It teaches us how to to **pass functions as parameters** to provide **just enough intelligence about the data** for the generic code to work properly.

Learning Goals

- Learn how to write C code that works with for **all data types**.
- Learn about how to use **`void *`** and **overcome its shortcomings**.

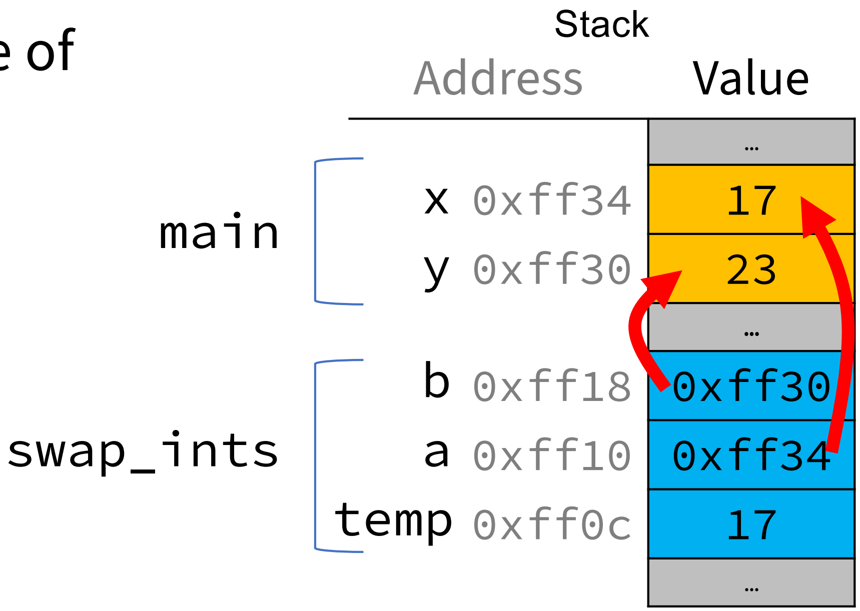
Strongly Typed Data Exchange

Let's once again implement a routine capable of **exchanging two numbers**.

```
int main(int argc, char *argv[]) {
   int x = 17;
   int y = 23;
   swap_ints(&x, &y);
   printf("x = %d, y = %d\n", x, y);
   return 0;
}

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

Assuming the line in red has just executed, the **state of memory** immediately afterward is presented up and to the right.



Stack

		Address	Value
			…
main	x	0xff34	17
	y	0xff30	23
			…
swap_ints	b	0xff18	0xff30
	a	0xff10	0xff34
	temp	0xff0c	17
			…

High-level POV: **swap_ints** accepts the **locations** of two **int**s and **exchanges them** using a well understood algorithm.

Low-level POV: **swap_ints** exchanges four-byte patterns at the provided addresses, and those patterns incidentally represent **int**s.

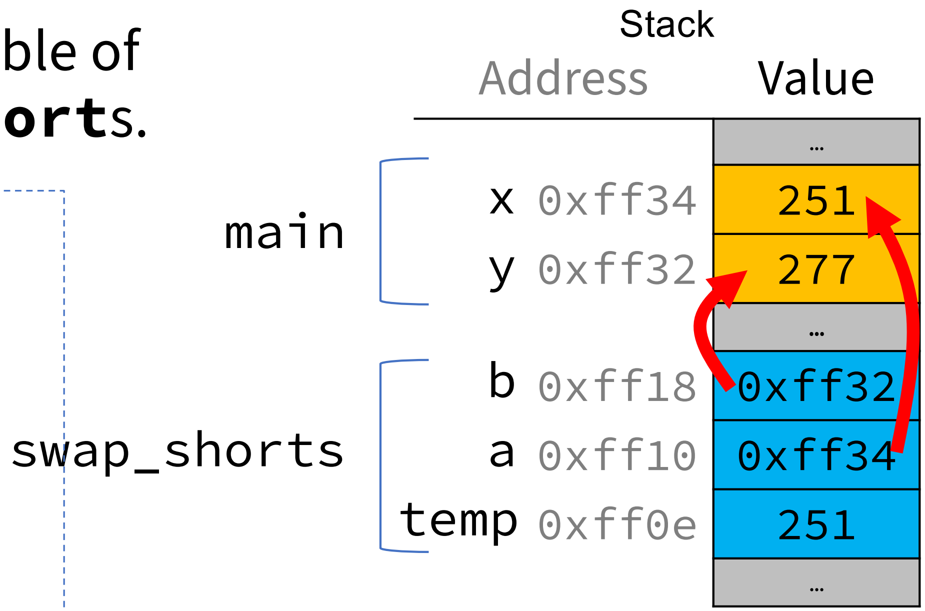
Strongly Typed Data Exchange

Let’s once again implement a routine capable of exchanging two numbers—**this time as `short`**s.

```
int main(int argc, char *argv[]) {
   short x = 251;
   short y = 277;
   swap_shorts(&x, &y);
   printf("x = %d, y = %d\n", x, y);
   return 0;
}

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

Assuming the line in red has just executed, the **state of memory** immediately afterward is presented up and to the right.



Stack

		Address	Value
			…
main	x	0xff34	251
	y	0xff32	277
			…
swap_shorts	b	0xff18	0xff32
	a	0xff10	0xff34
	temp	0xff0e	251
			…

High-level POV: **`swap_shorts`** accepts the **locations** of two **`short`**s and **exchanges them** using the same algorithm that **`swap_ints`** does.

Low-level POV: **`swap_shorts`** swaps two 16-bit patterns. Code exchanges two **`short`**s if it exchanges their underlying bit patterns.

Strongly Typed Data Exchange

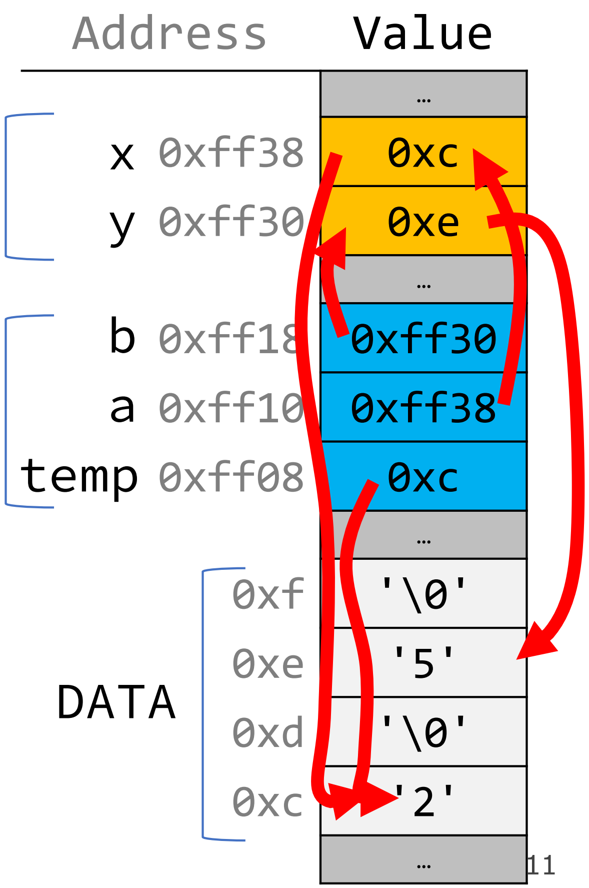
Let's implement it **once more**, this time **for the strings** in the back corner of the data segment.

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

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

The narrative is the same : **exchange the bit patterns** residing within the two orange boxes, whatever they happen to be.

Is it possible to implement a single function that works for all types?



		Address	Value
			…
main	x	0xff38	0xc
	y	0xff30	0xe
			…
swap_strings	b	0xff18	0xff30
	a	0xff10	0xff38
	temp	0xff08	0xc
			…
DATA		0xf	'\0'
		0xe	'5'
		0xd	'\0'
		0xc	'2'
			…

swap: Going Generic

These three functions all accomplish the same thing—**swapping two values**—but require three different **strongly typed signatures**.

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

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

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

Each of the three routines:

- **accepts pointers** to values that should be exchanged.
- **creates temporary storage** just big enough to store one of the two values.
- **reads the data** addressed by **a** and **copies that same data** into the temporary storage.
- **reads the data** addressed by **b** and **copies a bitwise replica** into the space addressed by **a.**
- **reads the data** residing in the temporary and **copies all of it** into the space addressed by **b**.

The primary difference worth pointing out: the **number of bytes moved** is different in each.

swap: Going Generic

```
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
}
```

swap: Going Generic

```
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!

swap: Going Generic

```
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!

swap: Going Generic

```
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!

swap: Going Generic

```
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
}
```

swap: Going Generic

```
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
}
```

swap: Going Generic

```
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.

swap: Going Generic

```
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.

swap: Going Generic

```
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 for the temporary.
How can we allocate **`nbytes`** of temp space?

swap: Going Generic

```
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?

swap: Going Generic

```
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!

swap: Going Generic

```
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**?

swap: Going Generic

```
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**?

swap: Going Generic

```
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.

The Byte Replicator: memcpy

memcpy is a function that copies a specified number of bytes from one address to another address.

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

It copies the next n bytes that src points to to the location contained in dest. (It also returns **dest**). It assumes the two regions of memory don't 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
```

swap: Going Generic

```
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.

swap: Going Generic

```
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** help us here?

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

swap: Going Generic

```
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
}
```

swap: Going Generic

```
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.

swap: Going Generic

```
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?

swap: Going Generic

```
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?

swap: Going Generic

```
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!

swap: Going Generic

```
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?

swap: Going Generic

```
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**!

swap: Going Generic

```
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));
```

swap: Going Generic

```
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));
```

swap: Going Generic

```
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));
```

swap: Going Generic

```
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));
```

Demo: void *s Gone Wrong

`swap.c`