Week 6 Part 1

Kyle Dewey

Overview

- Announcement repeat
- Pre-lab #5
- Pointers
- Arrays
- Exam #1 Lingering Questions

+3 points on exam

Pre-lab #5

Pointers

Pointers

- Data that "points to" other data
- In my humble opinion, the most difficult programming concept to grasp
  - Questions questions questions...

Pointers

- Non-programming example: place of residence
  - Data: Where you actually live
  - Pointer: The address of where you live

versus `123 Fake Street`

Pointers

- Programming example: `scanf`
  - Read in an input and put it someplace

```
int x;
scanf( “%i”, x ); // value of x - WRONG
scanf( “%i”, &x ); // where x is
                   // & is called the
                   // address-of
                   // operator
```

Why the `&`

- Analogy: ordering an integer online from `scanf` co.
- `scanf` co. needs to know where to send your brand-new integer

Why the `&`

```
int x;
scanf( "%i", &x ); // where x is
```

- Send the integer to this address (where `x` is)

Why the &

```
int x;
scanf( “%i”, x ); // what x holds
```

- You just shipped **a copy of** your entire house to `scanf` co.
- `scanf` co. is likely a little confused
- You still don’t have your integer

Leaving the Analogy

- The world: memory
- Memory is a linear sequence of bytes
- Where something is in memory: address
  - Each byte of memory can be addressed
- What something is in memory: value

Memory

Value	0x23	0xA4	0x2F	0x20	0xA4	0xB8	0xCA
Address	0	1	2	3	4	5	6

Types

- If a `*` follows a type name, it's a pointer to that type
  - `int*`: a pointer to an integer
  - Can also say an integer pointer for short

Usage

- The `&` (address-of) operator will get the address of a variable
  - If the variable's type is `int`, then using `&` on the variable will yield an `int*`

```
int x = 5;
int* pointer;
pointer = &x;
```

Usage

- `*` also acts to grab the value at the given address or put a new value in the given address
  - Called the **dereference** operator
  - Note this is `*variable` as opposed to `number * number`

Dereference

- Getting the value:

```
int x = 5;
int* pointer;
pointer = &x;
*pointer; // returns 5
```

Dereference

- Assigning a new value

```
int x = 5;
int* pointer;
pointer = &x;
*pointer = 10; // x is now 10
```

Useful Example

```
int readDigit( int* whereToPut ) {
  int readIn = getchar();
  if ( readIn >= ‘0’ &&
       readIn <= ‘9’ ) {
    *whereToPut = readIn - ‘0’;
    return 1;
  } else {
    return 0;
  }
}
```

Example #1

```
int x = 5;
int* pointer;
pointer = &x;
*pointer = 11;
// what does x equal?
```

Example #2

```
int x = 7;
int* pointer = &x;
int y = 3 + *pointer;
// what does y equal?
```

Example #3

```
int x = 7;
int y = 3;
int* xPointer = &x;
int* yPointer = &y;
*xPointer = y;
*yPointer = x;
// what do x and y equal?
```

Example #4

```
int x = 7;
int y = 3;
int* xPointer = &x;
int* yPointer = &y;
xPointer = yPointer;
yPointer = xPointer;
// what do x and y equal?
```

Example #5

```
int x = 7;
int y = 3;
int* xPointer = &x;
int* yPointer = &y;
int z = *xPointer + *yPointer;
*(&z) = 2 + 2;
// what do x, y, and z equal?
```

What about `char*`?

- The type of a string
- ...but this looks like a `char` pointer?
- ...and what about the boxed notation?
  - i.e. `string[ 0 ]`, `string[ 1 ]`...

Recall Strings

- Strings are a sequence of characters ended by a null byte

```
"Hello" = 'H','e','l','l','o','\0'
```

Importance of This

- A string is variable length
- A sequence of `char`s
- A `char` variable only holds **one** character
  - We want to hold a variable number of `char`s

`“Hello” = ‘H’,’e’,’l’,’l’,’o’,’\0’`

char*

- This code is invalid:

```
char string = “moo”;
```

- But this code is not:

```
char* string = “moo”;
```

Why a Pointer?

- All valid:

```
char* string1 = “moo”;
char* string2 = “cow”;
char* string3 = “bull”;
char* string4 = “foobar”;
char* string5 = “”;
```

- Pointers can be used for something special...

Arrays

Arrays

- Arrays are a sequence of elements of the same type
- Arrays can be of variable length, but once they are created they cannot be resized
  - (we will see an exception to this later)

Arrays and `char*`

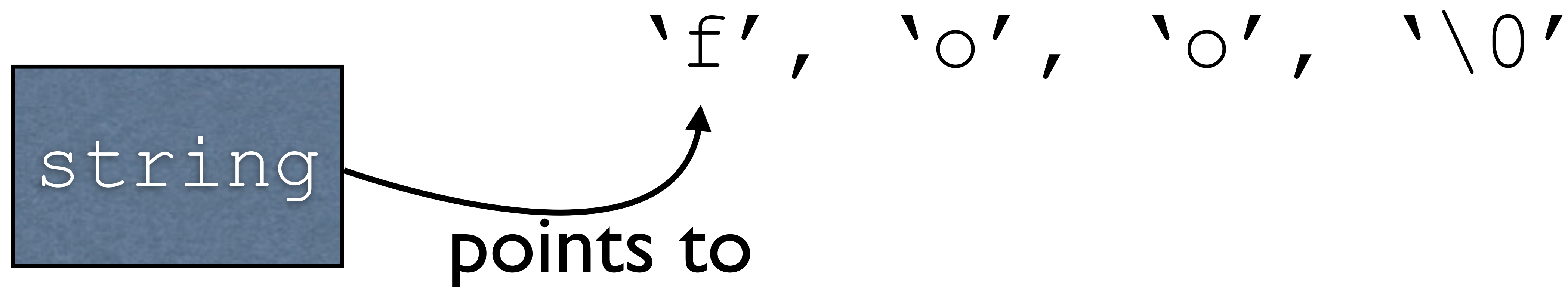
- A string is a sequence of `char`s...
- An array is a sequence of elements of the same type...
- Strings are arrays

```
"Hello" = 'H','e','l','l','o','\0'
```

Arrays and Pointers

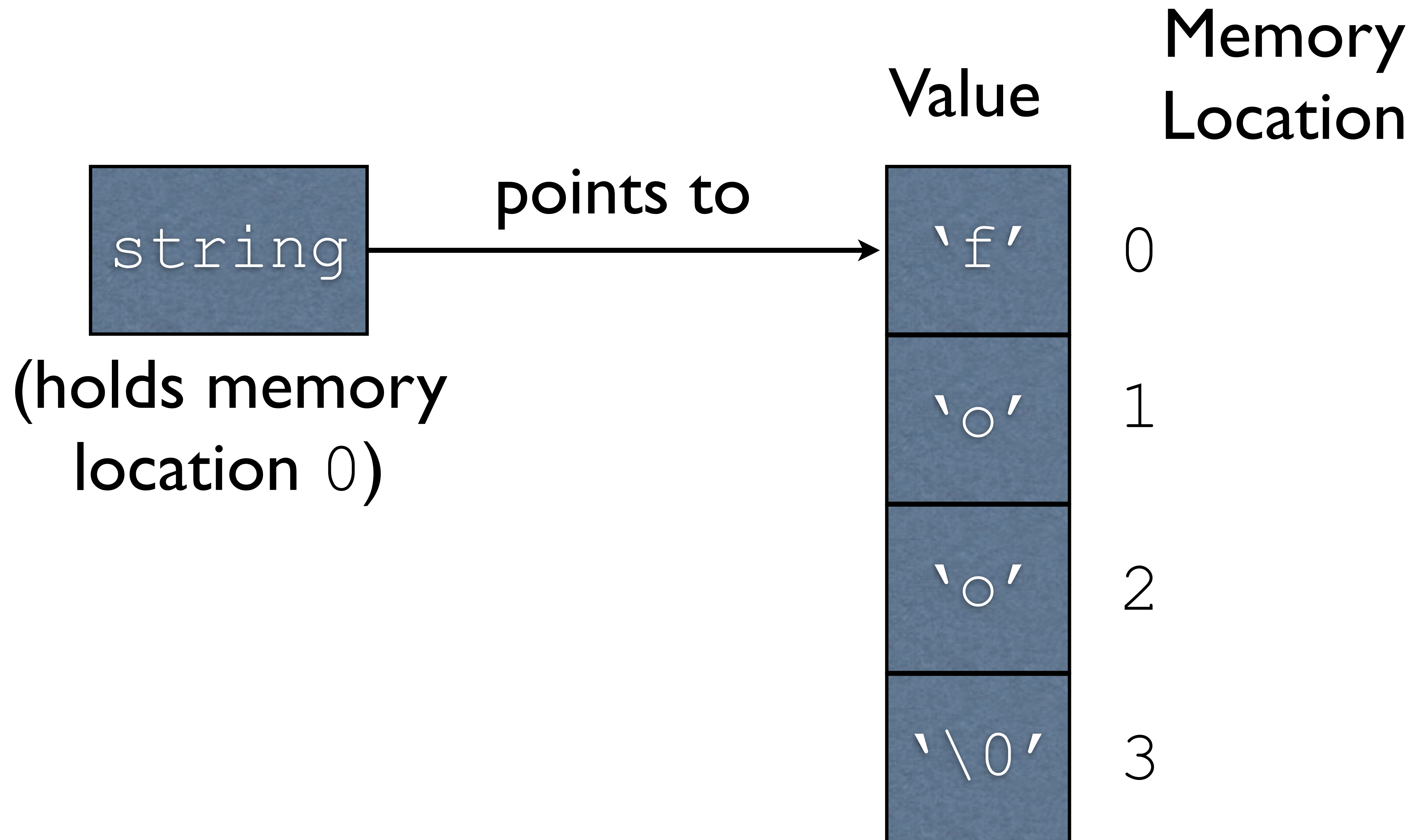
- Pointers can be used to **reference** (i.e. point to) arrays
- Example:

```
char* string = “foo”;
```



Another Look

```
char* string = “foo”;
```



Pointer Arithmetic

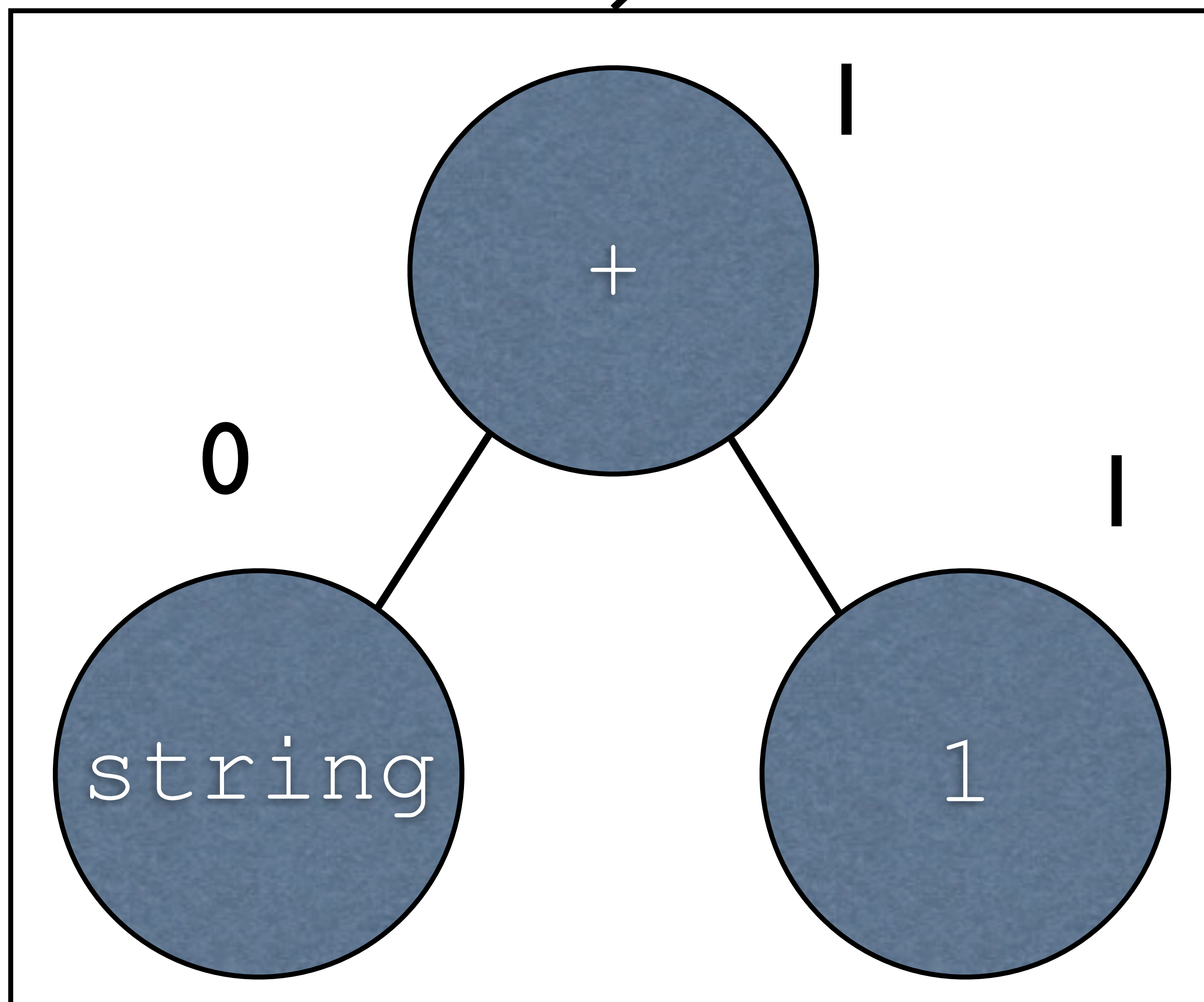
- Pointers hold memory addresses
- Memory addresses are sequential
- We can do arithmetic with them
  - NOTE: generally, addition by positive numbers is the only thing possible, and it's certainly the only thing people won't hate you for

Pointer Arithmetic

```
char* string = “foobar”;
printf( “%s”, string + 3 );
// prints “bar”
```

Pointer Arithmetic

```
char* string = "foo";
printf( "%s", string + 1 );
```



More on Arrays

- It's possible to make arrays of other kinds like so:

```
int arr[] = { 1, 2, 3, 4 };
// initialized to 1,2,3,4
```

- This is not a block!

More on Arrays

- It’s possible to make an array of a given length uninitialized:

```
int arr2[ 50 ];
// space for 50 integers
```

- Note that the size must be a **constant**

Array Length

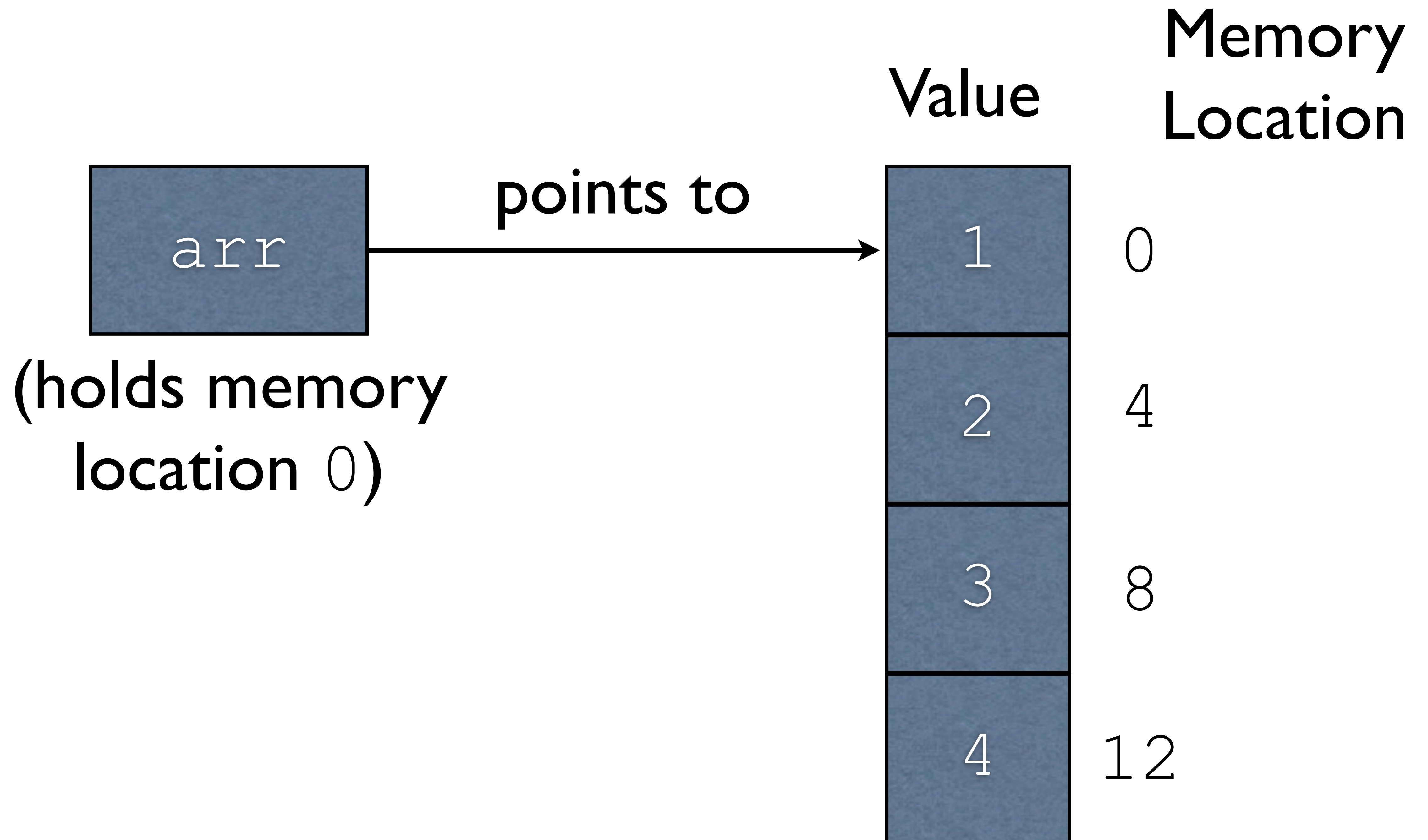
- The length of an array must be tracked separately
- Alternatively, the last element can be set to some **sentinel value**
  - For C strings, '\0' is a sentinel value

Pointer Arithmetic

- When we add, we may increment by more than a byte
- How much we increment by depends on the data type
  - A 1 byte `char` increments by 1 byte
  - A 4 byte `int` increments by 4 bytes

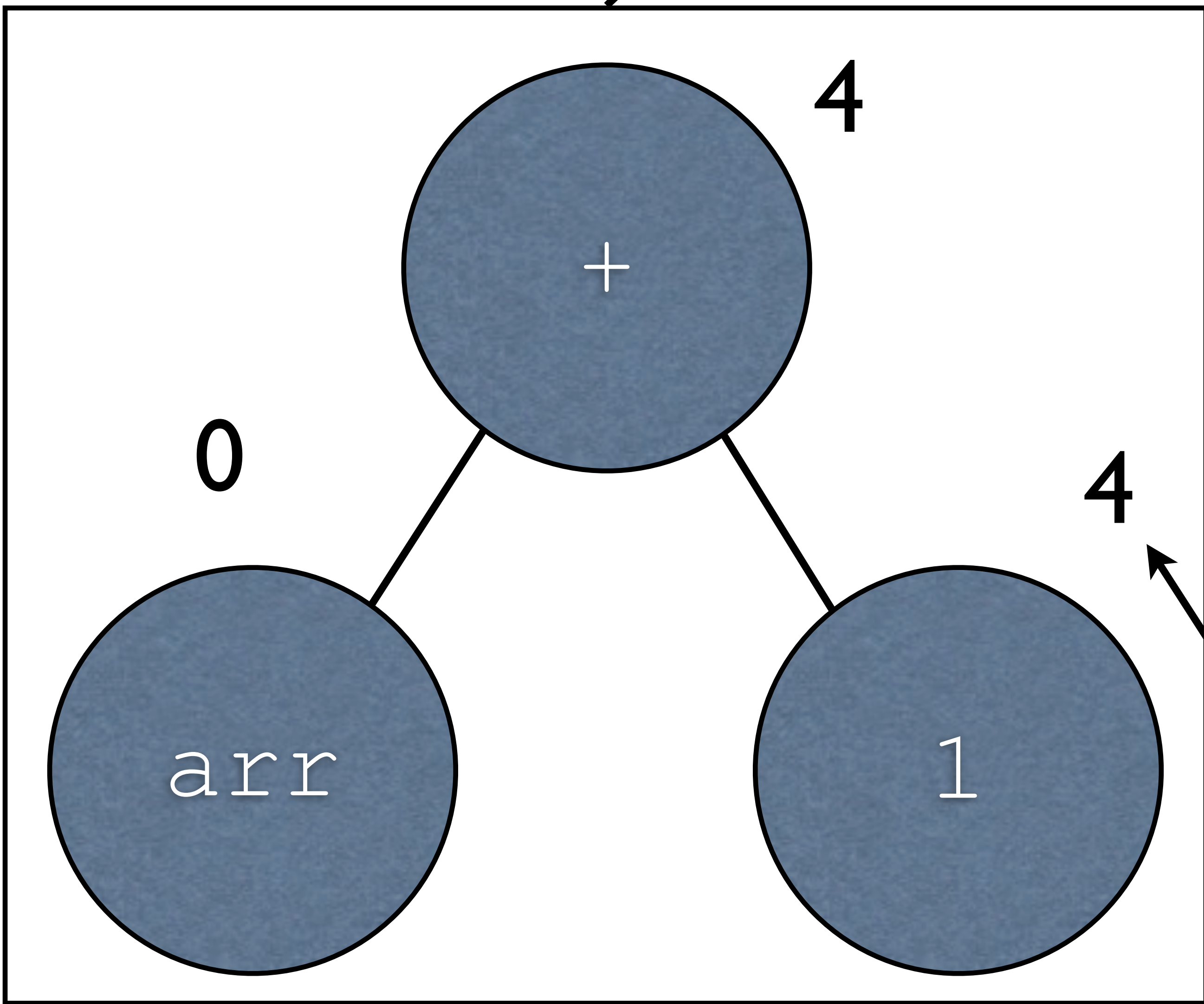
Another Look

```
int arr[] = { 1, 2, 3, 4 };
```



Pointer Arithmetic

```
int arr[] = { 1, 2, 3, 4 };
arr + 1
```



Getting Individual Elements

- We can do this:

```
int arr[] = { 1, 2, 3, 4 };
*(arr + 1)
```

- We can also use the equivalent boxed notation:

```
int arr[] = { 1, 2, 3, 4 };
arr[ 1 ]
```

Boxed Notation

- The notation `string[ 0 ]` refers to the 0th element of an array
  - The notation `string[ n ]` refers to the nth element of an array

Boxed Notation

- This is actually **syntactic sugar**

```
string[ n ]
```

...is equivalent to...

```
*(string + n)
```

Arrays and Loops

- `for` loops go very well with arrays
  - Arrays have fixed length
  - `for` loops are generally for a fixed number of iterations

Example

```
int sum( int* array, int length ) {
  int retval = 0;
  int x;
  for( x = 0; x < length; x++ ) {
    retval = retval + array[ x ];
  }
  return retval;
}
```

Question

- What is wrong with this code?

```
void printInts( int* arr, int length ){
  int x;
  for( x = 0; x <= length; x++ ) {
    printf( "%i\n", arr[ x ] );
  }
}
```

Answer

- There is no `arr[ length ]`
- Who knows what will happen?

```
void printInts( int* arr, int length ){
  int x;
  for( x = 0; x <= length; x++ ) {
    printf( “%i\n”, arr[ x ] );
  }
}
```

Side Note: String Concatenation

- Many questions regarding string concatenation in C for the project
- Short answer: don't use string concatenation
- Now for the long answer...

String Concatenation

- Say we are given two strings, `first` and `second`
- Say we are given a memory location named `result` where we put the result

```
void concat( char* first,
             char* second,
             char* result );
```

```
void concat( char* first,
             char* second,
             char* result ) {
  int firstLen = strlen( first );
  int secondLen = strlen( second );
  int x;
  for ( x = 0; x < firstLen; x++ ) {
    result[ x ] = first[ x ];
  }
  for ( x = 0; x < secondLen; x++ ) {
     result[ firstLen + x ] =
       second[ x ];
  }
  result[ firstLen + secondLen ] = '\0';
}
```

Works Except...

- Where did `result` come from?
  - Needs to be large enough for the result
- Herein lies the problem

```
void concat( char* first,
             char* second,
             char* result );
```

Memory Allocation

- Generally, we only know how big `result` must be at runtime
- Need **dynamic memory allocation**
  - Much later, and it's not easy

```
void concat( char* first,
             char* second,
             char* result );
```

Exam #1 Lingering Questions