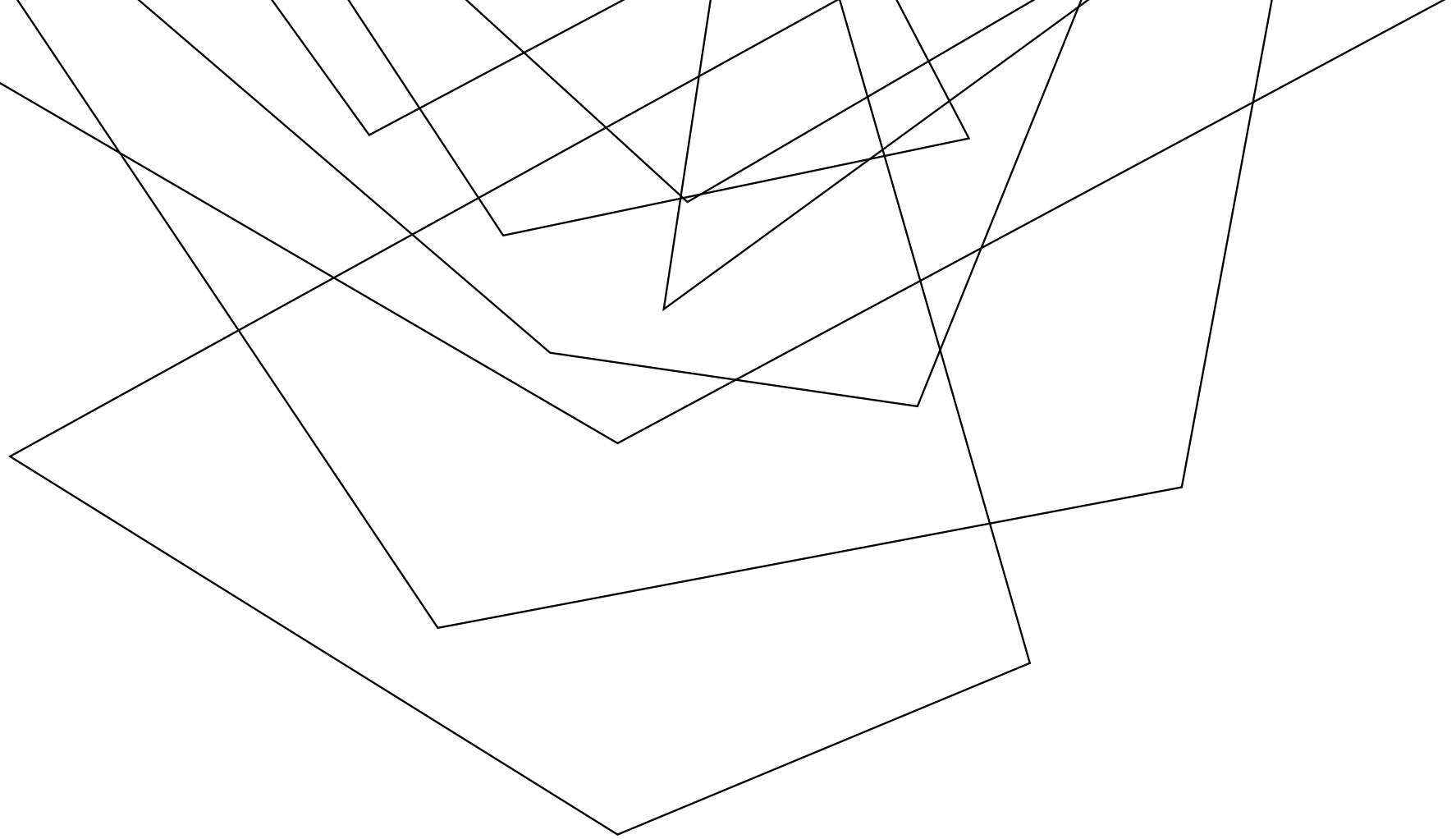
EXERCISE 5

LLVM MEMORY REVIEW

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

- *Write an LLVM IR function that simulates the following*

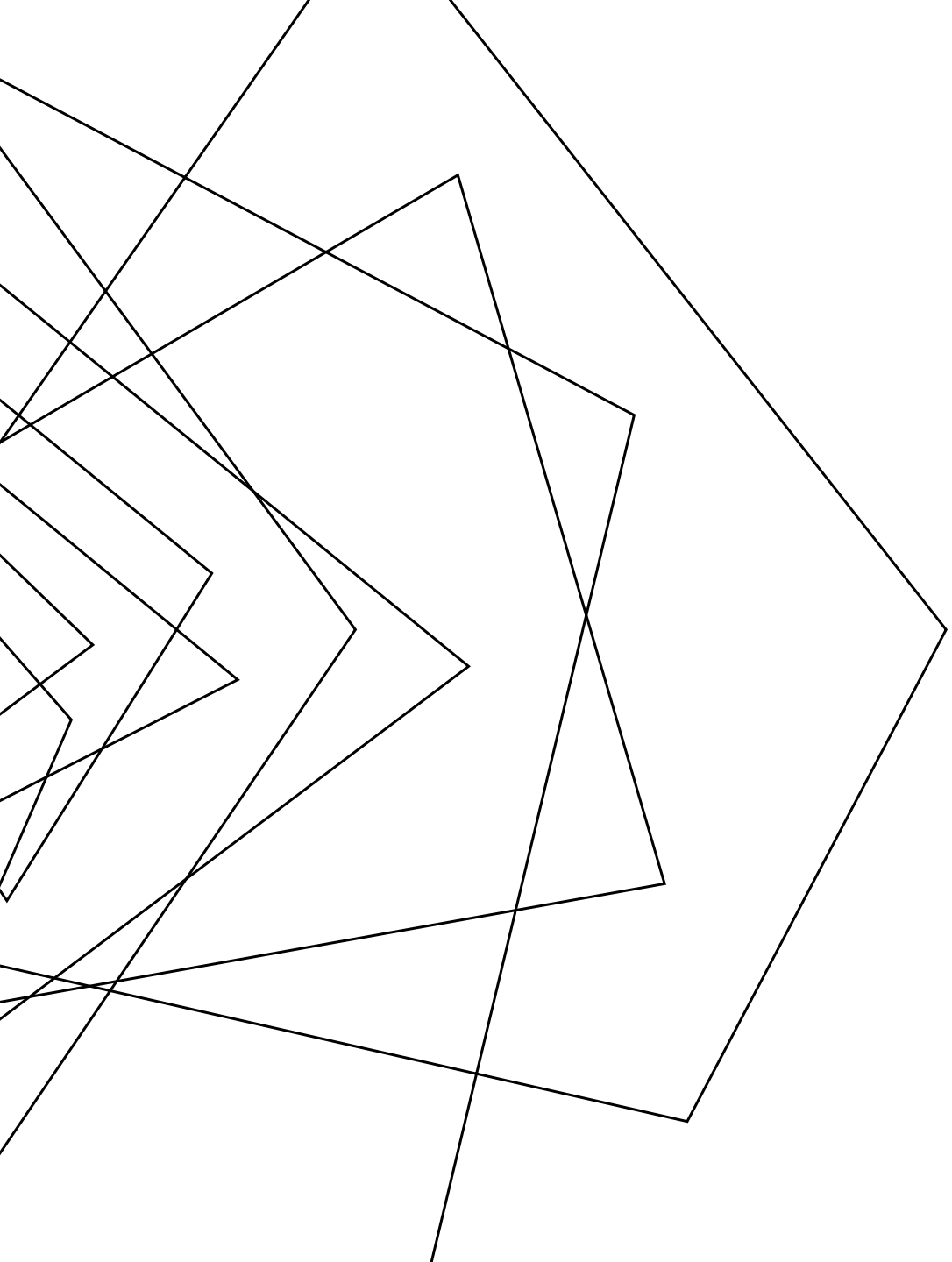
```
int a;
int main(int argc){
   return a + argc;
}
```



LLVM CALLS

EECS 677: Software Security Evaluation

Drew Davidson



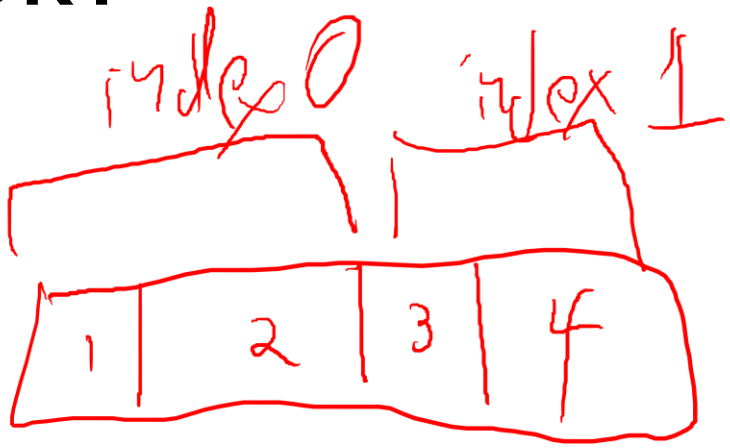
CLASS PROGRESS

WE'RE BASICALLY READY TO DO ~~STATIC ANALYSIS~~

SSE

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

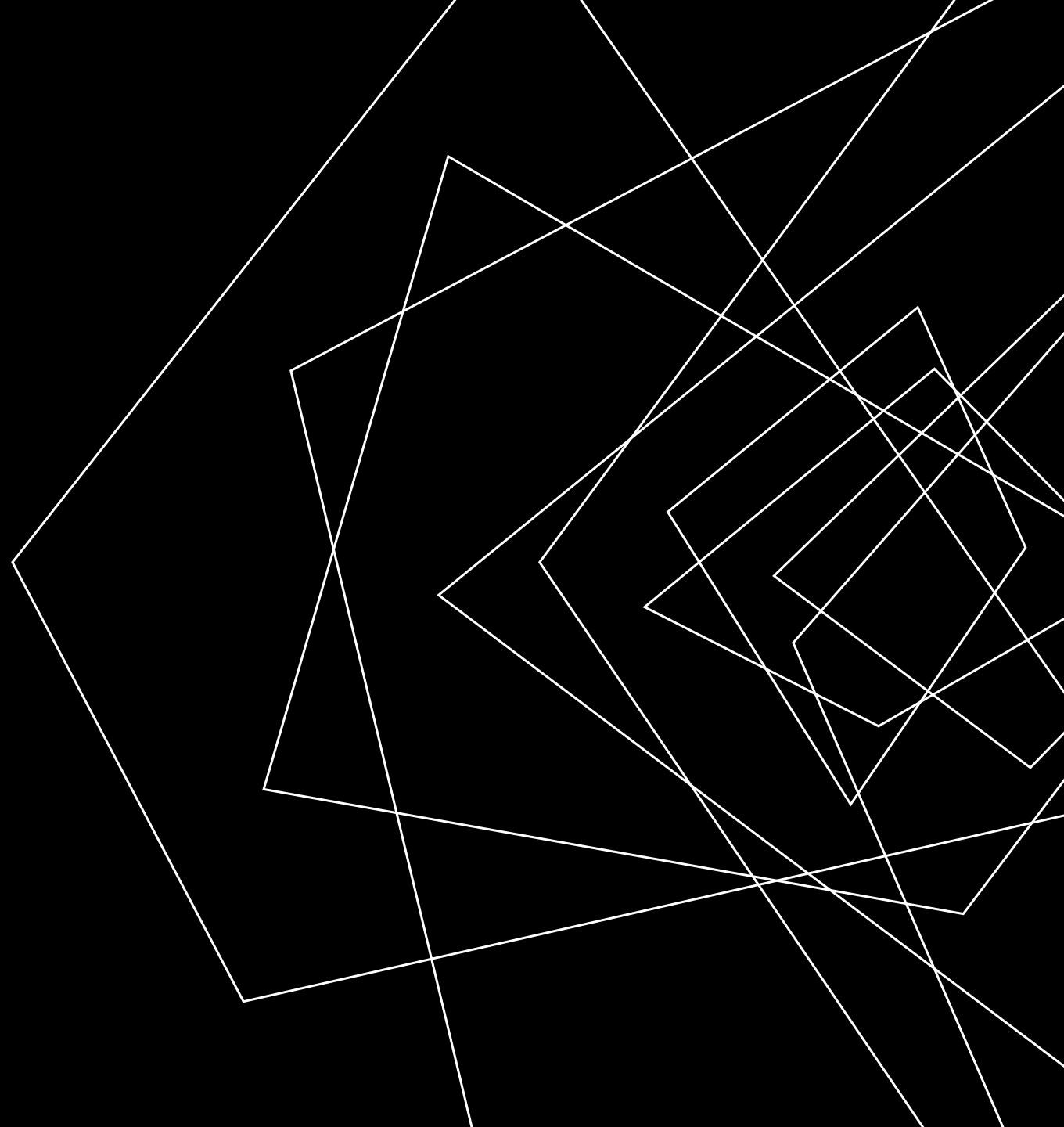
@ptrCstructs = constant [2 x {i8, i32}]
[{i8, i32} {i8 1, i32 2}, {i8, i32}{ i8 3, i32 4}] ,
align 4

ACCESSING MEMORY

- Store scalar:
  ```
  store i32 1, i32* %ptrL
  ```
- Global memory:
  ```
  %reg = load i32, i32* @ptrG
  ```

LECTURE OUTLINE

- GEP
- Function calls



GETELEMENTPTR

THE DREADED GEP

Here is the Basic format of a GEP

```
<result> = getelementptr <ty>, ptr <ptrval>{, [inrange] <ty> <idx>}*
```

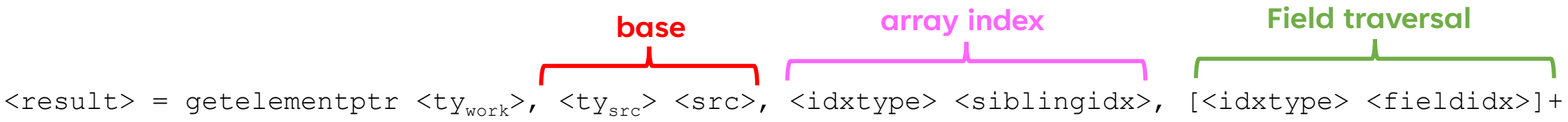
Here is a snippet of the documentation of the syntax:

The first argument is always a type used as the basis for the calculations. The second argument is always a pointer or a vector of pointers, and is the base address to start from. The remaining arguments are indices that indicate which of the elements of the aggregate object are indexed. The interpretation of each index is dependent on the type being indexed into. The first index always indexes the pointer value given as the second argument, the second index indexes a value of the type pointed to (not necessarily the value directly pointed to, since the first index can be non-zero), etc. The first type indexed into must be a pointer value, subsequent types can be arrays, vectors, and structs. Note that subsequent types being indexed into can never be pointers, since that would require loading the pointer before continuing calculation.

GETELEMENTPTR

THE DREADED GEP

LET ME (MAYBE?) SIMPLIFY THIS A BIT WITH A SLIGHT REFORMAT OF GEP



HERE IS MY EXPLANATION OF THIS VERSION OF GEP:

Assume **base** is a pointer into some array of somethings (possibly a nested data structure)
- Arg 1: `<ty_work>`: Specify the type of the somethings
- Arg 2: `<ty_src> <src>`: **base** address to start your computation
- Arg 3: `<idxtype> <siblingidx>`: **array index** to jump forward from the **base** address

(end of optional arguments)
- Arg 4+: `<idxtype> <fieldidx>`: **field traversal** to index into the fields of the nested data structure

GETELEMENTPTR

THE DREADED GEP

LET ME (MAYBE?) SIMPLIFY THIS A BIT WITH A SLIGHT REFORMAT OF GEP

base array index Field traversal

```
<result> = getelementptr <ty_work>, <ty_src> <src>, <idxtype> <siblingidx>, [<idxtype> <fieldidx>]+
```

Answer:

Very generic format to capture the large variety of ways that you need to index into memory

Basic GEP invocations handle simple cases

Complex GEP invocations handle complex cases

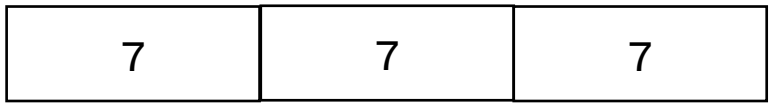
GETELEMENTPTR

THE DREADED GEP

LET ME (MAYBE?) SIMPLIFY THIS A BIT WITH A SLIGHT REFORMAT OF GEP

base array index Field traversal

```
<result> = getelementptr <ty_work>, <ty_src> <src>, <idxtype> <siblingidx>, [<idxtype> <fieldidx>]+
```




```
getelementptr %T2, ptr @ptr_n1,  i64 0,   i64 0
```

Answer:

Very generic format to capture the large variety of ways that you need to index into memory

Basic GEP invocations handle simple cases

Complex GEP invocations handle complex cases

GETELEMENTPTR: PICTORIALLY

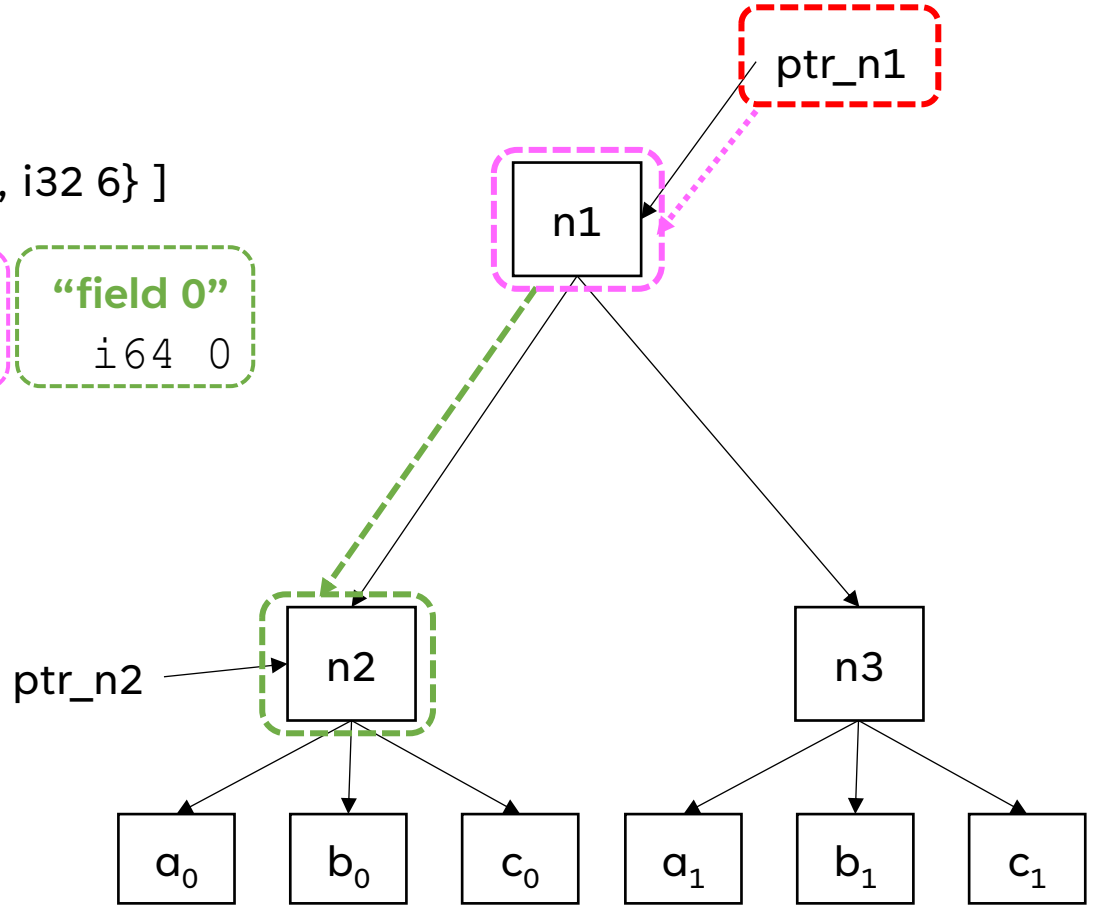
THE DREADED GEP

Can be helpful to walk through memory as a tree

```
%T1 = type { i32, i32, i32 }
%T2 = type [ 2 x %T1 ]
@ptr_n1 = global %T2 [ { i32 1, i32 2, i32 3 }, { i32 4, i32 5, i32 6} ]
```

```
ptr_n2 = getelementptr %T2, ptr @ptr_n1, i64 0, i64 0
```

"base addr" "sibling 0" "field 0"



GETELEMENTPTR: PICTORIALLY

THE DREADED GEP

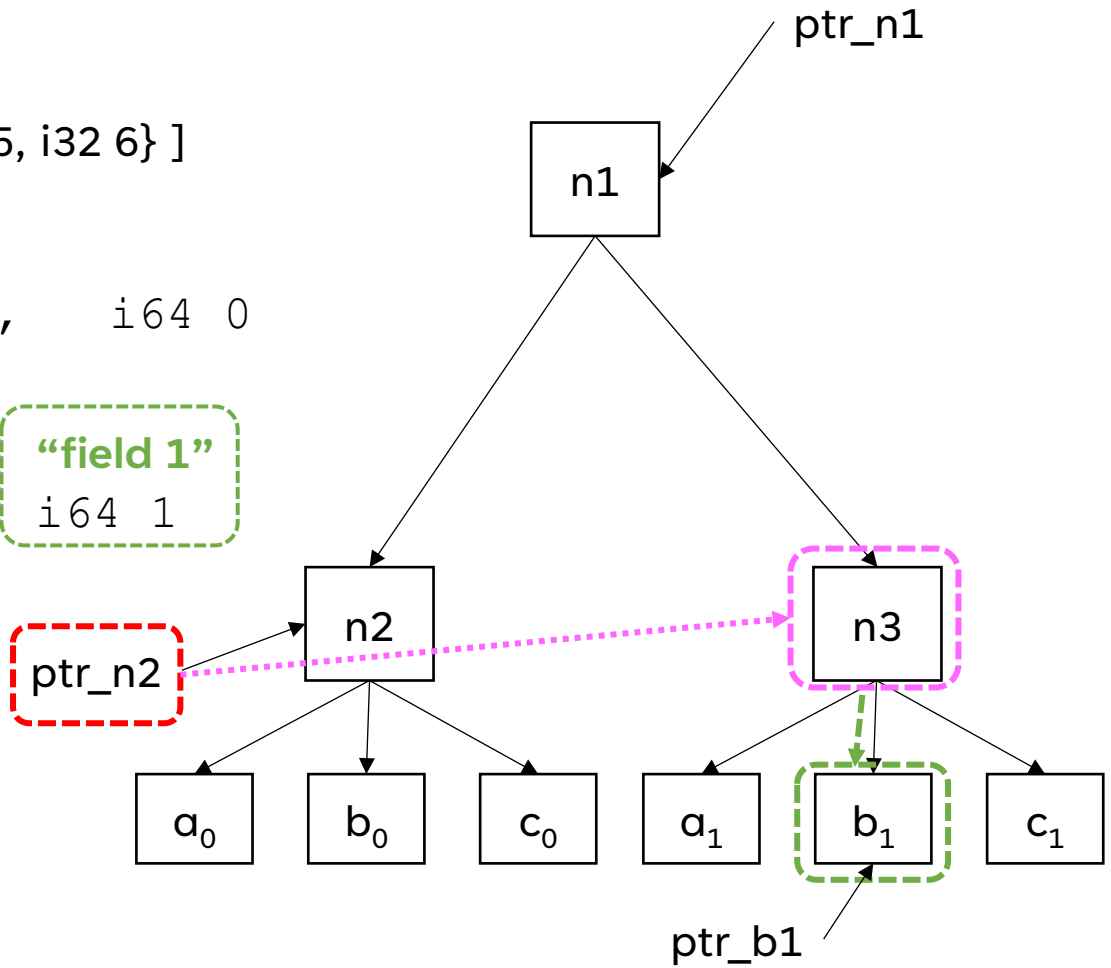
Can be helpful to walk through memory as a tree

```
%T1 = type { i32, i32, i32 }
%T2 = type [ 2 x %T1 ]
@ptr_n1 = global %T2 [ { i32 1, i32 2, i32 3 }, { i32 4, i32 5, i32 6} ]
```

```
ptr_n2 = getelementptr %T2, ptr @ptr_n1,  i64 0,   i64 0
```

```
ptr_b1 = getelementptr %T1, ptr @ptr_n2,  i64 1,   i64 1
```

"base addr" "sibling 1" "field 1"



GETELEMENTPTR: PICTORIALLY

THE DREADED GEP

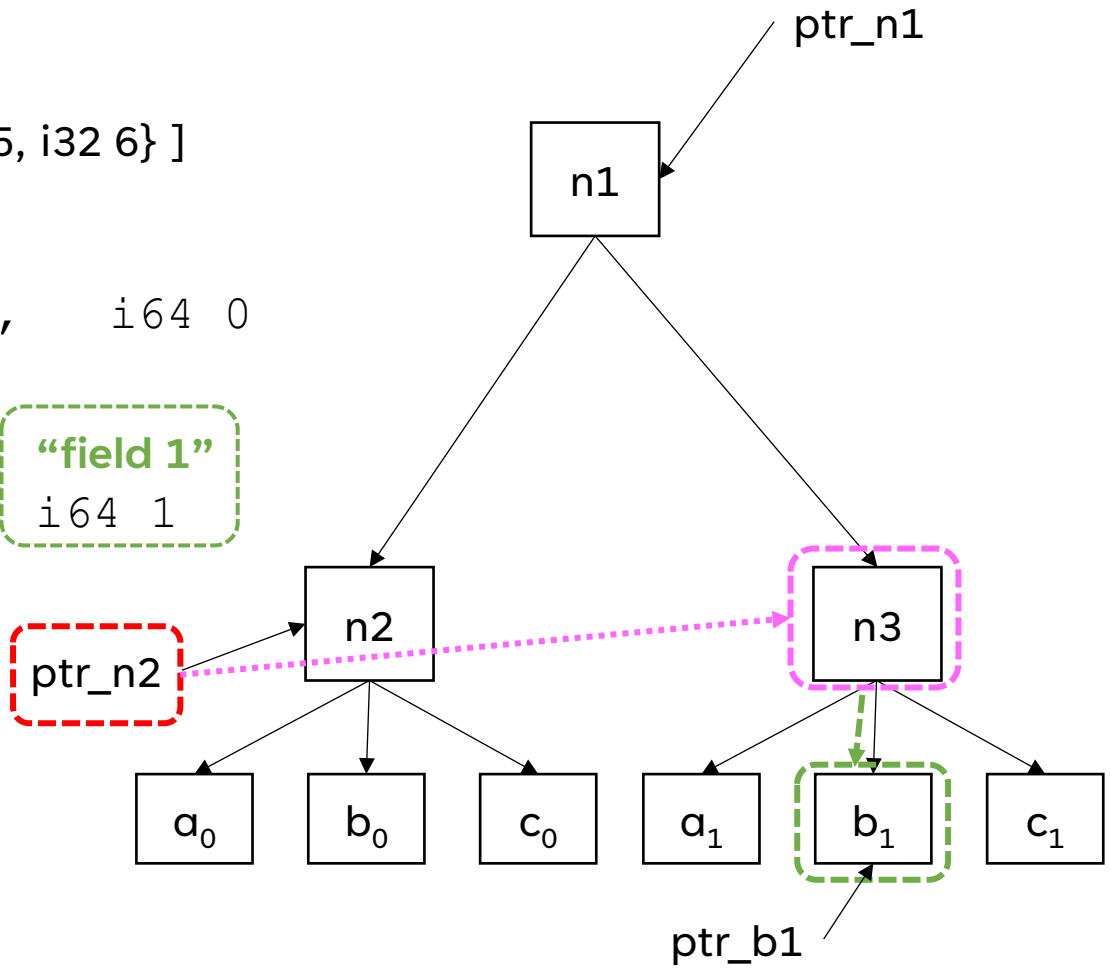
Can be helpful to walk through memory as a tree

```
%T1 = type { i32, i32, i32 }
%T2 = type [ 2 x %T1 ]
@ptr_n1 = global %T2 [ { i32 1, i32 2, i32 3 }, { i32 4, i32 5, i32 6} ]
```

```
ptr_n2 = getelementptr %T2, ptr @ptr_n1,  i64 0,   i64 0
```

```
ptr_b1 = getelementptr %T1, ptr @ptr_n2,  i64 1,   i64 1
```

"base addr" "sibling 1" "field 1"



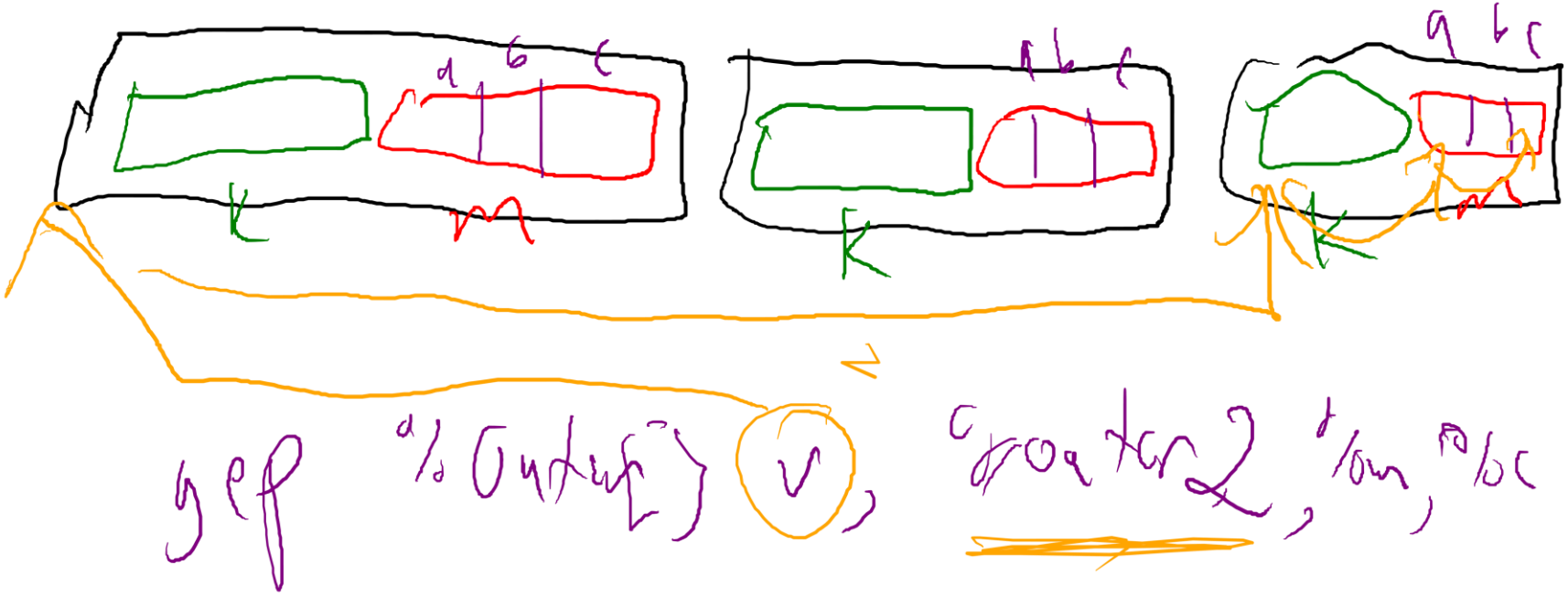
GETELEMENTPTR: YA GOTTA HANDLE C

THE DREADED GEP

MY THEORY: GEP IS DESIGNED TO ACCOMMODATE THE NEEDS OF C SOURCE CODE

```
struct Inner {
  int32_t a;
  int8_t b;
  char c;
};
struct Outer{
  int32_t k;
  struct Inner m;
}

struct Outer v[3];
int main(){
  v[2].m.c = 'X';
}
```



v is the base
2 is the 3rd in array

m is 2nd child
c is the 3rd child

GEP FOR SOURCE CODE

LLVM BITCODE

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

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

```
1 define i8 @getThirdElt([4 x i8]* %arrPtr){
2         %eltPtr = getelementptr [4 x i8], [4 x i8]* %arrPtr, i64 0, i64 2
3         %res = load i8, i8* %eltPtr
4         ret i8 %res
5 }
```

```
1 define i8 @getThirdOff(i8* %ptr){
2         %eltPtr = getelementptr i8, i8* %ptr, i64 2
3         %res = load i8, i8* %eltPtr
4         ret i8 %res
5 }
```

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*) @atoi(i8* %elt1)
```

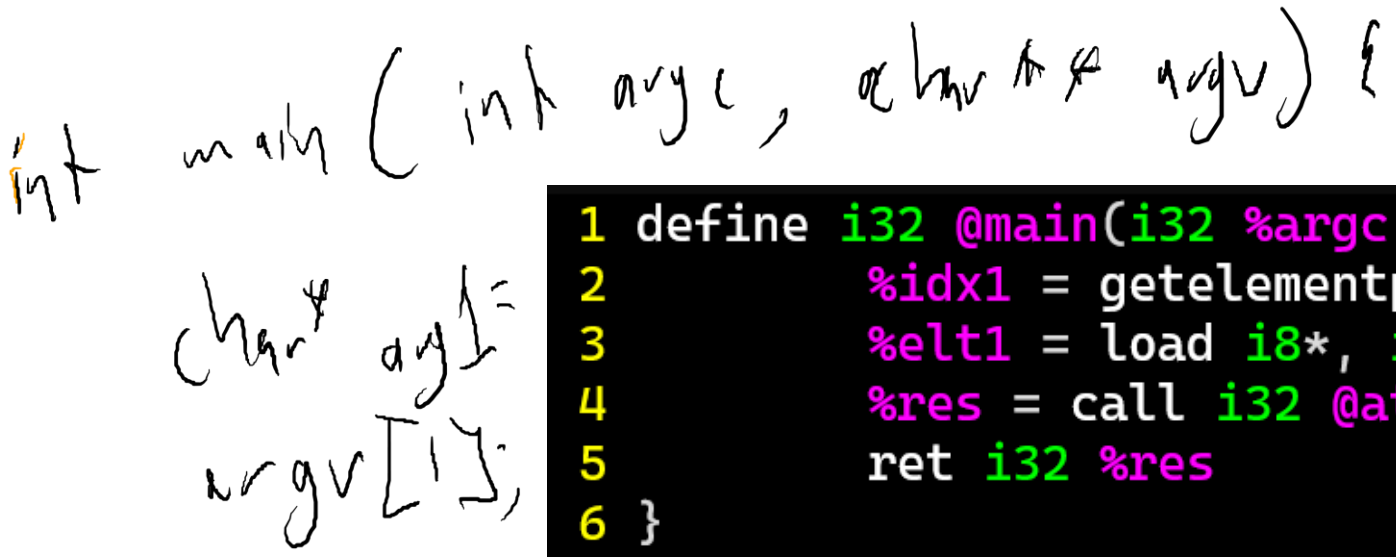
A hero arrives

```
declare i32 @atoi(i8*)
```

LLVM ATOI

LLVM BITCODE

LET'S INVOKE A SOMEWHAT-FAMILIAR FUNCTION...

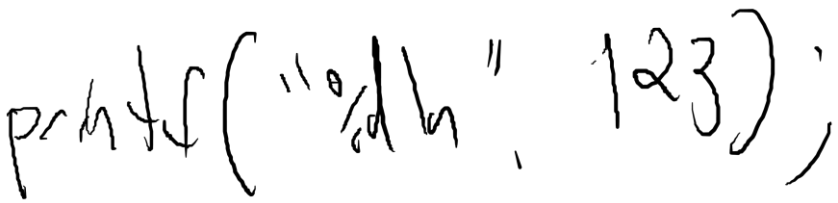



```
1 define i32 @main(i32 %argc, i8** %argv) {
2         %idx1 = getelementptr i8*, i8** %argv, i64 1
3         %elt1 = load i8*, i8** %idx1, align 8
4         %res = call i32 @atoi (i8* %elt1)
5         ret i32 %res
6 }
7
8 declare i32 @atoi(i8*)
```

LLVM PRINTF

LLVM BITCODE

GETTING MORE CONVENIENT OUTPUT FROM OUR PROGRAMS



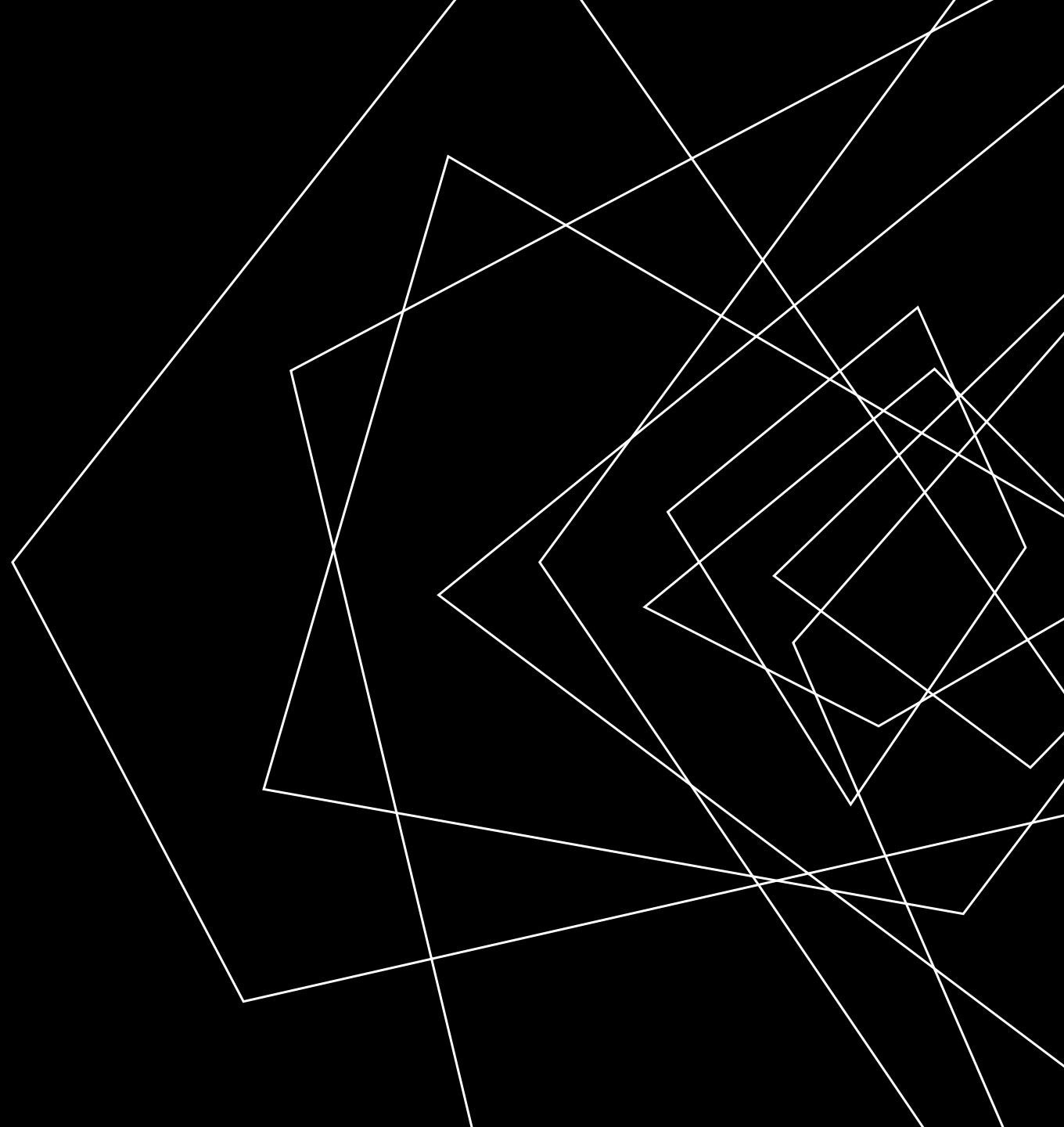

```
1 @.str = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
2
3  define i32 @main() {
4   %strPtr = getelementptr [4 x i8], [4 x i8]* @.str, i64 0, i64 0
5   %reg = call i32 (i8*, ...) @printf(i8* %strPtr, i32 123)
6   ret i32 %reg
7 }
8
9 declare dso_local i32 @printf(i8*, ...)
```

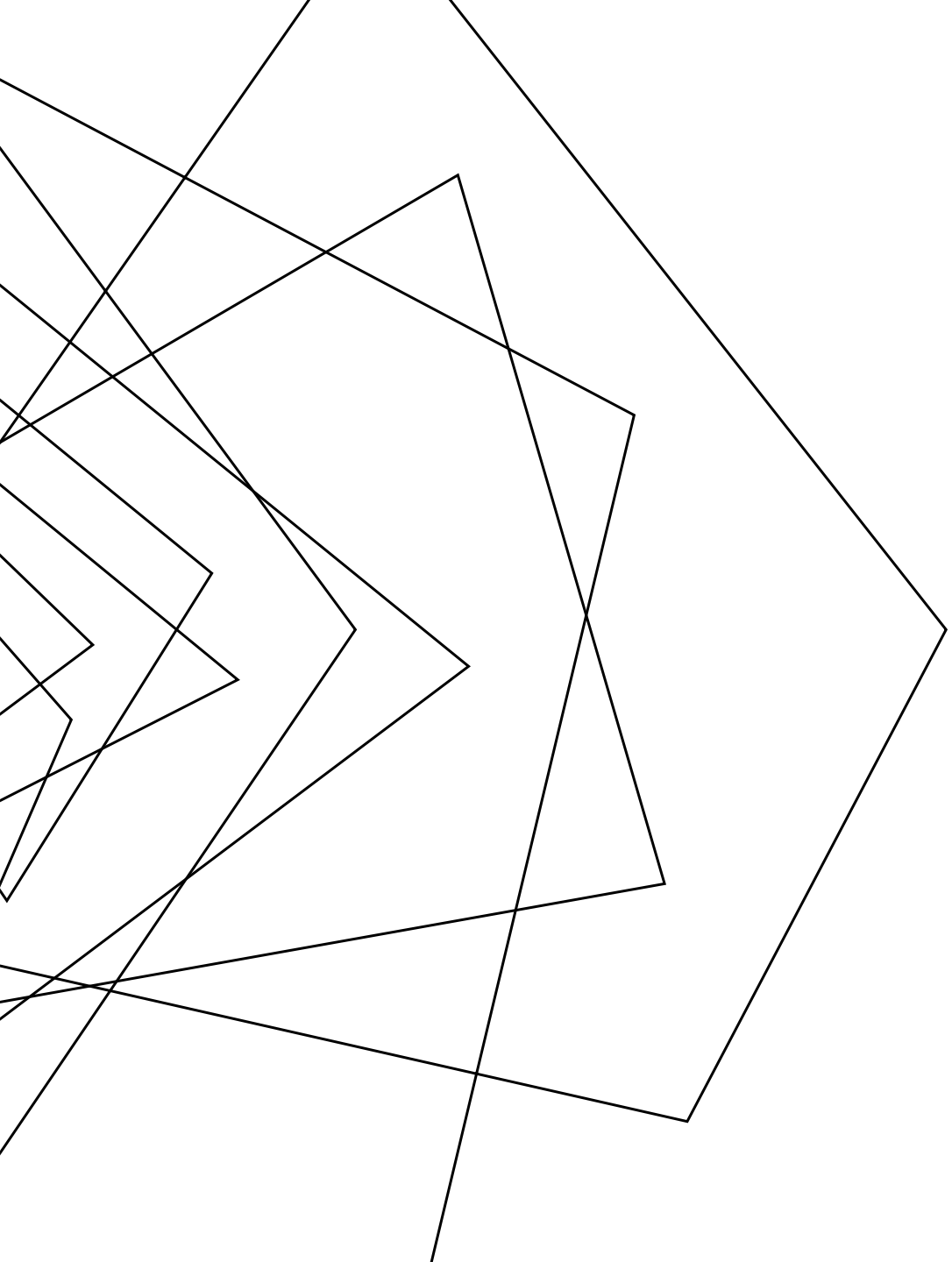
RUNNING THE LLVM TOOLS: CLANG

LLVM BITCODE

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

WRAP-UP





NEXT TIME

WRITING AN ANALYSIS