Introductions, C Refresher

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

pollev.com/tqm

- How are you?

Administrivia

- First Assignment (HW00 simple_string)
  - Releases Tonight
  - “Due” Tuesday next week 9/2
  - Extended to be due Tuesday the 9th (course selection period ends)
  - Mostly a C refresher
  - Don’t put it off, another assignment (a more rigorous C refresher) will also be due the 9th.

- Pre semester Survey
  - Anonymous
  - Due Wednesday the 9th

Lecture Outline

- **Introduction & Logistics**
  - **Course Overview**
  - **Assignments & Exams**
  - **Policies**
- C “Refresher”
  - Context in this course
  - memory
  - Pointers
  - Arrays
  - Structs
  - The heap
  - const

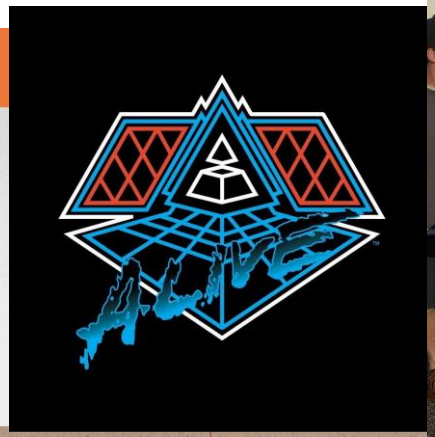
Instructor: Travis McGaha

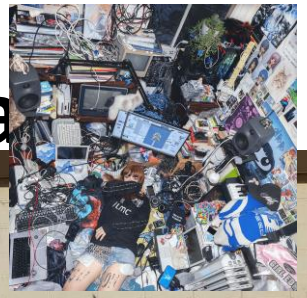
- UPenn CIS faculty member since August 2021
  - I am CREATING this course
  - BUT I have taught a lot of this content across other courses.
  - A lot of new stuff! But still some stability

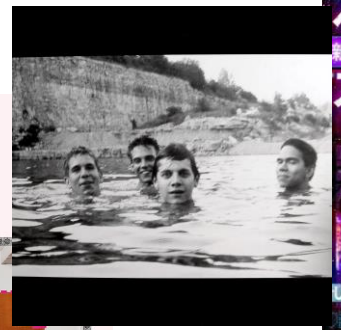
- And…

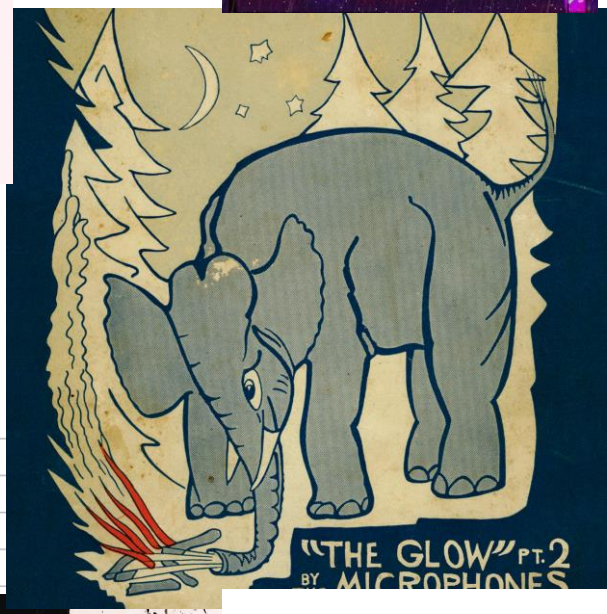
Instructor: Travis McGaha

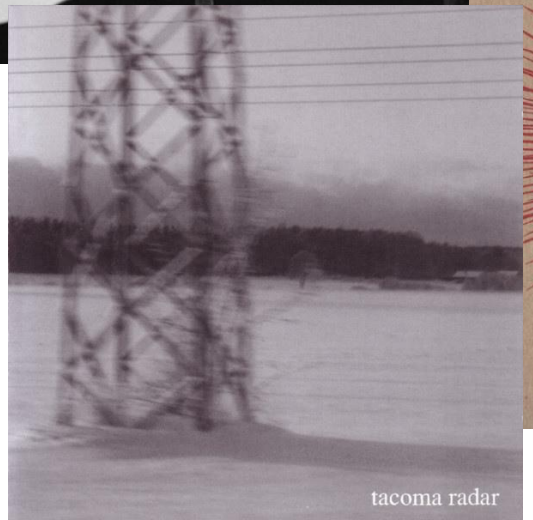
- I like most music

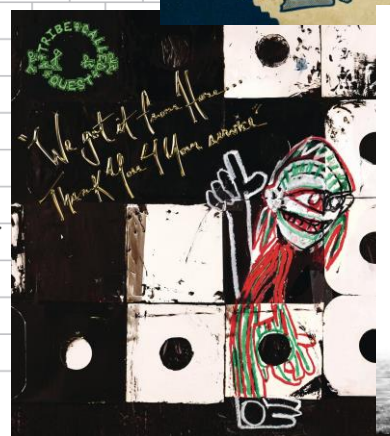










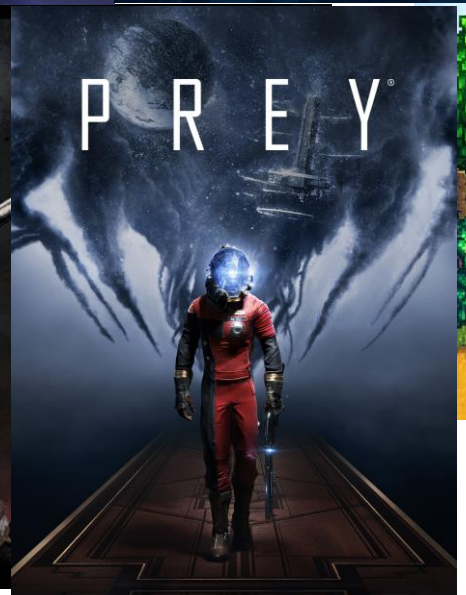


Instructor: Travis McGaha

- I like animals and going outside (especially birds, cats and mountains)

Instructor: Travis McGaha

- I like video games



Instructor: Travis McGaha

- I have a general dislike of food
  (Breakfast is pretty good tho)

Instructor: Travis McGaha

- I care a lot about your actual learning and that you have a good experience with the course
- I am a human being and I know that you are one too. If you are facing difficulties, please let me know and we can try and work something out.
- More on my personal website: https://www.cis.upenn.edu/~tqmcgaha/

1-5 on your fingers

- How confident are you in your C programming?
- What about your ability to write a program from scratch in general?

Course Overview

Course Overview

- C++
  - Taught in a way that hopefully prepares you for systems programming, whether you continue it in C++, C, Rust, or some other “value semantics” programming language.
- And Wider Systems Topics
  - Performance
  - Locality
  - Concurrency
- Program Design & Style

“Lies-to-children”

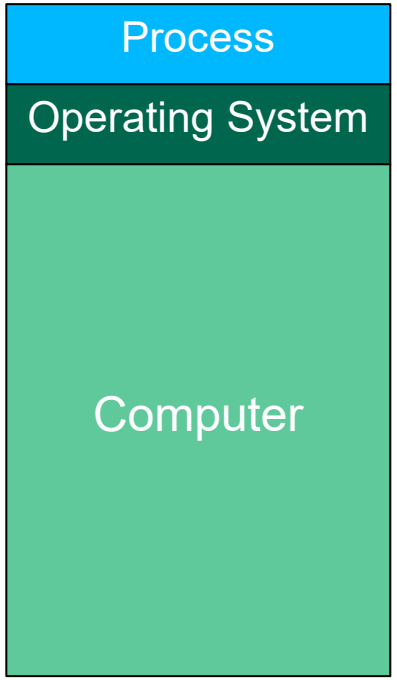
- "The necessarily simplified stories we tell children and students as a foundation for understanding so that eventually they can discover that they are not, in fact, true."
  - Andrew Sawyer (Narrativium and Lies-to-Children: 'Palatable Instruction in 'The Science of Discworld' ‘)

“Lies-to-children”

- "A lie-to-children is a statement that is false, but which nevertheless leads the child's mind towards a more accurate explanation, one that the child will only be able to appreciate if it has been primed with the lie"
  - Terry Pratchett, Ian Stewart & Jack Cohen (The Science of Discworld)

We lied to you (but in a good way)

- Is the LC4 model for a computer true? **Eh……. no**
- Is it a useful model? **Yes**



We lied to you (but in a good way)

- Is memory one giant array of bytes? **Eh……. no**
- Is this a useful model? **Yes**

I’m going to lie to you (but in a good way)

- "All models are wrong, but some are useful."
  - Same source as below.

- "If it were necessary for us to understand how every component of our daily lives works in order to function - we simply would not."
  - AnRel (UNHINGED: A Guide to Revolution for Nerds & Skeptics)

- This course will reveal more details, but there is still a ton I am leaving out. Even what I say that is accurate, will likely change in the future.

Prerequisites

- Course Prerequisites:
  - CIS 2400
  - Some of CIS 1200/1210

- What you should be familiar with already:
  - **C** programming experience
  - C Memory Model (we will build on this point and the previous point)
  - Computer Architecture **Model** (e.g. the high level, I won't talk about transistors)
  - Basic UNIX command line skills
  - Some basic data structures & algorithmic analysis

Learning Objectives

- To leave the class with a better understanding of:
  - C++
  - How a lot of software level structures work
  - How software “interfaces” with the Operating System
  - How a computer runs/manages multiple programs
  - Various system resources and how to apply those to code
    - Threads, networking, file I/O
- You should leave this with a solid foundation to explore higher level systems courses.
- Topics list/schedule can be found on course website
  - Note: This is tentative

Disclaimer

- A lot of the course is tentative
  - Travis has taught this before but is CHANGING A LOT this time

- This is a digest, **READ THE SYLLABUS**
  - https://www.cis.upenn.edu/~tqmcgaha/cis3990/25fa/documents/syllabus
  - Note: Syllabus is still being updated

What if I have already taken CIS 5XXX?

- Some of you have already taken CIS 5480 or some other upper-level systems course. Will you still benefit from the course?

- I am pretty sure you will
  - These other courses have probably taught you some of these things, but (from what I can tell) they go over a lot of this stuff fast.
  - Yes, a lot of this is “fundamentals” but the fundamentals are what everything else builds on top of (so they deserve more time)

- This course has the most overlap with CIS 5480. Why?
  - Cause I have taught that course before
  - Because it is in-part the “current” “intermediate” systems course.
    I do not think it is doing a good job at being an intermediate, so I hope this course will “free up” CIS 5480 so it can focus on being an Operating Systems course.

Course Components pt. 1

- Lectures (~26)
  - Introduces concepts, slides & recordings available on canvas
  - In lecture polling & Activities.
- Recitations (12)
  - Reiterates lecture content, lecture clarifications, assignment & exam preparation
- Programming Projects (~10)
  - Due every ~1 week
  - Applications of course content
  - Usually have everything you need for an assignment when it is released
- Check-in “Quizzes” (~12)
  - Unlimited attempt low-stake quizzes on Ed to make sure you are caught up with the material

Course Components pt. 2

- Final Project (1)
  - Due at the end of the semester
  - Can be done solo or in partners (tentatively)
  - Further Details TBD
- Exams (2)
  - Two in-person exams, some personal notes will be allowed
  - Details TBD
- Textbook (0)
  - No Textbook, but using a C++ reference would probably be useful
  - https://cplusplus.com/
  - https://en.cppreference.com/w/

Policies for Growth

- Course policies are designed to be flexible and to provide opportunities for you to get feedback and GROW

Course Grading (Tentative)

- Breakdown:
  - Homework assignments (54%)
  - Final Project (14%)
  - Exams (27%)
    - Midterm 9%
    - Final 18%
  - Oral Concept Discussion (5%)
- Engagement Credits to determine +/-

- Final Grade Calculations:
  - I would LOVE to give everyone an A+ if it is earned
  - Final grade cut-offs will be decided privately at the end of the Semester

Engagement Credits

- "Engagement Credits" keep track of the various ways you may interact with the course.
- The % on the previous slide determines whether you get some kind of an "A", some kind of a "B", etc.
- These credits will be used to decide whether you get a "+", "-" or neither on the letter grade you earn

Activity	Points per	# of occurances
Lecture Attendance	1	~27
Check-in Diagnostic	3	~12
Recitation Attendance	3	~12
Course Surveys	4	~3
Staff Endorsed Ed Answer	2	Uncapped
Other Activities	Varies	Varies

Engagement Credits

- "Engagement Credits" keep track of the various ways you may interact with the course.
- The % on the previous slide determines whether you get some kind of an "A", some kind of a "B", etc.
- These credits will be used to decide whether you get a "+", "-" or neither on the letter grade you earn
  - If you earn an A, you will need
    - 110 credits for an A+
    - 90 credits for an A
    - < 90 credits for an A-
  - If you earn a B, you will need
    - 100 credits for a B+
    - 80 credits for a B
    - < 80 credits for a B-

HW Late Policy

- Check-ins are due before Monday's lecture and cannot be turned in late
- HW's cannot be turned in late, but they can be reopened
  - When you submit a check-in you can also say you want to re-open ONE homework assignment.
    That homework assignment will be re-opened till the next check-in is due.
  - You can re-open the same assignment multiple times
  - The final project can't be re-opened

- End of the semester is the end. No submissions past that.
  (unless there is particularly special circumstances)

HW Grading

- Roughly every other programming homework assignment is graded on style in addition to correctness from an autograder
- Style grading is done manually and only if you get 100% on the autograder.
  - We let you re-open assignments later in the semester, so you can eventually get full correctness and fix style
  - It can take a lot of work to grade these manually, especially with the re-opens, so we need to make sure we stick to a reasonable number of submissions to grade and regrade.
  - For each style issue we will leave a comment and mark a rubric item in gradescope.

HW Grading

- Roughly every other programming homework assignment is graded on style in addition to correctness from an autograder
- Notably, each rubric item / style issue will necessarily result in a deduction on your grade. We instead categorize the overall quality of your assignment.
  - Excellent (E) - 100% - Code Quality and displayed mastery of the content is perfect, or has only a few very minor flaws.
  - Satisfactory (S) - 80% - Code Quality and mastery of material is pretty good
  - Needs improvement (N) - 50% - Code works, but demonstrates an incomplete mastery of topics and/or code-quality.
  - Unassessable (U) - 0% - Code either doesn't pass the already existing automated tests, or has enough major flaws to require an almost rewrite of the code.
- More details in syllabus

Midterm Clobber Policy

- The Final Exam is cumulative
  - Made up of “midterm” material and “post midterm” material
  - If you do better on the “midterm” section of the final, your midterm grade can be overwritten.
  - Accounts for the exam’s being too easy/hard by comparing to the standard deviation & mean of the exam.
  - Formula is in the syllabus

  - This does not work in reverse, if you do poorly on the “midterm” part of the final I will not improve that section of your final exam.
    - The point with this policy is to demonstrate growth

Course Infrastructure

- Course website
  - Schedule, syllabus, assignment specifications, materials …
  - If you need something it is probably linked from the course site
- Docker
  - Coding environment for hw's, code is submitted to GradeScope
- GradeScope
  - Used for exam grades & HW submissions
- Github
  - You will have a repo for the course and you will submit to gradescope via your repo

Course Infrastructure

- Poll Everywhere
  - Used for some lecture polls
- Ed
  - Course discussion board, in-class activities and for check-in quizzes
- Canvas
  - Grades, lecture recordings & surveys

Collaboration Policy Violation

- Generative AI
  - I **strongly** recommend **against** using generative AI like ChatGPT, Co-Pilot or related technologies. I am not denying that ChatGPT can be a useful tool for getting something made, but I not convinced that it benefits your growth and learning of the material in this class.
  - You learn by doing things. If you aren't doing the critical thinking, the learning, the programming yourself, then that experience (and knowledge) will not stick with you.
    - AI Tools in Society: Impacts on Cognitive Offloading and the Future of Critical Thinking
    - (More in syllabus)
- You will not help your overall grade and happiness:
  - Quizzed individually during project demo, exams on project in finals
  - If you can't explain your code in OH, we can turn you away.
    - This is different than being confused on a bug or with C, this is ok
  - Personal lifelong satisfaction from completing the course

Getting Help

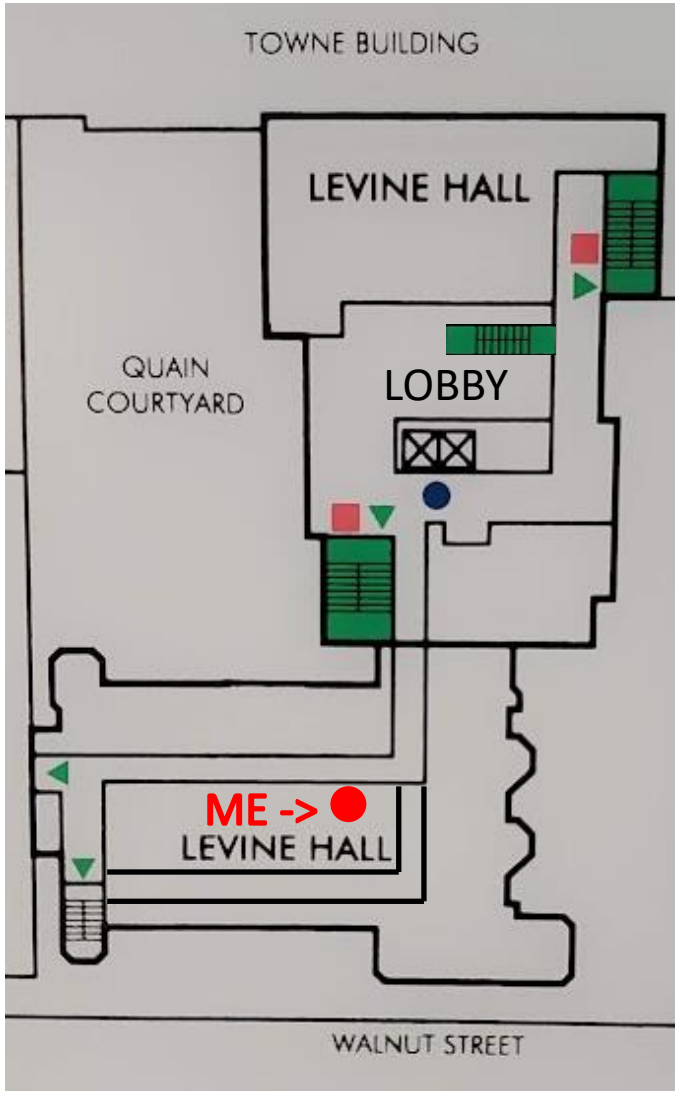
- Ed
  - Announcements will be made through here
  - Ask and answer questions
  - Sign up if you haven't already!

- Office Hours:
  - Can be found on calendar on front page of canvas page
  - Starts next week (hopefully)

- 1-on-1's:
  - Can schedule 1-on-1's with Travis
  - Should attend OH and use Ed when possible, but this is an option for when OH and Ed can't meet your needs

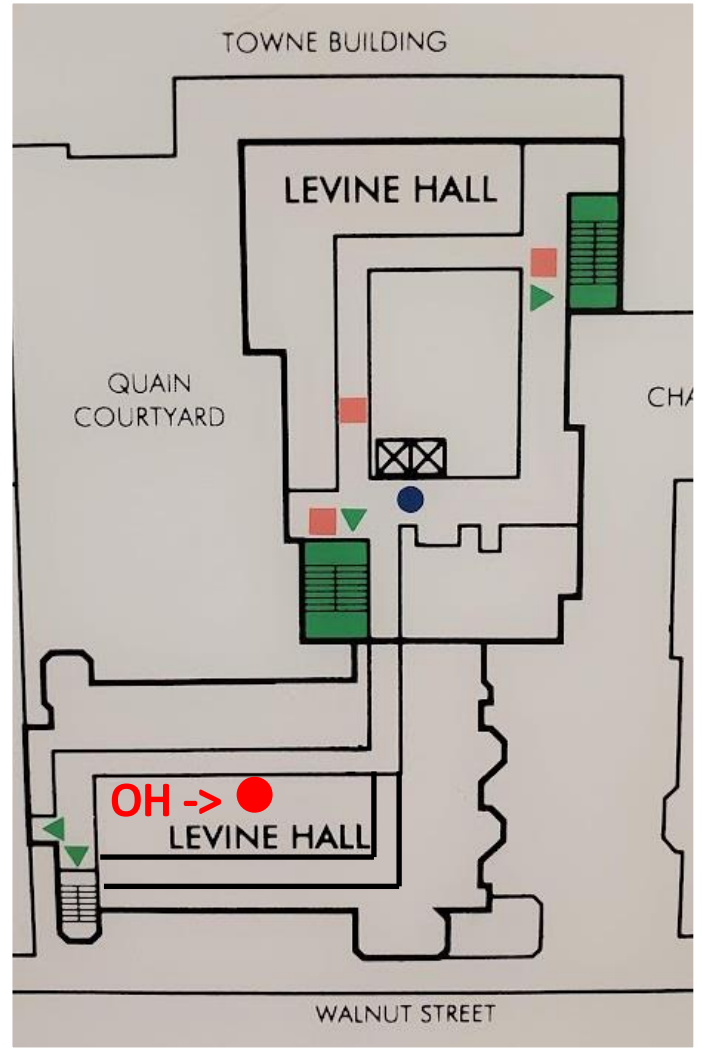
OH Locations

- Can see Office Hours on the course site calendar
- Travis' Office Hours are in his office (Levine 269 C)
- 2nd floor Levine ->



OH Locations

- Can see Office Hours on the course site calendar
- Theodor's Office Hours are in Levine 3rd floor bump space
- Map of 3rd floor ->



Collaboration Policy Violation

- We do look for it:
  - Careful grading by teaching staff for most classes
  - Measure of Software Similarity (MOSS): http://theory.stanford.edu/~aiken/moss/
  - Successfully used in several classes at Penn

- Penalty depends on severity.
  - Zero on the assignment, (5%) deduction on final grade. F grade if caught twice.
  - First-time offenders may be reported to Office of Student Conduct with no exceptions. Possible suspension from school
  - Your friend from last semester who gave the code will have their grade retrospectively downgraded.
  - Posting code publicly is a BIG NO NO
  - If you come to us first before we catch you, we will give you the opportunity to “make up”

We Care

- We are still figuring things out, but we do care about you and your experience with the course
  - There is a pre-semester survey available on canvas now. Please fill this out honestly and we will do our best to incorporate people's answers
  - Please reach out to course staff if something comes up and you need help
- **PLEASE DO NOT CHEAT OR VIOLATE ACADEMIC INTEGRITY**
  - **We know that things can be tough, but please reach out if you feel tempted. We want to help**
  - **Read more on academic integrity in the syllabus**
- We do not just say this, hopefully the policies also show that we care.

pollev.com/tqm

- Any questions, comments or concerns so far?

Lecture Outline

- Introduction & Logistics
  - Course Overview
  - Assignments & Exams
  - Policies
- **C "Refresher"**
  - **Context in this course**
  - **memory**
  - **Pointers**
  - Arrays
  - Structs
  - The heap
  - const

Context of C in this course

- You will be writing C++ in this course, not C
  - Most C is legal C++
  - For the first few assignments you will write C++ code that also mostly works as C code
- C++ is not C
  - C is the foundation for C++, but ***modern*** C++ is very different
  - We will refresh ourselves on this C foundation but quickly move on to C++
- Recitation tomorrow:
  - More C refresher
  - Not recorded, but materials will be posted

Aside: Hello World in C++

```cpp
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- Looks simple enough…
  - Let's walk through the program step-by-step to highlight some differences

Aside: Hello World in C++

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- `iostream` is part of the ***C++*** standard library
  - Note: you don't write "`.h`" when you include C++ standard library headers
    - But you *do* for local headers (*e.g.* `#include "Deque.hpp"`)
  - `iostream` declares stream *object* instances
    - *e.g.* `cin`, `cout`, `cerr`

Aside: Hello World in C++

```
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- `cstdlib` is the ***C*** standard library's `stdlib.h`
  - Nearly all C standard library functions are available to you
    - For C header `math.h`, you should `#include <cmath>`
  - We include it here for `EXIT_SUCCESS`

Aside: Hello World in C++

```
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- `using namespace std;`
  - It is there because I said so (can't use it in header files tho)
  - We include it here so that I can say `cout` instead of `std::cout`

```
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

int main() {
  std::cout << "Hello, World!" << std:: endl;
  return EXIT_SUCCESS;
}
```

Aside: Hello World in C++

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- "`cout`" is an object instance declared by `iostream`, C++'s name for `stdout`
  - `std::cout` is an object of class `ostream`
    - http://www.cplusplus.com/reference/ostream/ostream/
  - Used to format and write output to the console
  - We use << to send data to cout to get printed

Aside: Hello World in C++

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- `endl` is a pointer to a "manipulator" function
  - This manipulator function writes newline (`'\n'`) to the `ostream` it is invoked on and then flushes the `ostream`'s buffer
  - This *enforces* that something is printed to the console at this point

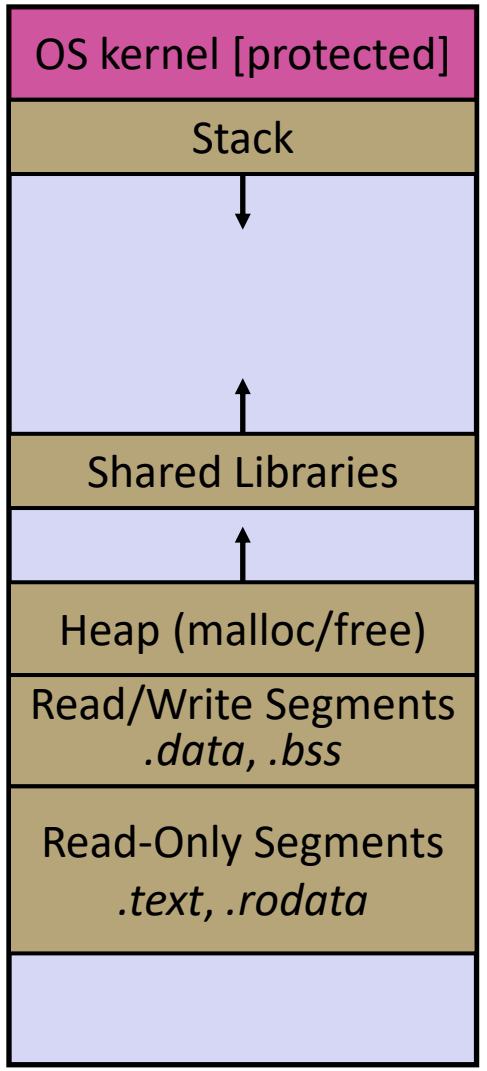
Aside: Error Printing

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

using namespace std;

int main() {
  // ...
  cerr << "ERROR: Invalid Argument!" << endl;
  return EXIT_SUCCESS;
}
```

- "`cerr`" is used if we want to print an error message

Memory

- Where all data, code, etc are stored for a program
- Broken up into several segments:
  - The stack
  - The heap
  - The kernel
  - Etc.
- Each “unit” of memory has an address



Memory as an array of bytes

- Everything in memory is made of bits and bytes
  - Bits: a single 1 or 0
  - Byte: 8 bits
- Memory is a giant array of bytes where everything* is stored
  - Each byte has its own address (“index”)
- Some types take up one byte, others more

```
int main() {
  char c = 'A';
  char other = '0';
  int x = 3034;
}
```

Memory is Huge

- Modern computers are called “64-bit”
  - Addresses are 64-bits (8-bytes)
  - There are 2^{64} possible memory locations, each location is 1-byte
  - 2^{64} is **18,446,744,073,709,551,616.**
  - Pointers must be 64-bits (8-bytes) to be able to hold any address on the computer.

Pointers

POINTERS ARE EXTREMELY IMPORTANT IN C & C++

- Variables that store addresses
  - It stores the address to somewhere in memory
  - Must specify a type so the data at that address can be interpreted
- Generic definition: `type* name;` or `type *name;` (equivalent)
  - Example: `int *ptr;`
    - Declares a variable that can contain an address
    - Trying to access that data at that address will treat the data there as an int

Pointer Operators

- *Dereference* a pointer using the unary * operator
  - Access the memory referred to by a pointer
  - Can be used to read or write the memory at the address
  - Example:

```
int *ptr = ...; // Assume initialized
int a = *ptr; // read the value
*ptr = a + 2; // write the value
```

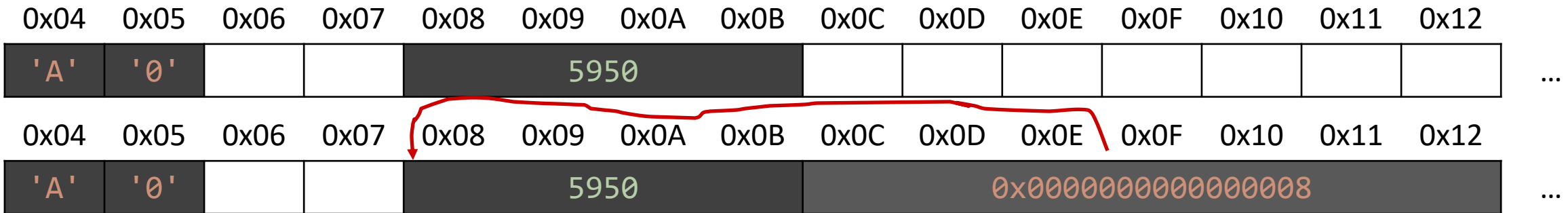
- Get the address of a variable with &
  - `&foo` gets the address of foo in memory
  - Example:

```
int a = 5950;
int *ptr = &a;
*ptr = 2; // 'a' now holds 2
```

Memory as an array of bytes

- Everything in memory is made of bits and bytes
  - Bits: a single 1 or 0
  - Byte: 8 bits
- Memory is a giant array of bytes where everything* is stored
  - Each byte has its own address (“index”)
- Some types take up one byte, others more

```
int main() {
  char c = 'A';
  char other = '0';
  int x = 5950;
  int* ptr = &x;
}
```



Aside: `nullptr`

- `nullptr` is a memory location that is guaranteed to be invalid
  - In C on Linux, `NULL` is `0x0` and an attempt to dereference `NULL` *causes a segmentation fault*
  - **In C++ (and modern C) we use `nullptr` instead**

- Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error
  - It's better to cause a segfault than to allow the corruption of memory!

```
int main(int argc, char** argv) {
  int* p = nullptr;
  *p = 1;  // causes a segmentation fault
  return EXIT_SUCCESS;
}
```

Arrays in C

- Definition: `type` `type name[size]`
  - Allocates `size*sizeof(type)` bytes of *contiguous* memory
  - Normal usage is a compile-time constant for `size`
    (*e.g.* `int scores[175];`)
  - Initially, array values are "garbage"

- Size of an array
  - **Not stored anywhere** – array does not know its own size!
  - The programmer will have to store the length in another variable or hard-code it in
  - No bounds checking!

Using Arrays

Optional when initializing

- Initialization: `type name[size] = {val0,…,valN};`
  - `{}` initialization can *only* be used at time of definition
  - If no size supplied, infers from length of array initializer

- Array name used as identifier for "collection of data"
  - `name[index]` specifies an element of the array and can be used as an assignment target or as a value in an expression
  - Array name (by itself) produces the address of the start of the array
    - Cannot be assigned to / changed

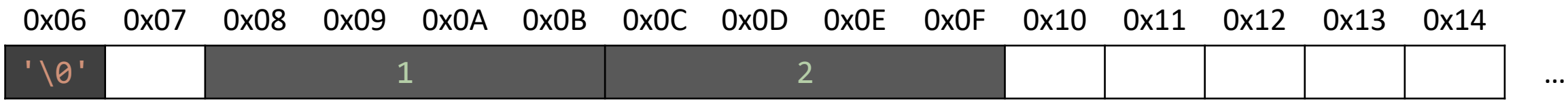
```
int primes[6] = {2, 3, 5, 6, 11, 13};
primes[3] = 7;
primes[100] = 0;  // memory smash!
```

No IndexOutOfBounds
Hope for segfault

Arrays in C

- Here is a memory diagram example:

```c
int main() {
  char c = '\0';

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

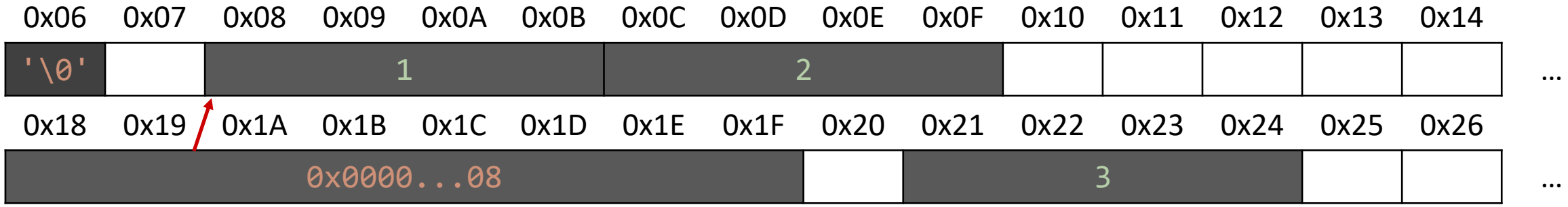


Pointers as C arrays

- Pointers can be set to an array

- Pointers can always be indexed into like an array
  - Pointers don't always have to point to the beginning of an array!

```
int main() {
  char c = '\0';

  int arr[2] = {1, 2};

  int* ptr = arr;

  int x = ptr[1] + 1;
}
```



The Heap

- For most program memory we care about, things are stored either in the **heap** or **stack**

- In C we allocated with `malloc()` and deallocated with `free()`

- In C++ we will use `new` and `delete`.
  - New still gives us a pointer to the heap
  - We must deallocate the pointer with delete when we are done with the pointer.

```
int main() {
  int* x = new int;

  *x = 3;

  // prints *x which is 3
  cout << *x << endl;

  delete x;
}
```

The Heap

- In C++ we will use `new` and `delete`.
  - New still gives us a pointer to the heap
  - Can use new to allocate an array!
  - Will need this to allocate an array of characters (so a C-style string) in the first homework assignment.
  - We deallocate arrays with `delete[]`

```
int main() {

  int* arr = new int[2];

  arr[0] = 5930;
  arr[1] = 5950;

  delete[] arr;
}
```

- We use the heap when we want memory to stay allocated past the lifetime of a function.
- Will talk more about what the heap is and why it is important next lecture. This should be enough for HW00 though.

Ed Discussion

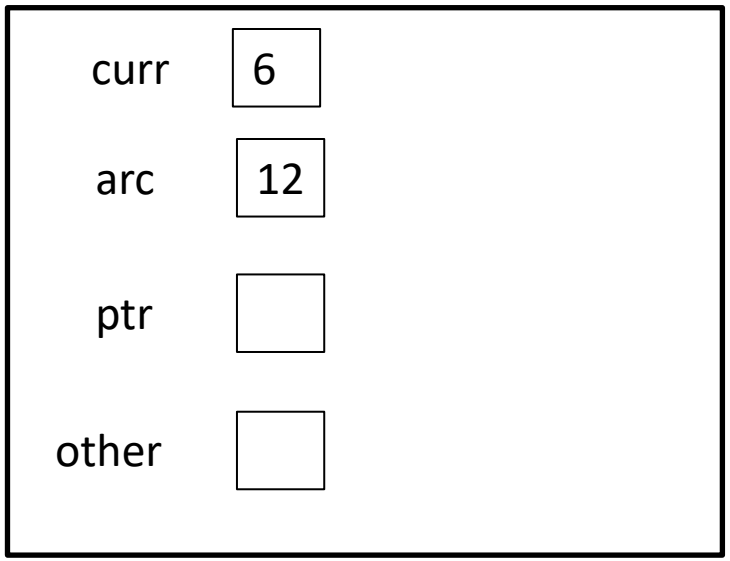
- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

```
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int** other = &ptr;
  ptr = &arc;
  **other = curr;
  *other += 3;

  // print curr and arc
  cout << curr << endl;
  cout << arc << endl;
}
```

Ed Discussion

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

`main()`'s stack frame

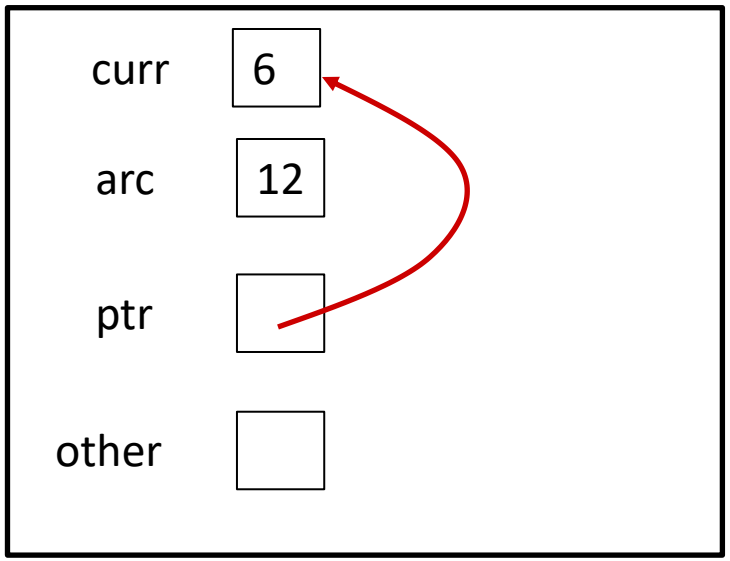



```
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int** other = &ptr;
  ptr = &arc;
  **other = curr;
  *other += 3;

  // print curr and arc
  cout << curr << endl;
  cout << arc << endl;
}
```

Ed Discussion

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

`main()`'s stack frame

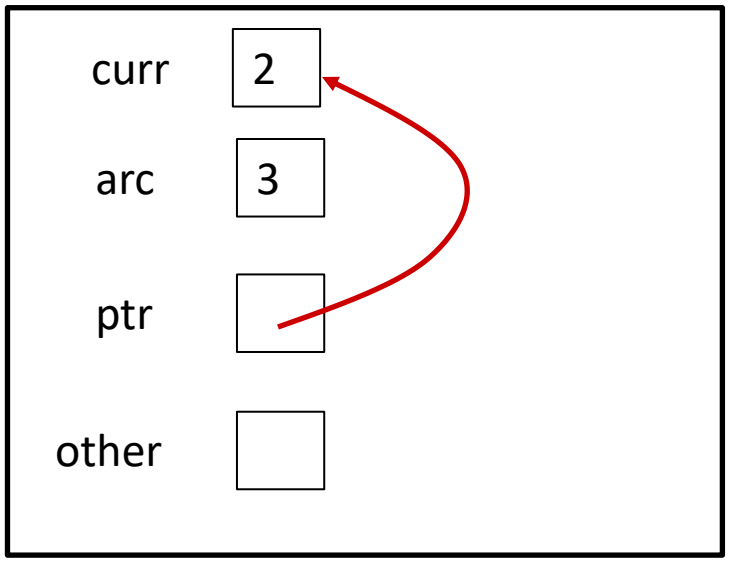



```
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int** other = &ptr;
  ptr = &arc;
  **other = curr;
  *other += 3;

  // print curr and arc
  cout << curr << endl;
  cout << arc << endl;
}
```

Ed Discussion

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

`main()`'s stack frame

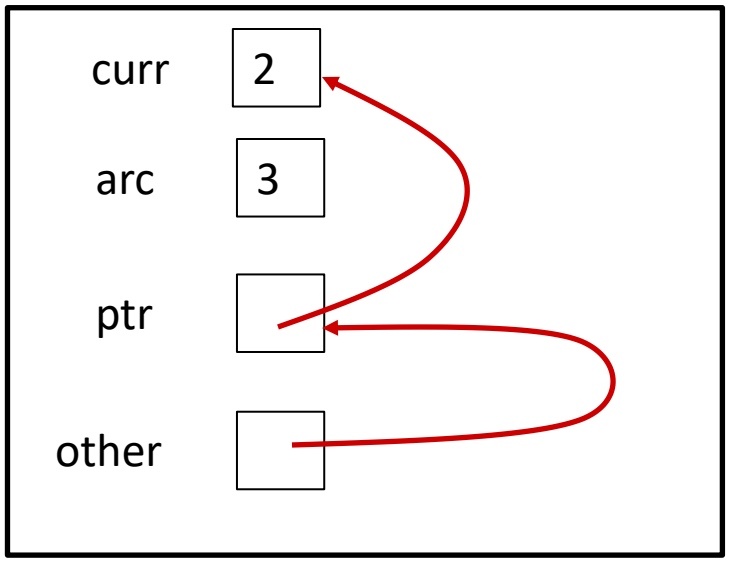



```
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int** other = &ptr;
  ptr = &arc;
  **other = curr;
  *other += 3;

  // print curr and arc
  cout << curr << endl;
  cout << arc << endl;
}
```

Ed Discussion

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

`main()`'s stack frame

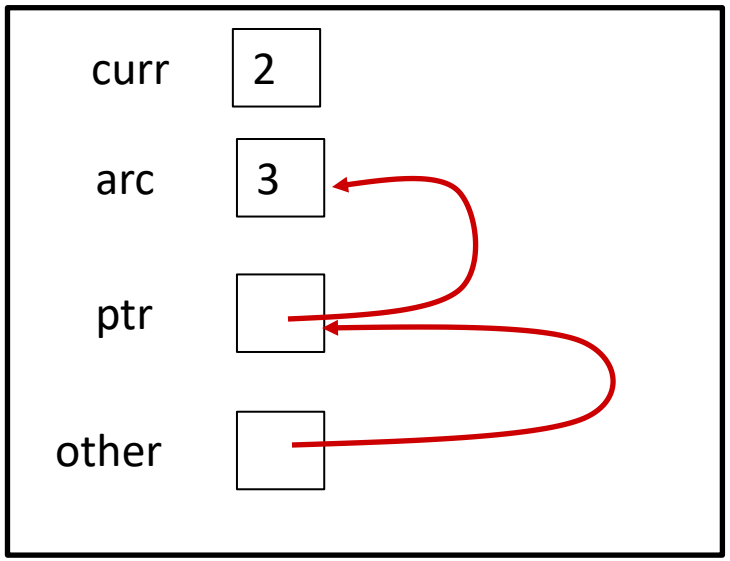



```
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int** other = &ptr;
  ptr = &arc;
  **other = curr;
  *other += 3;

  // print curr and arc
  cout << curr << endl;
  cout << arc << endl;
}
```

Ed Discussion

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

`main()`'s stack frame

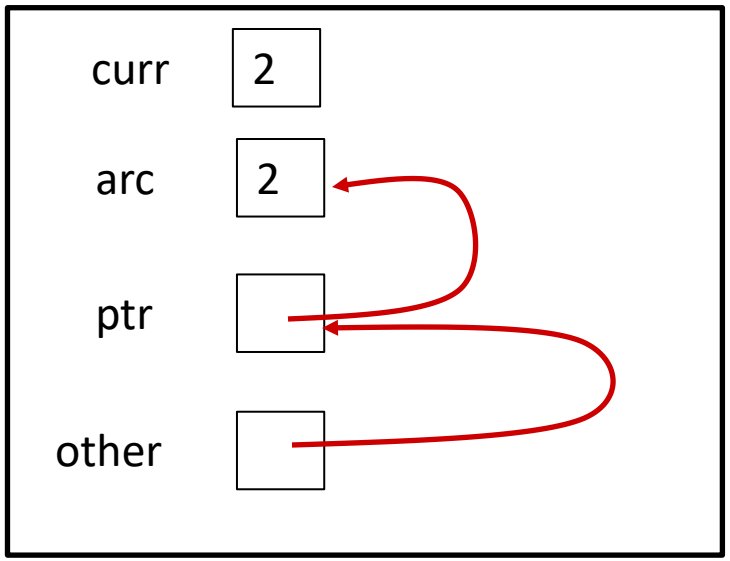



```
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int** other = &ptr;
  ptr = &arc;
  **other = curr;
  *other += 3;

  // print curr and arc
  cout << curr << endl;
  cout << arc << endl;
}
```

Ed Discussion

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

`main()`'s stack frame



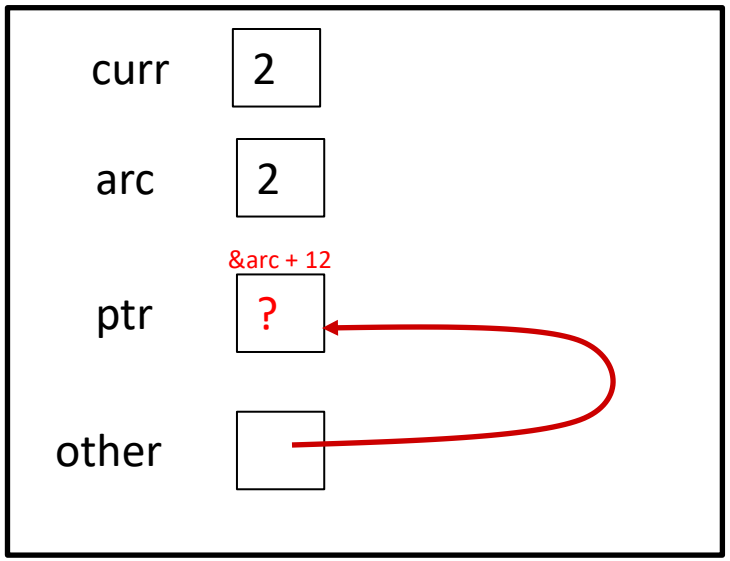

```
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int** other = &ptr;
  ptr = &arc;
  **other = curr;
  *other += 3;

  // print curr and arc
  cout << curr << endl;
  cout << arc << endl;
}
```

Ed Discussion

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

```
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int** other = &ptr;
  ptr = &arc;
  **other = curr;
  *other += 3;

  // print curr and arc
  cout << curr << endl;
  cout << arc << endl;
}
```

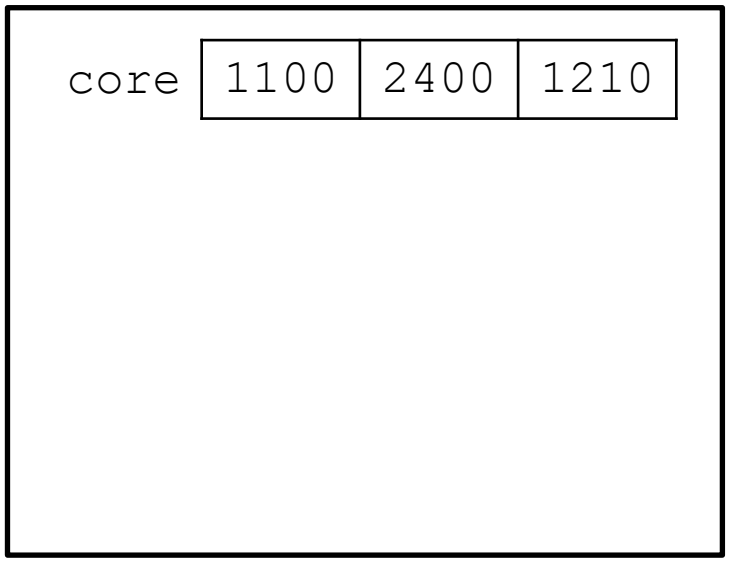
`main()`'s stack frame



Ed Discussion

- What is the final value of core after this code is run? Where is ptr pointing to after this code is run?
  - Hint: Draw it out!

`foo()`'s stack frame

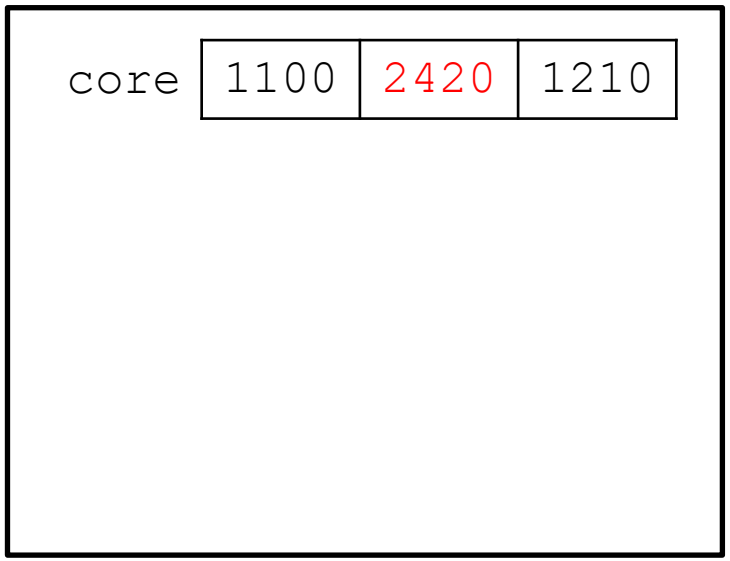



```
void foo() {
  int core[3] = {1100, 2400, 1210};

  core[1] += 20;

  int* ptr = &(core[1]);

  ptr[0] -= 900;

  ptr[1] = 5000;

  core[2] += 20;

  // STOP HERE
}
```

Ed Discussion

- What is the final value of core after this code is run? Where is ptr pointing to after this code is run?
  - Hint: Draw it out!

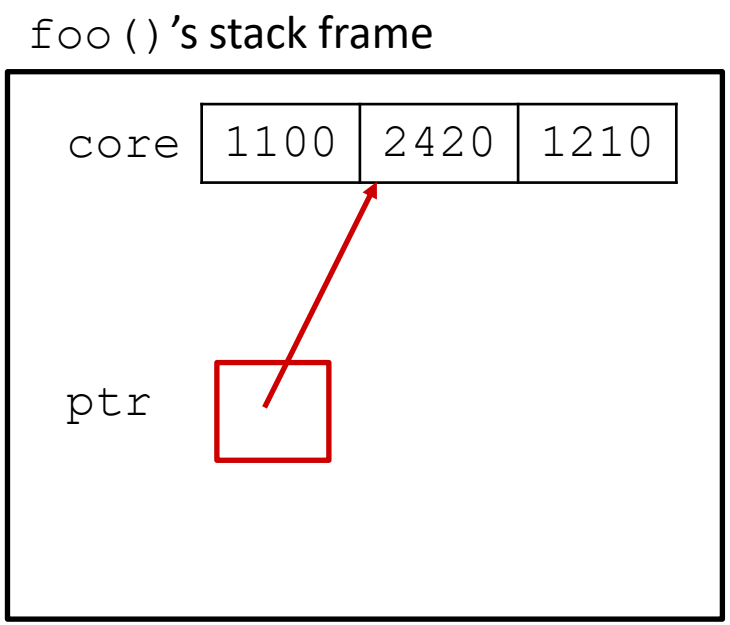
`foo()`'s stack frame




```
void foo() {
  int core[3] = {1100, 2400, 1210};

  core[1] += 20;

  int* ptr = &(core[1]);

  ptr[0] -= 900;

  ptr[1] = 5000;

  core[2] += 20;

  // STOP HERE
}
```

Ed Discussion

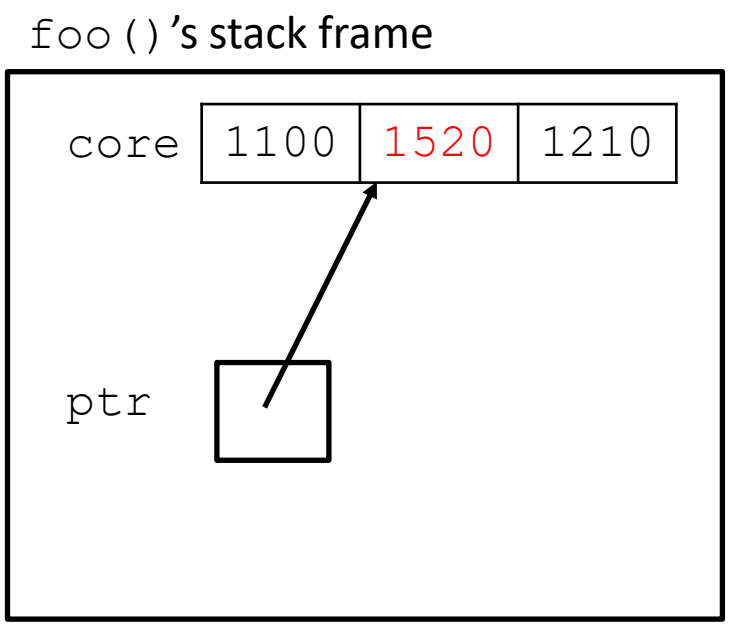
- What is the final value of core after this code is run? Where is ptr pointing to after this code is run?
  - Hint: Draw it out!




```
void foo() {
  int core[3] = {1100, 2400, 1210};

  core[1] += 20;

  int* ptr = &(core[1]);

  ptr[0] -= 900;

  ptr[1] = 5000;

  core[2] += 20;

  // STOP HERE
}
```

Ed Discussion

- What is the final value of core after this code is run? Where is ptr pointing to after this code is run?
  - Hint: Draw it out!

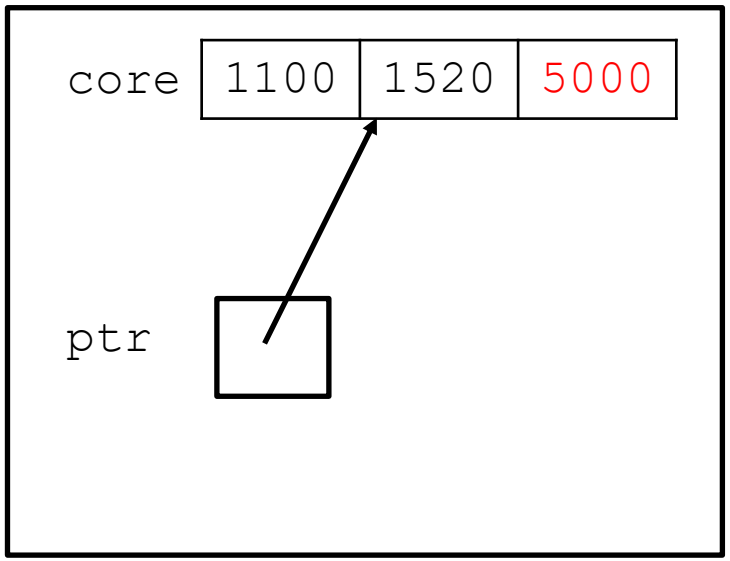



```
void foo() {
  int core[3] = {1100, 2400, 1210};

  core[1] += 20;

  int* ptr = &(core[1]);

  ptr[0] -= 900;

  ptr[1] = 5000;

  core[2] += 20;

  // STOP HERE
}
```

Ed Discussion

- What is the final value of core after this code is run? Where is ptr pointing to after this code is run?
  - Hint: Draw it out!

`foo()`'s stack frame

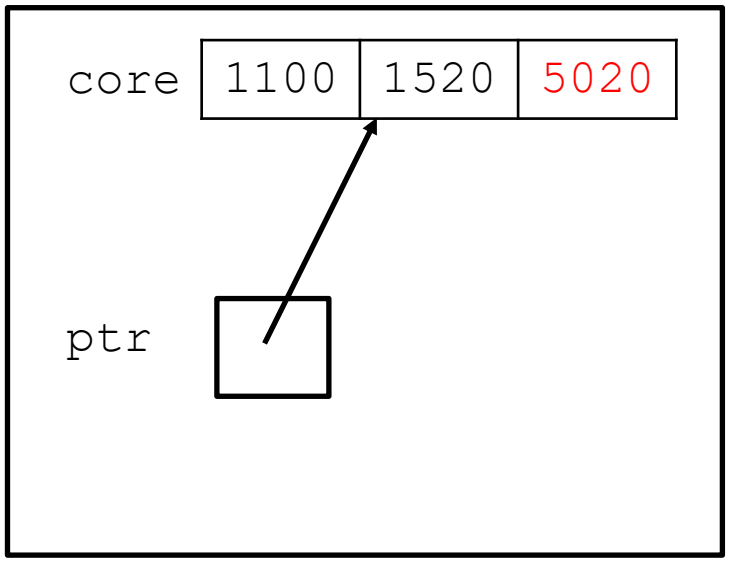



```
void foo() {
  int core[3] = {1100, 2400, 1210};

  core[1] += 20;

  int* ptr = &(core[1]);

  ptr[0] -= 900;

  ptr[1] = 5000;

  core[2] += 20;

  // STOP HERE
}
```

Ed Discussion

- What is the final value of core after this code is run? Where is ptr pointing to after this code is run?
  - Hint: Draw it out!

`foo()`'s stack frame



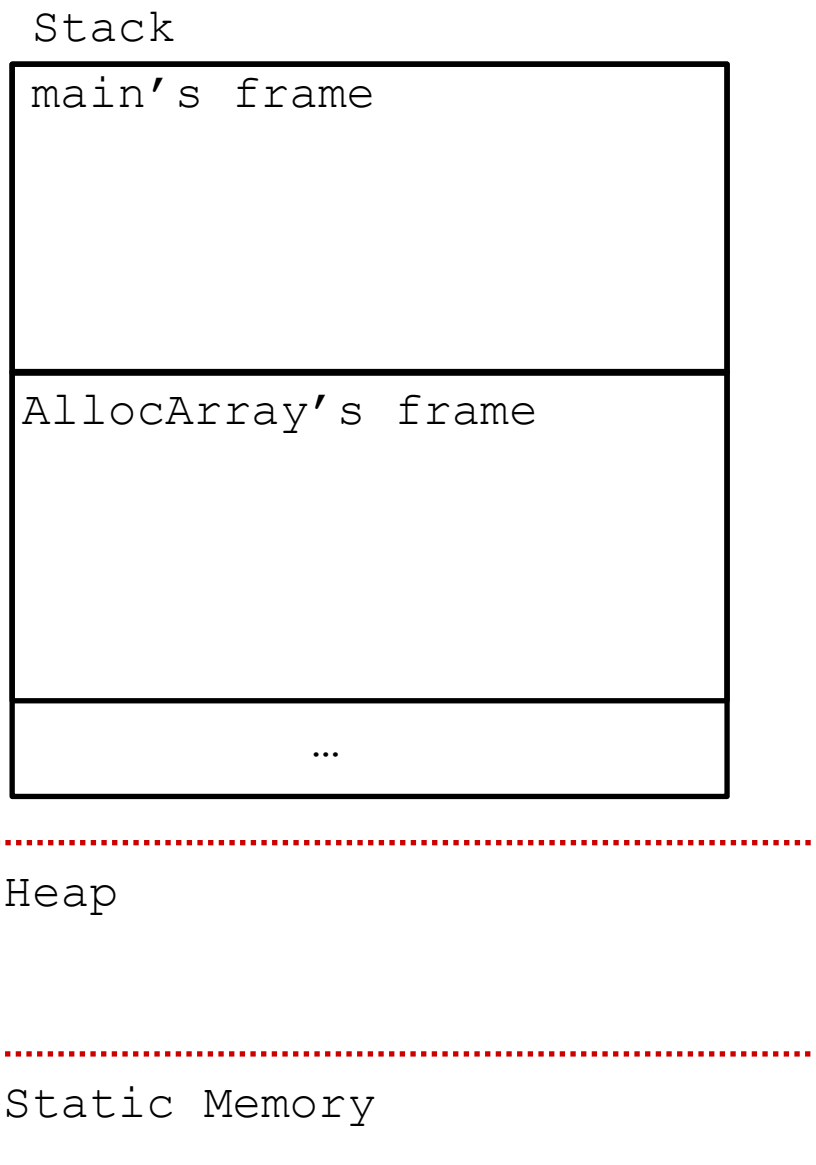

```
void foo() {
  int core[3] = {1100, 2400, 1210};

  core[1] += 20;

  int* ptr = &(core[1]);

  ptr[0] -= 900;

  ptr[1] = 5000;

  core[2] += 20;

  // STOP HERE
}
```

Ed Discussion

- Given this code, where are theses in memory? Assume the code is executing and is just about to finish the AllocArray function.
  - array
  - arr
  - my_len
  - num
  - arr[0]
  - main

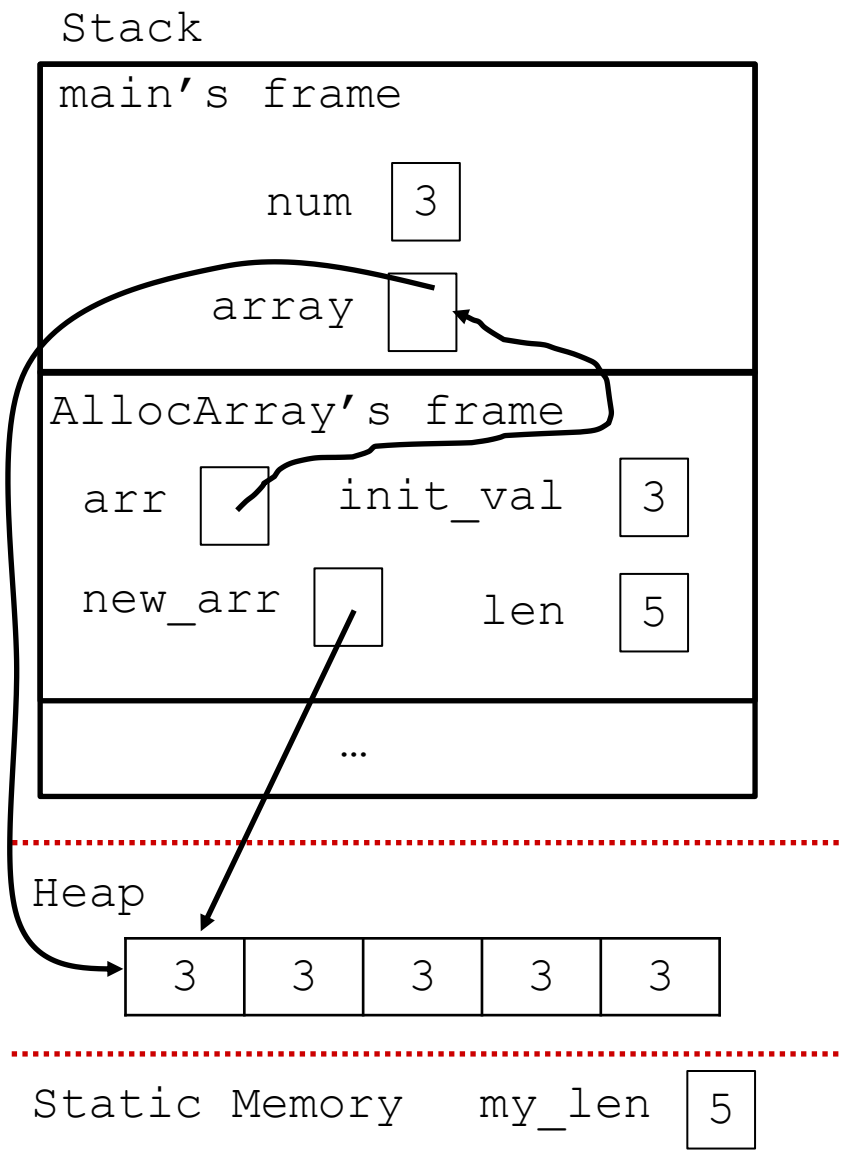
```
size_t my_len = 5;

// forward decl
void AllocArray(int** arr, size_t len, int init_val);

int main() {
  int num = 3;
  int* array = nullptr;
  AllocArray(&array, my_len, num);
}
void AllocArray(int** arr, size_t len, int init_val) {
  int* new_arr = new int[len];
  for (size_t i = 0; i < len; i++) {
    new_arr[i] = init_val;
  }
  *arr = new_arr;
  // ← WE ARE RIGHT HERE. ABOUT TO RETURN
}
```

Ed Discussion




```
size_t my_len = 5;

// forward decl
void AllocArray(int** arr, size_t len, int init_val);

int main() {
  int num = 3;
  int* array = nullptr;
  AllocArray(&array, my_len, num);
}
void AllocArray(int** arr, size_t len, int init_val) {
  int* new_arr = new int[len];
  for (size_t i = 0; i < len; i++) {
    new_arr[i] = init_val;
  }
  *arr = new_arr;
  // ← WE ARE RIGHT HERE. ABOUT TO RETURN
}
```

Ed Discussion




```
size_t my_len = 5;

// forward decl
void AllocArray(int** arr, size_t len, int init_val);

int main() {
  int num = 3;
  int* array = nullptr;
  AllocArray(&array, my_len, num);
}
void AllocArray(int** arr, size_t len, int init_val) {
  int* new_arr = new int[len];
  for (size_t i = 0; i < len; i++) {
    new_arr[i] = init_val;
  }
  *arr = new_arr;
  // ← WE ARE RIGHT HERE. ABOUT TO RETURN
}
```

Ed Discussion

- If we wanted to make sure everything was properly deallocated, how many calls to delete do we need?

  Where should we delete?

```
size_t my_len = 5;

// forward decl
void AllocArray(int** arr, size_t len, int init_val);

int main() {
  int num = 3;
  int* array = nullptr;
  AllocArray(&array, my_len, num);
}

void AllocArray(int** arr, size_t len, int init_val) {
  int* new_arr = new int[len];
  for (size_t i = 0; i < len; i++) {
    new_arr[i] = init_val;
  }
  *arr = new_arr;
}
```

Ed Discussion

- If we wanted to make sure everything was properly deallocated, how many calls to delete do we need?

  Where should we delete?

```
size_t my_len = 5;

// forward decl
void AllocArray(int** arr, size_t len, int init_val);

int main() {
  int num = 3;
  int* array = nullptr;
  AllocArray(&array, my_len, num);
  delete[] array;
}

void AllocArray(int** arr, size_t len, int init_val) {
  int* new_arr = new int[len];
  for (size_t i = 0; i < len; i++) {
    new_arr[i] = init_val;
  }
  *arr = new_arr;
```

Structured Data

- A `struct` is a C and C++ datatype that contains a set of fields
  - Similar to a Java class, but with no methods or constructors
  - Useful for defining new structured types of data
  - Acts similarly to primitive variables
- Generic declaration in C++:

```
struct Point {
  float x;
  float y;
};

Point pt;
Point origin = {0.0f, 0.0f};
pt = origin; // pt now contains 0.0f, 0.0f
```

Default values are still garbage!

<- Initializer List

Can be assigned into, used as parameters, etc.

Structured Data: copied not referenced

- A `struct` is a C and C++ datatype that contains a set of fields
  - Similar to a Java class, but with no methods or constructors
  - Useful for defining new structured types of data
  - Acts similarly to primitive variables
    - When we assign a struct, we copy the values in the struct

```
Point pt;
Point origin = {0.0f, 0.0f};
pt = origin; // pt now contains 0.0f, 0.0f

origin.first = 1.0f;

print(origin.first);
print(pt.first);
```

Accessing struct Fields

- Use "." to refer to a field in a struct
- Use "->" to refer to a field from a struct pointer
  - Dereferences pointer first, then accesses field

```
struct Point {
  float x, y;
};

int main(int argc, char** argv) {
  Point p1 = {0.0, 0.0};
  Point* p1_ptr = &p1;

  p1.x = 1.0;
  p1_ptr->y = 2.0;  // equivalent to (*p1_ptr).y = 2.0;
  return 0;
}
```

Const

- `const` is a keyword in C and C++ that means that a variable cannot be modified. It is "constant"

- If a struct is `const` in C or C++, then its members are also `const`.

```
int main() {
  const int x = 3;
  int y = 5;

  x += 1;  // ILLEGAL

  y += 1;

  const Point p = {0.0, 0.0};

  p.first = 1.0; // ILLEGAL
}
```

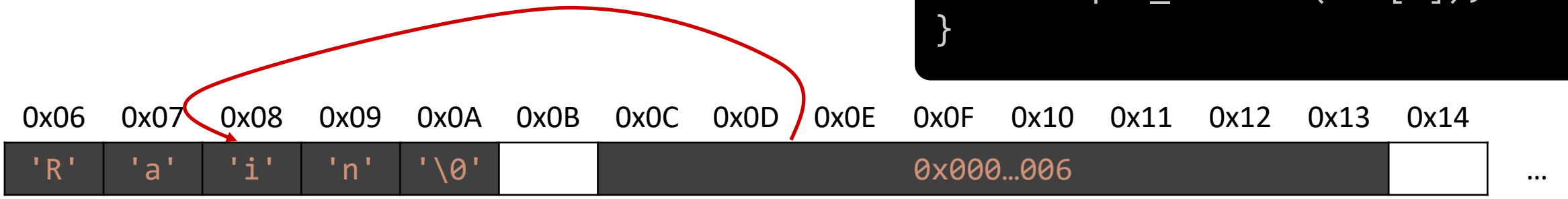
Strings in C

- Strings in C are just arrays of characters with a special character at the end to mark the end of the string: `'\0'`
  - Called the "null terminator" character

- C-strings are often referred to with a **`char[]`** or a **`char*`**

- Example:
  - `print(str)  // Rain`
  - `print(ptr_str)  // in`

```
int main() {
  char c = '\0';

  char str[5] = "Rain";

  char* ptr_str = &(str[2]);
}
```



Ed Discussion

- Finish writing this code:

- Strdup is a function that takes in a C string, makes a copy of it, and returns it

```
// takes in a pointer to constant chars
// (we cannot manipulate the chars but could manipulate the pointers)
char* strdup(const char* string) {




}
```

Ed Discussion

- Finish writing the CopyList function.

- It takes a pointer to the first node in a linked list. Each node has a pointer to the next node and an integer value.

- The function makes an independent copy of the list passed in and returns the copy.

```
struct node {
  node* next;
  int val;
};

node* CopyList(node* head) {




}
```

Ed Discussion

- Finish writing the PushFront function.
- It takes a pointer to the head pointer of a linked list. Each node has a pointer to the next node and an int.
- The function pushes a new element on to the front of the list.
- Note the node** parameter
- Follow up question why is it important that it is a node** for head?

```
struct node {
  node* next;
  int val;
};

void PushFront(node** head, int to_add) {



}
```

That's all for now!

- If we got through all this, you should have everything you need for the first homework assignment from this lecture and recitation
- We are going a little fast because I expect you have already seen all or most of this
- When we get to new material it won't be as fast
- Releasing tonight or tomorrow:
  - HW00
  - Pre-semester Survey