CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

UC Berkeley
Teaching Professor
Lisa Yan

UC Berkeley
Lecturer
Justin Yokota

AI & Machine Learning

BFloat16: The secret to high performance on Cloud TPUs

August 23, 2019

format was conceived. Bfloat16 is carefully used within systolic arrays to accelerate matrix multiplication operations on Cloud TPUs. More precisely, each multiply-accumulate operation in a matrix multiplication uses bfloat16 for the multiplication and $\boxed{\text{32-bit IEEE floating point for accumulation.}}$ In this

Google blog, TPU docs

Today's material is not required for Project 1. It is covered in Homework 03.

C Generics and Function Pointers

From Last Time

www.h-schmidt.net/FloatConverter/IEEE754.html

- Floating Point lets us…
  - Represent numbers with both integer and fractional parts making efficient use of available bits.
  - Store approximate values for very large and very small #s.
- IEEE 754 Floating Point Standard is the most widely accepted attempt to standardize interpretation of such numbers.
  - Every computer since ~1997 follows these conventions!
- Summary (single precision, or fp32):

31	30 … 23	22 … 0
s	exponent	significand
1 bit	8 bits	23 bits

Can store NaN, $\pm\infty$ ☑

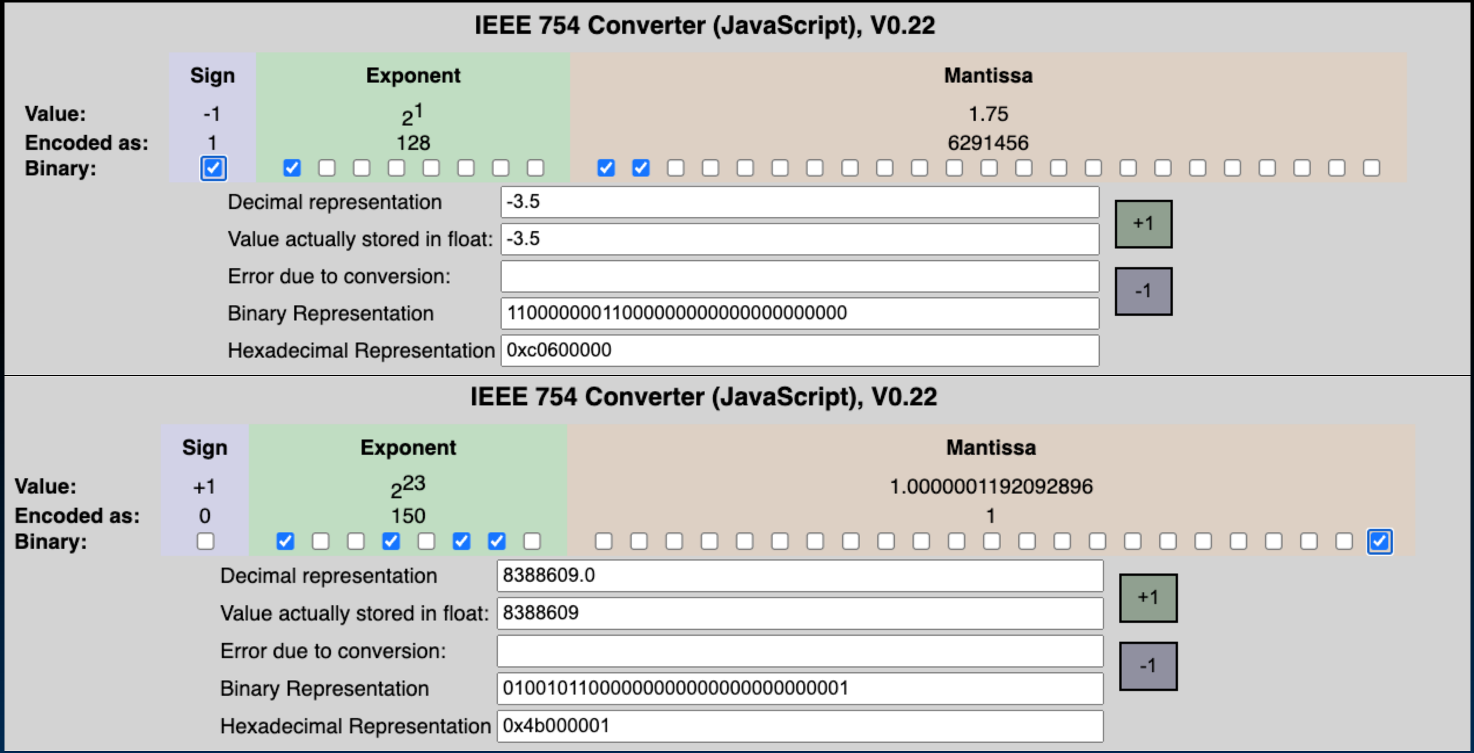
$$(-1)^{S} \times (1 + \text{significand}) \times 2^{(\text{exponent}-127)}$$

Exponent tells Significand how much (2^i) to count by (…, 1/4, 1/2, 1, 2, …)

For you: IEEE-754 Floating Point Converter

[for next time]

https://www.h-schmidt.net/FloatConverter/IEEE754.html



Yan, Yokota

Special Numbers

- Normalized numbers are only a fraction (heh) of floating point representations. For single-precision (32-bit):

Biased Exponent	Significand field	Object
0	all zeros	±0
0	nonzero	**Denormalized numbers**
1 – 254	anything	Normalized floating point
255	all zeros	±∞
255	nonzero	NaNs

Today: denorms
Full slidedeck in L06

The fields 0 and 255 accommodate **overflow**, **underflow**, and arithmetic errors.

Yan, Yokota

Overflow and Underflow

[from last time]

- Because 0 and 255 are reserved exponent fields, the range of normalized single-precision floating point is:
  - Largest magnitude:

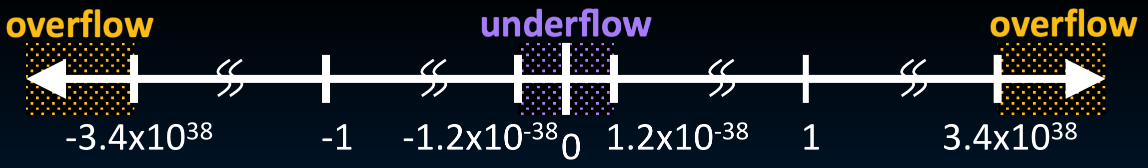
s	1111 1110	1…1 (23 bits)

$(1.1\ldots1) \times 2^{(254-127)} \approx 3.4 \times 10^{38}$

  - Smallest magnitude:

s	0000 0001	0…0 (23 bits)

$(1.0) \times 2^{(1-127)} \approx 1.2 \times 10^{-38}$



- What if a number falls outside this range?
  - Too large: overflow. **Magnitude of value is too large to represent!**
  - **Too small: underflow. Magnitude of value is too small to represent!**

Yan, Yokota

Denorms: Gradual Underflow (1/2)

- Problem:
  - There's a **gap** among representable FP numbers around zero!
  - Smallest normalized number:

0	0000 0001	000…000

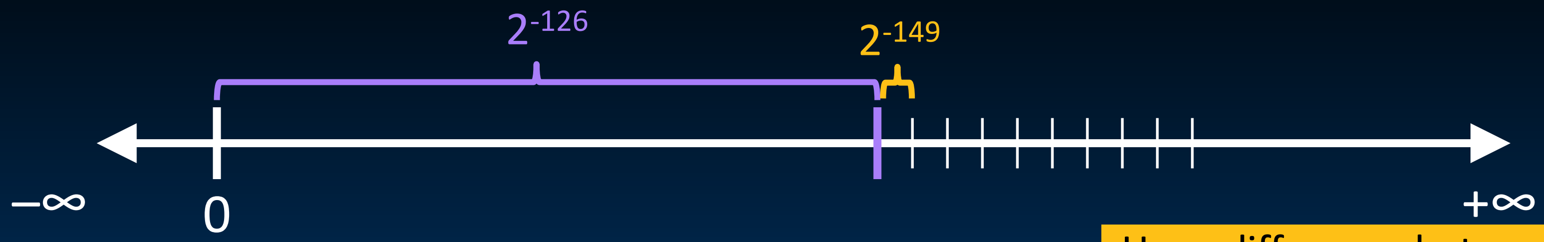
$(1.0) \times 2^{(1-127)} = 2^{-126}$

  - 2^{nd} smallest normalized number:

31	30 … 23	22 … 1 0
0	0000 0001	00000…00001

$(1.000......1_2) \times 2^{(1-127)}$

$= (1 + 2^{-23}) \times 2^{-126} = 2^{-126} + 2^{-149}$



Huge difference between 0 and smallest vs. smallest and 2^{nd} smallest! Occurs b/c of normalization and implicit leading 1.

Denorms: Gradual Underflow (2/2)

[for next time]

- Solution:
  - We still haven't used Exponent = 0, Significand nonzero.
  - **DEnormalized** number:
    - no (implied) leading 1
    - implicit exponent for all denorms = -126.

31	30 … 23	22 … 0
s	0000 0000	xxx…xxxx

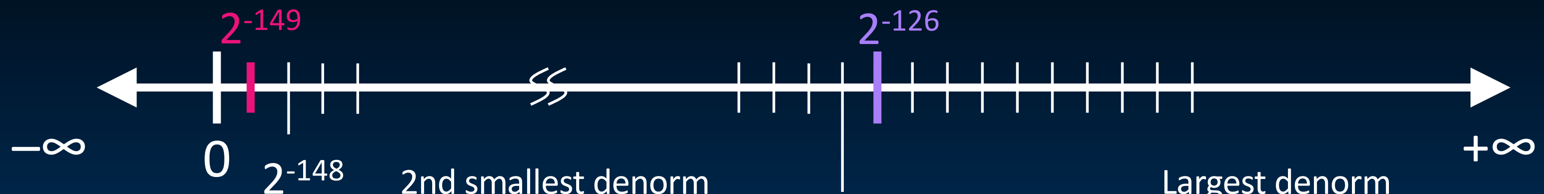
$$(-1)^s \times (0.x\ldots\ldots x_2) \times 2^{-126}$$

Smallest denormalized number

0	0000 0000	000…001

Smallest normalized number

0	0000 0001	000…000



2nd smallest denorm

0	0000 0000	000…010

Largest denorm

0	0000 0000	111…111

BTW: double Precision Floating Point

See more floating point reps in the L06 slides!

- binary64: Next Multiple of Word Size (64 bits)

31	30 … 20	19 … 0
S	Exponent	Significand
1 bit	11 bits	20 bits

Significand (cont'd)
32 bits

- Double Precision (vs. Single Precision)
  - C variable declared as `double`
  - Exponent bias now 1023.
  - Represent numbers from ~2.0×10^{-308} to ~2.0×10^{308}.
  - The primary advantage is greater **accuracy** due to larger significand.

Yan, Yokota

Agenda

Today's lecture

- Floats Wrap-up (denorms)
- Function Pointers
- Generic Functions
- Writing Generics: Episode I
- Writing Generics: Episode II
- Pointer Arithmetic in Generics

Yan, Yokota

Pointers to Different Data Types

[review]

- Pointers are used to point to a variable of a particular data type.
  - Normally a pointer can only point to one type.
- `void *` is a type that can point to anything (generic pointer).
  - Use sparingly to help avoid program bugs… and security issues… and a lot of other bad things!
- You can even have pointers to functions…
  - `int (*fn) (void *, void *) = &foo;`
    - `fn` is a function that accepts two `void *` pointers and returns an `int` and is initially pointing to the function `foo`.
  - `(*fn)(x, y);` will then call the function

```
int *xptr;
char *str;
struct llist *foo_ptr;
```

(more ~~later~~ **now**)

Agenda

Function Pointers

- Floats Wrap-up (denorms)
- Function Pointers
- Generic Functions
- Writing Generics: Episode I
- Writing Generics: Episode II
- Pointer Arithmetic in Generics

Yan, Yokota

Function Pointers

- Pointers are used to point to a variable of a particular data type.
  - Normally a pointer can only point to one type.
- `void *` is a type that can point to anything (generic pointer).
  - Use sparingly to help avoid program bugs… and security issues… and a lot of other bad things!
- You can even have pointers to functions…
  - `int (*fn) (void *, void *) = &foo;`
    - fn is a function that accepts two `void *` pointers and returns an `int` and is initially pointing to the function foo.
  - `(*fn)(x, y);` will then call the function

```
int *xptr;
char *str;
struct llist *foo_ptr;
```

- A **function pointer** is a variable storing the starting address of a function.
  - Function pointers are also typed!

Function Pointers allow us to define **higher-order functions**, such as map, filter, and generic sort.

(1/3) Function Pointer Example: The Output

```c
/* map a function onto int array */
void mutate_map(int arr[], int n,
                int(*fp)(int)) {
  for (int i = 0; i < n; i++)
    arr[i] = (*fp)(arr[i]);
}
int multiply2 (int x) {  return  2 * x;   }
int multiply10(int x) {  return 10 * x;   }
int main() {
  int arr[] = {3,1,4};
  int n = sizeof(arr)/sizeof(arr[0]);
  print_array(arr, n); // 1
  mutate_map (arr, n, &multiply2);
  print_array(arr, n); // 2
  mutate_map (arr, n, &multiply10);
  print_array(arr, n); // 3
  return 0;
}
```

```
$ ./map_func
3 1 4
6 2 8
60 20 80
```

(see code in Drive)

Yan, Yokota

(2/3) Function Pointer Example

```c
/* map a function onto int array */
void mutate_map(int arr[], int n,
                int(*fp)(int)) {
  for (int i = 0; i < n; i++)
    arr[i] = (*fp)(arr[i]);
}
int multiply2 (int x) {  return  2 * x;  }
int multiply10(int x) {  return 10 * x;  }
int main() {
  int arr[] = {3,1,4}
  int n = sizeof(arr)/sizeof(arr[0]);
  print_array(arr, n); // 1
  mutate_map (arr, n, &multiply2);
  print_array(arr, n); // 2
  mutate_map (arr, n, &multiply10);
  print_array(arr, n); // 3
  return 0;
}
```

```
$ ./map_func
3 1 4
6 2 8
60 20 80
```

- The `fp` parameter is a function **pointer**.
  - Type: `int` parameter, `int` retval

(see code in Drive)

Yan, Yokota

Function Pointer Example

```c
/* map a function onto int array */
void mutate_map(int arr[], int n,
                int(*fp)(int)) {
  for (int i = 0; i < n; i++)
    arr[i] = (*fp)(arr[i]);
}
int multiply2 (int x) {  return  2 * x;  }
int multiply10(int x) {  return 10 * x;  }
int main() {
  int arr[] = {3,1,4}
  int n = sizeof(arr)/sizeof(arr[0]);
  print_array(arr, n); // 1
  mutate_map (arr, n, &multiply2);
  print_array(arr, n); // 2
  mutate_map (arr, n, &multiply10);
  print_array(arr, n); // 3
  return 0;
}
```

```
$ ./map_func
3 1 4
6 2 8
60 20 80
```

- The `fp` parameter is a function pointer.
  - Type: `int` parameter, `int` retval
- Call the function pointed to by the function pointer.

(see code in Drive)

Yan, Yokota

Function Pointer Example

```c
/* map a function onto int array */
void mutate_map(int arr[], int n,
                int(*fp)(int)) {
  for (int i = 0; i < n; i++)
    arr[i] = (*fp)(arr[i]);
}
int multiply2 (int x) {  return  2 * x;  }
int multiply10(int x) {  return 10 * x;  }
int main() {
  int arr[] = {3,1,4}
  int n = sizeof(arr)/sizeof(arr[0]);
  print_array(arr, n); // 1
  mutate_map (arr, n, &multiply2);
  print_array(arr, n); // 2
  mutate_map (arr, n, &multiply10);
  print_array(arr, n); // 3
  return 0;
}
```

```
$ ./map_func
3 1 4
6 2 8
60 20 80
```

- The `fp` parameter is a function pointer.
  - Type: `int` parameter, `int` retval
- Call the function pointed to by the function pointer.
- Assign the function pointer a value.

Berkeley UNIVERSITY OF CALIFORNIA (see code in Drive)

Yan, Yokota

Function Pointer Example

```c
/* map a function onto int array */
void mutate_map(int arr[], int n,
                int(*fp)(int)) {
  for (int i = 0; i < n; i++)
    arr[i] = (*fp)(arr[i]);
}
int multiply2 (int x) {  return  2 * x;  }
int multiply10(int x) {  return 10 * x;  }
int main() {
  int arr[] = {3,1,4}
  int n = sizeof(arr)/sizeof(arr[0]);
  print_array(arr, n); // 1
  mutate_map (arr, n, &multiply2);
  print_array(arr, n); // 2
  mutate_map (arr, n, &multiply10);
  print_array(arr, n); // 3
  return 0;
}
```

```
$ ./map_func
3 1 4
6 2 8
60 20 80
```

- The `fp` parameter is a function pointer.
  - Type: `int` parameter, `int` retval
- Call the function pointed to by the function pointer.
- Assign the function pointer a value.

(C quirk: Outside of declaration, functions implicitly convert to pointers.)

- `(*fp)(arg)` and `fp = &fname` are optional, but strongly recommended for readability. [link, link2]

(see code in Drive)

Yan, Yokota

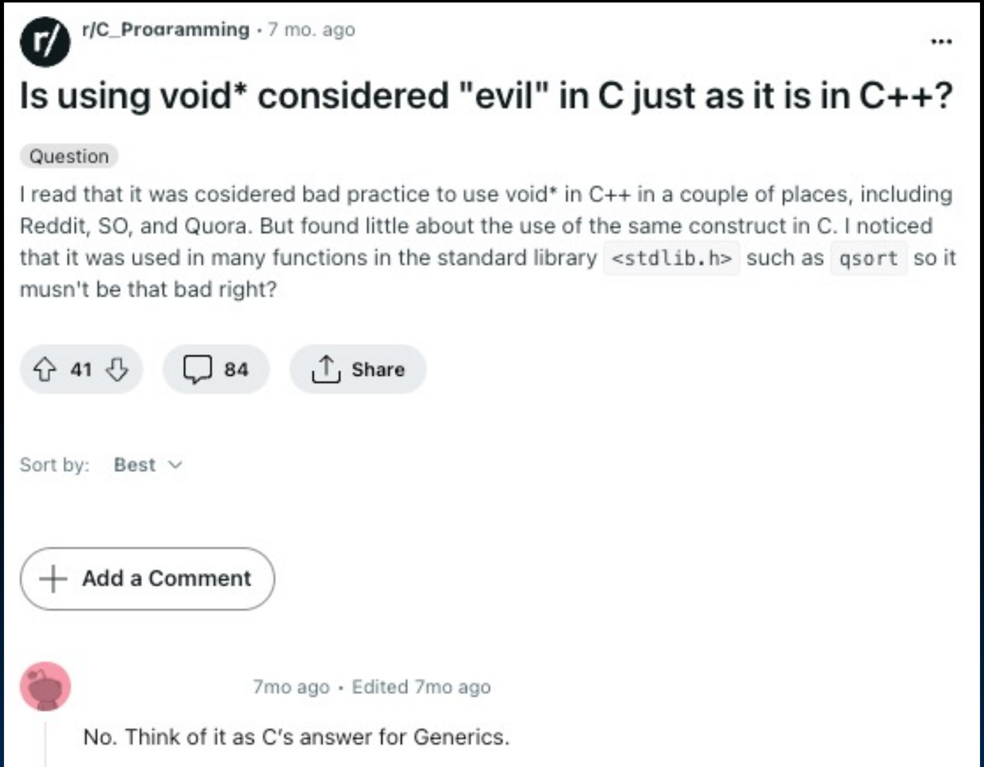
Agenda

Generic Functions

- Floats Wrap-up (denorms)
- Function Pointers
- Generic Functions
- Writing Generics: Episode I
- Writing Generics: Episode II
- Pointer Arithmetic in Generics

Yan, Yokota

When should we use void *?


r/C_Programming · 7 mo. ago

Is using void* considered "evil" in C just as it is in C++?

Question

I read that it was cosidered bad practice to use void* in C++ in a couple of places, including Reddit, SO, and Quora. But found little about the use of the same construct in C. I noticed that it was used in many functions in the standard library `<stdlib.h>` such as `qsort` so it musn't be that bad right?

41 · 84 · Share

Sort by: Best

Add a Comment

7mo ago · Edited 7mo ago

No. Think of it as C's answer for Generics.

Generics is short for **generic functions**.

Yan, Yokota

Why generic functions?

- We want to write general-purpose code.
  - Generic code reduces code duplication and means you can make improvements and fix bugs in one place rather than many.
  - Generics are used throughout C for functions to sort any array, search any array, free arbitrary memory, and more.
- In general, generics:
  - Should generally work regardless of argument type
  - Update blocks of memory, regardless of data type stored in those blocks

Yan, Yokota

Generics example: Dynamic Memory

[review]

```
MALLOC(3)                  Linux Programmer's Manual                  MALLOC(3)

NAME
       malloc, free, calloc, realloc, reallocarray - allocate and free dynamic
       memory

SYNOPSIS
       #include <stdlib.h>

       void *malloc(size_t size);
       void free(void *ptr);
       void *calloc(size_t nmemb, size_t size);
       void *realloc(void *ptr, size_t size);
       void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

Pro Tip: run `man malloc`! The Linux manual page describes the heap API well (as well as what behavior is undefined).

```
typedef struct { ... } TreeNode;
TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));
```

(interesting Ed convo on malloc casting convention: #114db)

Heap `stdlib.h` functions are general purpose and therefore accept/return generic pointers.

Agenda

(This joke will be explained later)

~~Star Wars~~
Writing Generics: Episode I

(1999)

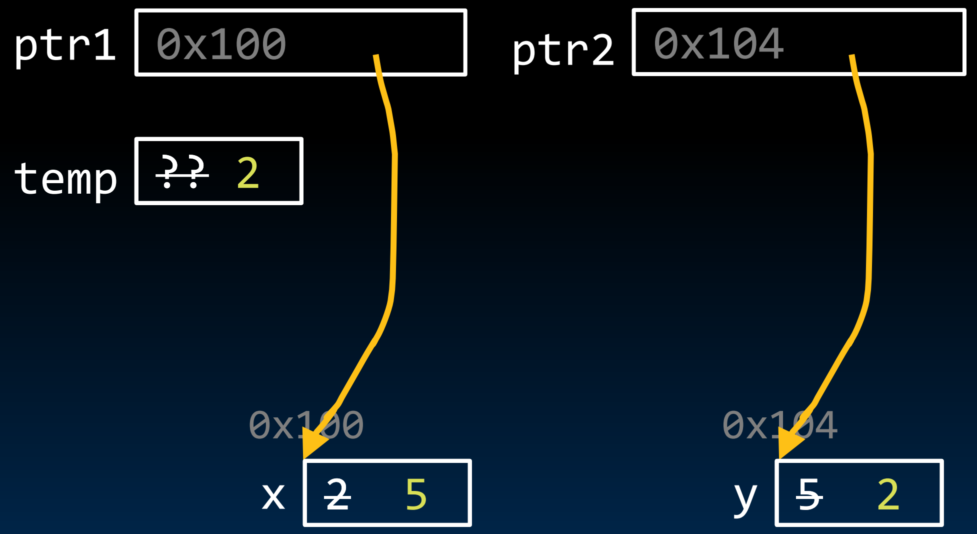
- Floats Wrap-up (denorms)
- Function Pointers
- Generic Functions
- Writing Generics: Episode I
- Writing Generics: Episode II
- Pointer Arithmetic in Generics

Swap integers

```c
void swap_int(int *ptr1, int *ptr2) {
  int temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}


int main() {
  int x = 2;
  int y = 5;
  swap_int(&x, &y);
  // now, x=5 and y=2
  ...
  return 0;
}
```

- Remember: C is pass-by-value.
  - Pass in arguments to `swap_int` as **pointers** to swap local variables x, y in main.



Berkeley UNIVERSITY OF CALIFORNIA
Yan, Yokota

Swap other types

```c
void swap_int(int *ptr1, int *ptr2) {
  int temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}
```

```c
void swap_short(short *ptr1,
                short *ptr2) {
  short temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}
```

```c
void swap_string(char **ptr1,
                 char **ptr2) {
  char *temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}
```

- Remember: C is pass-by-value.
  - Pass in arguments to swap_int as **pointers** to swap local variables x, y in main.
- With different types, the logic is still similar.

Yan, Yokota

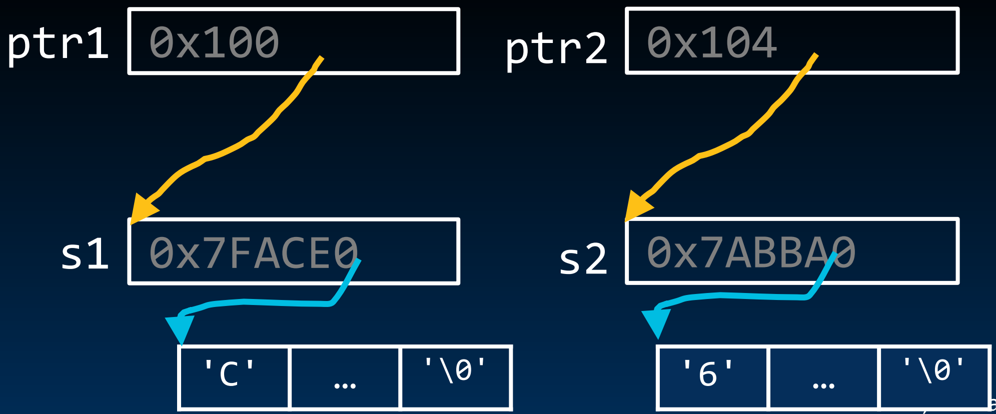
Swap other types

```
void swap_int(int *ptr1, int *ptr2) {
  int temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}

void swap_short(short *ptr1,
                short *ptr2) {
  short temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}

void swap_string(char **ptr1,
                 char **ptr2) {
  char *temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}
```

- Remember: C is pass-by-value.
  - Pass in arguments to swap_int as **pointers** to swap local variables x, y in main.
- With different types, the logic is still similar.



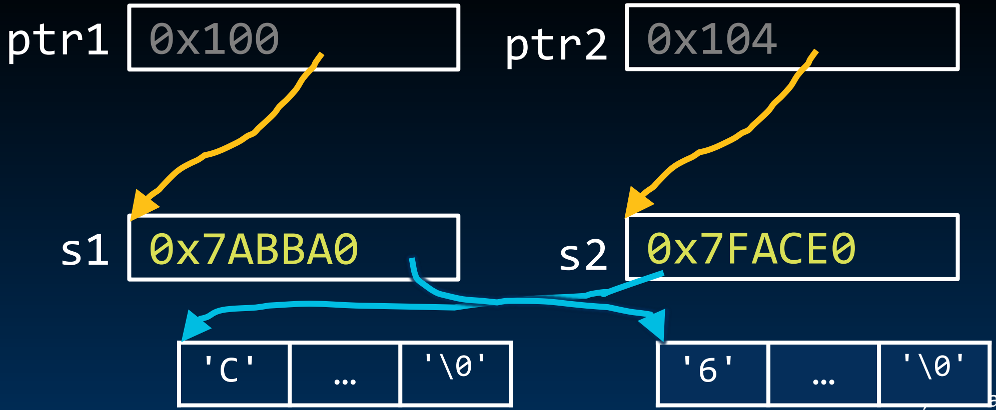
Swap other types

```c
void swap_int(int *ptr1, int *ptr2) {
  int temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}

void swap_short(short *ptr1,
                short *ptr2) {
  short temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}

void swap_string(char **ptr1,
                 char **ptr2) {
  char *temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}
```

- Remember: C is pass-by-value.
  - Pass in arguments to swap_int as **pointers** to swap local variables x, y in main.
- With different types, the logic is still similar.



Generic swap...? An attempt

```
void swap_int(int *ptr1,
              int *ptr2) {
  int temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}
```

...

```
void swap_string(char **ptr1,
                 char **ptr2) {
  char *temp = *ptr1;
  *ptr1 = *ptr2;
  *ptr2 = temp;
}
```

```
void swap(void *ptr1, void *ptr2) {
    // 1. store a copy of data1 in temporary storage
    // 2. copy data2 to location of data1
    // 3. copy data in temporary storage to location of data2
}
```

Can you write this function?

Dereferencing void *

- What happens in each case? Select all that apply.

```c
// Case 1
1 void *ptr = …;
2 printf("%p\n", *ptr);
```

```c
// Case 2
1 void **doubleptr = …;
2 printf("%p\n", *doubleptr);
```

A. Dereference pointer in Line 2
B. Compile error
C. Compile warning
D. Runtime error/undefined behavior
E. Don’t know

Yan, Yokota

Dereferencing void *

- What happens in each case? Select all that apply.

```
// Case 1
1 void *ptr = …;
2 printf("%p\n", *ptr);
```

B, C

```
warning: dereferencing 'void *' pointer
error: invalid use of void expression
```

```
// Case 2
1 void **doubleptr = …;
2 printf("%p\n", *doubleptr);
```

A

- Dereferencing doubleptr gives another pointer, of type void *!
- sizeof(*doubleptr) ==
  sizeof(void *) ==
  sizeof(<any pointer type>)

A. Dereference pointer in Line 2
B. Compile error
C. Compile warning
D. Runtime error/undefined behavior
E. Don't know

To dereference a pointer, we must know the number of bytes to access from memory **at compile time**.

Generics employ generic pointers and therefore cannot use the dereference operator!

Yan, Yokota

The joke, explained

Agenda

(This joke will be explained ~~later)~~ now

~~Star Wars~~

Writing Generics: Episode I

Yan, Yokota

Agenda

Writing Generics: Episode II

- Floats Wrap-up (denorms)
- Function Pointers
- Generic Functions
- Writing Generics: Episode I
- Writing Generics: Episode II
- Pointer Arithmetic in Generics

Yan, Yokota

Generic memory copying

- To access some number of bytes in memory with a generic-typed `void *` pointer, we use two generics in the `string` standard library:

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

  - Copy n bytes from memory area `src` to memory area `dest`. Return a pointer to `dest`.
  - `man memcpy`: *"The memory areas **must not overlap**."*

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

  - Copy n bytes from memory area `src` to memory area `dest`. Return a pointer to `dest`.
  - `man memmove`: *"copying takes place **as though** the bytes in `src` are first copied into a **temporary array** … then copied … to `dest`."*

- Use `memcpy` for performance reasons (unless you know memory areas overlap).
  - Not only is `memcpy` faster, but some implementations of `memmove` actually employ temporary storage (like in C99), which risks running out of memory. [source1, source2]

From dereferencing to memcpy

- Let's rewrite Line 3 to use `memcpy`:

```
  void swap_int(int *ptr1, int *ptr2) {
1   int temp = *ptr1;
2   *ptr1 = *ptr2;
3   *ptr2 = temp;
  }
```

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

function signature

```
memcpy( ptr1 , ptr2 , sizeof(int) );
```

- We must know how many bytes are pointed to!
- Let's add this parameter to `swap`:

```
void swap(void *ptr1, void *ptr2, size_t nbytes);
```


Yan, Yokota

Constructing generic swap

Suppose:

- ptr1 points to nbytes of memory (call this data1)
- ptr2 points to nbytes of memory (call this data2)
- Assume no overlap in these two memory areas

```
void swap_int(int *ptr1,
              int *ptr2) {
  int temp = *ptr1;   // 1
  *ptr1 = *ptr2;      // 2
  *ptr2 = temp;       // 3
}
```

```
void swap(void *ptr1, void *ptr2, size_t nbytes) {
  // 1. store a copy of data1 in temporary storage
  char temp[nbytes];
  memcpy(temp, ptr1, nbytes);

  // 2. copy data2 to location of data1
  memcpy(ptr1, ptr2, nbytes);

  // 3. copy data in temporary storage to location of data2
  memcpy(ptr2, temp, nbytes);
}
```

In C, `sizeof(char)` == 1 byte, regardless of architecture.

To create temporary "generic" storage, declare a local character array (i.e., **buffer**).

If temp is a local **byte array**, this resolves to a pointer.

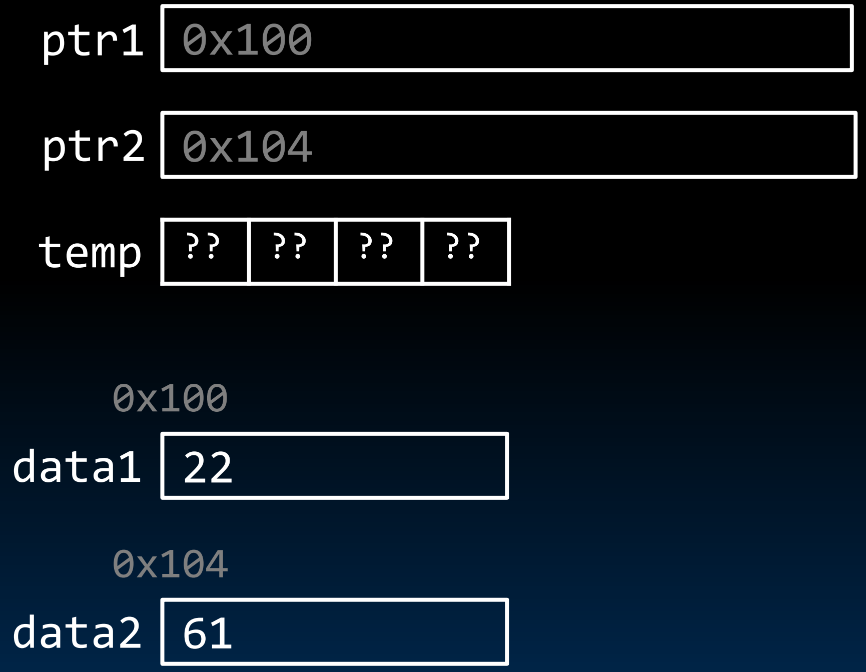
Yan, Yokota

Generic swap

```c
void swap(void *ptr1,
          void *ptr2,
          size_t nbytes) {
  char temp[nbytes];
  memcpy(temp, ptr1, nbytes);
  memcpy(ptr1, ptr2, nbytes);
  memcpy(ptr2, temp, nbytes);
}

int main() {
  int data1 = 22
  int data2 = 61;
  swap(&data1,
       &data2,
       sizeof(data1));
  return 0;
}
```

(assume sizeof(int) == 4)



(see more code examples in Drive)

Yan, Yokota

Generic swap

(assume `sizeof(int) == 4`)

```
void swap(void *ptr1,
          void *ptr2,
          size_t nbytes) {
  char temp[nbytes];
  memcpy(temp, ptr1, nbytes);
  memcpy(ptr1, ptr2, nbytes);
  memcpy(ptr2, temp, nbytes);
}

int main() {
  int data1 = 22
  int data2 = 61;
  swap(&data1,
       &data2,
       sizeof(data1));
  return 0;
}
```

ptr1	0x100
ptr2	0x104
temp	22

	address	value
data1	0x100	61
data2	0x104	22

(see more code examples in Drive)

Yan, Yokota

Agenda

Pointer Arithmetic in Generics

- Floats Wrap-up (denorms)
- Function Pointers
- Generic Functions
- Writing Generics: Episode I
- Writing Generics: Episode II
- Pointer Arithmetic in Generics

Yan, Yokota

Now your turn! swap_ends

- Finally, let's consider generics that operate on a generic array of values.
  - We'll need to do pointer arithmetic!
- Use the swap function to swap the first and last elements in an array:

```
void swap(void *ptr1, void *ptr2, size_t nbytes) {…}






int main() {
  int arr[] = {1, 2, 3, 4, 5}, n = sizeof(arr)/sizeof(arr[0]);
  swap_ends(arr, n, sizeof(arr[0]); // to implement
  …
}
```

0x100 … 0x114

~~1~~ 5	2	3	4	~~5~~ 1

arr

(assume sizeof(int) == 4)

Yan, Yokota

Now your turn! swap_ends

- Finally, let's consider generics that operate on a generic array of values.
  - We'll need to do pointer arithmetic!
- Use the swap function to swap the first and last elements in an array:

```
void swap(void *ptr1, void *ptr2, size_t nbytes) {…}
void swap_ends(void *arr,
               size_t nelems,
               size_t nbytes) {
    …
}

int main() {
  int arr[] = {1, 2, 3, 4, 5}, n = sizeof(arr)/sizeof(arr[0]);
  swap_ends(arr, n, sizeof(arr[0]);
  …
}
```

"Array" parameter needs # elements in array

Generics need size of each element (in bytes)

(assume sizeof(int) == 4)

0x100 0x114

~~1~~ 5	2	3	4	~~5~~ 1

arr

Yan, Yokota

Now your turn! swap_ends

[concept check]

- Finally, let's consider generics that operate on a generic array of values.
  - We'll need to do pointer arithmetic!
- Use the swap function to swap the first and last elements in an array:

```
void swap(void *ptr1, void *ptr2, size_t nbytes) {…}
void swap_ends(void *arr,
               size_t nelems,
               size_t nbytes) {
    swap(arr, __???__, nbytes);
}

int main() {
  …
  swap_ends(arr, n, sizeof(arr[0]);
  …
}
```

A. `arr + nelems`
B. `arr + nelems*nbytes`
C. `(char *) arr + nelems*nbytes`
D. `arr + nelems - 1`
E. `arr + (nelems - 1)*nbytes`
F. `(char *) arr + (nelems - 1)*nbytes`
G. Something else

0x100 0x114

~~1~~ 5	2	3	4	~~5~~ 1

arr

(assume sizeof(int) == 4)

Yan, Yokota

What should go in the blank?

0

A. arr + nelems

0%

B. arr + nelems*nbytes

0%

C. (char *) arr + nelems*nbytes

0%

D. arr + nelems - 1

0%

E. arr + (nelems - 1)*nbytes

0%

F. (char *) arr + (nelems - 1)*nbytes

0%

G. Something else

0%

Generic byte arithmetic with (char *) typecast

Pointer arithmetic in generics must be **bytewise** arithmetic.
1. Cast void * pointer to char *.
2. Pointer arithmetic is then effectively byte-wise!

```
void swap(void *ptr1, void *ptr2, size_t nbytes) {…}
void swap_ends(void *arr,
               size_t nelems,
               size_t nbytes) {
    swap(arr, ___???___, nbytes);
}

int main() {
  …
  swap_ends(arr, n, sizeof(arr[0]));
  …
}
```

A. arr + nelems
B. arr + nelems*nbytes
C. (char *) arr + nelems*nbytes
D. arr + nelems - 1
E. arr + (nelems - 1)*nbytes
F. (char *) arr + (nelems - 1)*nbytes
G. Something else

⚠ Caution! Option E (no explicit cast to char *) will work on hive but is not standard C [source].

(assume sizeof(int) == 4)

0x100 0x114

~~1~~ 5	2	3	4	~~5~~ 1

arr

Yan, Yokota

And in Conclusion…

- Function pointers enable higher-order functions in C.
  - map, filter, sorting, etc.
- Generic functions (i.e., generics), use `void *` pointers to operate on memory.
  - Generics are widely present in the C standard library! (`malloc`, `memcpy`, `qsort`, …)
  - Generics require a solid understanding of memory! By manipulating arbitrary bytes, you risk violating data boundaries, e.g., "Frankenstein"-ing two halves of `int`s.
- Reminders when writing generics:
  - Generic pointers do not support dereferencing, as the number of bytes to access from memory is not known at compile-time.
  - Instead, use byte handling functions (`memcpy`, `memmove`).
  - Pointer arithmetic: first cast to byte arrays with `(char *)`.

Yan, Yokota

The C standard library header string.h [optional]

- string.h contains not only functions for handling strings, but also those for handling memory.
  - Common memory functions: memcpy, memmove, memset, …
- While the header name is a bit misleading, memory handling functions operate byte-by-byte, and strings are effectively (null-terminated) byte arrays.
- glibc’s strncpy implementation:

```
char *strncpy(char *dest, const char *src, size_t n) {
    size_t size = strnlen(src, n); // max(strlen(src), n)
    if (size != n)
        memset(dest + size, '\0’, n – size); // write '\0's
    return memcpy(dest, src, size);
}
```

https://code.woboq.org/userspace/glibc/string/strncpy.c.html

Full code for Function Pointer example

```c
/* map a function onto int array */
void mutate_map(int arr[], int n,
                int(*fp)(int)) {
  for (int i = 0; i < n; i++)
    arr[i] = (*fp)(arr[i]);
}
int multiply2 (int x) {  return  2 * x;   }
int multiply10(int x) {  return 10 * x;   }
int main() {
  int arr[] = {3,1,4};
  int n = sizeof(arr)/sizeof(arr[0]);
  print_array(arr, n); // 1
  mutate_map (arr, n, &multiply2);
  print_array(arr, n); // 2
  mutate_map (arr, n, &multiply10);
  print_array(arr, n); // 3
  return 0;
}
```

```
$ ./map_func
3 1 4
6 2 8
60 20 80
```

(see code in Drive)

(C quirk: Outside of declaration, functions implicitly convert to pointers.)

`(*fp)(arg)` and `fp = &fname` are optional, but strongly recommended for readability. [link, link2]

Yan, Yokota

Full code for swap_ends

```c
void swap(void *ptr1, void *ptr2, size_t nbytes) {
  char temp[nbytes];
  memcpy(temp, ptr1, nbytes);
  memcpy(ptr1, ptr2, nbytes);
  memcpy(ptr2, temp, nbytes);
}
void swap_ends(void *arr,
               size_t nelems,
               size_t nbytes) {
    swap(arr, (char *) arr + (nelems - 1)*nbytes, nbytes);
}

int main() {
  int arr[] = {1, 2, 3, 4, 5}, n = sizeof(arr)/sizeof(arr[0]);
  swap_ends(arr, n, sizeof(arr[0]));
  …
}
```

(see code in Drive)

⚠ Caution! With generic pointers, omitting the explicit char * cast to) will work on hive (gcc uses GNU C) but is not standard C [source].

(assume sizeof(int) == 4)

0x100 … 0x114

~~1~~ 5	2	3	4	~~5~~ 1

arr

Yan, Yokota