Slides by Andrew DeOrio

EECS 280

Programming and Introductory Data Structures

Midterm Exam Review

Exam time and location

- Wednesday, October 14th from 7pm to 8:30
  - *Michigan time (80 min exam)*
- Location – see email

Schedule

- Wednesday (day of exam) and Thursday (day after exam)
  - No lecture
- No lab this week
- All office hours following the exam are canceled

Policies

- Closed book
- Closed notes
- One "cheat sheet"
  - 8.5"x11", double-sided, hand-written, with your name on it
- No calculators or electronics
  - None needed
- Given under the engineering honor code

Engineering honor code

- Exams in the CoE are given under the honor code, which hold that students are honorable and trustworthy people
- No proctor
- Staff available outside for questions

- You must sign the honor pledge on the exam
  "I have neither given nor received unauthorized aid on this examination, nor have I concealed any violations of the Honor Code."

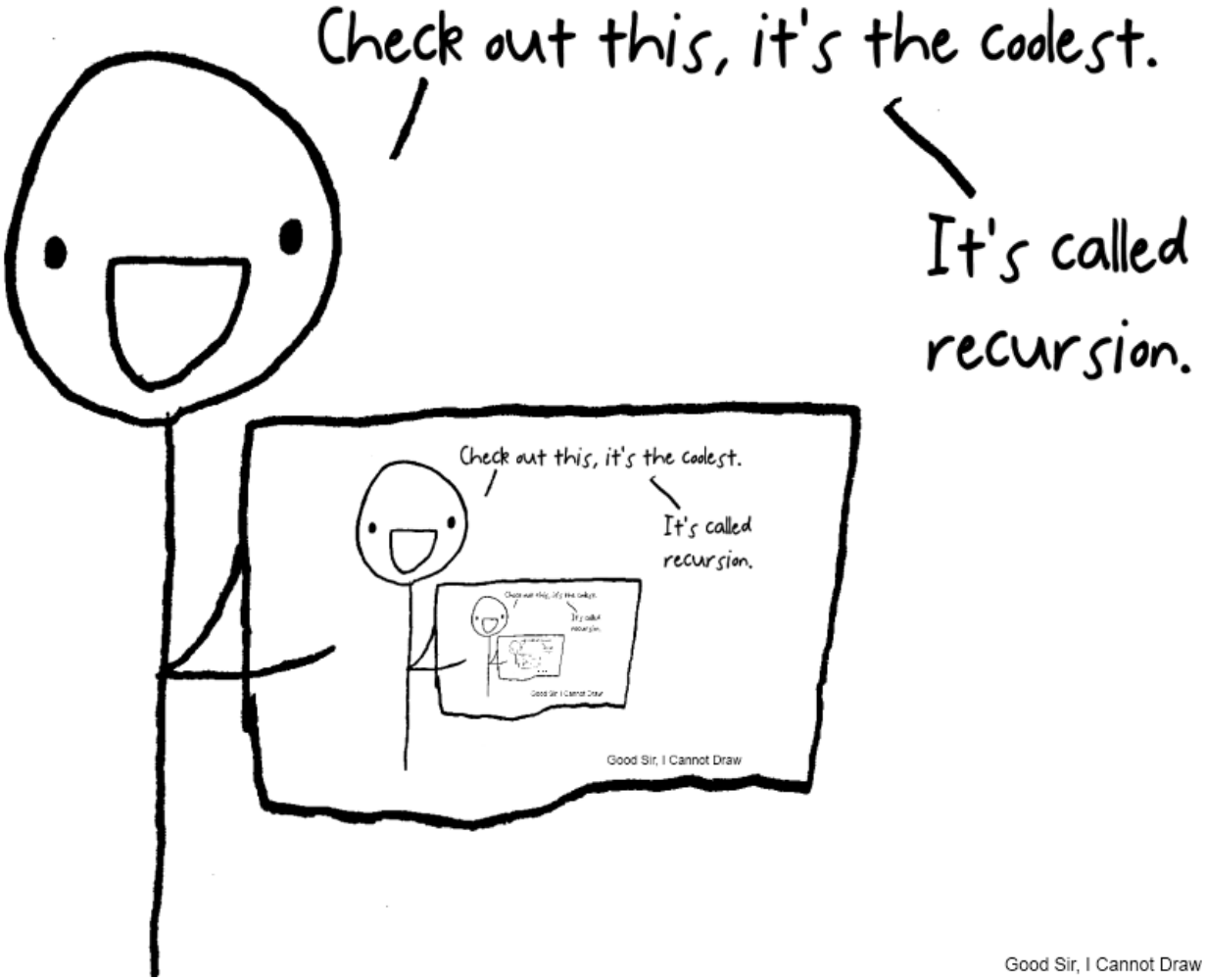
Study materials

- Practice exams posted on CTools / Google Drive
- Labs
  - Including optional exercises
- Lecture slides
  - Exercises from lecture
- Text book
- Study groups

Topics

- Everything we have covered up to and including Abstract Data Types
- Topics you should definitely study:
  - Recursion and tail recursion
    - Lists and trees (project 2)
  - Function pointers
  - Pointers and arrays
  - Strings and C-Strings
  - Structs

Recursion



Good Sir, I Cannot Draw

Group Exercise: pow(x, y)

- Write a tail-recursive version of the power function that returns x to the power of y. It needs a helper function - why?
- Here is a non-tail-recursive, but recursive version of this function:

```
int power(int x, int y) {
  if (y == 0) {
    return 1;
  } else {
    return x * power(x, y-1);
  }
}
```

- Before you begin, why isn't this tail recursive?

Group Exercise: pow(x, y)

```
int power_tail(int x, int y, int result) {
  if (y == 0) {
    return result;
  } else {
    return power_tail(x, y-1, result*x);
  }
}

int power(int a, int b) {
  return power_tail(a, b, 1);
}
```

Group Exercise: pow(x, y)

- How was this conversion made?

1. Notice that a multiply is needed before making the original recursive call to `power`.
2. But, a place is needed to store the result of that multiply. `x` and `y` can't because they are already doing useful things. So, an extra argument, `result`, must be added.
3. Since an extra argument is added, the type signature of power must change. This means that a helper function must be created to do the actual tail recursion (`power_tail`).
4. Finally, choose an initial value for `result`. 1 works because it was the value returned by the base case in the original recursive solution. Similarly, rather than return 1 in the new base case, `result` is returned. In a sense, the direction of the multiplication has been "reversed".

Function Pointers

- I couldn’t find anything funny about function pointers on the internet.

Exercise

```
bool all_of(list_t list, bool(*fn)(int)) {
  //EFFECTS: returns true if fn returns true for all

  if (list_isEmpty(list)) //base case
    return true;
  if (!fn(list_first(list))) //check current item
    return false;
  return all_of(list_rest(list), fn); //recurse
}
```

- Write these two functions. Use `all_of()` and helper functions

```
bool all_even(list_t list);
bool all_odd(list_t list);
```

Exercise

```
bool is_even(int i) {
  //EFFECTS: returns true if i is even
  return (i % 2) == 0;
}

bool all_even(list_t list) {
  //EFFECTS: returns true if all elements are even
  return all_of(list, is_even);
}
```

Exercise

```
bool is_odd(int i) {
  //EFFECTS: returns true if i is odd
  return (i % 2) == 1;
}

bool all_odd(list_t list) {
  //EFFECTS: returns true if all elements are odd
  return all_of(list, is_odd);
}
```

Arrays and pointers

Pointer Exercise: Code these

```
//REQUIRES: "a" points to an array of length "size"
//EFFECTS:  Returns a pointer to the first
//  occurrence of "search" in "a".
//  Returns NULL if not found.
int * find (int *a, unsigned int size, int search);


//REQUIRES: "s" is a NULL-terminated C-string
//EFFECTS:  Returns a pointer to the first
//  occurrence of "search" in "s".
//  Returns NULL if not found.
char * strchr (char *s, char search);
```

Do not use array indexing, e.g., `a[i]` or `*(a+i)`

Pointer Exercise: Code these

```
//REQUIRES: "a" points to an array of length "size"
//EFFECTS:  Returns a pointer to the first
//  occurrence of "search" in "a".
//  Returns NULL if not found.
int * find (int *a, unsigned int size, int search) {
 for (int *i=a; i<a+size; ++i) {
    if (*i == search) return i;  //found
  }
  return NULL; //not found
}
```

Pointer Exercise: Code these

```
//REQUIRES: "s" is a NULL-terminated C-string
//EFFECTS:  Returns a pointer to the first
//  occurrence of "search" in "s".
//  Returns NULL if not found.
char * strchr (char *s, char search) {
 while (*s) {
    if (*s == search) return s; //found
    ++s;
  }
  return NULL; //not found
}
```

Strings


ARRGH! MY MAP OF LISTS OF MAPS TO STRINGS IS TOO HARD TO ITERATE THROUGH! I'LL JUST ASSIGN EVERYTHING A NUMBER AND USE A *!#*!@ ARRAY

C strings vs. C++ strings

C++ string

```
#include <string>
const string hello =
"hello";
hello.length();
string s;

s = hello; //copy
if (a == b)
 // do something
```

C string

```
/* Write the C string
version here */
```

C strings vs. C++ strings

C++ string

```
#include <string>
const string hello =
"hello";
hello.length();
string s;

s = hello; //copy
if (a == b)
 // do something
```

C string

```
#include <cstring>
const char* hello =
"hello";
strlen(hello);
const int MAXSIZE=1024;
char s[MAXSIZE];
strcpy(s, hello);
if (strcmp(a,b) == 0)
  // do something
```

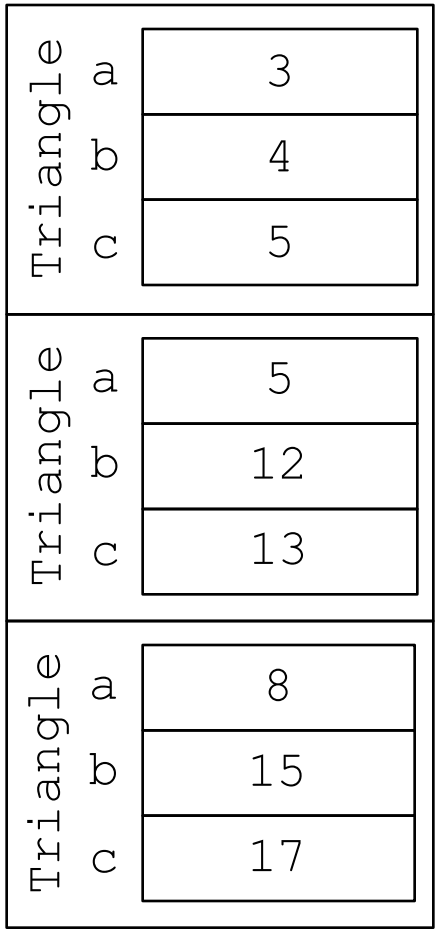
Compound Types

Exercise: arrays of structs

- Call `Triangle_area()` on each `Triangle` in the array using *traversal by pointer*

```
double Triangle_area(const Triangle *t);


const int SIZE = 3;
Triangle triangles[SIZE];
// initialization code ...
```



Exercise: arrays of structs

```
const int SIZE = 3;
Triangle triangles[SIZE];
//initialize triangles...
```

++t moves the pointer to the next struct

```
for (Triangle *t=triangles; t<triangles+SIZE; ++t)
  cout << "area = " << Triangle_area(t) << endl;
```

```
$ ./a.out
area = 6
area = 30
area = 60
```