Introduction to Memory

5

CS/COE 0449
Introduction to Systems Software

wilkie
(with content borrowed from Vinicius Petrucci and Jarrett Billingsley)

Spring 2019/2020

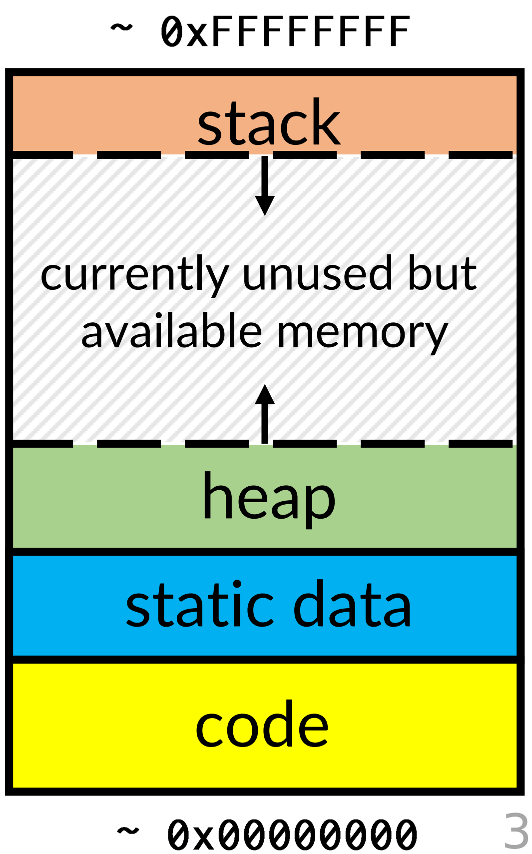
The Memory Model

If you forget how addressing works, I have a few pointers for you.

The C Memory Model

- **Memory is a continuous series of bits.**
  - It can be logically divided into bytes or words.
- **We will treat it as *byte-addressable* which means individual bytes can be read.**
  - This is not always the case!!
  - Consider masking and shifting to know the workaround!
- **With byte-addressable memory, each and every byte (8 bits) has its own unique address.**
  - It's the place it lives!! Memory is JUST LIKE US!
  - Address starts at 0, second byte is at address 1, and increases ("upward") as you add new data.

Potential Layout
(32-bit addresses)



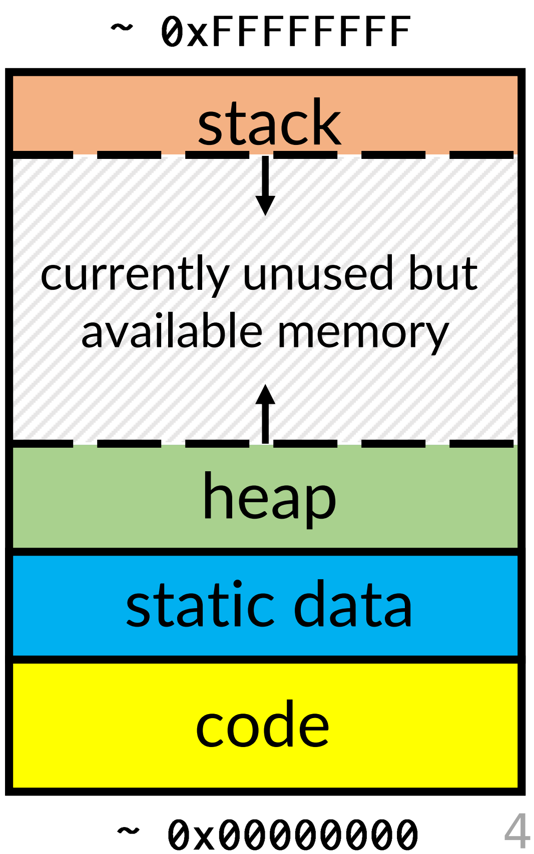
The C Memory Model

- **There are two main parts of a program: *code* and *data***
  - "code" is sometimes called "text"

- **Where in memory should each go?**
  - Should we interleave them?
  - Which do you think is usually largest?

- **How do we use memory dynamically?**
  - That is, only when we know we need it, in the moment.

Potential Layout
(32-bit addresses)

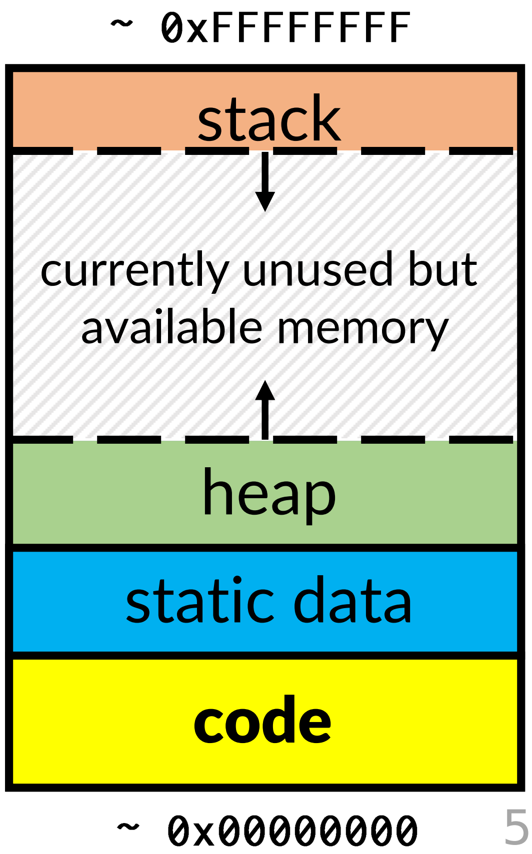


The C Memory Model: Code

- **Code has a few known properties:**
  - It *likely* should not change.
  - It must be loaded before a program can start.

```
int my_static_var = 1;

int factorial(int n) {
  if (n <= 1) { return my_static_var; }
  return n * factorial(n - 1);
}

void main(void) {
  factorial(5);
}
```

Potential Layout
(32-bit addresses)

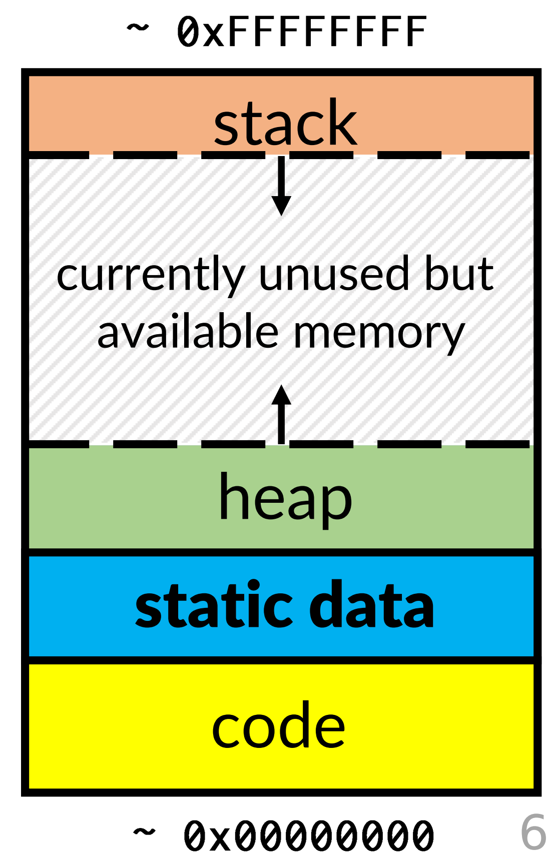


The C Memory Model: Static Data

- **Static Data is an oft forgotten but useful section.**
  - It ***does*** change. (contrary to its name)
  - It generally must be loaded before a program starts.
  - The size of the data and section is fixed.

```c
int my_static_var = 1;

int factorial(int n) {
  if (n <= 1) { return my_static_var; }
  return n * factorial(n - 1);
}

void main(void) {
  factorial(5);
}
```

Potential Layout
(32-bit addresses)



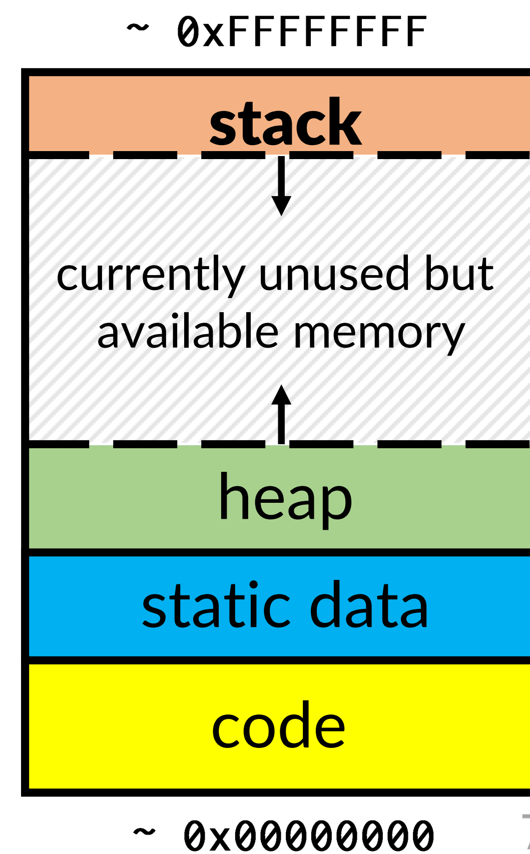
The C Memory Model: The Stack

- **The Stack is a space for temporary dynamic data.**
  - Holds local variables and function arguments.
  - Allocated when functions are called. Freed on return.
  - Grows “downward”! (Allocates lower addresses)

```
int my_static_var = 1;

int factorial(int n) {
  if (n <= 1) { return my_static_var; }
  return n * factorial(n - 1);
}

void main(void) {
  factorial(5);
}
```

Stack Allocation allows recursion. However, the more you recurse, the more you use! (Stack is only freed on return)

Potential Layout
(32-bit addresses)



Revisiting our past troubles:

```c
#include <stdio.h>    // Gives us 'printf'
#include <stdlib.h>   // Gives us 'rand' which returns a random-ish int

void undefined_local() {
  int x;
  printf("x = %d\n", x);
}

void some_calc(int a) {
  a = a % 2 ? rand() : -a;
}

int main(void) {
  for (int i = 0; i < 5; i++) {
    some_calc(i * i);
    undefined_local();
  }
  return 0;
}
```

4. Stack Allocation (No initialization!) (`int x;`)

It reuses what is already there!!

2. Stack Allocation (`(int a)`)

1. Function Call (`some_calc(i * i);`)

3. Function Call (`undefined_local();`)


```
Output:

x = 0
x = 1804289383
x = -4
x = 846930886
x = -16
```

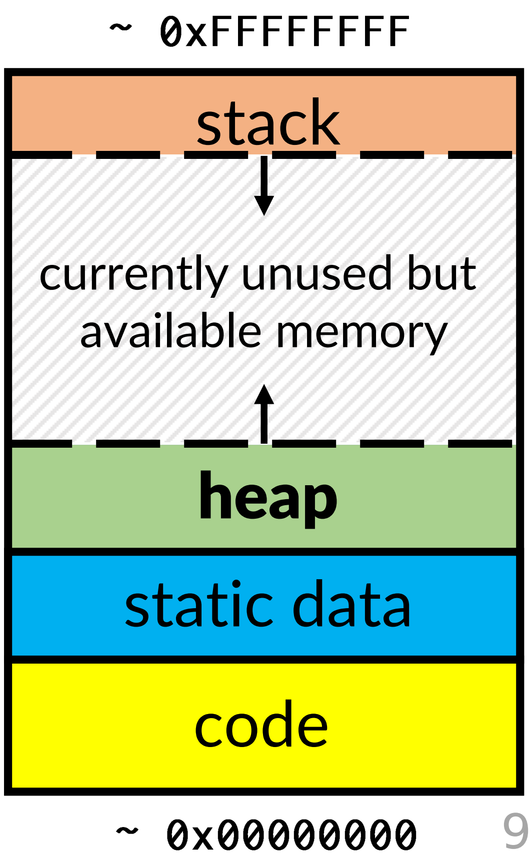
Q: Hmm. Where is the value for ‘x’ coming from? Why?

The C Memory Model: The Heap

- **The Heap is the dynamic data section!**
  - Managing this memory can be very complex.
  - No garbage collection provided!!
  - We will revisit it in greater detail very soon.

```c
#include <stdlib.h> // For 'malloc'

void main(void) {
  // I want 10 integers in my array.
  // malloc returns the address in the
  // heap. But, wait, what's that * ??
  int* data = malloc(sizeof(int) * 10);
}
```

Potential Layout
(32-bit addresses)



Pointers

They point to things. They are not the things.

The "Memory Address" Variable Type

- In C, we have integer types, floating point types…
- Now we introduce our dedicated address type!

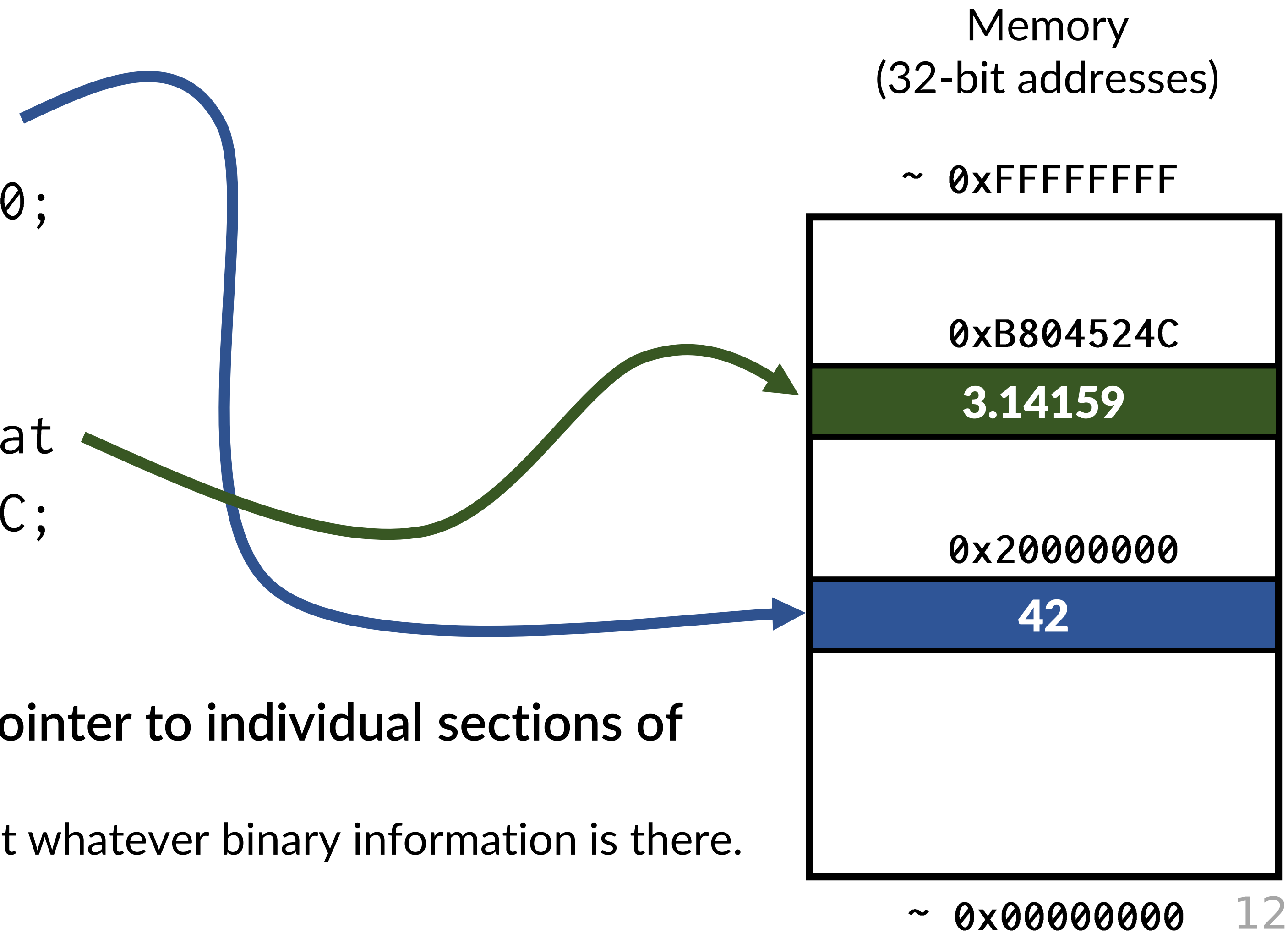
- A **pointer** is a specific variable type that holds a memory address.

- You can create a pointer that *points* to any address in memory.
- Furthermore, you can tell it what type of data it should interpret that memory to be: Just place that `*` at the end.

```
int* my_integer_somewhere;
float* hey_its_a_float;
struct Song* ah_our_trusty_song_type;
```

Interpreting Pointers: Basics

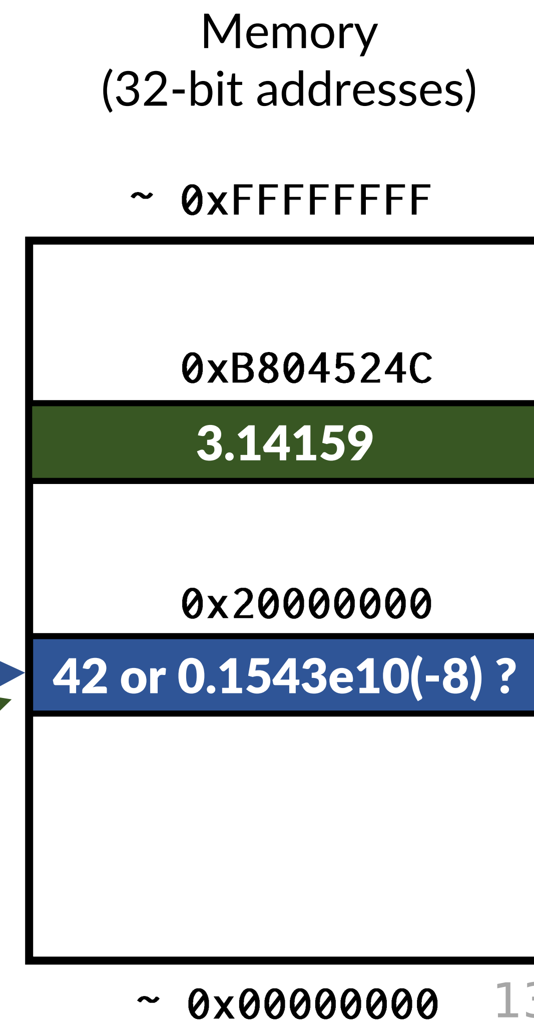
```
int* my_int
  = 0x20000000;

float* my_float
  = 0xB804524C;
```



- **Pointers can pointer to individual sections of memory.**
  - They interpret whatever binary information is there.

Interpreting Pointers: Hmm

```c
int* my_int
   = 0x20000000;


float* my_float
   = 0x20000000;
```



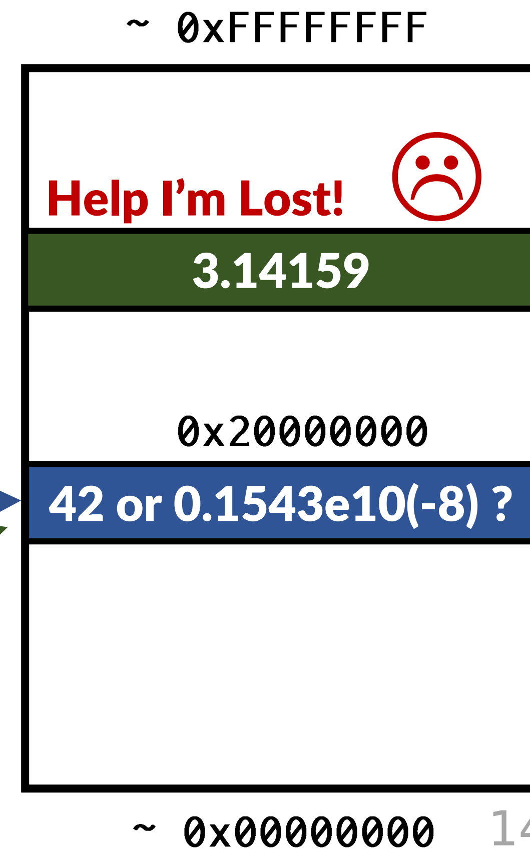
- Pointers can refer to the same address as other pointers just fine.
  - ***They interpret whatever binary information is there.***

Interpreting Pointers: A Sign of Trouble

```
int* my_int
  = 0x20000000;
```

```
float* my_float
  = 0x20000000;
```

Memory
(32-bit addresses)



- Without the pointer, allocated data may linger forever without a way to reference it again!
  - ***C does not manage freeing memory for you.***

Dereferencing Pointers: A Star is Born

- So, we have some ambiguity in our language.
- If we have a **variable that holds an address**, normal operations *change the address not the value* referenced by the pointer.

- We use the ***dereference operator*** (*)

```
int* dataptr = 0x00800000; // this address is arbitrary
dataptr = 0xffffffff;      // Reassigns the ADDRESS
*dataptr = 42;             // Reassigns the VALUE

int data = 0xc0de;         // Initializes a new variable
data = *dataptr;           // Assigns VALUE from pointer
```

Dereferencing Pointers: A Star is Born

- Remember: C implicitly coerces whatever values you throw at it…
- Incorrectly assigning a value to an address or vice versa will be…
  - … Well … It will be surprising to say the least.
- Generally, compilers will issue a warning.
  - But warnings mean it still compiles!! (You should eliminate warnings in practice)

```
int* dataptr = 0x00800000; // this address is arbitrary


int* secondptr = dataptr;  // Assigns ADDRESS
int* thirdptr = *dataptr;  // VALUE casted to ADDRESS?
```

```
example.c:4:17: warning: initialization of 'int *'
from 'int' makes pointer from integer without a cast
```

Referencing Data: An… &… is Born?

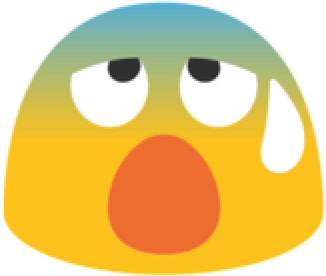
- Again… ambiguity. When do you want the address or the data?
- We can pull out the address to data and assign that to a pointer.
  - *Sometimes we refer to pointers as 'references' to data.*

- We use the ***reference operator*** (`&`)

```
int* dataptr = 0x00800000; // this address is arbitrary

int data = 0xc0de;                  // Initializes a new variable
dataptr = &data;    ← reference!    // Assigns ADDRESS to pointer
*dataptr = 42;      ← dereference!  // Assigns 42 to memory!
printf("%d\n", data);               // Prints 42!
```

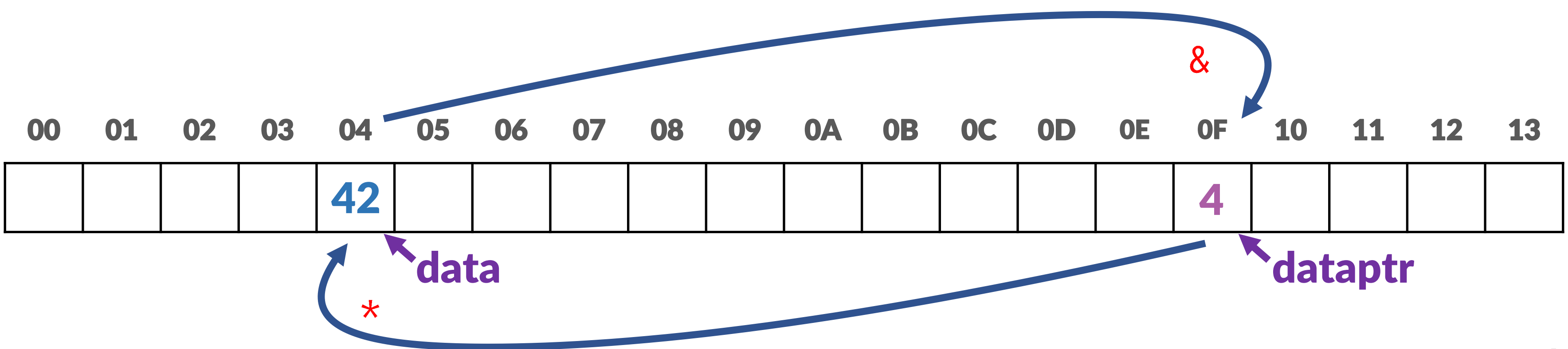
Turtles all the way down

```
int data = 42;
int* dataptr = &data;           // store address of data

// pointer to a pointer of an int:
int** dataptrptr = &dataptr; // store address of dataptr

// dereference dataptrptr... then dereference that...
*(*dataptrptr) = -64;           // store VALUE into 'data'
```



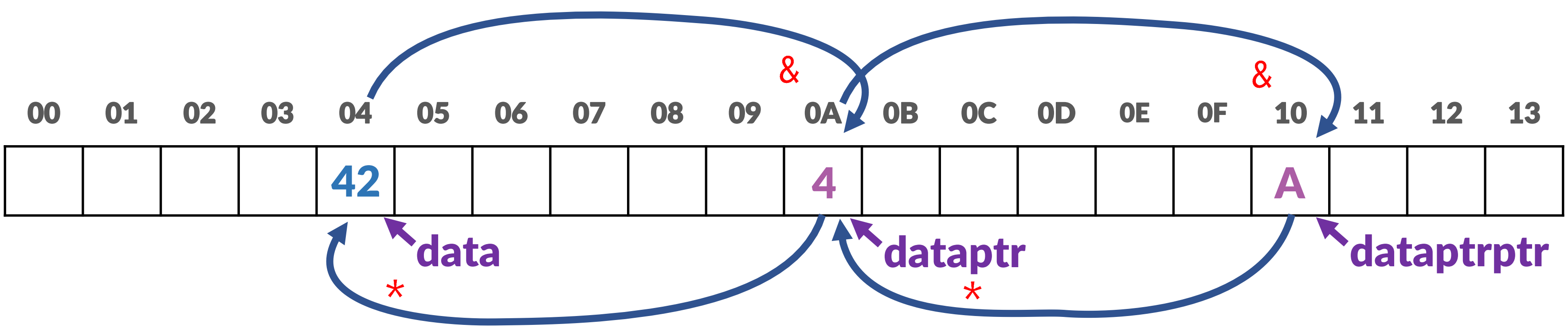
Like skipping rocks on the lake...

```
int data = 42;                   // Initializes a new variable
int* dataptr = &data;            // Assigns ADDRESS to pointer
printf("%d\n", data);            // Prints 42!
printf("%d\n", *dataptr);        // Prints 42!
printf("%p\n", dataptr);         // Prints the ADDRESS of data
          // However, 'data' could be ANYWHERE
```



Like skipping rocks on the lake...

```c
int data = 42;                  // Initializes a new variable
int* dataptr = &data;           // Assigns ADDRESS to pointer
int** dataptrptr = &dataptr;
printf("%p\n", dataptr);
printf("%p\n", dataptrptr);
printf("%d\n", **dataptrptr);    // Prints "42"!
```

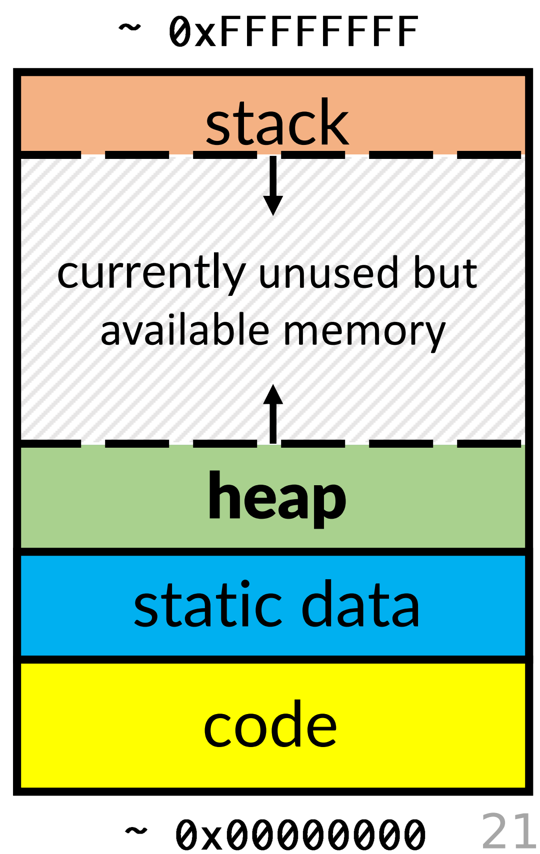


The C Memory Model: The Heap

- **The Heap is the dynamic data section!**
  - You interact with the heap entirely with pointers.
  - `malloc` returns the address to the heap with at least the number of bytes requested. Or `NULL` on error.

```
#include <stdlib.h> // For 'malloc'

void main(void) {
  // I want 10 integers in my array.
  // malloc returns the address in the
  // heap. WAIT, that's an array??!?
  int* data = malloc(sizeof(int) * 10);
}
```

Potential Layout
(32-bit addresses)



Arrays

It is what all my fellow teachers desperately need: Arrays.

(Support your local teacher's union)

Many ducks lined up in a row

- An array is simply a continuous span of memory.

- You can declare an array on the stack:

```c
void main(void) {
  int array[5]; // 5 integers... with garbage in them
}
```

- You can declare an array on the heap:

```c
void main(void) {
  // 5 integers... with garbage in them
  int* array = (int*)malloc(sizeof(int) * 5);
}
```

writing in a pedantic style, you would write the cast here.

Initialization

- You can initialize them depending on how they are allocated:

- You can initialize an array as it is allocated on the stack:

```
void main(void) { // Unspecified values default to 0:
  int array[5] = {1, 42, -3}; // [1, 42, -3, 0, 0]
}
```

- And the heap (for values other than 0, you'll need a loop):

```
void main(void) {
  // 5 integers... 'calloc' sets the memory to 0.
  int* array = (int*)calloc(5, sizeof(int));
}
```

Q: Why is using `sizeof` important here?

Carelessness means the Stack; Can stab you in the back!

— “A poem about betrayal” by wilkie

- Remember: Variables declared on the stack are **temporary**.
- All arrays can be considered pointers, but addresses to the stack are not reliable:

```
int* powers_of_two(void) {
  int array[5] = {1, 2, 4, 8, 16};
  return array;
}
```

Stack allocation

Arrays are indeed just pointers! This is an address on the stack.

Stack deallocation (oh no!)

- This may work sometimes.
  - However calling a new function will overwrite the array. ***Don’t trust it!!***
- Instead: Allocate on the heap and pass in a buffer. (next slide)

Appropriate use of arrays. Approp-array-te.

```c
#include <stddef.h>  // For 'size_t'
#include <stdlib.h>  // For 'malloc', 'calloc', and 'free'
#include <stdio.h>   // For 'printf'

void powers_of_two(int* buffer, size_t length) {
  int value = 1;
  for (int i = 0; i < length; i++) {
    buffer[i] = value;
    value *= 2;
  }
}

void main(void) {
  int* buffer = calloc(10, sizeof(int));
  powers_of_two(buffer, 10);
  for (int i = 0; i < 10; i++) {
    printf("%d\n", buffer[i]);
  }
  free(buffer);  // Make sure you free any memory you use!
}
```

Arrays don't store length. Gotta pass it in.

Pointers allow for passing arguments "by reference"

Pointers can indeed be array-like!

Heap allocation!
Although we overwrite all values, using calloc to initialize array elements to 0 reduces surprises.

Q: What happens if we pass 20 instead of 10 to powers_of_two?

Quick notes on function arguments, here...

- **All arguments are passed "by value" in C.**
  - This means the values are copied into temporary space (the stack, usually) when the functions are called.
  - This means changing those values does not change their original sources.
- **However, we can pass "by reference" indirectly using pointers:**
  - Similar to how you pass "by reference" in Java by using arrays.

```
void powers_of_two(int* buffer, size_t length) {
  int value = 1;
  for (int i = 0; i < length; i++) {
    buffer[i] = value; // Changes the value in the array.
    value *= 2;
  }
}
```

The "value" of the argument is the address.

Careful! No guard rails... You might run off the edge...

- **Since arrays are just pointers... and the length is not known...**
  - Accessing any element is correct regardless of actual intended length!
  - No array bounds checking is the source of many very serious bugs!
    - Can pull out and leak arbitrary memory.
    - Can potentially cause the program to execute arbitrarily code.

What if this is too big?

A simple mistake, but it will gleefully write to it!

```
void powers_of_two(int* buffer, size_t length) {
  int value = 1;
  for (int i = 0; i <= length; i++) {
    buffer[i] = value; // Does exactly what you say.
    value *= 2;
  }
}
```

Pointer arithmetic (Warning: it's wacky)

- **Because pointers and arrays are essentially the same concept in C…**
  - Pointers have some strange interactions with math operations.
- **Ideally pointers should "align" to their values in memory.**
  - Goal: Incrementing an `int` pointer should go to the next `int` in memory.
  - That is, not part way between two `int` values.
- **Therefore, pointer sum is scaled to the element size.**
  - Multiplication and other operators are undefined and result in a compiler error.

```
int* ptr = (int*)0x400; // Arbitrary for illustration
ptr++;                  // ptr is 0x404 (assuming 32-bit int)
ptr *= 2;               // Error: multiplication not valid!
```

Pointer arithmetic in practice:

```c
#include <stddef.h>  // For 'size_t'
#include <stdlib.h>  // For 'malloc', 'calloc', and 'free'
#include <stdio.h>

void powers_of_two(int buffer[], size_t length) {
  int value = 1;
  for (int i = 0; i < length; i++) {
    *buffer++ = value; // Assigns and then moves the pointer to the next item.
    value *= 2;
  }
}

void main(void) {
  int* buffer = calloc(10, sizeof(int));
  powers_of_two(buffer, 10);
  for (int i = 0; i < 10; i++) {
    printf("%d\n", buffer[i]);
  }
  free(buffer);  // Make sure you free any memory you use!
}
```

Alternative (and less common) way of expressing a pointer.

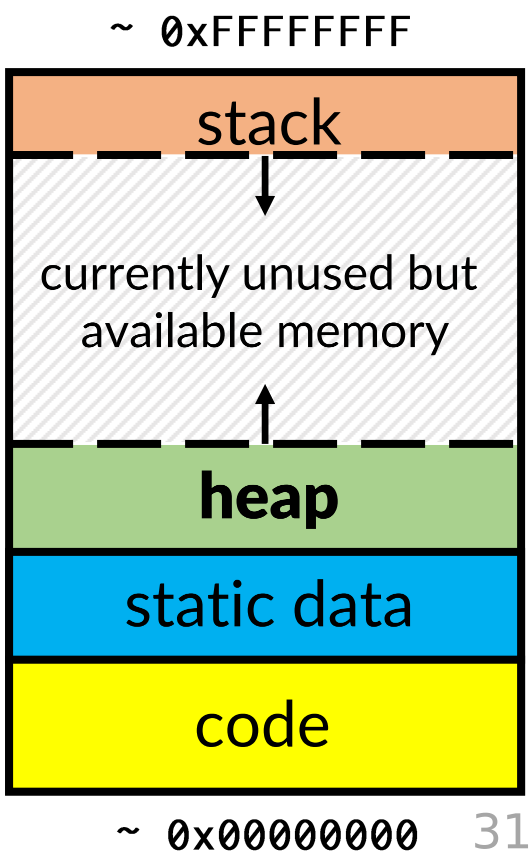
The `++` (postfix-increment) happens AFTER the dereference.
This is defined by the C language and is really confusing in practice.
(but you'll see it. often.)

The C Memory Model: The Heap

- **The Heap is the dynamic data section!**
  - You interact with the heap entirely with pointers.
  - `malloc` returns the address to the heap with at least the number of bytes requested. Or `NULL` on error.

```
#include <stdlib.h> // For 'malloc'
void main(void) {
  // I want 10 integers in my array.
  // malloc returns the address in the
  // heap. This can be used as an array.
  int* data = malloc(sizeof(int) * 10);
  data[5] = 42;
  free(data); // Good to free memory!
}
```

Potential Layout
(32-bit addresses)



Strings

No longer just for cats!

Strings

- **They are arrays and, as such, inherit all their limitations/issues.**
  - The size is not stored.
  - They are essentially just pointers to memory.

- **Text is represented as an array of `char` elements.**

- **Representing text is hard!!!**
  - Understatement of the dang century.
  - Original ASCII is 7-bit, encodes Latin and Greek
    - Hence `char` being the C integer byte type.
  - Extended for various locales haphazardly.
    - 7-bits woefully inadequate for certain languages.
  - Unicode mostly successfully unifies a variety of glyphs.
    - Tens of thousands of different characters! More than a byte!!

How long is your string?

- **Arrays in C are just pointers and as such <u>do not store their length</u>.**
  - They are simply continuous sections of memory!
  - Up to you to figure out how long it is!
    - Misreporting or assuming length is often a big source of bugs!

- **So, there are two common ways of expressing length:**
  - Storing the length alongside the array.
  - Storing a special value within the array to mark the end. (A **sentinel** value)

- **Strings in C commonly employ a sentinel value.**
  - Such a valid must be something considered invalid for actual data.
  - How do you know how long such an array is?
    - You will have to search for the sentinel value! Incurring a $O(n)$ time cost.

The string literal.

- **String literals should be familiar from Java.**
  - However, in C, they are `char` pointers. (That is: `char*`)
  - The contents of the literal are read-only (immutable) so it is a: `const char*`
    - Modifying it crashes your program!!
    - A pointer that can't change pointing to an immutable string is a `const char* const`

Let's ignore this! ☺ (for now)

```c
#include <stdio.h> // For 'printf'

void main(void) {
  // You could specify it as: char my_string[] = "...";
  // But that would allocate the string on the stack!
  const char* my_string = "Hello World.";
  printf("%s\n", my_string);
}
```

The variable is allocated on the stack, which is a pointer. The string itself is likely in the static data segment!

How long is your string? Let's find out.

- **The `strlen` standard library function reports the length of a string.**
  - This is done in roughly $O(n)$ time as it must find the sentinel.
  - The following code investigates and prints out the sentinel:

```
#include <stdio.h>  // For 'printf'
#include <string.h> // For 'strlen'

void main(void) {
  const char* my_string = "Hello World.";
  int length = strlen(my_string);
  printf("length: %d\n", length);
  printf("sentinel: %x\n", my_string[length]);
}
```

When good strings go bad.

- **What happens if that sentinel… was not there?**
  - Well… it would keep counting garbage memory until it sees a 0.

This syntax copies the string literal on to the stack.
This allows us to modify it. (otherwise, it is immutable)

```
#include <stdio.h>  // For 'printf'
#include <string.h> // For 'strlen'

void main(void) {
  char my_string[] = "Hello World.";
  int length = strlen(my_string);
  my_string[length] = 42; // Corrupt the sentinel
  length = strlen(my_string); // Uh oh.
}
```

The length here depends on the state of memory in the stack.

Using stronger strings. A… rope… perhaps.

- To ensure that malicious input is less likely to be disastrous…
  - We have alternative standard functions that set a maximum length.

```
#include <stdio.h>  // For 'printf'
#include <string.h> // For 'strnlen'

void main(void) {
  char my_string[] = "Hello World.";
  int length = strlen(my_string);
  my_string[length] = 42; // Corrupt the sentinel
  length = strnlen(my_string, 12); // That's fine.
}
```

strnlen will stop after the 12th character if it does not see a sentinel.

Comparing “Apples” to “Oranges”

- **When you compare strings using `==` it compares the addresses!**
  - Since string literals are constant, they only exist in the executable once.
  - All references will refer to the same string!

```c
#include <stdio.h> // For 'printf'
#include <string.h> // For 'strncmp'

void main(void) {
  char* string1 = "apples";
  char* string2 = "apples";

  if (string1 == string2) {
    printf("same\n"); // This runs!
  }
}
```

Comparing "Apples" to "Oranges"

- **When the addresses differ, they are not equal.**
  - So, you have to be careful when comparing them.
  - This is similar to Java when considering `==` versus `String.equals()`

```
#include <stdio.h> // For 'printf'
#include <string.h> // For 'strncmp'

void main(void) {
  char string1[] = "apples";
  char string2[] = "apples";

  if (string1 == string2) {
    printf("same\n"); // This does not run!
  }
}
```

Comparing “Apples” to “Oranges”

- **To compare values instead, use the standard library’s `strcmp`.**
  - This will perform a byte-by-byte comparison of the string.
    - Upon finding a difference, it returns rough difference between those contrary bytes.
    - When they are the same, then the difference is 0!
  - Therefore, it is case sensitive! It also has a $O(n)$ time complexity.

```
#include <stdio.h> // For 'printf'
#include <string.h> // For 'strncmp'

void main(void) {
  char* string1 = "apples";
  char* string2 = "apples";

  if (strcmp(string1, string2) == 0) {
    printf("same\n"); // This runs!
  }
}  // You could write it as: if(!strcmp(string1, string2))
```

`strcmp` **will return 0 when the strings are equal.**

Appropriate string construction. A-rope-riate.

```
#include <stdio.h>  // For 'printf' and 'scanf'
#include <string.h> // For 'strnlen' etc
#include <stdlib.h> // For 'calloc' and 'free'

#define MAX_STRING 100

void main(void) {
  const char* str_start = "Hello, ";
  const char* str_end = "!";
  char* str_name = calloc(MAX_STRING + 1, sizeof(char));
  char* my_buffer = calloc(MAX_STRING + 1, sizeof(char));

  printf("Type in your name: ");   // Let someone type in their name
  scanf("%100s", str_name);        // The term %100s has it record at most 100 characters to str_name.

  strncpy(my_buffer, str_start, MAX_STRING);
  strncat(my_buffer, str_name, MAX_STRING);
  strncat(my_buffer, str_end, MAX_STRING);
  printf("%s\n", my_buffer);

  free(str_name);
  free(my_buffer);
} // Prints "Hello, wilkie!" depending on what you've typed in.
```

- C is a very deliberate language.

`calloc` **is important here! Ensures string has a length of 0. (is initially empty, not garbage!)**

Like a ballroom. Empty, but spacious.

`strncpy` **is the bounded form of** `strcpy`**. Overwrites string.**

`strncat` **is the bounded form of** `strcat`**. Concatenates to end of existing string.**

Memory/Strings: Summary

- Memory Allocation
  - `#include <stdlib.h>`
  - `malloc(size_t length)` Returns pointer to length bytes
  - `calloc(size_t count, size_t size)` Returns pointer to (count*size) bytes, zeros them
  - `free(void* ptr)` Deallocates memory at ‘ptr’ so it can be allocated elsewhere

- Strings
  - `#include <string.h>`
  - `strcpy(char* dst, const char* src)` Copies src to dst overwriting dst.
  - `strncpy(char* dst, const char* src, size_t max)` Copies up to ‘max’ to dst.
  - `strcat(char* dst, const char* src)` Copies string from src to end of dst.
  - `strncat(char* dst, const char* src, size_t max)` Copies up to ‘max’ to end of dst.
  - `strcmp(const char* a, const char* b)` Returns difference between strings. (0 if equal)
  - `strncmp(const char* a, const char* b, size_t max)` Compares up to ‘max’ bytes.

  - Generally safer to use the bounded forms.

Input/Output: Summary

- Input
  - `#include <stdio.h>`
  - `scanf("%s", my_buffer)` Copies string input by user into buffer (unsafe!)
  - `scanf("%10s", my_buffer)` Copies up to 10 chars into buffer (my_buffer needs to be >= 11 bytes for sentinel)
  - `scanf("%d", &my_int)` Interprets input and places value into int variable.

`scanf` updates your variable, so you need to pass the address.
(`my_buffer` does not need it. Strings are already `char*`)

- Output
  - `#include <stdio.h>`
  - `printf("%s", my_buffer)` Prints string. (technically unsafe)
  - `printf("%10s", my_buffer)` Prints up to 10 chars from string. (safe as long as my_buffer is >= 10 bytes)
  - `printf("%d", my_int)` Prints int variable. (d for decimal, unfortunately)
  - `printf("%x", my_int)` Prints int variable in hexadecimal. (x for hex)
  - `printf("%l", my_int)` Prints long variable.
  - `printf("%ul", my_int)` Prints unsigned long variable.

  - Lots more variations! Generally `scanf` and `printf` share terms. Look them up!