EXERCISE #4

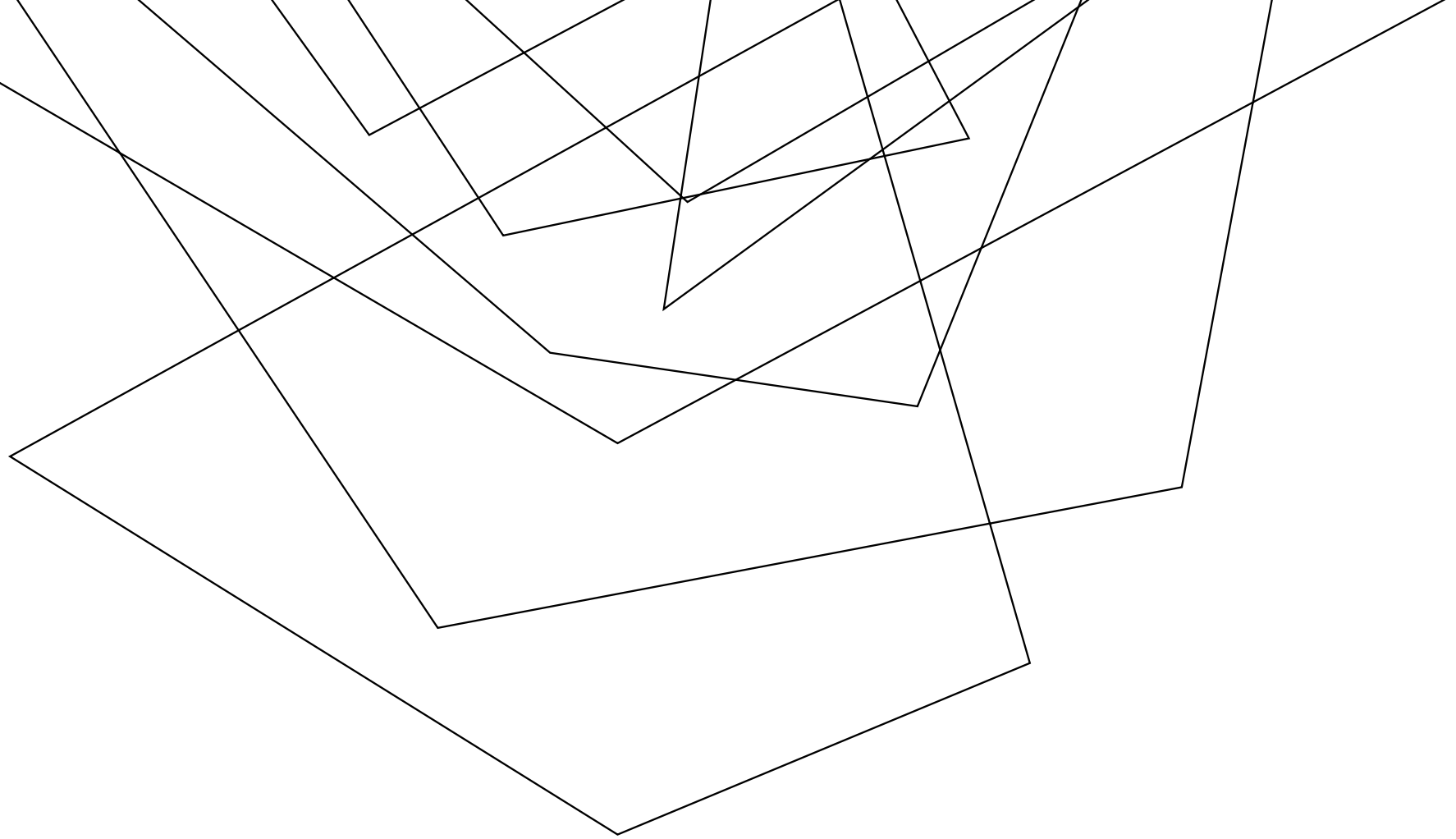
LLVM MEMORY REVIEW

Write your name and answer the following on a piece of paper

- *Write an LLVM IR function that takes a pointer to an array of ints and returns the element at index 2*

```
int foo(int * arr){
  return arr[2];
}
```

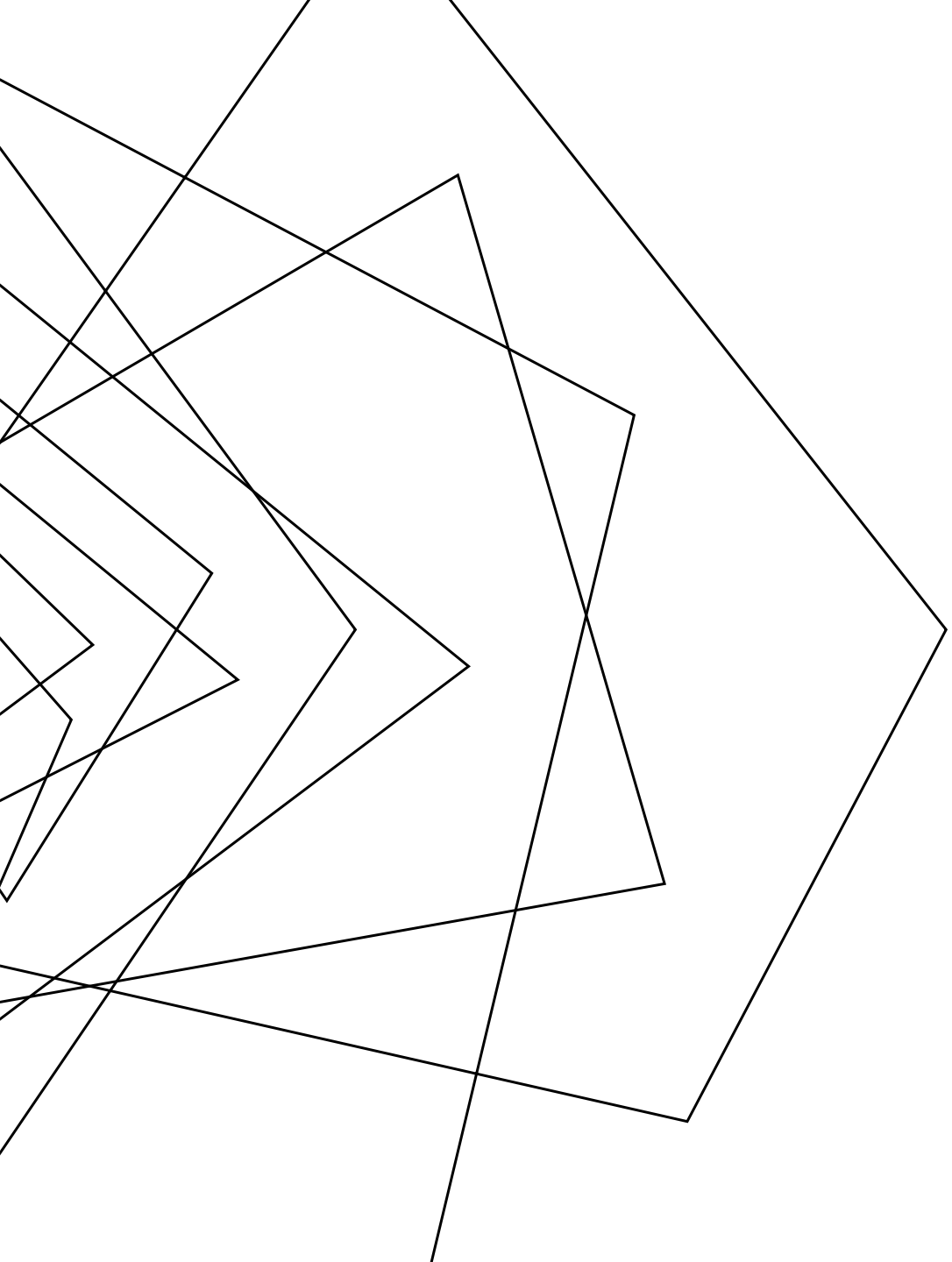
```
define i32 @foo(i32* %arg){
  %eltPtr = getelementptr i32, i32* %0, i64 2
  %res = load i32, i32* %eltPtr
  return i32 %res
}
```



LLVM CALLS

EECS 677: Software Security Evaluation

Drew Davidson



CLASS PROGRESS

WE'RE BASICALLY READY TO DO STATIC ANALYSIS

LAST TIME: LLVM MEMORY

REVIEW: LAST LECTURE

DESCRIBED LLVM'S CONCEPT OF NAMED MEMORY

- No mathematical relationship between distinct named memory items
- Guaranteed mathematical relation WITHIN named memory items (as exploited by GEP)

DECLARING MEMORY

- Local memory:

```
%ptrL = alloca i32, align 4
```

- Global memory:

```
@ptrG = global i32 2, align 4
```

- (Global) constant: Guaranteed mathematical relation

```
@ptrC = constant i32 2, align 4
```

```
%ptrLarr = alloca [8 x i32], align 4
```

```
%ptrGstruct = global i32 {i32, i8}, align 4
```

```
@ptrCstructs = constant [2 x {i8, i32}]
    [{i8, i32} {i8 1, i32 2}, {i8, i32}{ i8 3, i32 4} ] ,
    align 4
```

LAST TIME: LLVM MEMORY

REVIEW: LAST LECTURE

DECLARING MEMORY

- Local memory:
  %ptrL = alloca i32, align 4
- Global memory:
  @ptrG = global i32 2, align 4
- (Global) constant: Guaranteed mathematical relation
  @ptrC = constant i32 2, align 4

%ptrLarr = alloca [8 x i32], align 4

%ptrGstruct = global i32 {i32, i8}, align 4

@ptrCstructs = constant [2 x {i8, i32}]
[{i8, i32} {i8 1, i32 2}, {i8, i32}{ i8 3, i32 4}] ,
align 4

ACCESSING MEMORY

- Store scalar:

```
store i32 1, i32* %ptrL
```

- Global memory:

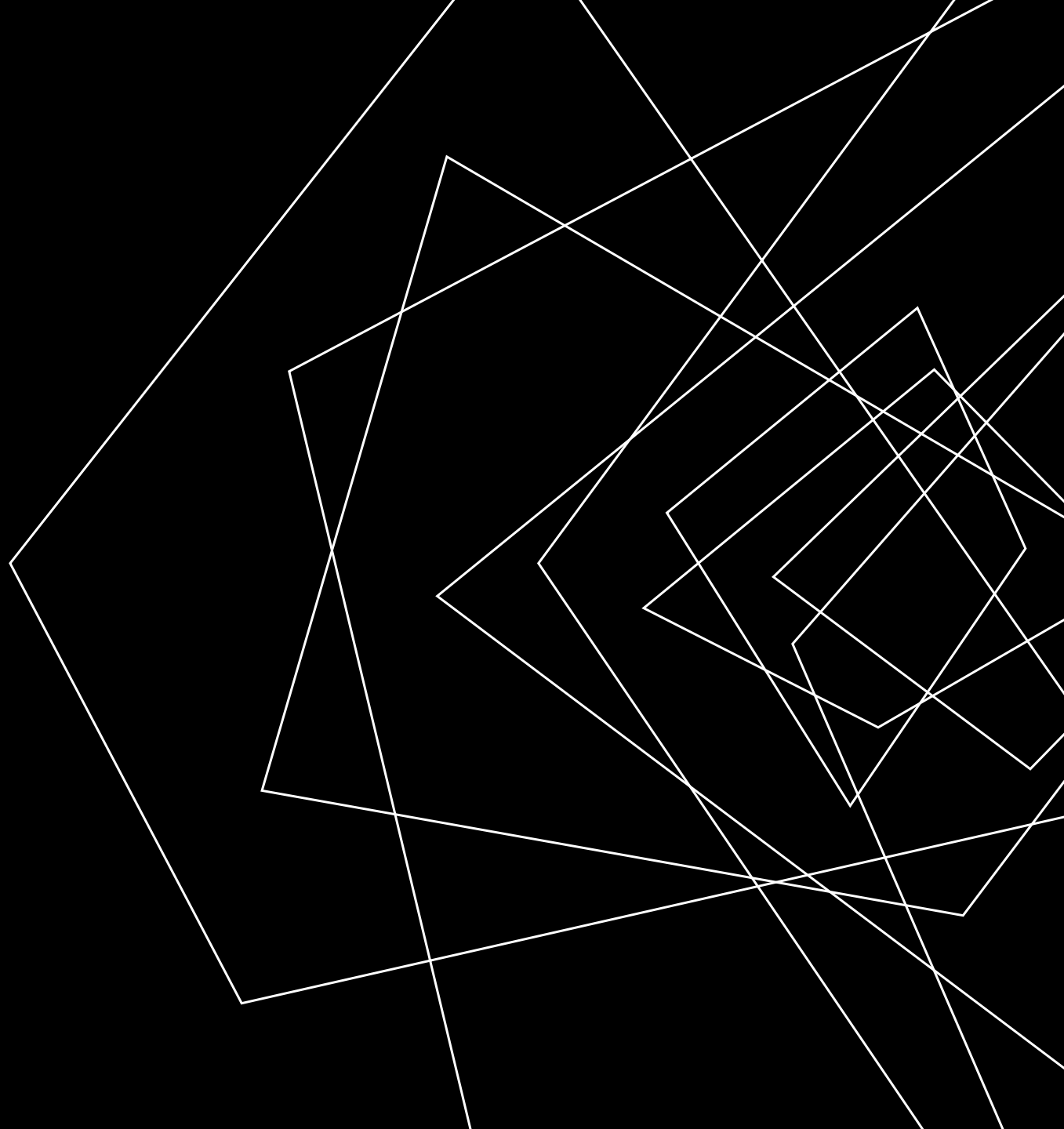
```
%reg = load i32, i32* @ptrG
```

- Load index of aggregate type

```
%eltPtr = getelementptr [2 x {i8, i32}],
    [2 x {i8, i32}]* @ptrCstructs, i64 0, i64 0, i32 1
%res = load i32, i32* %ret
```

LECTURE OUTLINE

- A little more GEP intuition
- Function calls



SOME TIME WITH GEP

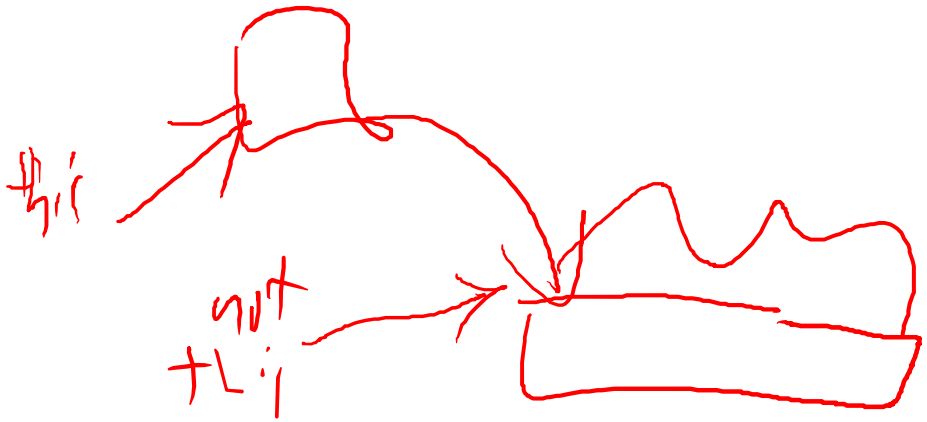
LLVM BITCODE

```
char getThirdElt(char arr[4]){
        return arr[2];
}

char getThirdOff(char * ptr){
        return ptr[2];
}
```

```
define i8 @getThirdElt([4 x i8]* %arrPtr) {
  %eltPtr = getelementptr [4 x i8], [4 x i8]* %arrPtr, i64 0, i64 2
  %res = load i8, i8* %eltPtr, align 1
  ret i8 %res
}
```

```
define i8 @getThirdOff(i8* %ptr) {
  %eltPtr = getelementptr inbounds i8, i8* %ptr, i64 2
  %res = load i8, i8* %eltPtr, align 1
  ret i8 %res
}
```



WRITING “INTERESTING” PROGRAMS

LLVM BITCODE

WE CAN NOW WRITE TURING COMPLETE PROGRAMS

BUT THESE PROGRAMS ARE VERY BORING!

We need to interact with external functionality

LLVM CALLS

LLVM BITCODE

General syntax

call <callee sig> <function name> (<argList>)

```
%res = call  i32(i32,i32)      @max      (i32 1, i32 2)
```

Somewhat surprisingly, does not (always) require the full function signature of the callee!

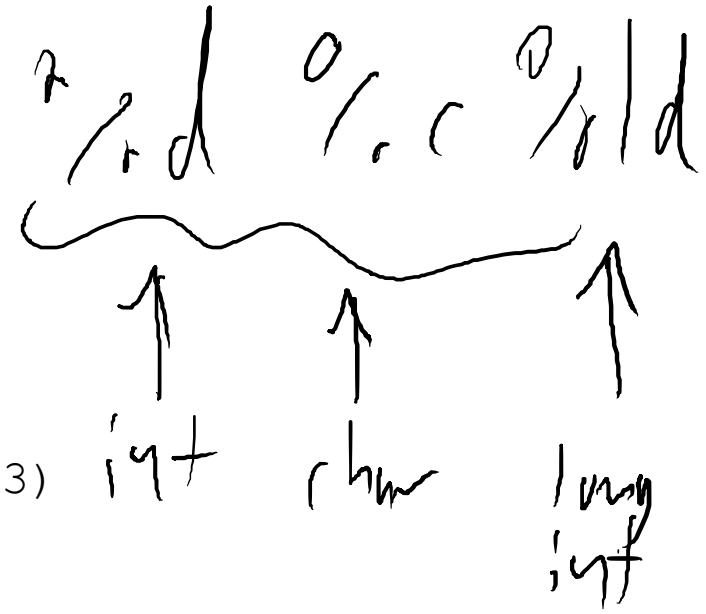
Constrained Syntax

call <return type> <function name> (<argList>)

```
%res = call  i32          @max      (i32 1, i32 2)
```

The general syntax IS required if a function has varargs

```
%len = call i32 (i8*, ...) @printf(i8* %strPtr, i32 123)
```



LLVM EXTERNAL CALLS

LLVM BITCODE

EXAMPLE

```
%len = call i32 (i8*, ...) @printf(i8* %strPtr, i32 123)
```

LLVM ATOI

LLVM BITCODE

To the Terminal!

LLVM PRINTF

LLVM BITCODE

To the Terminal!

RUNNING THE LLVM TOOLS: CLANG

LLVM BITCODE

```
clang -S -emit-llvm foo.c -o foo.ll -disable-O0-optnone
```

FOLLOW-UP WHAT IS THE #0 HERE?

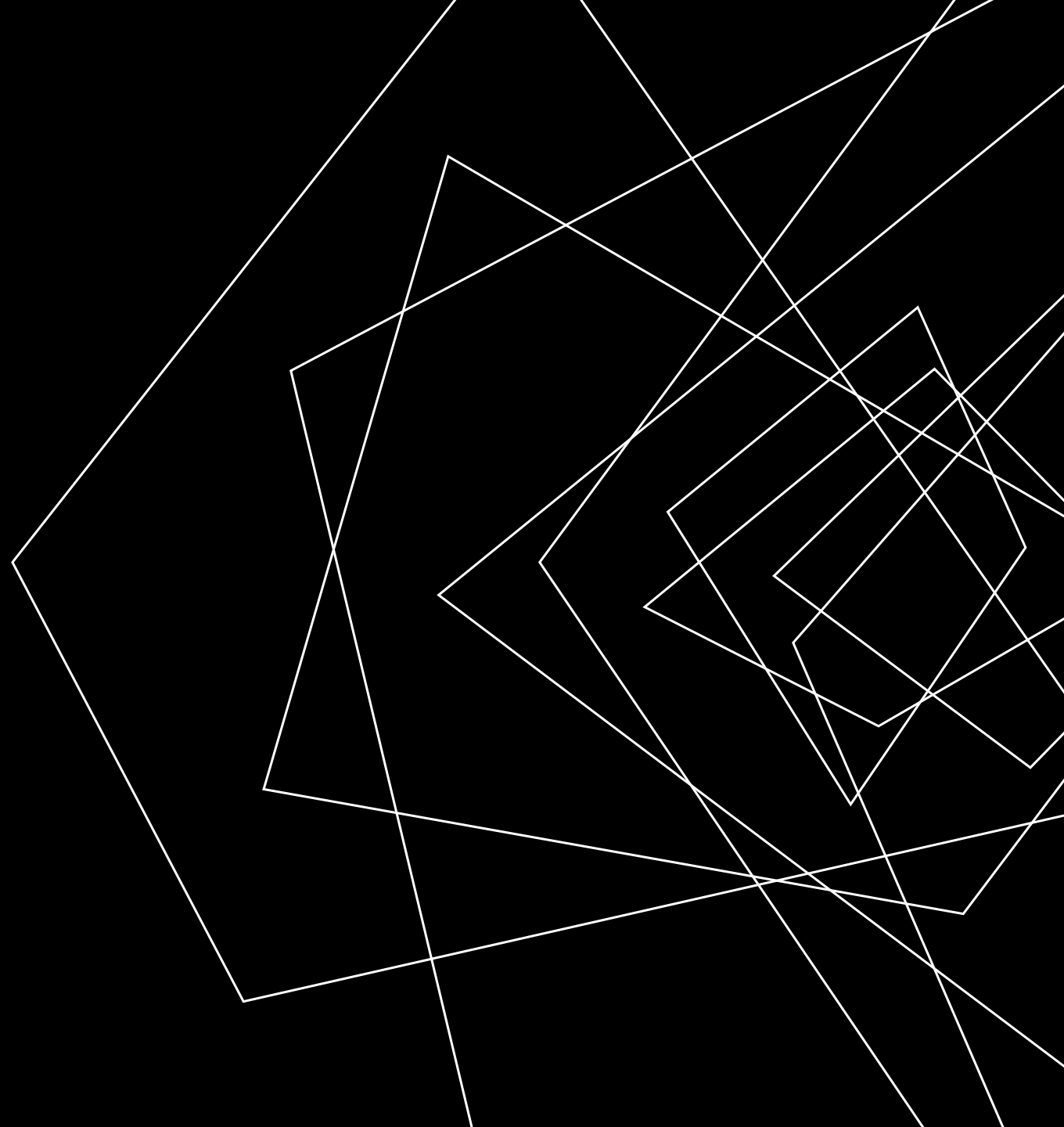
REVIEW: LAST LECTURE

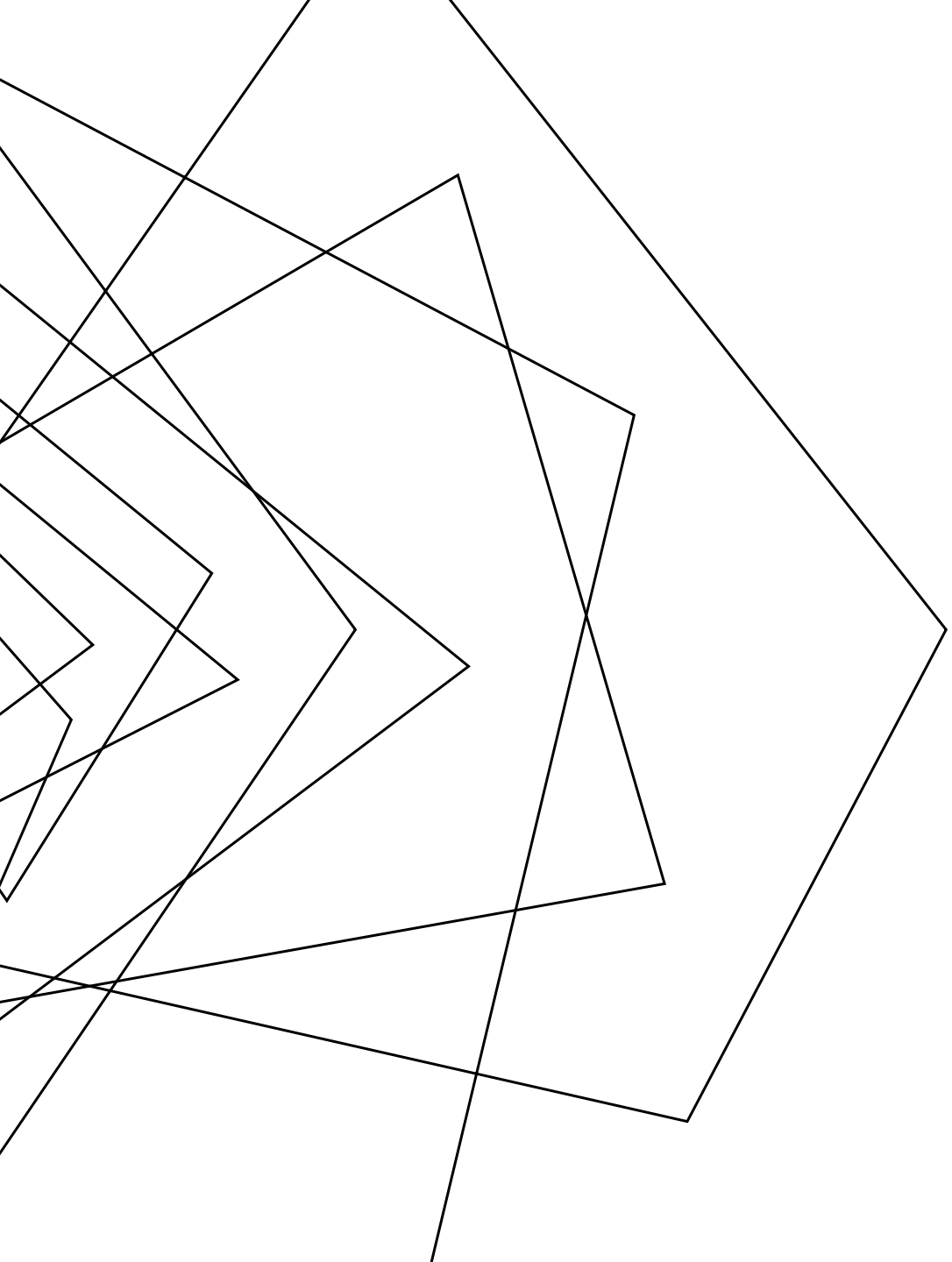
```
define i32 @main() #0 {
  ...
}
```

It’s an alias for an attribute list

```
attributes #0 = { optnone }
```

WRAP-UP





NEXT TIME

WRITING AN ANALYSIS