CS24 Week 1 Lecture 2

Kyle Dewey

Overview

- C Review
  - Multiple files
  - File I/O
  - Command-line arguments
  - Pointers
  - Allocation

Multiple Files

Situation 1

- You have written a *library* of routines for manipulating images
- You want to share these with other programmers
- How can we go about this?

Situation 2

- You are at Google working on their search engine
- The search engine is divided into these components:
  - An external interface
  - A database of various webpages
  - A sophisticated search algorithm
- How can all parties work together?

Situation 3

- You are working on a large project, and putting everything in one file leads to a mess
  - 10s of thousands of lines of code
  - By the time you're at line 2,000, you can't remember what 200 did
  - Editing is a nightmare

Solution: Multiple Files

- Splitting code up into multiple files allows for easier collaboration, and helps *hide details* from us
- Generally, the fewer details you must know, the better
  - Mark of good software design

Header Files

- In C/C++, this is accomplished via header files
- A header file defines an *interface*
- Code can *include* other header files to gain access to the interfaces
- The interfaces are implemented in separate files

Header Files Example

Basic File I/O

Question

- Say a program is not permitted to read or write to files, the terminal, the network, or any other source
- Can the resulting program do anything useful?

I/O (Input/Output)

- The way programs interact with the outside world
- Without it, programs are simply things that turn computers into space heaters

File I/O

- When working with files, we must `open` a file before we can `read` from it
- When we are done with a file, we must `close` it
  - What happens if we forget to close it?

Reading from a File

- Can read one character at a time
  - See `cat1.c`, which uses `fgetc` for this

Reading from a File

- Can also read multiple characters at a time
  - See `cat2.c`, which uses `fgets` for this

Questions

- What extra bit is needed to read multiple characters at a time?
    - What happen if we get this extra bit wrong?
- Why read multiple characters at a time?

Command Line Arguments

UNIX Commands

- We have seen a bunch of UNIX commands used at this point
- How exactly do these programs interpret what they are supposed to do?
  - How does `emacs` know which file to open?
  - How does `cd` know which directory to go to?

Command Line Arguments

- A standard way to tell programs what and how to do
- In C/C++, we can get access to the command line arguments via the parameters to the `main` function

Command Line Arguments Example (`echo.c`)

Command Line Arguments

- What is `argc`? What is it set to?
- What is `argv`? What is it set to?

Pointers

Question

- What is a pointer?
  - Conceptually?
  - In reality (the value held)?
- What can pointers point to?

What is Printed?

```
int x = 5;
int y = 7;
int* p = &x;

if (*p == 5) {
   *p = 8;
   p = &y;
}

*p = 1;

printf("%d %d", x, y);
```

Question

- Why ever use a pointer over a value?

Answers

- Why ever use a pointer over a value?
  - Copying around values can get expensive
  - Allows for indirect access to other program portions
  - You do not know how big the value is it points to (dynamic allocation)

Question

- Why use a value over a pointer?

Answers

- Why use a value over a pointer?
  - Values are generally easier to reason about
  - Sometimes you need a copy

Pointers Versus Arrays

Pointers Versus Arrays

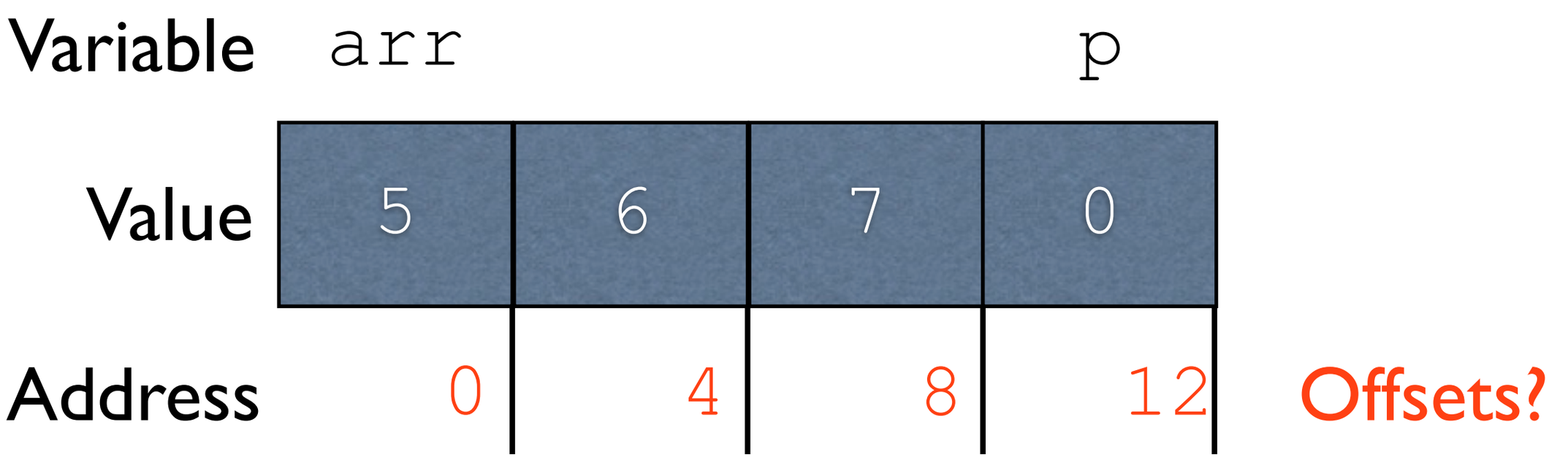
- For a single-dimensional array, they are effectively the same

```
int arr[3] = {5, 6, 7};
int* p = arr;
```

Pointers Versus Arrays

- For a single-dimensional array, they are effectively the same

```
int arr[3] = {5, 6, 7};
int* p = arr;
```



Pointers Versus Arrays

- For a single-dimensional array, they are effectively the same

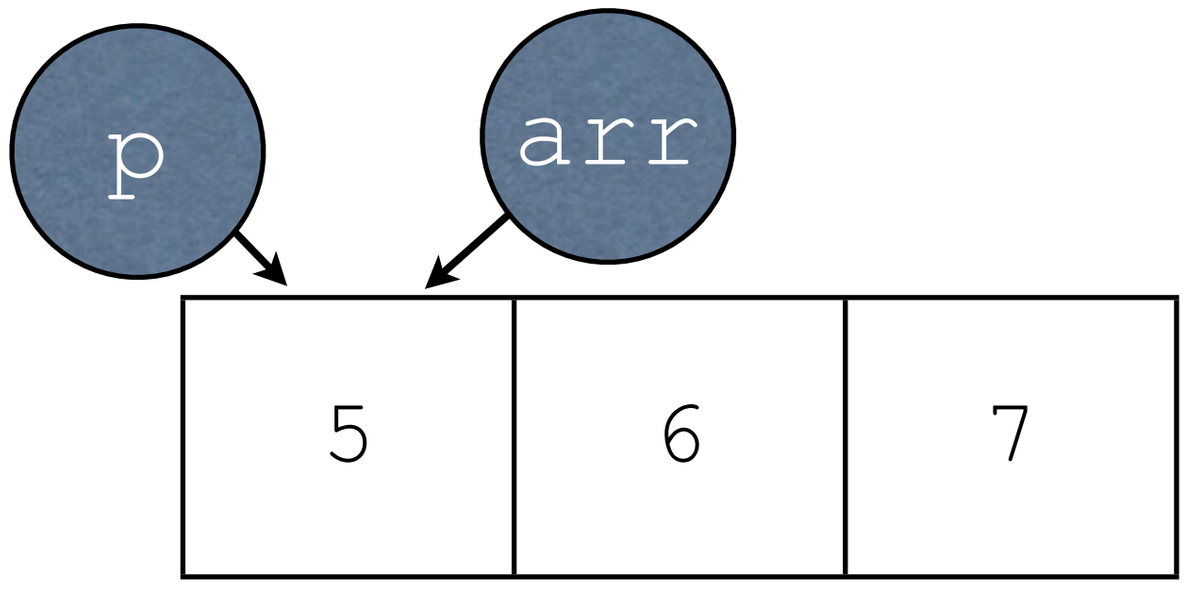
```
int arr[3] = {5, 6, 7};
int* p = arr;
```

Could be just about anywhere

Memory Diagram

- A memory diagram for the same program:

```
int arr[3] = {5, 6, 7};
int* p = arr;
```

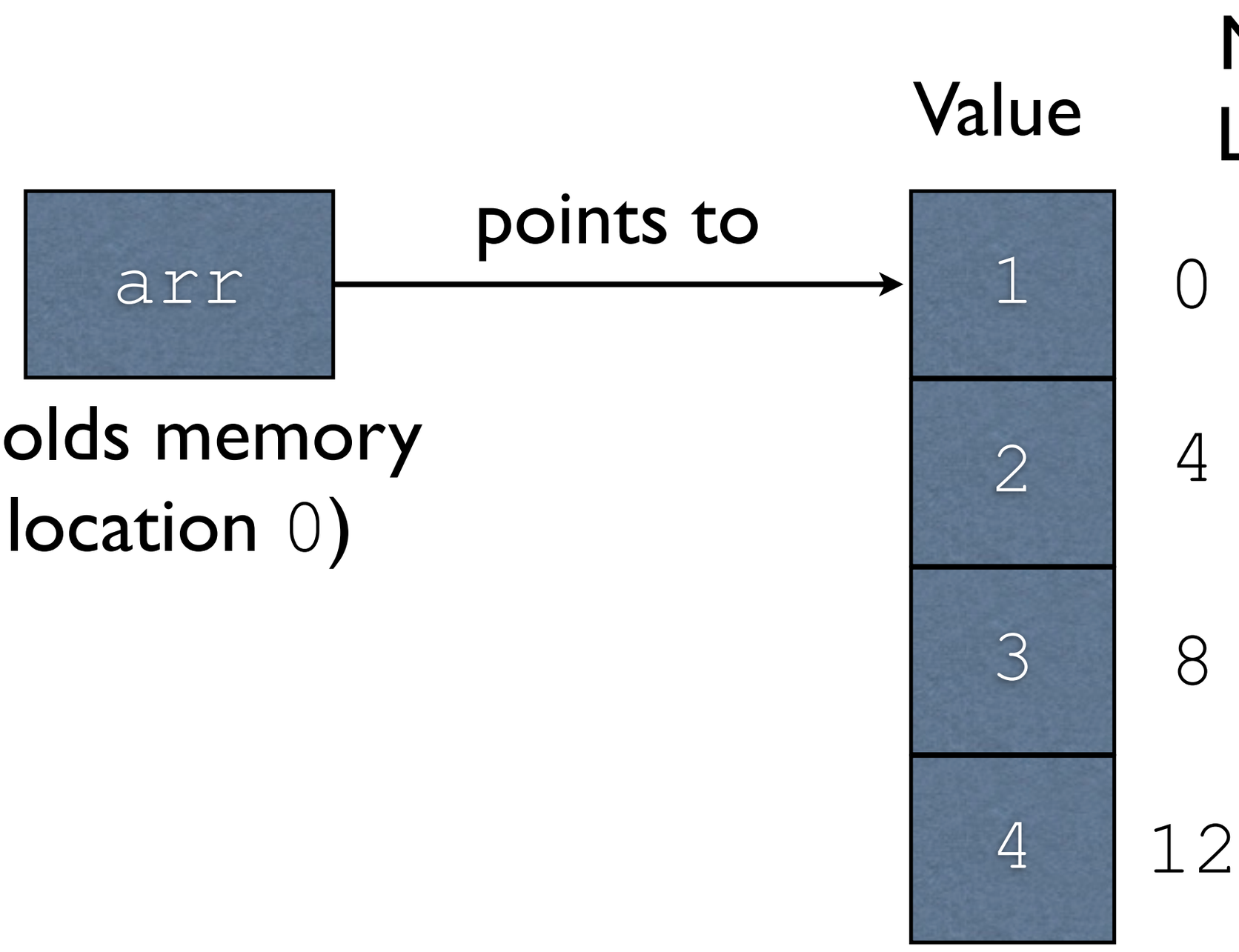


Pointers Versus Arrays

- You can add a value to a pointer to increment (or decrement) that many places in memory

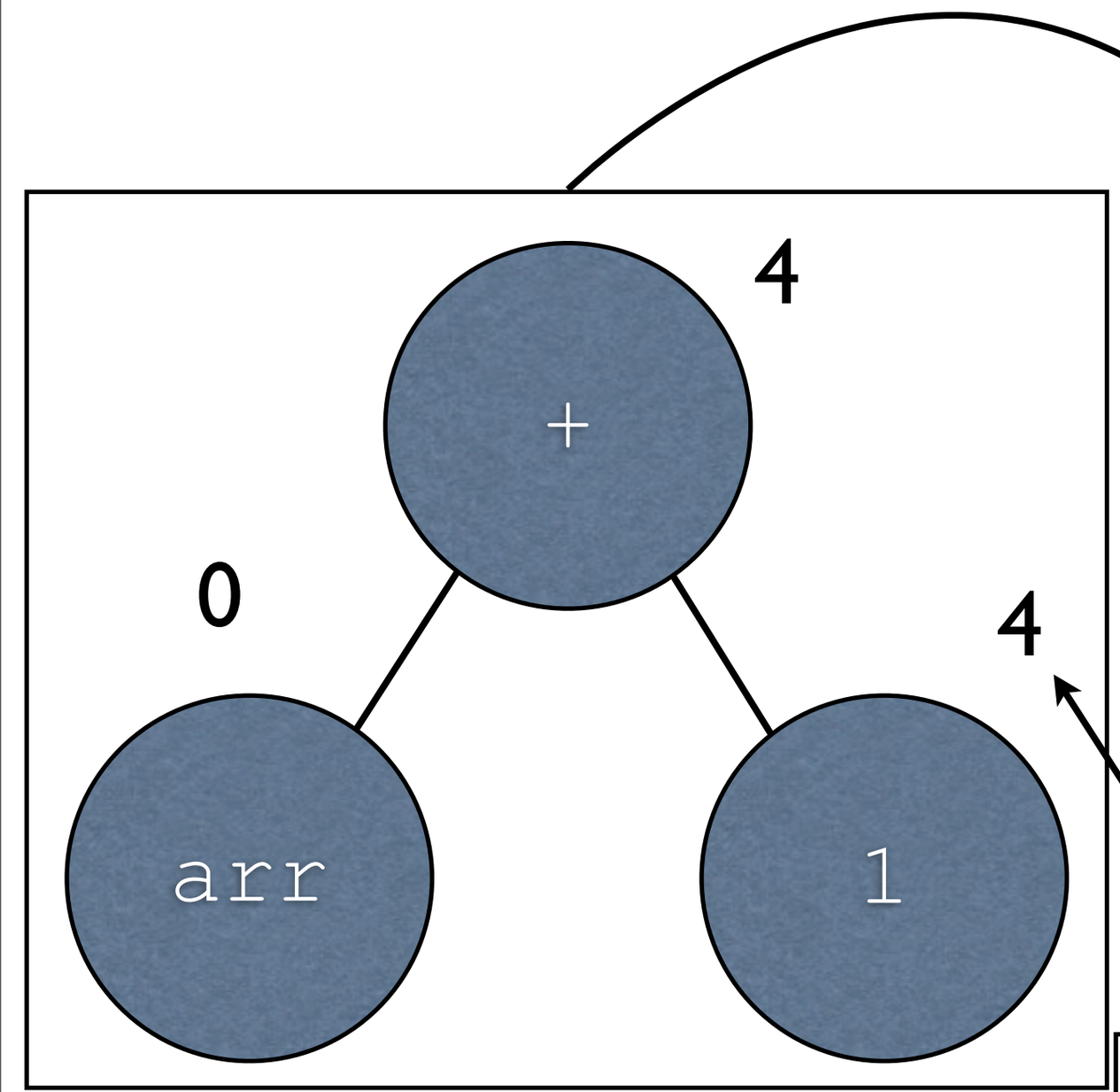
Pointers Versus Arrays

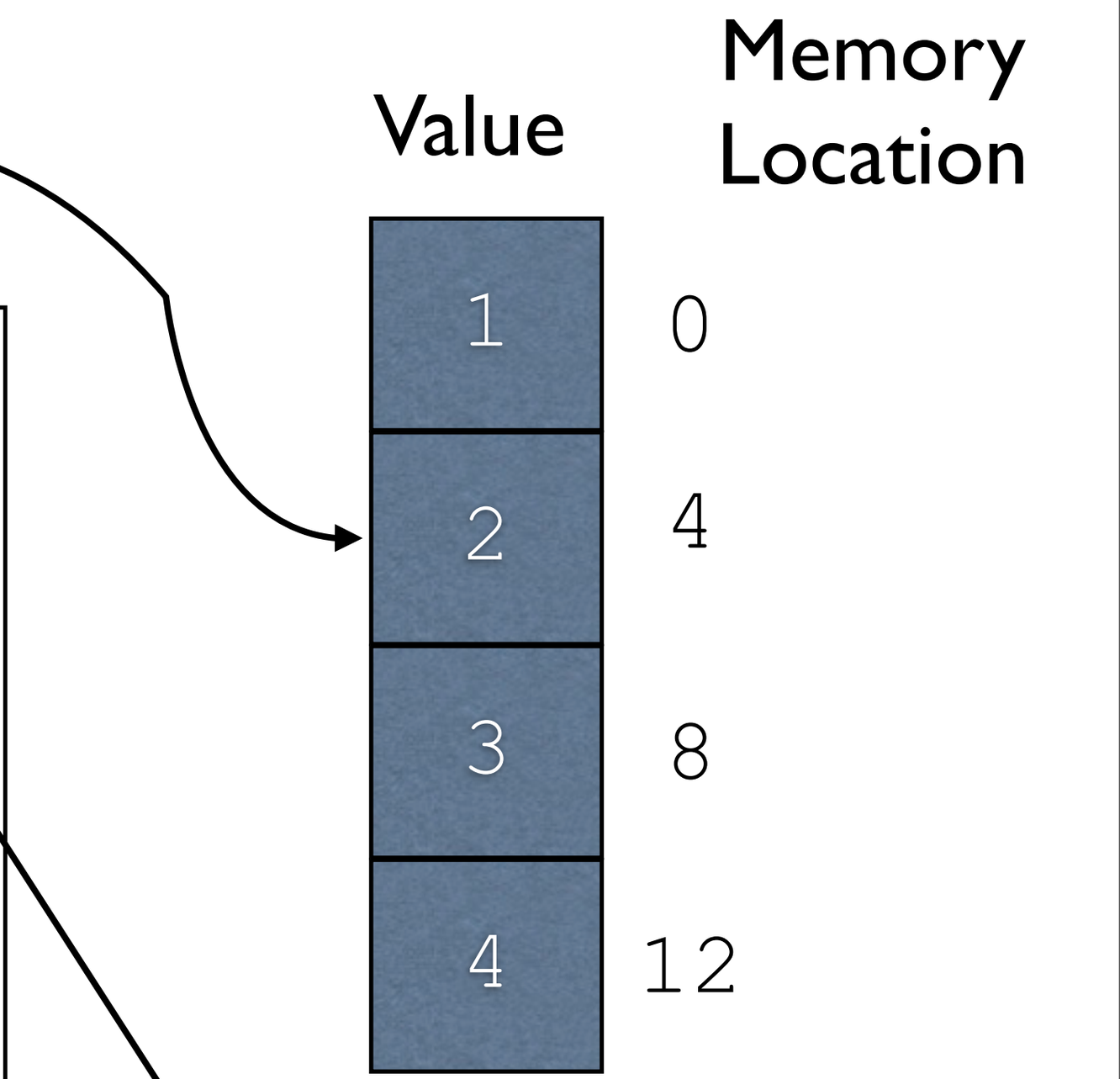
```
int arr[] = { 1, 2, 3, 4 };
```



Pointers Versus Arrays

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






```
1 * sizeof( int )
```

Pointers Versus Arrays

- 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 ]
```

Pointers Versus Arrays

- Still some differences

```
int arr1[3] = {1, 2, 3};
int arr2[3] = {5, 6, 7};
arr1 = arr2; // not legal
```

- Generally, pointers can act like arrays, but arrays cannot act like pointers

void*
(Void Pointers)

`void*`

- Like any other pointer, it refers to some memory address
- However, it has no associated type, and cannot be dereferenced directly
  - Question: why can't it be dereferenced?

No Dereferencing

```
void* p = 2;
*p; // get what's at p
```

Value	0x21	0x00	0x01	0x52	0xF0	0xAB	0x2C
Address	0	1	2	3	4	5	6

- `void*` is a value without context
- Without context, there is no way to know how to interpret the value (`int`? `char`? `double`?)

How to Use a `void*`

- A `void*` cannot be dereferenced directly
- However, it is possible to cast a `void*` to another type

```
char* str = “moo”;
void* p = str;
printf( “%s\n”, (char*)p );
```

How to Use a `void*`

- A `void*` also coerces into other pointer types

```
char* str = “moo”;
void* p = str;
char* str2 = p; // no errors
```

Caveat

- A `void*` also coerces into other pointer types
- The compiler will trust you blindly

```
char* str = “moo”;
void* p = str;

// no compiler errors, but
// this is probably not what
// is desired
int* nums = p;
```

Why a `void*`?

- Allows for generic data structures
  - A list of `int`s looks a lot like a list of `char`s
- Can refer to some block of memory without context
  - Up next: why anyone would want to do that

Dynamic Memory Allocation

Motivation

- We want to read in a dictionary of words
- Before reading it in:
  - We don't know how many words there are
  - We don't know how big each word is

```
apple
banana      <<empty>>      aardvark
pear
```

Possible Solution

- Allocate the maximum amount you could ever need
- Question: why is this generally not a good solution? (2 reasons)

```
// 1000 words max with
// 100 characters max per word
char dictionary[1000][100];
```

Problems

- Most things do not have a good "maximum" you can get a grasp of
- Your program always needs the maximum amount of memory, and usually the vast majority is completely wasted

What is Desired

- A way to tell the computer to give a certain amount of memory to a program as it runs
- Only what is explicitly requested is allocated

Dynamic Memory Allocation

- Dynamic: as the program runs
- Memory allocation: set aside memory

malloc

- The most generic way to allocate memory
- Takes the number of bytes to allocate
- **Returns a `void*` to the block of memory allocated**

```
// size_t is an integral defined
// elsewhere
void* malloc( size_t numBytes );
```

Using `malloc`

- The `sizeof` operator comes in handy
  - Returns an integral size as a `size_t`
- For example: allocate room for 50 integers dynamically:

```
// dynamically
int* nums1;
nums1 = malloc( sizeof( int ) * 50 );

int nums2[ 50 ]; // statically
```

Importance

- Static allocation can only be done with constants
- Dynamic allocation can be done with variables

```
int numToAllocate;
scanf( “%i”, &numToAllocate );
int* nums =
  malloc(sizeof( int ) * numToAllocate);
int nums2[ numToAllocate ]; // ERROR
```

Memory Contents

- The contents of the memory allocated by malloc is undefined
- You will need to initialize it yourself with a loop (or by using the `memset` function)

free

- Once we are done using a block of memory, call free on it
- If a block is never freed, it is called a **memory leak**
  - Memory is still allocated but wasted

```
int* nums;
nums = malloc( sizeof( int ) * 50 );
...
// done with nums
free( nums );
```

malloc1.c,

malloc2.c

On Calling `free`

- With static allocation, the compiler handles deallocation for you
- With dynamic allocation, you must call `free` yourself
- The simple act of knowing when to call `free` can be hard
  - In general, mathematically unsolvable!