Designing C Modules

CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:

Dao Yi	Farrell Fileas	Lukas Joswiak
Nathan Lipiarski	Renshu Gu	Travis McGaha
Yibo Cao	Yifan Bai	Yifan Xu

Poll Everywhere

pollev.com/cse333

About how long did Exercise 3 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I prefer not to say

Administrivia

- Exercise 4: Released today, due Monday

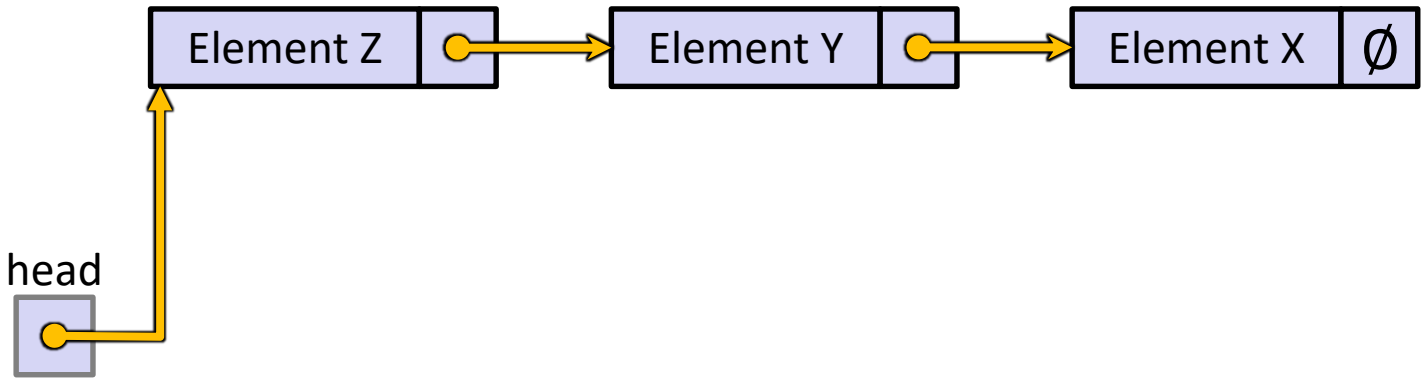
- Homework 1 due in less than a week
  - You should be 😱 😱 😱 if you haven't checked your repo for a missing HW1 yet!
  - Advice: be *sure* to read headers carefully while implementing
  - Advice: use git add/commit/push often to save your work

Lecture Outline

- **Generic Data Structures in C**
- Structuring Interfaces
  - C Preprocessor Intro
- Choosing Your Integer Type: `int8_t` vs `uint64_t` vs …

Simple Linked List in C

- Each node in a linear, singly-linked list contains:
  - Some element as its payload
  - A pointer to the next node in the linked list
    - This pointer is NULL (or some other indicator) in the last node in the list



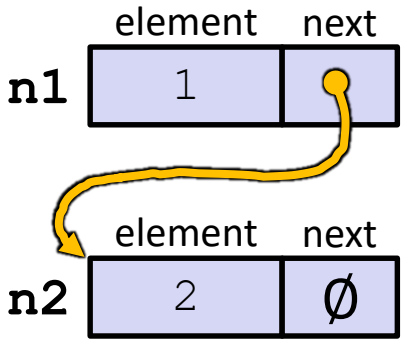
Linked List (Attempt #1)

- Let's represent a linked list node with a struct
  - Assume each element is an int32_t

```
typedef struct node_st {
  int32_t element;
  struct node_st *next;
} Node;

Node* Push(Node *head,
    int32_t elt) {
  Node *n =
    (Node*) malloc(sizeof(Node));
  assert(n != NULL
  n->element = elt;
  n->next = head;
  return n;
}
```

```
int main(int argc, char **argv) {
  Node n1, n2;

  n1.element = 1;
  n1.next = &n2;
  n2.element = 2;
  n2.next = NULL;
  return EXIT_SUCCESS;
}
```

manual_list.c

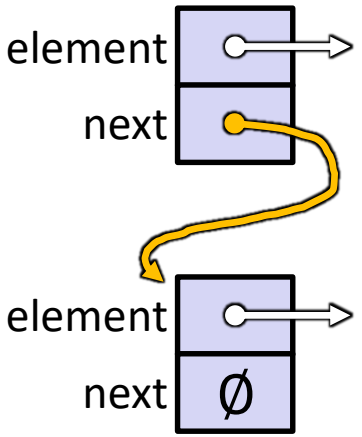


Linked List (Attempt #2)

- Let's generalize the linked list element type
  - Let customer decide type (instead of always `int32_t`)
  - Idea: let them use a generic pointer (*i.e.* a `void*`)

```
typedef struct node_st {
  void *element;
  struct node_st *next;
} Node;

Node* Push(Node *head, void *e) {
  Node *n = (Node*) malloc(sizeof(Node));
  assert(n != NULL);  // crashes if false
  n->element = e;
  n->next = head;
  return n;
}
```

manual_list_void.c



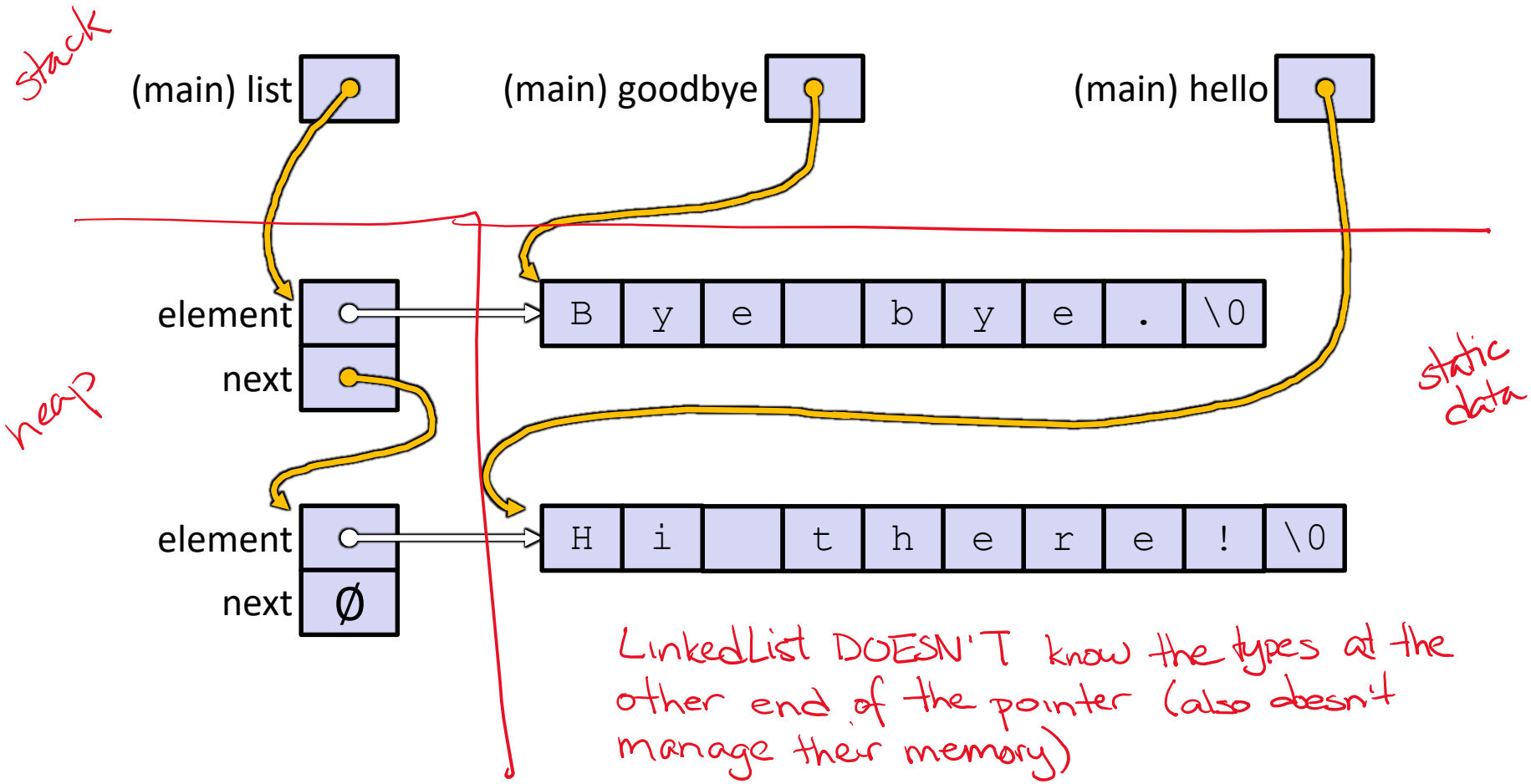
Using a Generic Linked List

- Type casting needed to deal with `void*` (raw address)
  - Before pushing, should convert to `void*`
  - Must convert back to data type when accessing

```
typedef struct node_st {
  void *element;
  struct node_st *next;
} Node;

Node* Push(Node *head, void *e);   // assume last slide's code

int main(int argc, char **argv) {
  char *hello = "Hi there!";
  char *goodbye = "Bye bye.";
  Node *list = NULL;

  list = Push(list, (void*) hello);
  list = Push(list, (void*) goodbye);
  printf("payload: '%s'\n", (char*) ((list->next)->element) );
  return EXIT_SUCCESS;
}
```

manual_list_void.c

Resulting Memory Diagram



Something's Fishy …  …

- A (benign) memory leak!

```
int main(int argc, char **argv) {
  Node * list = NULL;
  list = Push(list, 1);
  list = Push(list, 2);
  return EXIT_SUCCESS;
}
```

push_list.c

```
bash$ gcc -Wall -g -o push_list push_list.c

bash$ valgrind --leak-check=full ./push_list
```

Lecture Outline

- Generic Data Structures in C
- **Structuring Interfaces**
  - **C Preprocessor Intro**
- Choosing Your Integer Type: `int8_t` vs `uint64_t` vs …

Multi-File C Programs

- Let's create a linked list *module*
  - A module is a self-contained piece of an overall program
    - Has *externally visible* functions that customers can invoke
    - Has *externally visible* `typedef`s, constants, and perhaps global variables, that customers can use
    - May have *internal functions*, `typedef`s, or global variables that customers should *not* look at
  - The module's *interface* is its set of public functions, `typedef`s, and global variables

C Header Files

- Header: a file whose only purpose is to be `#include`'d
  - Generally has a filename `.h` extension
  - Holds the variables, types, and function prototype declarations that make up the interface to a module
  - Can have <system-defined headers> or "programmer-defined"

- Main Idea:
  - Every `name.c` is intended to be a module that has a `name.h`
  - `name.h` declares the interface to that module
  - Other modules can use `name` by `#include`-ing `name.h`
    - They should assume as little as possible about the implementation in `name.c`

C Module Conventions (1 of 3)

- `.h` files only contain *declarations*, never *definitions*
- `.c` files never contain prototype declarations for functions that are intended to be exported through the module interface
  - Those function prototype declarations belong in the `.h` file
- Public-facing functions are `ModuleName_functionname()`, take a pointer to "`this`" as first argument
  - i.e. the instance we want to operate on
- How do we keep the declaration and definitions in sync?

`#include` and the C Preprocessor

- The C preprocessor (`cpp`) transforms your source code before the compiler runs
  - Input is a C file (text) and output is still a C file (text)
  - Processes the directives it finds in your code (`#directive`)
    - *e.g.* `#include "ll.h"` is replaced by the post-processed content of `ll.h`
    - *e.g.* `#define PI 3.1415` defines text to be replaced later
    - Several others that we'll see soon…
  - Run on your behalf by `gcc` during compilation
- `cpp` is a *sequential* and *stateful* search-and-replace text-processer!

C Module Conventions (2 of 3)

- ***NEVER*** `#include` a `.c` file; only `#include` `.h` files
- `#include` all of headers you reference, even if another header (transitively) includes some of them
- Any `.c` file with an associated `.h` file should be able to be compiled into a `.o` file
  - The `.c` file should `#include` the `.h` file; the compiler will check definitions and declarations for consistency

C Module Conventions (3 of 3)

- If a function is declared in a header file (`.h`) and defined in a C file (`.c`):
  - *The header needs full documentation because it is the public specification*
  - No need to copy/paste the comment into the C file
    - Two copies that can get out of sync
- If the prototype & implementation are in same C file:
  - One school of thought: Full comment on the prototype, no comment (or "declared above") on code
    - 333 project code is like this
  - Another school: Prototype is for the compiler and doesn't need comment; comment the code to keep them together

Poll Everywhere

pollev.com/cse333

What will be the preprocessor's output?

A. long long int z = 1 + 2 + 1
B. long long int z = 1 + 2 + FOO
C. verylong z = 1 + 2 + 1
D. verylong z = 1 + 2 + FOO
E. I'm not sure …

```
#define BAR 2 + FOO

typedef long long int verylong;
```

cpp_example.h

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char **argv) {
  int x = FOO;    // a comment
  int y = BAR;
  verylong z = FOO + BAR;
  return EXIT_SUCCESS;
}
```

cpp_example.c

Compare against <stdio.h>

C Preprocessor Example

Preprocessor State

FOO	1
BAR	2+1

- We can manually run the preprocessor:
  - `cpp` is the preprocessor (can also use `gcc -E`)
  - "`-P`" option suppresses some extra debugging annotations

```
#define BAR 2 + FOO 1

typedef long long int verylong;
```

cpp_example.h

copy-n-pasted

```
#define FOO 1

#include "cpp_example.h"

int main(int argc, char **argv) {
  int x = FOO; 1 // a comment
  int y = BAR; 2+1
  verylong z = FOO + BAR; 1+2+1
  return EXIT_SUCCESS;
}
```

cpp_example.c

```
bash$ cpp -P cpp_example.c out.c
bash$ cat out.c

typedef long long int verylong;
int main(int argc, char **argv) {
  int x = 1;
  int y = 2 + 1;
  verylong z = 1 + 2 + 1;
  return 0;
}
```

Program Using a Linked List

```c
#include <stdlib.h>
#include <assert.h>
#include "ll.h"

Node* Push(Node *head,
           void *element) {
  ... // implementation here
}
```

ll.c

```c
typedef struct node_st {
  void *element;
  struct node_st *next;
} Node;

Node* Push(Node *head,
           void *element);
```

ll.h

```c
#include "ll.h"

int main(int argc, char **argv) {
  Node *list = NULL;
  char *hi = "hello";
  char *bye = "goodbye";

  list = Push(list, (void*)hi);
  list = Push(list, (void*)bye);

  ...

  return 0;
}
```

example_ll_customer.c

Compiling the Whole Program

- Four parts:
  - 1) Compile `example_ll_customer.c` into an object file
  - 1) Compile `ll.c` into an object file
  - 2) Link both object files into an executable
  - 3) Test, Debug, Rinse, Repeat

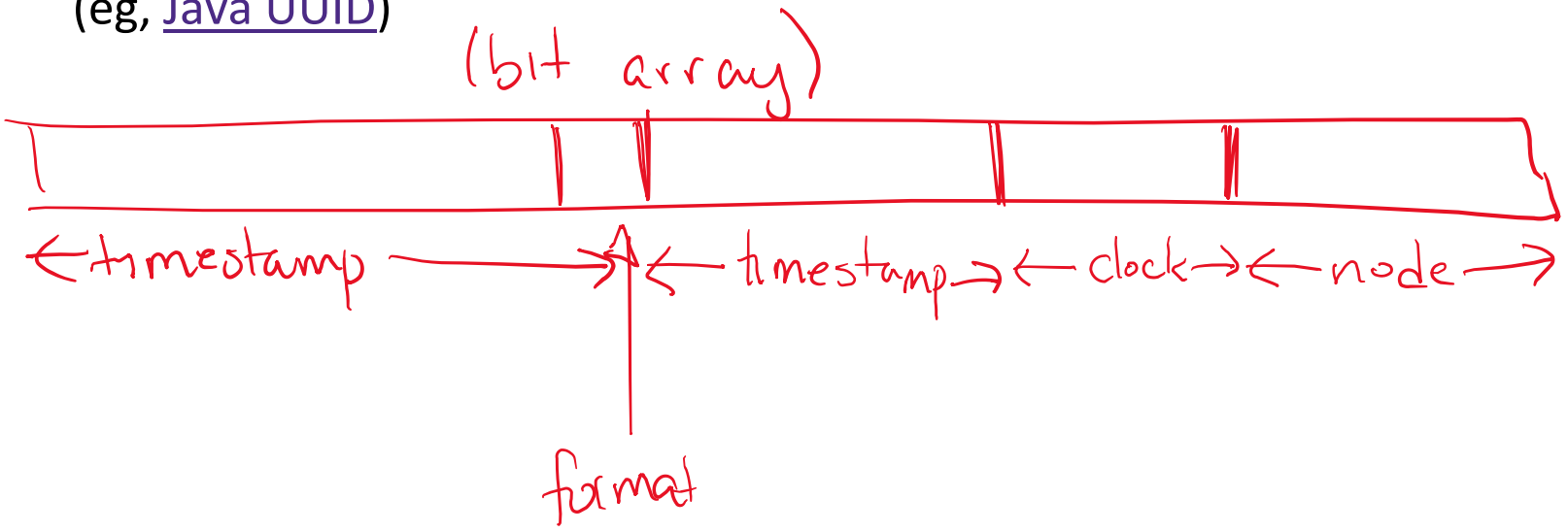
```
bash$ gcc -Wall -g -c -o example_ll_customer.o example_ll_customer.c
bash$ gcc -Wall -g -c -o ll.o ll.c
bash$ gcc -g -o example_ll_customer ll.o example_ll_customer.o
bash$ ./example_ll_customer
Payload: 'yo!'
Payload: 'goodbye'
Payload: 'hello'
bash$ valgrind -leak-check=full ./example_ll_customer
... etc ...
```

Lecture Outline

- Generic Data Structures in C
- Structuring Interfaces
  - C Preprocessor Intro
- **Choosing Your Integer Type: `int8_t` vs `uint64_t` vs …**

What is the Use Case?

- Counters vs IDs
  - IDs uniquely identify some entity, and can be implemented as a counter
    - But IDs can also be hashes, random numbers, or some combination (eg, Java UUID)



How Big?

- $2^8 = 256$
- $2^{16} = 65{,}536 = 64k$
- $2^{32} \approx 4B$
  - ≈ number of people on the internet, as of Jul 2019
  - ≈ 156 years, counted in seconds*
- $2^{64} \approx 1.8 \times 10^{19}$
  - ≈ number of atoms in the universe
  - ≈ 584 years, counted in nanoseconds (billionths)

- $2^{128} \approx 3.4 \times 10^{38}$
- $2^{256} \approx 3.4 \times 10^{77}$

`signed` vs `unsigned`

- unsigned integers are defined differently from signed integers
  - Comparisons and conversions have nasty edge cases
  - Dangerous enough that using "unsigned" to express "this will never be negative" is discouraged by Google C++ Style Guide

- Unsigned's best use: raw bit patterns

Counting: `int` vs defined-size `int`s

- Most modern architectures are 32-bit or larger:
  - You can reasonably assume `sizeof(int)` >= 32-bits
  - 2^{32} is a reasonable upper bound for "number of items held in memory"

- Google C++ Style Guide: "*We use int very often, for integers we know are not going to be too big, e.g. loop counters*"

What If It's Not In Memory?

- If it's read by another program (even us! In a different invocation)
  - Eg, write to network, write to disk, …

- If the thing we're counting has constraints
  - Eg, counter for the number of people in the world
  - Constraints can cut both ways! Eg, representing a human's age

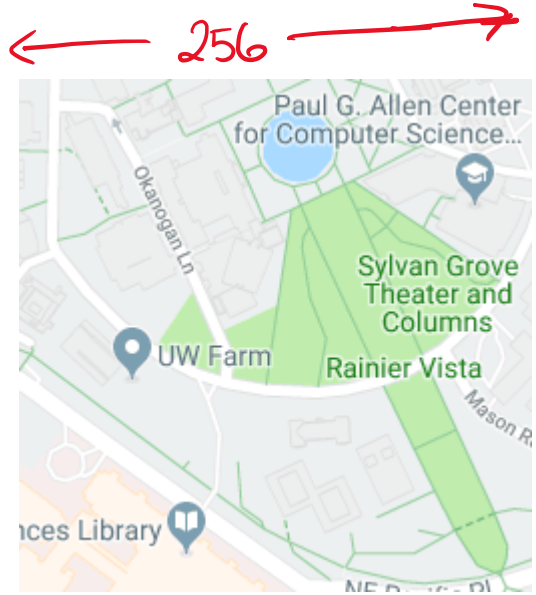
Case Studies (1 of 3)

- Rendering instructions for a 256 x 256 px "map tile"
  - … with quarter-pixel resolution

```
message Vertex {
  required int?? x;
  required int?? y;
}
```

how big?

probbuf



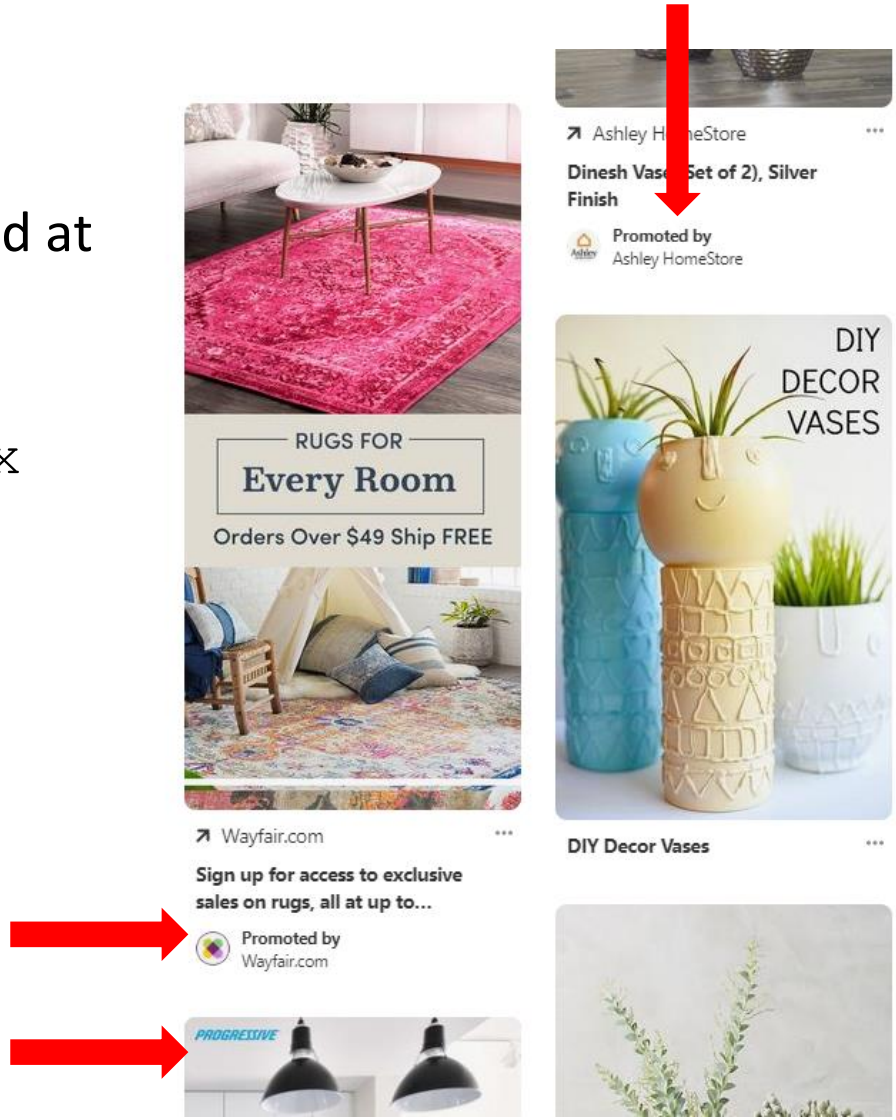
- Note: int16_t and int8_t are very specialized

Case Studies (2 of 3)

- Crashdump IDs for an internet-sized company
  - Every time a program on your machine crashes, generate a dump for that program and upload to a server (which will then generate an ID)

Case Studies (3 of 3)

- Ad "impressions"
  - When a user views a specific ad at a specific time
  - Need to be big enough for `num_users x num_ads x timestamp`



Poll Everywhere

pollev.com/cse333

What Types Should We Use?

map coordinate | crash dump id | impression id

A. int16_t / int64_t / int64_t
B. int32_t / int64_t / int64_t
C. int16_t / uint64_t / uint64_t
D. int32_t / uint32_t / uint64_t
E. I'm not sure …

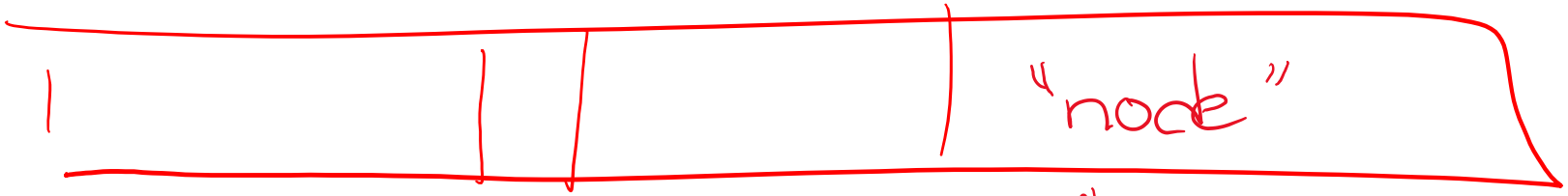
Depends on how we generate the ids!

- counter: int64_t
- hash: uint64_t

Case Studies (3 of 3 continued …)

- When is 64-bits *not enough*?!?!
  - If you add structure to your identifier!
  - $2^{48} \approx 281T$

Recall Java UUID (64-bits):



How do you fit an IPv6 address in here?

Extra Exercise #1

- Extend the linked list program we covered in class:
  - Add a function that returns the number of elements in a list
  - Implement a program that builds a list of lists
    - *i.e.* it builds a linked list where each element is a (different) linked list
  - Bonus: design and implement a "Pop" function
    - Removes an element from the head of the list
    - Make sure your linked list code, and customers' code that uses it, contains no memory leaks

Extra Exercise #2

- Implement and test a binary search tree
  - https://en.wikipedia.org/wiki/Binary_search_tree
    - Don't worry about making it balanced
  - Implement key insert() and lookup() functions
    - Bonus: implement a key delete() function
  - Implement it as a C module
    - `bst.c`, `bst.h`
  - Implement `test_bst.c`
    - Contains main() and tests out your BST

Extra Exercise #3

- Implement a Complex number module
  - `complex.c`, `complex.h`
  - Includes a typedef to define a complex number
    - a + bi, where a and b are `doubles`
  - Includes functions to:
    - add, subtract, multiply, and divide complex numbers
  - Implement a test driver in `test_complex.c`
    - Contains `main()`