CSE 333: Systems Programming

Section 1
Introduction, structs, arrays
My name is Elliott, and I’m a fifth-year masters student

I enjoy operating systems, distributed systems, and programming in C++

I interned twice at Google with the Dremel team, which develops a SQL server for querying large data sets and returning results real time
About your TA

Office hours:
- Monday 12:30 to 1:20 in CSE 002
- Wednesday 12:30 to 1:20 in CSE 216
- Whenever I’m in 002 (fairly often during the week)

In general, seek help through the GoPost before sending email—other students likely have the same question
Section format

- Some lecture material/discussion of projects
- Lab exercise
  - A short coding exercise related to class material
  - Must compile without warnings and pass valgrind without memory leaks
  - Work with a partner if you like
  - Show a TA your solution to receive credit for it
Section format

Section question

- Come up with an answer to *one* of the questions listed at the end of this slide deck
- Tell the TAs your answer when you show them your lab exercise solution
Section format

- 3 points possible per section
  - 1 for attending section
  - 1 for having a lab exercise solution without compiler warnings or memory leaks
  - 1 for answering one of the section questions

- If you miss a section, you can email Chuong and me your code along with answers to *all* of the section questions to receive 2/3 points
Ex0/hw0

- Success?

- Some suggestions for exercises
  - "Good style" for this class is based on the Google style guide, so follow it when in doubt
  - Keep it short and simple—dense code with a few comments sprinkled in

- Expect exercise grades/feedback prior to the next lecture after turning them in
**Structs**

- Used for encapsulating data
- Can contain primitive types (int, double, etc.), arrays, other structs, and unions, among other types
- Accesses are made through the ‘\(->\)’ operator for pointers to structs and ‘.’ for values
- More on this later; just need basics for the lab exercise
**Example:**

typedef struct {
   int a, b;
} sample;

int main(int argc, char* argv[]) {
   sample s;
   s.a = 10;
   s.b = 5;
   sample* s_ptr = &s;
   printf("s.a is %d and s.b is %d\n", s.a, s.b);
   printf("s_ptr->a is %d and s_ptr->b is %d\n", s.a, s.b);
   return 0;
}
Arrays

* Just a block of data of a particular type and size
* Raw pointers can be treated as arrays and vice versa, with some minor caveats

```c
int* a = (int*) malloc(sizeof(int) * 3);
int* b = (int*) malloc(sizeof(int));
int c[5] = {0}; // stack-allocated array
a[2] = 6;
b[0] = 4;
c[2] = 2;
*a = c[2]; // what does this do?
free(a);
free(b);
```
Lab exercise!

- Play around with arrays and get a brief introduction to structs
- Create a way to access arrays “safely” through bounds-checking
- Clone the section repository to get the skeleton code (pull up this slide deck on your laptop to copy/paste instead)

```
git clone ssh://[username]@attu.cs.washington.edu/projects/instr/12au/cse333/section/central.git
```
Lab exercise questions

- The code for the SafeArray implementation passes the SafeArray struct by value. What are the benefits of passing SafeArray by value (if any)? What are the drawbacks (if any)?

- What are the performance implications of using these functions for safely accessing arrays? Why does Java, for example, perform bounds-checking on arrays while C does not?