

CS 241: Systems Programming

Lecture 15. Strings

Fall 2019

Prof. Stephen Checkoway

Review of last lecture

Arrays are contiguous sequences of objects

- `double blah[x];` // declares an array of length `x`

Pointers hold the address of an object (or 0)

- `long *p;` // declares a pointer `p` that can point to a long
- `p = &x;` // sets the value of `p` to the address of `x`
- `long y = *p;` // sets `y` to the value of the long pointed to by `p`
- `*p = 5;` // sets the value of the long pointed to by `p` to be 5

```
long x = 5;  
long y = -10;  
long *p = &x;  
long *q = &y;  
*p = *q + 2;  
q = p;  
p = 0;  
printf("%ld\n", *q); // What is printed?
```

A. -10

D. 5

B. -8

E. 7

C. 0

```
long x = 5;  
long y = -10;  
long *p = &x;  
long *q = &y;  
*p = *q + 2;  
q = p;  
*p = 0;  
printf("%ld\n", *q); // What is printed?
```

A. -10

D. 5

B. -8

E. 7

C. 0

Array decay

C has array *objects* but not array *values*

When an array is used as an value, it **decays** into a pointer to its 0th element

```
#include <stdio.h>
```

```
int main(void) {  
    int arr[] = { 25, 18, -9 };  
    int *ptr = arr; // decay to pointer, same as &arr[0]  
    ptr[1] = 77;    // We can use [] syntax with pointers too!  
    for (size_t idx = 0; idx < sizeof arr / sizeof arr[0]; ++idx) {  
        printf("arr[%zu] = %d\n", idx, arr[idx]);  
    }  
    return 0;  
}
```

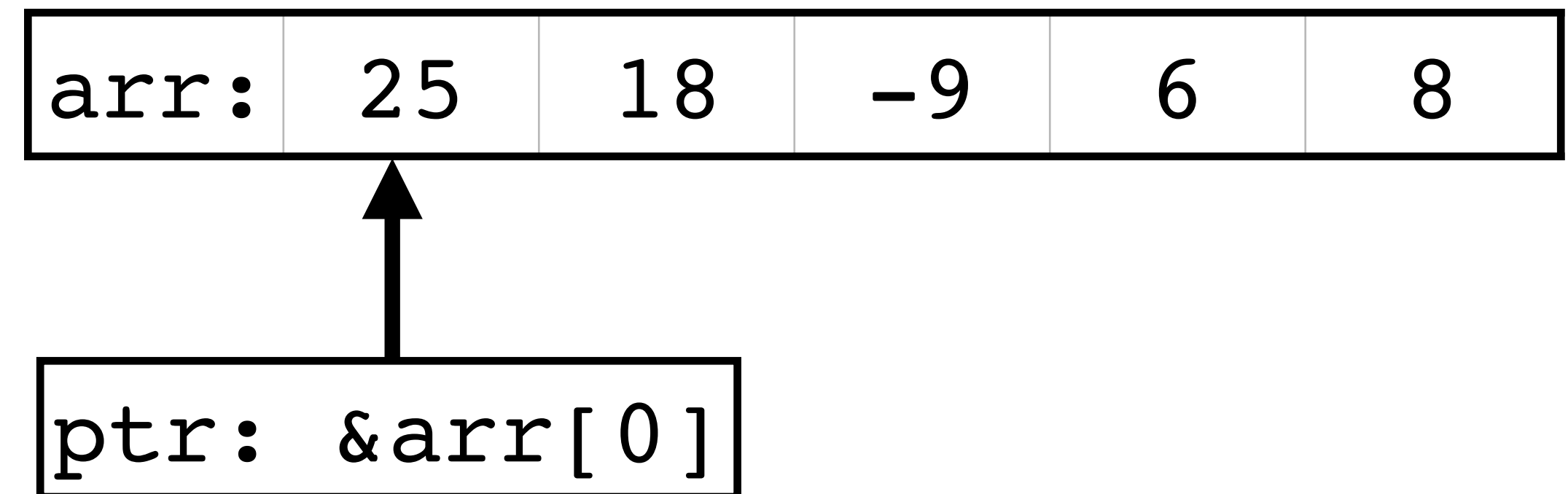
Prints:

arr[0] = 25

arr[1] = 77

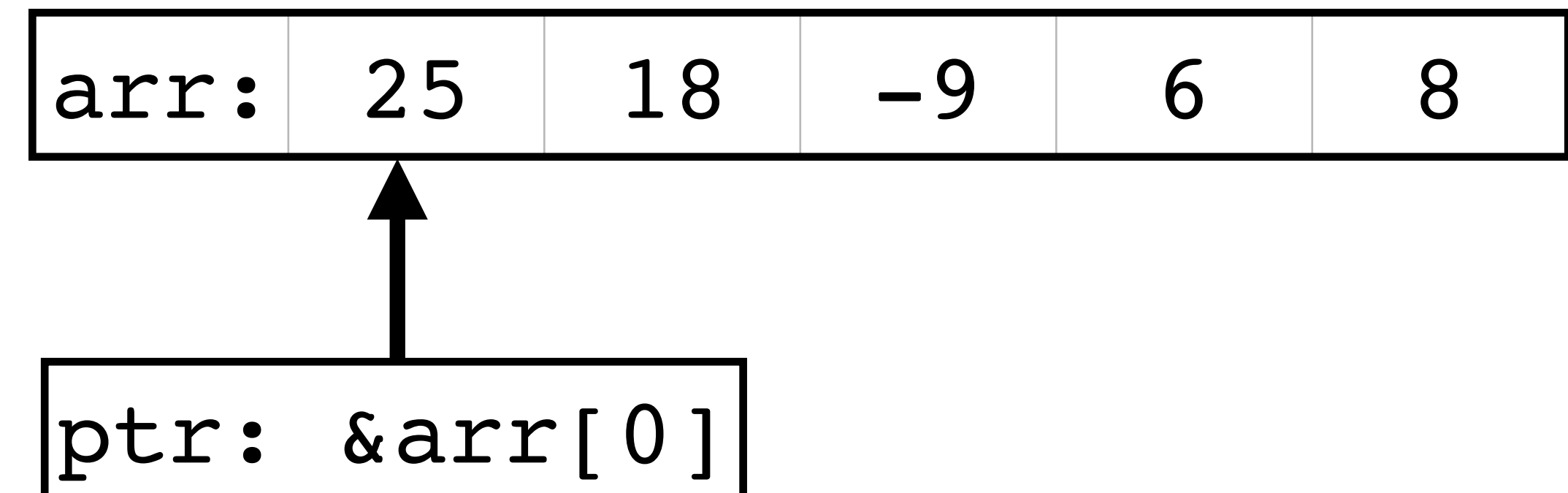
arr[2] = -9

Pointer arithmetic



Pointer arithmetic

```
int arr[] = { 25, 18, -9, 6, 8 };  
int *ptr = arr;
```

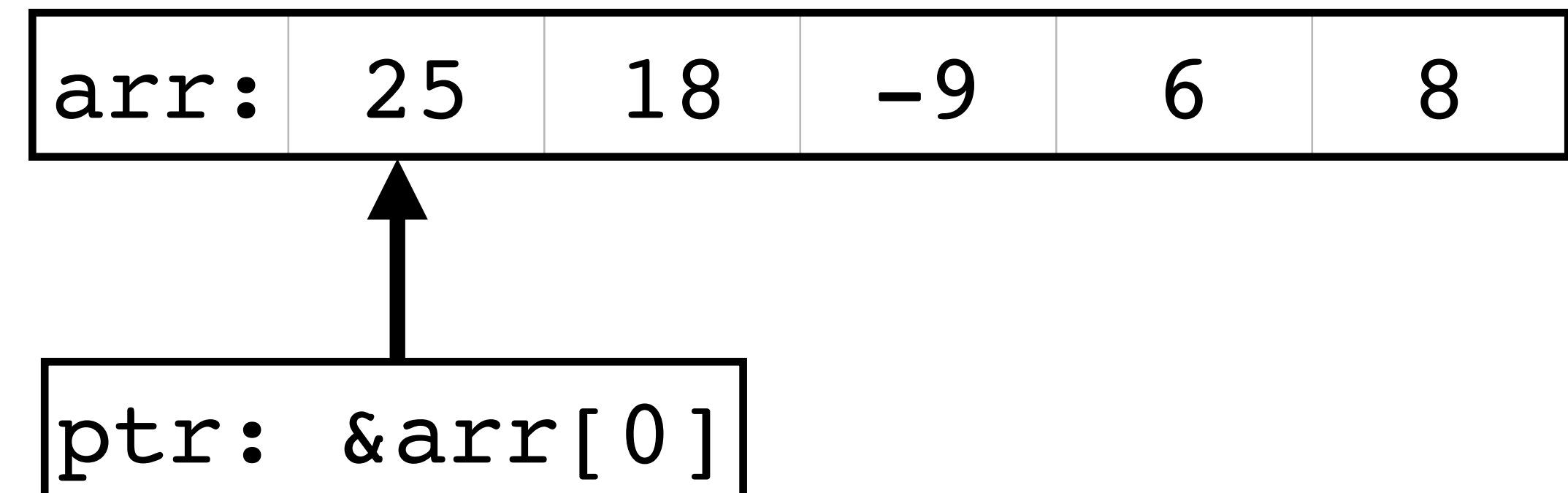


Pointer arithmetic

```
int arr[] = { 25, 18, -9, 6, 8 };  
int *ptr = arr;
```

Adding pointers and integers

- `ptr` points to the 0th element of `arr`
- `ptr + 1` points to the 1st element of `arr`
- `ptr + 2` points to the 2nd element of `arr`
- ...



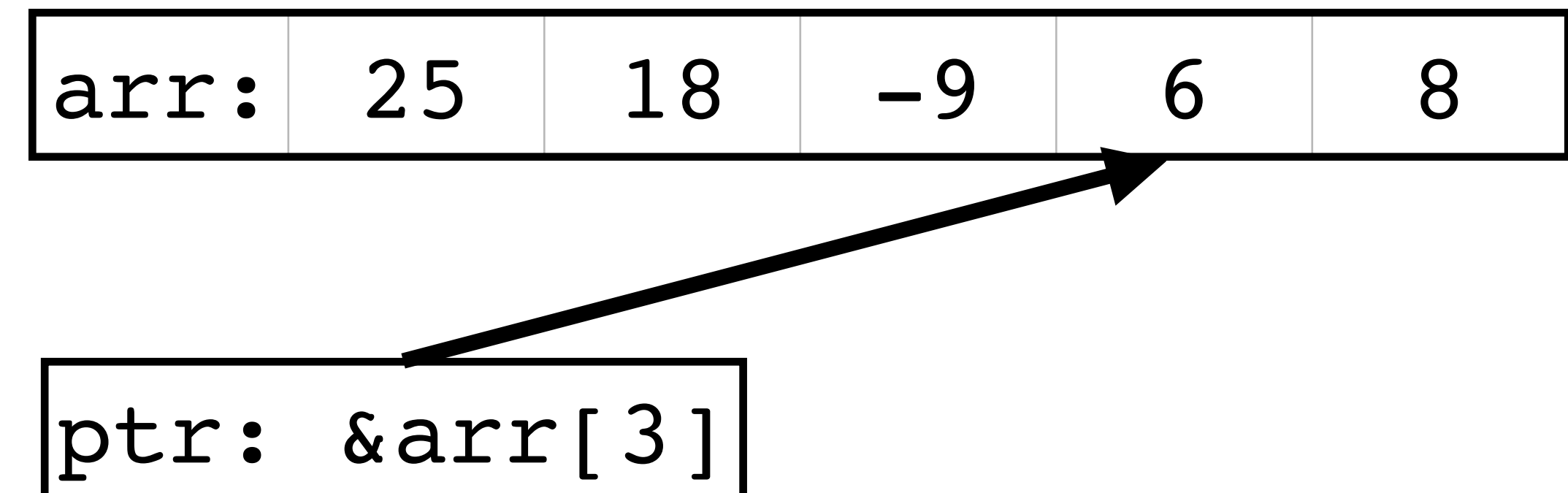
Pointer arithmetic

```
int arr[] = { 25, 18, -9, 6, 8 };  
int *ptr = arr;
```

Adding pointers and integers

- `ptr` points to the 0th element of `arr`
- `ptr + 1` points to the 1st element of `arr`
- `ptr + 2` points to the 2nd element of `arr`
- ...

```
ptr += 3;
```



Pointer subtraction

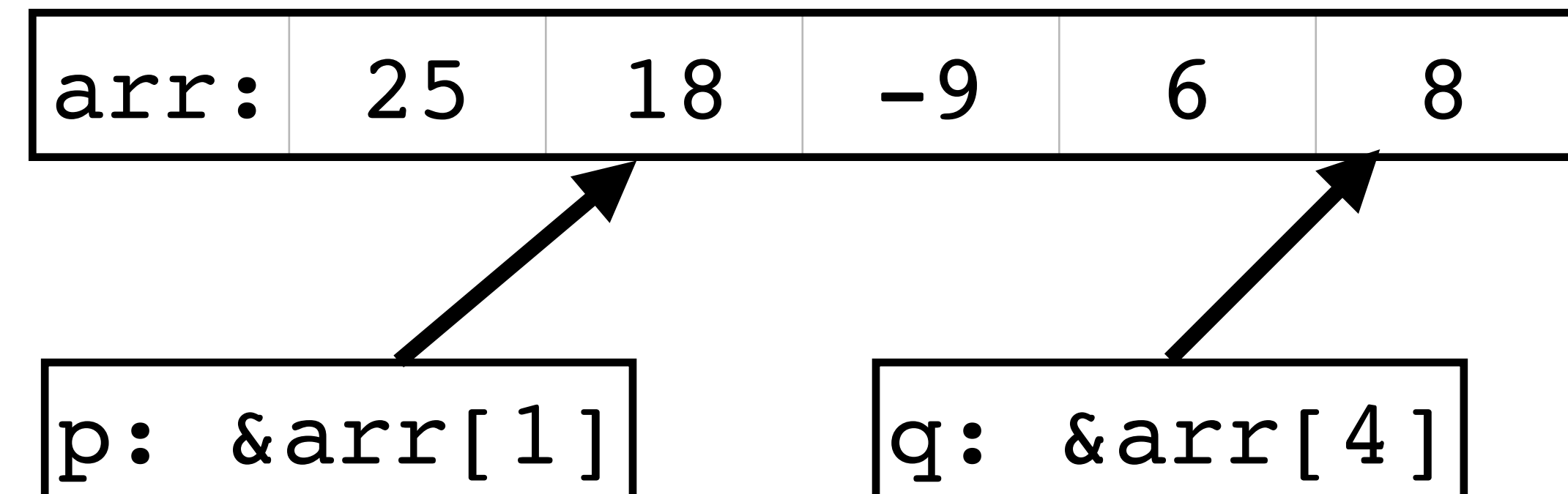
Two pointers into the same array object may be subtracted

```
int arr[] = { 25, 18, -9, 6, 8 };
```

```
int *p = &arr[1];
```

```
int *q = &arr[4];
```

```
ptrdiff_t difference = q - p;
```



Pointer undefined behavior

Pointers into array objects can point at any element of the array *or* just beyond the end of the array

Doing pointer arithmetic to get a different value is UB

- This is a massive source of software vulnerability

- Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).
- Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary `*` operator that is evaluated (6.5.6).

```

void foo(size_t n, int *p) {
    for (int *end = p + n; p != end; ++p)
        printf("%d\n", *p);
}
void bar() {
    int arr[] = { 0, 5, 4, 8, -8, 100, 0x80 };
    foo(sizeof arr/sizeof arr[0], arr);
} // What does bar do?

```

- A. Prints each element of the arr array
- B. Prints 0 seven times
- C. Prints all but the last element of the arr array
- D. Undefined behavior because end points beyond the end of the array pointed to by p
- E. Undefined behavior because arr[0] is 0 so it divides by 0

Fun pointer facts (read later)

If $x + y$ is a pointer, then

- $x[y]$ is $*(x + y)$ which equals $*(y + x)$ which is $y[x]$
- $\&x[y]$ is $\&*(x + y)$ [same as $x + y$] which equals $\&*(y + x)$ which is $\&y[x]$

Fun pointer facts (read later)

If $x + y$ is a pointer, then

- $x[y]$ is $*(x + y)$ which equals $*(y + x)$ which is $y[x]$
- $\&x[y]$ is $\&*(x + y)$ [same as $x + y$] which equals $\&*(y + x)$ which is $\&y[x]$

```
int arr[10];  
for (int i = 0; i < 10; ++i)  
    arr[i] = i;
```

```
int *p = &arr[4];  
int *q = &4[arr];  
int *r = &*(arr + 4);  
printf("p = %p; *p = %d\n", p, *p);  
printf("q = %p; *q = %d\n", q, *q);  
printf("r = %p; *r = %d\n", r, *r);
```

```
int x = arr[8];  
int y = 8[arr];  
printf("x = %d; y = %d\n", x, y);
```

Fun pointer facts (read later)

If $x + y$ is a pointer, then

- $x[y]$ is $*(x + y)$ which equals $*(y + x)$ which is $y[x]$
- $\&x[y]$ is $\&*(x + y)$ [same as $x + y$] which equals $\&*(y + x)$ which is $\&y[x]$

```
int arr[10];  
for (int i = 0; i < 10; ++i)  
    arr[i] = i;
```

```
int *p = &arr[4];  
int *q = &4[arr];  
int *r = &*(arr + 4);  
printf("p = %p; *p = %d\n", p, *p);  
printf("q = %p; *q = %d\n", q, *q);  
printf("r = %p; *r = %d\n", r, *r);
```

```
int x = arr[8];  
int y = 8[arr];  
printf("x = %d; y = %d\n", x, y);
```

```
p = 0x7ffee6bf31a0; *p = 4  
q = 0x7ffee6bf31a0; *q = 4  
r = 0x7ffee6bf31a0; *r = 4  
x = 8; y = 8
```


Strings

C has no string type

Strings are char arrays where the last byte is 0 (not '0')

- We say C strings are NUL-terminated (or null-terminated)

```
char x[] = "CS 241"; // identical to
```

```
char y[] = { 'C', 'S', ' ', '2', '4', '1', 0 };
```

```
char *str = "FOO";
```

```
// str is a pointer to the { 'F', 'O', 'O', 0 } array
```

```
str = x; // now str points to the x array
```


String literals are read-only

// This is valid because it creates a new array object x

```
char x[ ] = "CS 2xx";  
x[4] = '4';  
x[5] = '1';
```

// This is invalid because the pointer points to a read-only object

```
char *y = "CS 2xx";  
x[4] = '4'; // This will likely crash right here  
x[5] = '1';
```

String functions <string.h>

String functions <string.h>

```
size_t strlen(char const *str);
```

- returns the length of the string str (not including the 0 byte!)

String functions <string.h>

```
size_t strlen(char const *str);
```

- returns the length of the string str (not including the 0 byte!)

```
char *strcpy(char *dest, char const *src);
```

- Copies the string src to dest
- **DO NOT USE**, no bounds checking!

String functions <string.h>

```
size_t strlen(char const *str);
```

- returns the length of the string str (not including the 0 byte!)

```
char *strcpy(char *dest, char const *src);
```

- Copies the string src to dest
- **DO NOT USE**, no bounds checking!

```
char *strcat(char *dest, char const *src);
```

- Concatenates the string src to dest
- **DO NOT USE**, no bounds checking!

String functions <string.h>

```
size_t strlen(char const *str);
```

- returns the length of the string str (not including the 0 byte!)

```
char *strcpy(char *dest, char const *src);
```

- Copies the string src to dest
- **DO NOT USE**, no bounds checking!

```
char *strcat(char *dest, char const *src);
```

- Concatenates the string src to dest
- **DO NOT USE**, no bounds checking!

```
int strcmp(char const *s1, char const *s2);
```

- Compares strings s1 and s2 character by character returning a negative if s1 is lexicographically less than, equal to, or greater than s2

Safe string handling functions

Safe string handling functions

BSD provides safer alternatives

- On linux, need to include `<bsd/string.h>` and link with `-lbsd`

Safe string handling functions

BSD provides safer alternatives

- On linux, need to include `<bsd/string.h>` and link with `-lbsd`

```
size_t strlcpy(char *dest, char const *src, size_t size);
```

- Copies up to `size-1` characters from `src` to `dest` and NUL-terminates
- Returns `strlen(src)` (and you can check that it is less than `size`)

Safe string handling functions

BSD provides safer alternatives

- On linux, need to include `<bsd/string.h>` and link with `-lbsd`

```
size_t strncpy(char *dest, char const *src, size_t size);
```

- Copies up to `size-1` characters from `src` to `dest` and NUL-terminates
- Returns `strlen(src)` (and you can check that it is less than `size`)

```
size_t strlcat(char *dest, char const *src, size_t size);
```

- Appends up to `size-strlen(dest)-1` characters from `src` to `dest` and NUL-terminates
- Returns `strlen(dest) + strlen(src)`

Example

```
bool foo(char const *name) {  
    char buffer[100];  
    size_t size = strlcpy(buffer, "Hello ", sizeof buffer);  
    if (size >= sizeof buffer)  
        return false;  
    size = strlcat(buffer, name, sizeof buffer);  
    if (size >= sizeof buffer)  
        return false;  
    size = strlcat(buffer, "! Welcome.", sizeof buffer);  
    if (size >= sizeof buffer)  
        return false;  
    // buffer now contains "Hello ${name}! Welcome."  
}
```

const

We can tell the compiler that data should not be modified

```
int const x = 5;  
x = 6; // Invalid because x is declared const
```

The address of non-const variable may be assigned to a pointer to a const but the variable may not be modified via that pointer

```
int y = 28;  
int const *p = &y; // Valid  
y = 15; // Valid because y isn't const  
*p = 6; // Invalid because p is a pointer to const!
```

String literals aren't const (but should be)

Help the compiler help you: always use

```
char const *str = "Foo bar";
```

This makes modification illegal

```
str[0] = 'T'; // Invalid because str points to const
```

Pointer function parameters

If a function has a pointer parameter and it does not modify the data pointed to, make it a pointer to const

- `printf(char const *format, ...);` // doesn't modify format
- `puts(char const *str);` // doesn't modify str

Constant pointers

Constant pointers

You *can* make a constant pointer, if you want

Constant pointers

You *can* make a constant pointer, if you want

const applies **left**

Constant pointers

You *can* make a constant pointer, if you want

const applies **left**

- `int x;`

Constant pointers

You *can* make a constant pointer, if you want

const applies **left**

- `int x;`
- `int *p = &x; // non-const pointer to non-const int`

Constant pointers

You *can* make a constant pointer, if you want

const applies **left**

- `int x;`
- `int *p = &x;` // non-const pointer to non-const int
- `int const *q = &x;` // non-const pointer to int const

Constant pointers

You *can* make a constant pointer, if you want

const applies **left**

- `int x;`
- `int *p = &x;` // non-const pointer to non-const int
- `int const *q = &x;` // non-const pointer to int const
- `int *const r = &x;` // const pointer to non-const int

Constant pointers

You *can* make a constant pointer, if you want

const applies **left**

- `int x;`
- `int *p = &x;` // non-const pointer to non-const int
- `int const *q = &x;` // non-const pointer to int const
- `int *const r = &x;` // const pointer to non-const int
- `int const *const s = &x;` // const pointer to int const

In-class exercise

<https://checkoway.net/teaching/cs241/2019-fall/exercises/Lecture-15.html>

Grab a laptop and a partner and try to get as much of that done as you can!