# 600.220 Intermediate Programming

## Pointer Operations

# Outline

- Pointer operations
- Pointer arithmetic for arrays
- Pointer difference type
- Pointers and c-strings

# Pointer Operations

- Assignment
- Comparisons
- Arithmetic

# Pointer Assignment

- `ptr1 = ptr2;` assignment works for pointers of the same type
- only the memory address in ptr2 is copied
- they reference the same memory location
  - `*ptr1 = 10;` leads to `*ptr2 == 10` being true

# Pointer Comparisons

- `ptr1 == ptr2` and `ptr1 != ptr2` compare the addresses inside the pointer variables for equality (do they point to the same memory location?)
- `ptr == NULL` will check if ptr is 0 (good initialization value to use)
- `ptr1 < ptr2` compares addresses
  - useful if pointer variables reference memory in the same array for example (similar to comparing indices)

# Pointer Arithmetic

- Operators +, -, +=, -= can be used with other pointers or integers for the 2nd operand
- Most often used on pointers into arrays
- Doesn't add the actual number, it adds that number times how many bytes each element takes up based on the pointer base type
  - for variable `int * p`, code `p+1` will in fact add 4 bytes (`sizeof(int)`) to `p`'s address

# Pointer Arithmetic and Arrays

- For a declared array like `int a[10]`, `a` is "really" just a (non-modifiable) address that starts a block of memory.
- Writing `a` is generally the same as writing `&a[0]`
- `a[3]` is a synonym for `*(a + 3)` (offset three from pointer to start of array)
- `&a[3]` is a synonym for `a + 3`

# Pointer Arithmetic and Arrays - example

```c
// pointerArith.c:
#include <stdio.h>
#include <stdlib.h>
int main() {

   int array[] = {2, 4, 6};

   printf("array[1] = %d, ", array[1]);
   printf(" *(array + 1) = %d, ", *(array+1));
   printf(" array = %p\n", (void *) array);
   printf(" &array[1] = %p, ",(void *) &array[1]);
   printf(" array + 1 = %p\n", (void *) (array + 1));
   return 0;
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic pointerArith.c
$ ./a.out
array[1] = 4,  *(array + 1) = 4,  array = 0x7ffee992989c
 &array[1] = 0x7ffee99298a0,  array + 1 = 0x7ffee99298a0
```

# Checkpoint Poll!

What is the correct output?

```c
#include <stdio.h>

int main() {
  int a[] = {1, 1, 2, 3, 4};
  printf("%d ", (*a) + 7);
  printf("%d ", *(a + 4));
  printf("%d ", *(&a[1] + 1));
  return 0;
}
```

A. 2 4 3

B. 8 4 2

C. 8 5 2

D. 7 4 2

E. The program does not compile and/or has an error.

# Pointer difference (subtraction)

- `ptrdiff_t` is a predefined type in library `stddef.h`
- this is the resulting type when subtracting pointers (memory addresses)
- essentially equivalent to the `long` integer type

```c
// pointerDiff.c:
#include <stdio.h>
#include <stddef.h>
int main() {

   int array[] = {2, 4, 6, 8, 10, 12, 14, 16};
   int * start = &array[0];  // first element address
   int * end = &array[7];    // last element address
   ptrdiff_t capacity = end - start + 1;  // number of elements in array

   printf("start = %p, ", (void *) start);
   printf("end = %p, ", (void *) end);
   printf("capacity = %ld\n", capacity);  // print as long int
   return 0;
}
```

```
$ gcc -std=c99 -Wall -Wextra -pedantic pointerDiff.c
$ ./a.out
start = 0x7ffee2a4e880, end = 0x7ffee2a4e89c, capacity = 8
```

# Checkpoint Poll!

What is the correct output?

```
char str1[] = "original";
char * str2;

str2 = str1;
*str2 = 'O';
str2 += 3;
*str2 = 'G';
str2 += 3;

printf("%s %s\n", str1, str2);
```

A. `original Original`

B. `original OriGinal`

C. `OriGinal OriGinal`

D. `OriGinal al`

E. The program does not compile and/or has an error.

# Pointers vs. C-strings - common errors

Given these declarations

```
char str1[] = "original";
char * str2;
```

- Why is this bad code? `strcpy(str2, str1);`
- Why is this bad code? `str1 += 3;`
- Why doesn't `str2 = str1;` make a copy of "original"?

# Pointers vs. C-strings - common error explanations

Given these declarations

```
char str1[] = "original";
char * str2;
```

- `strcpy(str2, str1);` will crash because memory for str2 was never allocated
- `str1 += 3;` will not compile because we cannot change the address stored in a statically declared array variable
- `str2 = str1;` only copies the memory address stored in str1, not the whole array