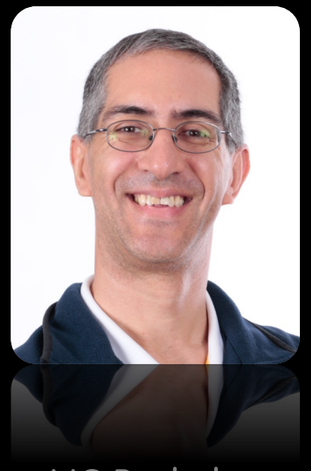


UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

C Generics and Function Pointers

Agenda

Today's lecture

- Function Pointers
- Generic Functions
- Writing Generics: Episode I
- Writing Generics: Episode II
- Pointer Arithmetic in Generics

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.

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

`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

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

Agenda

Function Pointers

- Function Pointers
- Generic Functions
- Writing Generics: Episode I
- Writing Generics: Episode II
- Pointer Arithmetic in Generics

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!

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

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

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

```
/* 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)

Garcia, Kao

(2/3) Function Pointer Example

```
/* map a function onto int array */
1  void mutate_map(int arr[], int n,
2                  int(*fp)(int)) {
3    for (int i = 0; i < n; i++)
4      arr[i] = (*fp)(arr[i]);
5  }
6  int multiply2 (int x) {  return  2 * x;  }
7  int multiply10(int x) {  return 10 * x;  }
8  int main() {
9    int arr[] = {3,1,4}
10   int n = sizeof(arr)/sizeof(arr[0]);
11   print_array(arr, n); // 1
12   mutate_map (arr, n, &multiply2);
13   print_array(arr, n); // 2
14   mutate_map (arr, n, &multiply10);
15   print_array(arr, n); // 3
…    return 0;
   }
```

The `fp` parameter is a function **pointer**.

Type: `int` parameter, `int` retval

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

(see code in Drive)

Garcia, Kao

Function Pointer Example

```
/* 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;
}
```

The `fp` parameter is a function pointer.

Type: `int` parameter, `int` retval

Call the function pointed to by the function pointer.

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

(see code in Drive)

Function Pointer Example

```
/* 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;
}
```

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.

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

(see code in Drive)

Function Pointer Example

```
/* 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]

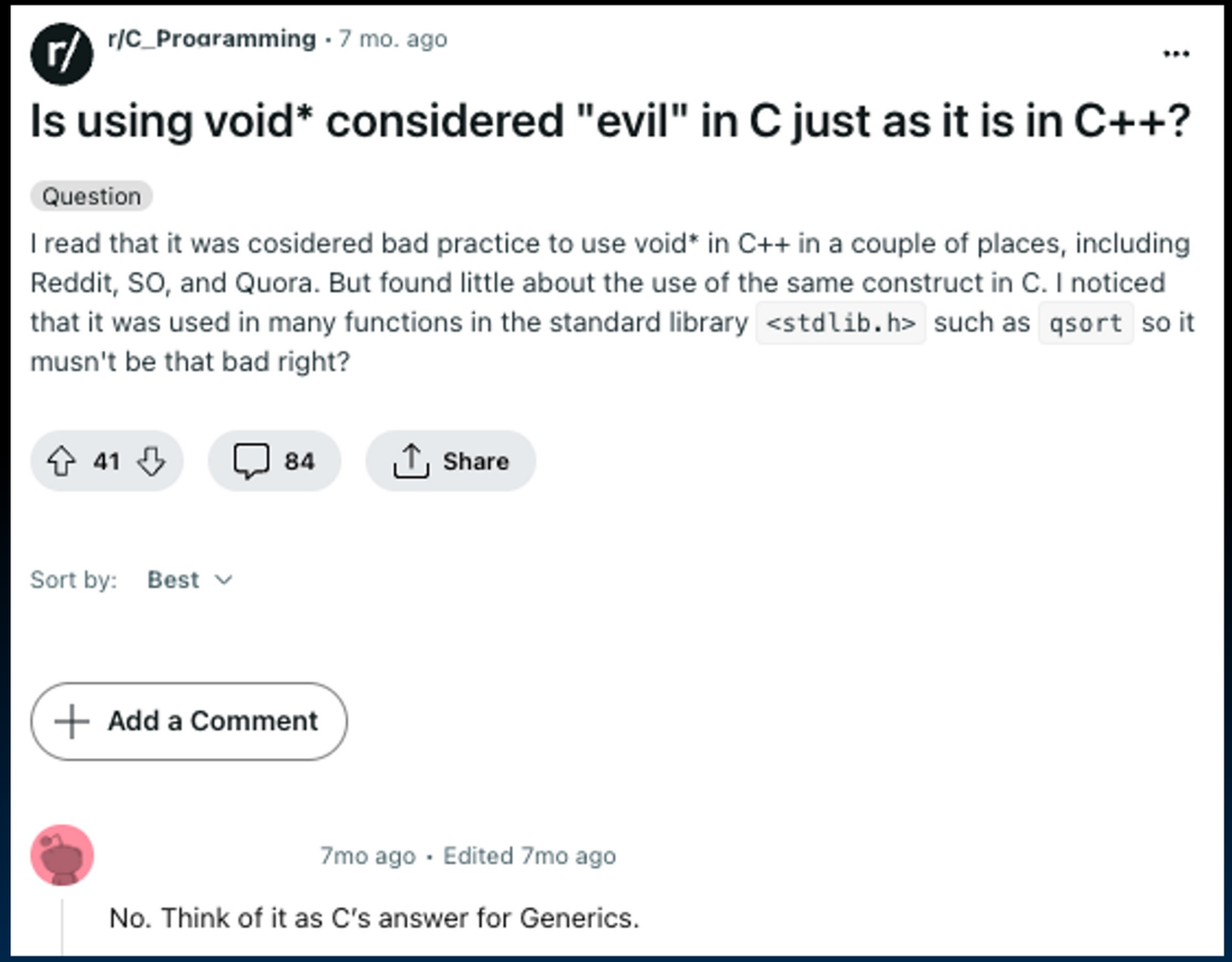
(see code in Drive)

Agenda

Generic Functions

- Function Pointers
- Generic Functions
- Writing Generics: Episode I
- Writing Generics: Episode II
- Pointer Arithmetic in Generics

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**.

source [accessed 01/29/24]

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

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

Writing Generics: Episode I

Or, how to be a star (*) lord!

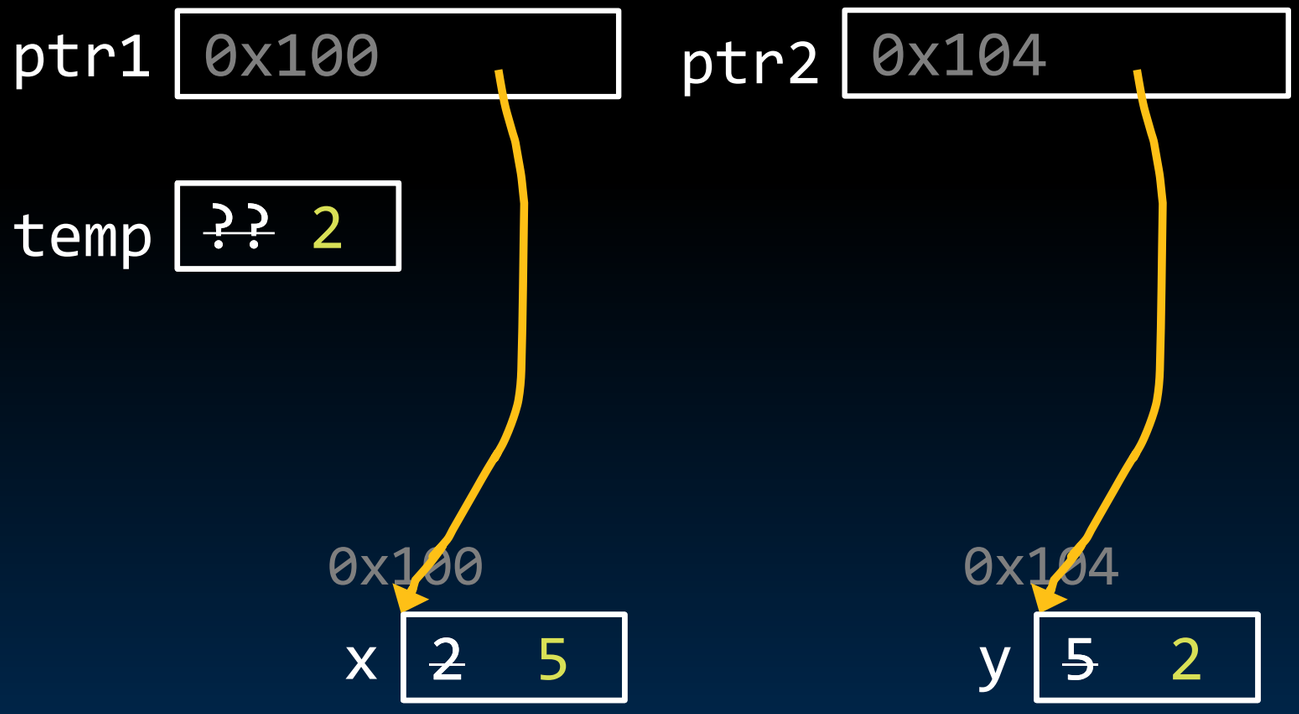
- Function Pointers
- Generic Functions
- Writing Generics: Episode I
- Writing Generics: Episode II
- Pointer Arithmetic in Generics

Swap integers

```
1  void swap_int(int *ptr1, int *ptr2) {
2    int temp = *ptr1;
3    *ptr1 = *ptr2;
4    *ptr2 = temp;
5  }


6  int main() {
7    int x = 2;
8    int y = 5;
9    swap_int(&x, &y);
10   // now, x=5 and y=2
11   ...
12   return 0;
13 }
```

Remember: C is pass-by-value.

Pass in arguments to `swap_int` as **pointers** to swap local variables `x`, `y` in `main`.



Swap other types

```
1  void swap_int(int *ptr1, int *ptr2) {
2    int temp = *ptr1;
3    *ptr1 = *ptr2;
4    *ptr2 = temp;
5  }

6  void swap_short(short *ptr1,
                   short *ptr2) {
7    short temp = *ptr1;
8    *ptr1 = *ptr2;
9    *ptr2 = temp;
10 }

11 void swap_string(char **ptr1,
                   char **ptr2) {
12   char *temp; temp = *ptr1;
13   *ptr1 = *ptr2;
14   *ptr2 = temp;
15 }
```

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.

Garcia, Kao

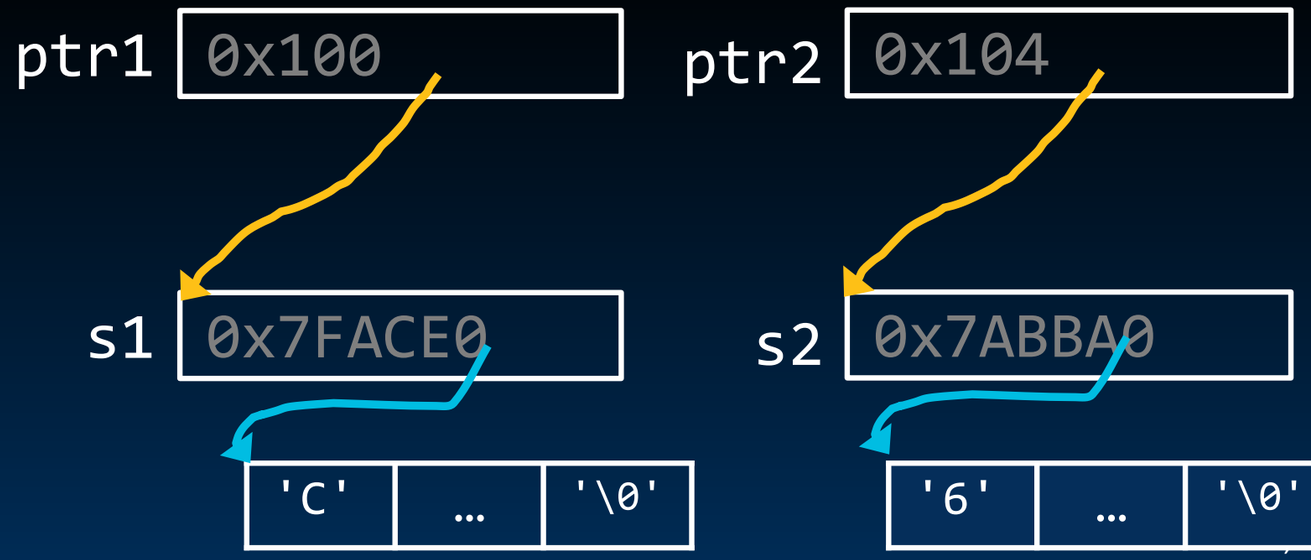
Swap other types

```
1  void swap_int(int *ptr1, int *ptr2) {
2    int temp = *ptr1;
3    *ptr1 = *ptr2;
4    *ptr2 = temp;
5  }

6  void swap_short(short *ptr1,
                   short *ptr2) {
7    short temp = *ptr1;
8    *ptr1 = *ptr2;
9    *ptr2 = temp;
10 }

11 void swap_string(char **ptr1,
                   char **ptr2) {
12   char *temp; temp = *ptr1;
13   *ptr1 = *ptr2;
14   *ptr2 = temp;
15 }
```

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.



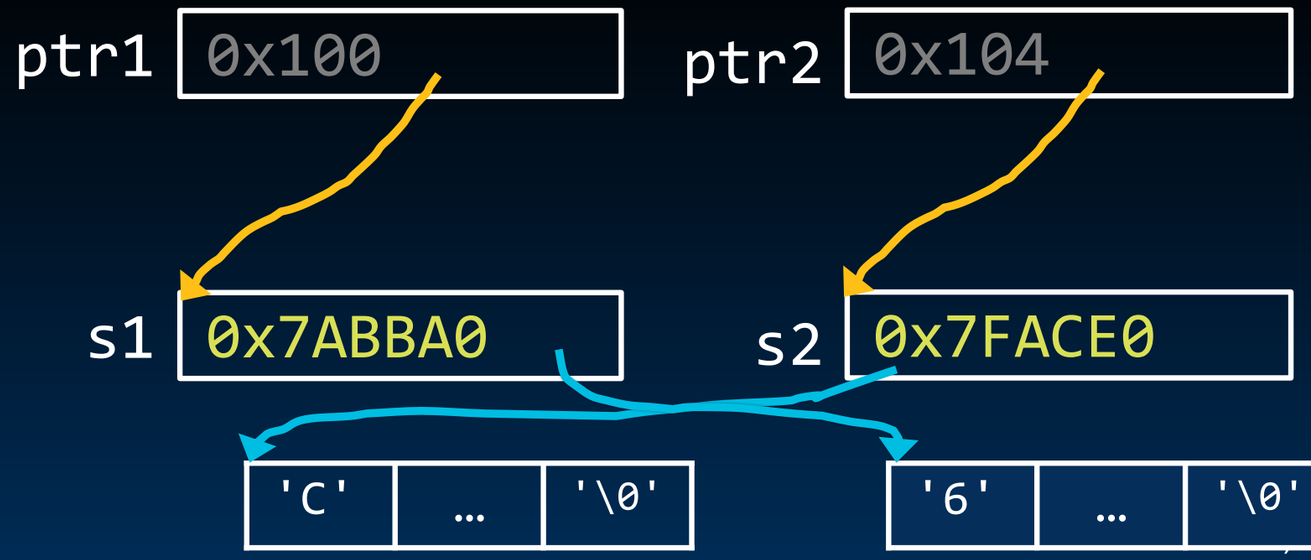
Swap other types

```
1  void swap_int(int *ptr1, int *ptr2) {
2    int temp = *ptr1;
3    *ptr1 = *ptr2;
4    *ptr2 = temp;
5  }

6  void swap_short(short *ptr1,
                   short *ptr2) {
7    short temp = *ptr1;
8    *ptr1 = *ptr2;
9    *ptr2 = temp;
10 }

11 void swap_string(char **ptr1,
                   char **ptr2) {
12   char *temp; temp = *ptr1;
13   *ptr1 = *ptr2;
14   *ptr2 = temp;
15 }
```

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; 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 *

[concept check]

What happens in each case? Select all that apply.

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

```
// 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

Dereferencing void *

What happens in each case? Select all that apply.

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

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

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

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!

Agenda

Writing Generics: Episode II

- Function Pointers
- Generic Functions
- Writing Generics: Episode I
- Writing Generics: Episode II
- Pointer Arithmetic in Generics

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`:

```
1   void swap_int(int *ptr1, int *ptr2) {
2     int temp = *ptr1;
3     *ptr1 = *ptr2;
4     *ptr2 = temp;
5   }
```

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

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

function signature

- 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);
```

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.

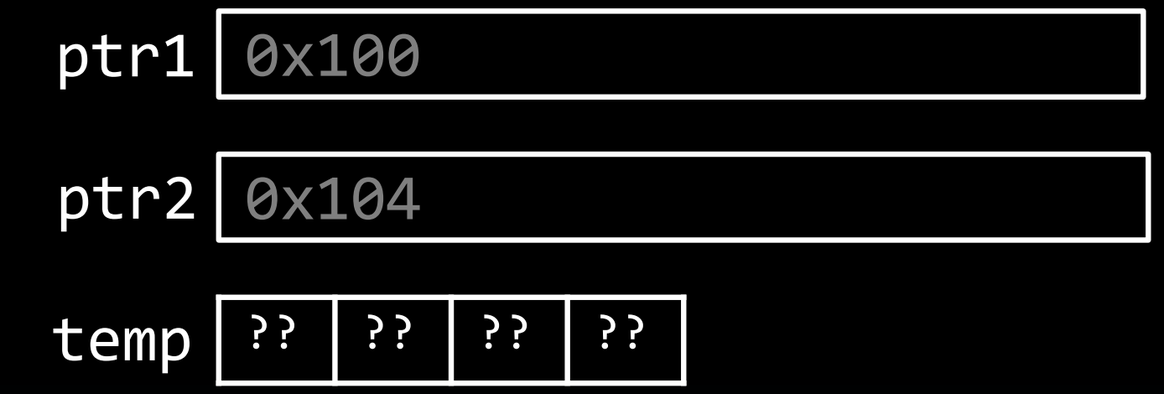
To create temporary "generic" storage, declare a local character array (i.e., **buffer**).

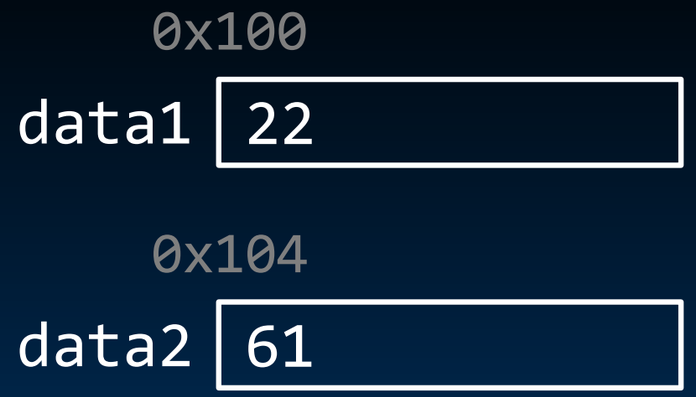
If temp is a local **byte array**, this resolves to a pointer.

Generic swap

```
1  void swap(void *ptr1,
             void *ptr2,
             size_t nbytes) {
2    char temp[nbytes];
3    memcpy(temp, ptr1, nbytes);
4    memcpy(ptr1, ptr2, nbytes);
5    memcpy(ptr2, temp, nbytes);
6  }

7  int main() {
8    int data1 = 22
9    int data2 = 61;
10   swap(&data1,
          &data2,
          sizeof(data1));
11   return 0;
12 }
```

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



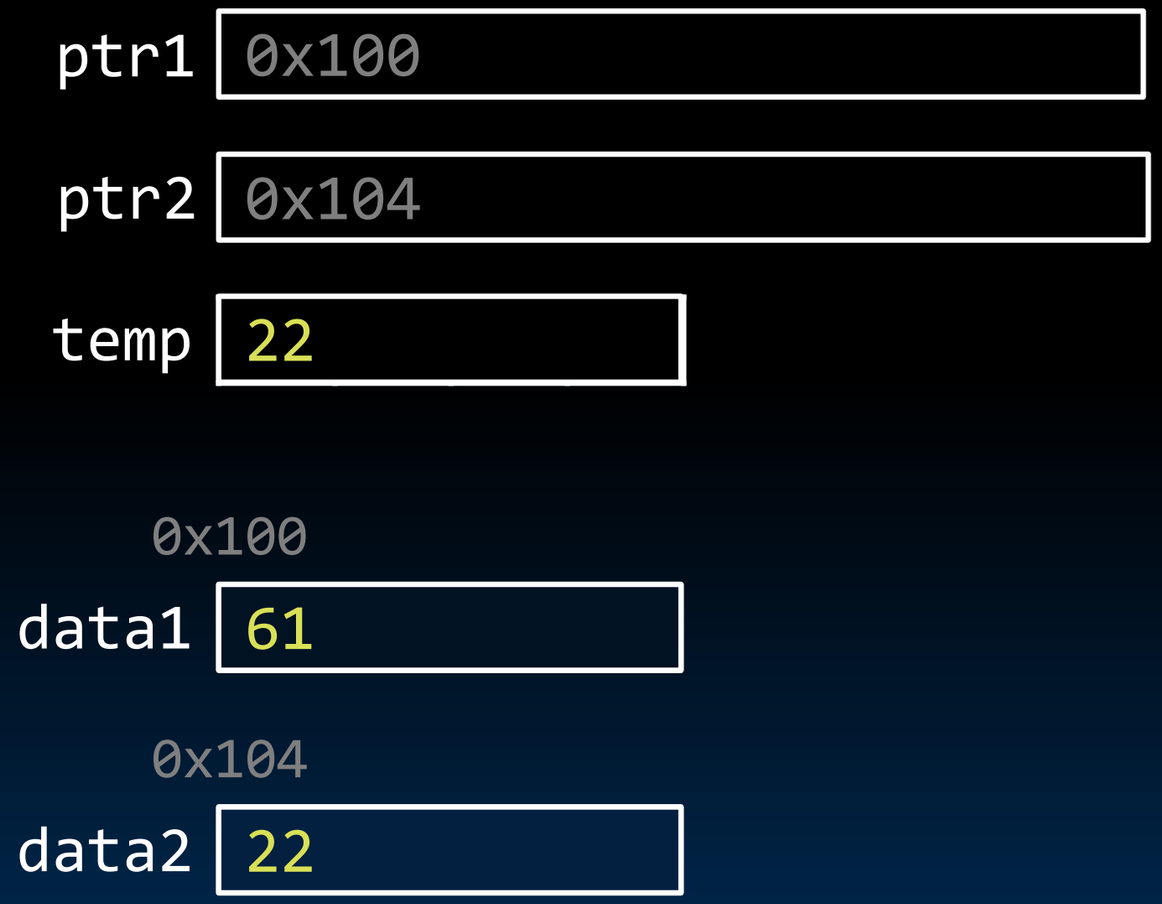


(see more code examples in Drive)

Generic swap

```
1  void swap(void *ptr1,
             void *ptr2,
             size_t nbytes) {
2    char temp[nbytes];
3    memcpy(temp, ptr1, nbytes);
4    memcpy(ptr1, ptr2, nbytes);
5    memcpy(ptr2, temp, nbytes);
6  }

7  int main() {
8    int data1 = 22
9    int data2 = 61;
10   swap(&data1,
          &data2,
          sizeof(data1));
11   return 0;
12 }
```

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



(see more code examples in Drive)

Agenda

Pointer Arithmetic in Generics

- Function Pointers
- Generic Functions
- Writing Generics: Episode I
- Writing Generics: Episode II
- Pointer Arithmetic in Generics

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:

```
1  void swap(void *ptr1, void *ptr2, size_t nbytes) {…}
2
3
4
5
6

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

(assume sizeof(int) == 4)

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:

```
1  void swap(void *ptr1, void *ptr2, size_t nbytes) {…}
2  void swap_ends(void *arr,
3                 size_t nelems,
4                 size_t nbytes) {
5      …
6  }

7  int main() {
8    int arr[] = {1, 2, 3, 4, 5}, n = sizeof(arr)/sizeof(arr[0]);
9    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)

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:

```
1  void swap(void *ptr1, void *ptr2, size_t nbytes) {…}
2  void swap_ends(void *arr,
3                 size_t nelems,
4                 size_t nbytes) {
5      swap(arr, __???__, nbytes);
6  }

7  int main() {
8    …
9    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

(assume sizeof(int) == 4)

L07 C4 What should go in the blank to complete swap_ends?

0

arr + nelems

0%

arr + nelems * nbytes

0%

arr + nelems - 1

0%

arr + (nelems - 1) * nbytes

0%

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

0%

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!

```
1  void swap(void *ptr1, void *ptr2, size_t nbytes) {…}
2  void swap_ends(void *arr,
3                 size_t nelems,
4                 size_t nbytes) {
5      swap(arr, ___???___, nbytes);
6  }

7  int main() {
8    …
9    swap_ends(arr, n, sizeof(arr[0]);
…    …
     }
```

(assume sizeof(int) == 4)

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].

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 *)`.

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

```
/* map a function onto int array */
1  void mutate_map(int arr[], int n,
2                  int(*fp)(int)) {
3    for (int i = 0; i < n; i++)
4      arr[i] = (*fp)(arr[i]);
5  }
6  int multiply2 (int x) {  return  2 * x;  }
7  int multiply10(int x) {  return 10 * x;  }
8  int main() {
9    int arr[] = {3,1,4}
10   int n = sizeof(arr)/sizeof(arr[0]);
11   print_array(arr, n); // 1
12   mutate_map (arr, n, &multiply2);
13   print_array(arr, n); // 2
14   mutate_map (arr, n, &multiply10);
15   print_array(arr, n); // 3
…    return 0;
   }
```

(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]

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

Full code for swap_ends

```
1  void swap(void *ptr1, void *ptr2, size_t nbytes) {
     char temp[nbytes];
     memcpy(temp, ptr1, nbytes);
     memcpy(ptr1, ptr2, nbytes);
     memcpy(ptr2, temp, nbytes);
   }
2  void swap_ends(void *arr,
3                 size_t nelems,
4                 size_t nbytes) {
5      swap(arr, (char *) arr + (nelems - 1)*nbytes, nbytes);
6  }

7  int main() {
8    int arr[] = {1, 2, 3, 4, 5}, n = sizeof(arr)/sizeof(arr[0]);
9    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)