

EXERCISE 4

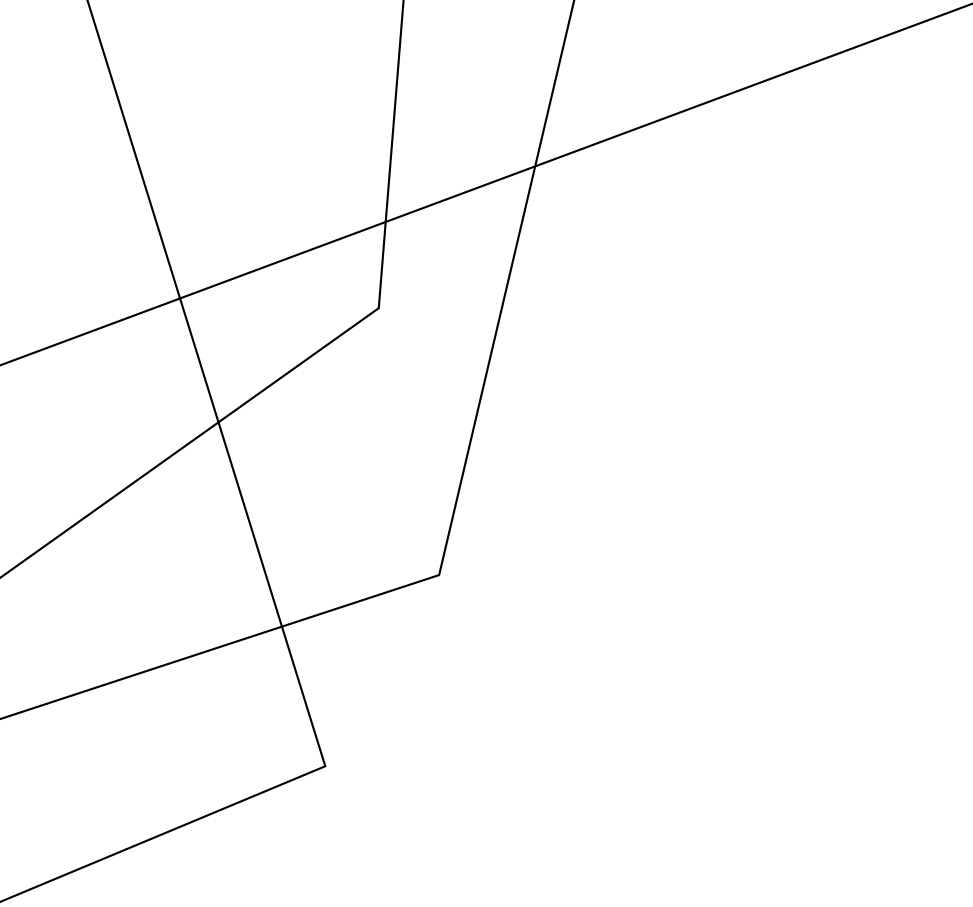
LLVM REGISTER OPERATION REVIEWS

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

- *Draw the CFG and equivalent c code for the following LLVM IR program*

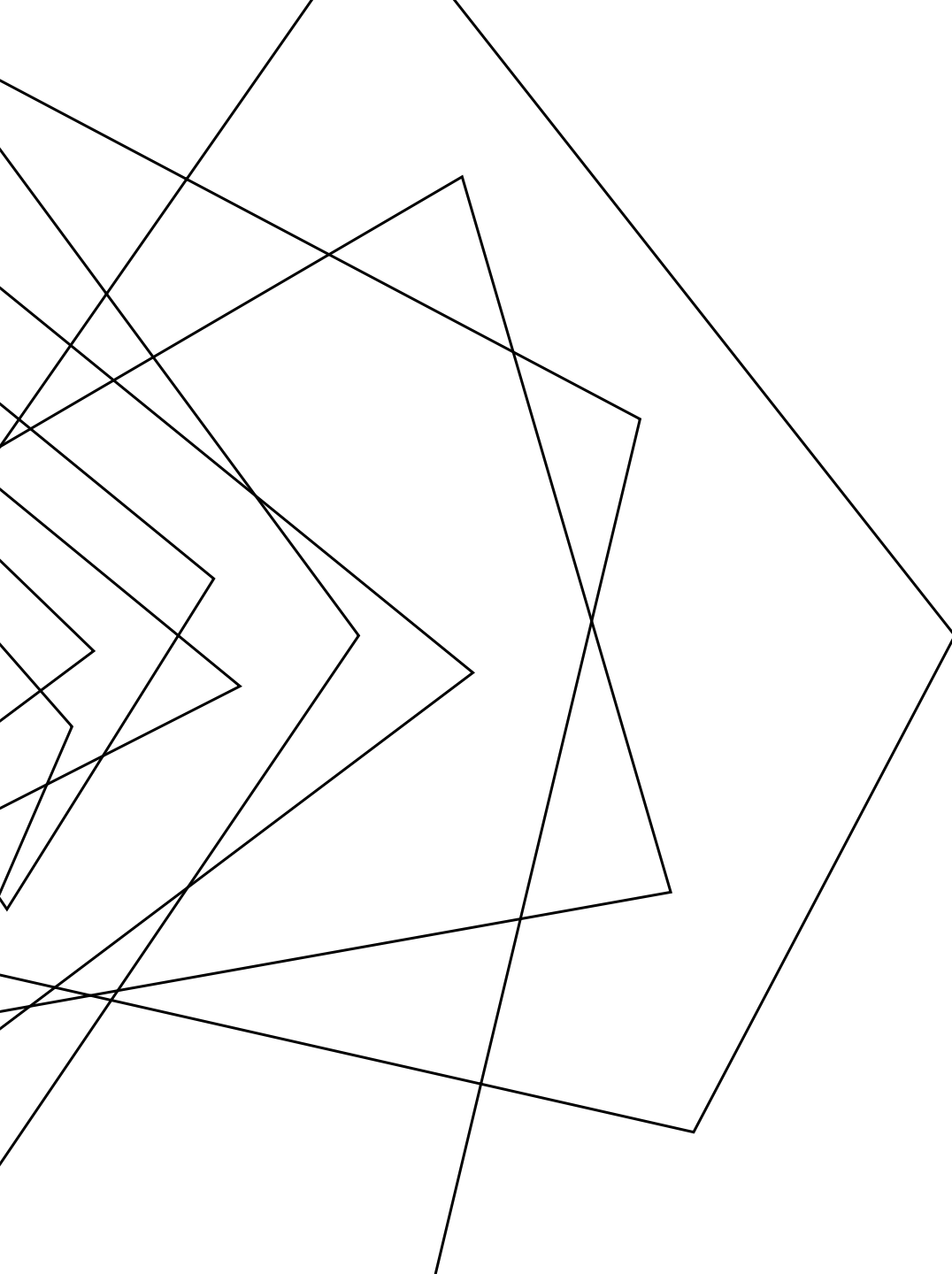
```
define i32 @main(i32 %argc) #0 {
entry:
    %i_init = add i32 0, 0
    br label %loop_head
loop_head:
    %i_join = phi i32 [%i_init, %entry], [%i_loop, %loop_body]
    %done = icmp slt i32 %i_join, %argc
    br i1 %done, label %loop_body, label %loop_after
loop_body:
    %i_loop = add i32 %i_join, 1
    br label %loop_head
loop_after:
    ret i32 %i_join
}
```

```
int main(int argc){
    int i = 0;
    while (i < argc){
        i = i + 1;
    }
    return i;
}
```

Abstract geometric lines in the top-left corner, consisting of several thin black lines forming a series of overlapping, irregular shapes.

All grades up to date!!

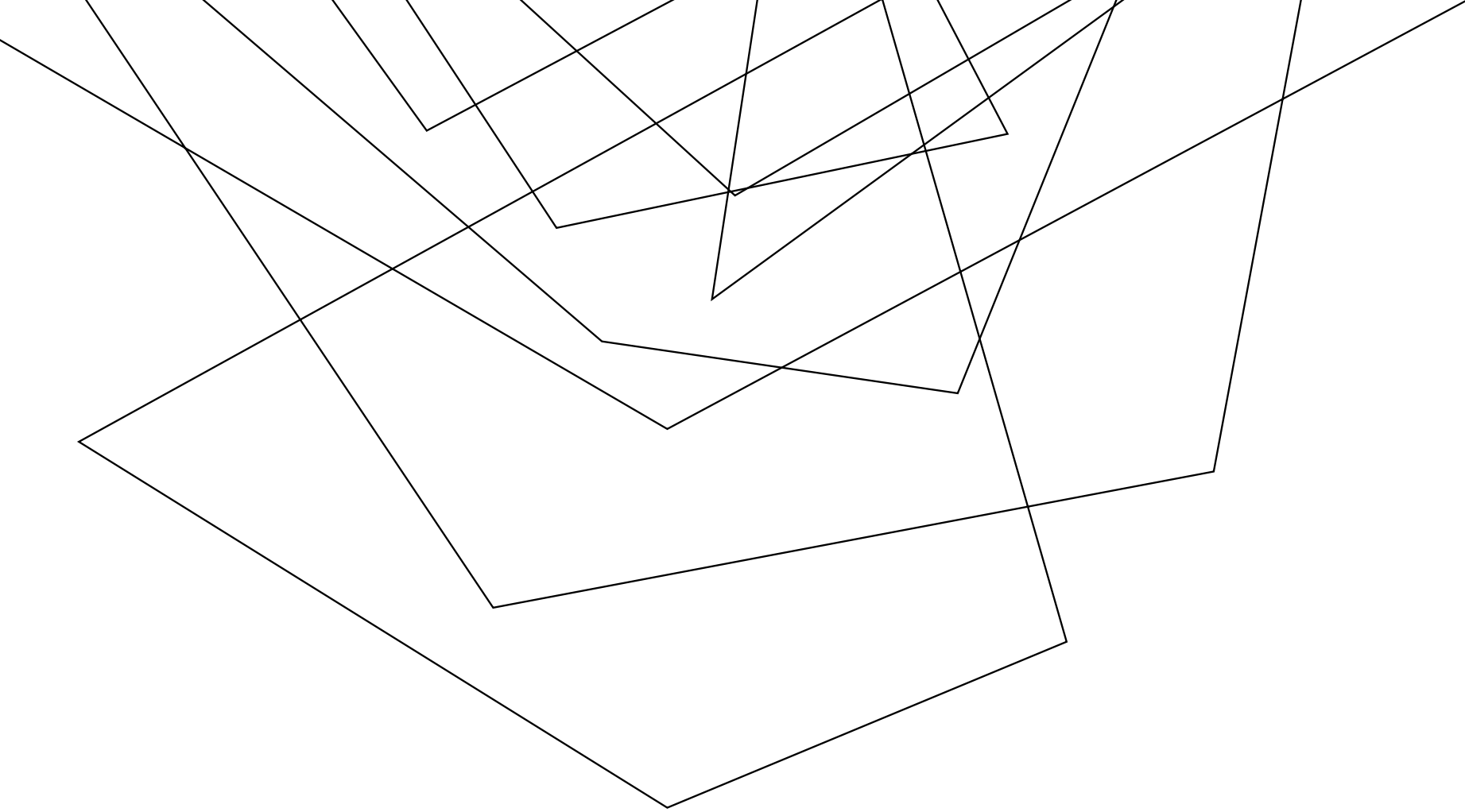
**ADMINISTRIVIA
AND
ANNOUNCEMENTS**



CLASS PROGRESS

WE'RE GEARING UP TO BUILD OUR OWN
PROGRAM ANALYSES

- WORKING THROUGH A GOOD
PROGRAM REPRESENTATION
- LLVM BITCODE IS A NICE “GENERIC”
TARGET



LLVM BITCODE MEMORY

EECS 677: Software Security Evaluation

Drew Davidson

LAST TIME: LLVM BITCODE & REGISTERS

REVIEW: LAST LECTURE

LOW-LEVEL LANGUAGE

- Targets an abstract machine
- Uses a system of (infinite) named registers to perform computation
- Registers must be in SSA format

SSA FORMAT

REVIEW: LAST LECTURE

STATIC SINGLE ASSIGNMENT

- A variable may be assigned at only one program point

PHI INSTRUCTIONS

REVIEW: LAST LECTURE

```
dest = phi <type> [val1, pred1], [val2, pred2] ...
```

RESOLVE THE NEED TO UNIFY REGISTERS

- Each argument is a pair [V,B] where
 - B is a predecessor basic block to the current block containing the phi instruction
 - V is a value to assign when block B is the dynamic predecessor

PHI INSTRUCTIONS

REVIEW: LAST LECTURE

LET'S IMPLEMENTATION
THIS CODE:

```
int main(int argc){
    int res = 1;
    if (argc > 1){
        res = 7;
    }
    return res;
}
```

```
define i32 @main(i32 %argc) #0 {
entry:
    %res_entry = add i32 0, 1
    %cond = icmp sgt i32 %argc, 1
    br i1 %cond, label %if_body, label %after_if
if_body:
    %res_body = add i32 0, 7
    br label %after_if
after_if:
    %res_join = phi i32 [%res_entry, %entry], [%res_body, %if_body]
    ret i32 %res_join
}
```

PHI INSTRUCTIONS

REVIEW: LAST LECTURE

LET'S IMPLEMENTATION
THIS CODE:

```
int main(int argc){
    int i = 0;
    while (i < argc){
        i = i + 1;
    }
    return i;
}
```

```
define i32 @main(i32 %argc) #0 {
entry:
    %i_init = add i32 0, 0
    br label %loop_head
loop_head:
    %i_join = phi i32 [%i_init, %entry], [%i_loop, %loop_body]
    %done = icmp slt i32 %i_join, %argc
    br i1 %done, label %loop_body, label %loop_after
loop_body:
    %i_loop = add i32 %i_join, 1
    br label %loop_head
loop_after:
    ret i32 %i_join
}
```

ASIDE: FANCY SYNTAX HIGHLIGHTING

DREW'S COOL TOOLS

TIRED:

```
define i32 @main(i32 %argc) #0 {
entry:
    %res_entry = add i32 0, 1
    %cond = icmp sgt i32 %argc, 1
    br i1 %cond, label %if_body, label %after_if
if_body:
    %res_body = add i32 0, 7
    br label %after_if
after_if:
    %res_join = phi i32 [%res_entry, %entry], [%res_body, %if_body]
    ret i32 %res_join
}
```

WIRED:

```
define i32 @main(i32 %argc) #0 {
entry:
    %res_entry = add i32 0, 1
    %cond = icmp sgt i32 %argc, 1
    br i1 %cond, label %if_body, label %after_if
if_body:
    %res_body = add i32 0, 7
    br label %after_if
after_if:
    %res_join = phi i32 [%res_entry, %entry], [%res_body, %if_body]
    ret i32 %res_join
}
```

ASIDE: FANCY SYNTAX HIGHLIGHTING

DREW'S COOL TOOLS

TURNS OUT SYNTAX HIGHLIGHTERS ARE AVAILABLE FOR SEVERAL EDITORS

- vim (my personal choice)
- emacs
- vscode

FILES ARE IN THE GIT REPO

<https://github.com/llvm/llvm-project>

Under the directory llvm/utils

If you don't want to download all that code, check

https://analysis.cool/llvm_syntax.tgz

TIRED:

```
define i32 @main(i32 %argc) #0 {
entry:
    %res_entry = add i32 0, 1
    %cond = icmp sgt i32 %argc, 1
    br i1 %cond, label %if_body, label %after_if
if_body:
    %res_body = add i32 0, 7
    br label %after_if
after_if:
    %res_join = phi i32 [%res_entry, %entry], [%res_body, %if_body]
    ret i32 %res_join
}
```

WIRED:

```
define i32 @main(i32 %argc) #0 {
entry:
    %res_entry = add i32 0, 1
    %cond = icmp sgt i32 %argc, 1
    br i1 %cond, label %if_body, label %after_if
if_body:
    %res_body = add i32 0, 7
    br label %after_if
after_if:
    %res_join = phi i32 [%res_entry, %entry], [%res_body, %if_body]
    ret i32 %res_join
}
```

THIS TIME: LLVM MEMORY

LECTURE OVERVIEW

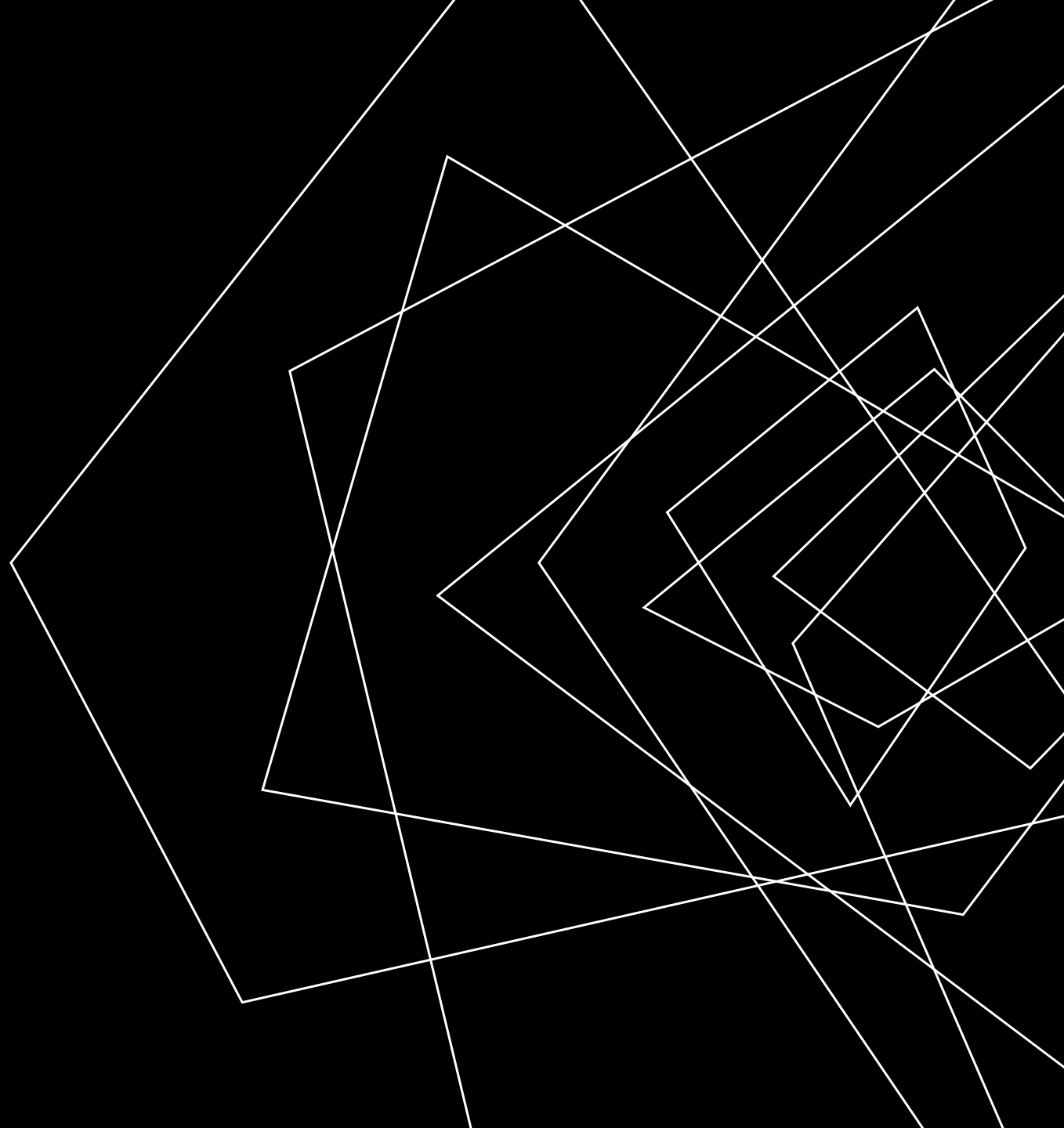
DEALING WITH MEMORY

- Ultimately we'll need to consider storage other than the infinite virtual register abstraction

Thanks
Fr th
MMRS

LECTURE OUTLINE

- LLVM Memory
- Load/Store
- The dreaded GEP



CONCRETE MEMORY

LLVM MEMORY

LLVM BITCODE ATTEMPTS TO
REPRESENT COMMONALITIES OF
MEMORY ON REAL ARCHITECTURES

So, what is computer memory like
(from the CS perspective)?

A 1-D array of cells

Numeric addresses (of some size)

Cells contain numeric values (of some size)



ASCRIBED MEANING

LLVM MEMORY

3GL ~~LANGUAGE~~ NOTIONS ARE
SIMULATED THROUGH CONVENTION

Functions

Variables

Complex data types (arrays, structs, classes)

0x1	0x2	0x3	0x4
-----	-----	-----	-----



ABSTRACTING MEMORY

LLVM MEMORY

ENCODES THE CONCEPTS OF LOCAL AND GLOBAL MEMORY

Local memory: within a function activation

Global memory: static in the data section

Notably absent: heap memory

With infinite registers,
Why have local memory?

Because we might take the address of a local

```
int main(){  
    int a;  
    int * p;  
    p = &a; (p points to a)  
}
```

ALLOCATION

LLVM MEMORY

ALLOCATING GLOBAL MEMORY

```
@glb1 = global i32 2, align 4  
@cnst2 = constant i32 3, align 4
```

ALLOCATING LOCAL MEMORY

```
%reg = alloca i64, align 8
```

Note: some architectures either require or suggest (for speed)
that memory be type aligned, e.g.:

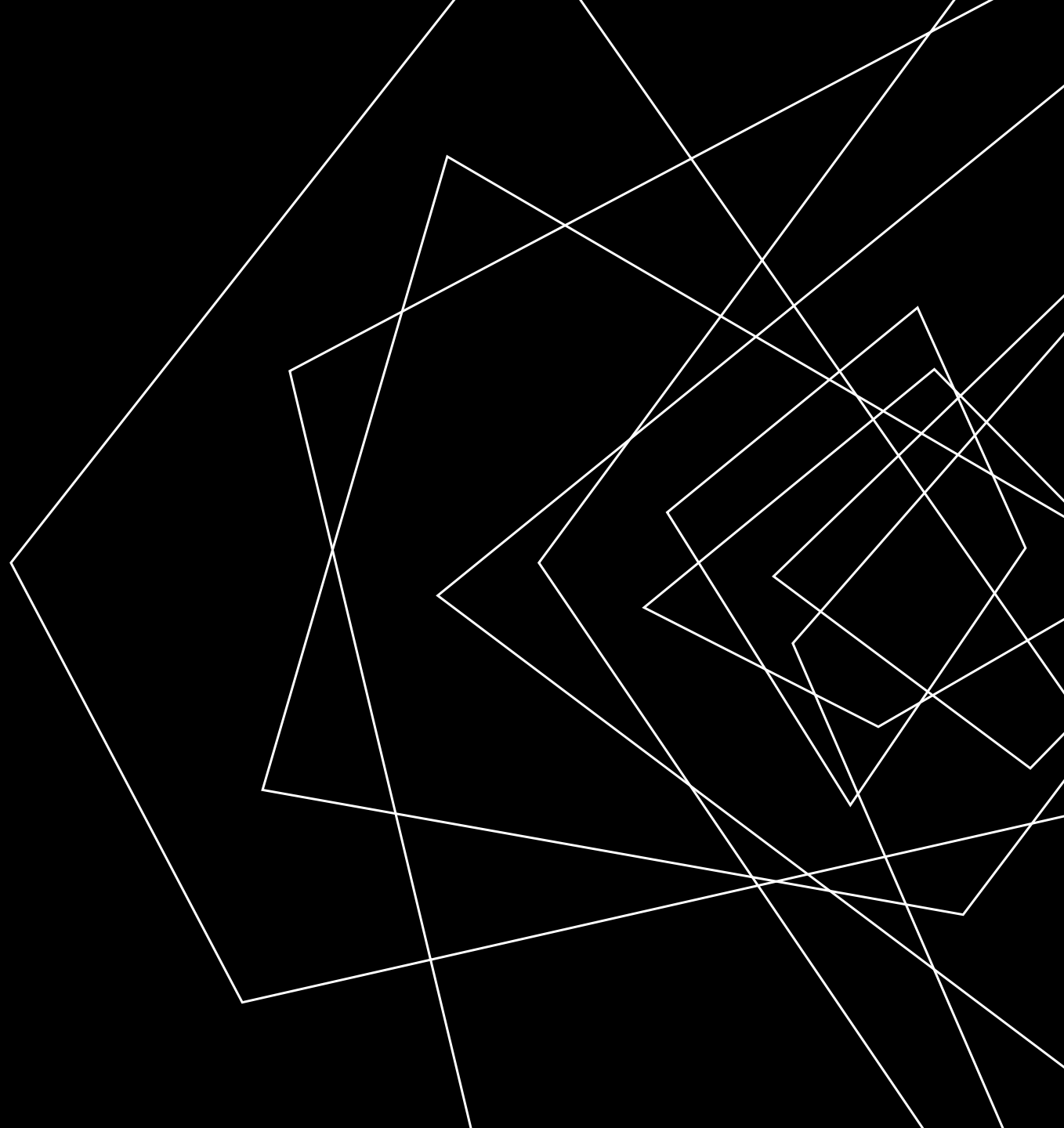
A 4-byte type (like i32) is allocated in a memory address that is a multiple of 4

An 8-byte type (like i64) is allocated in a memory address that is a multiple of 8

To enforce this requirement, allocation can use the align <Num> argument

LECTURE OUTLINE

- LLVM Memory
- Load/Store
- The dreaded GEP



POINTER TYPES

LLVM MEMORY

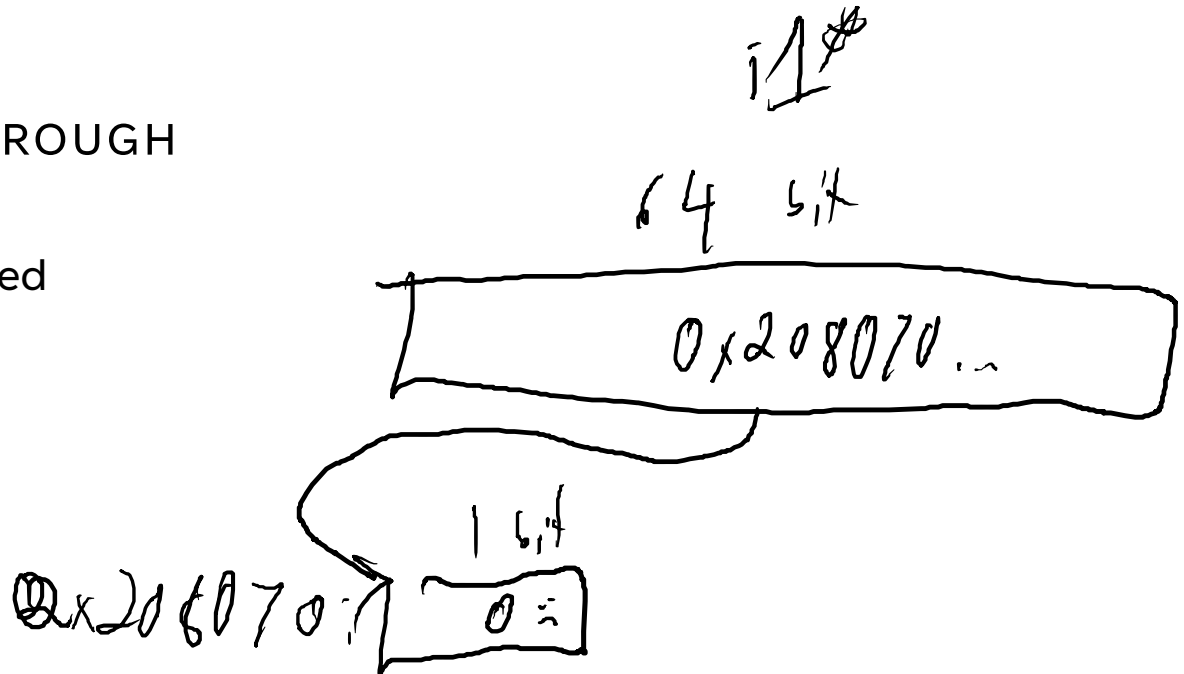
MEMORY LOCATIONS ARE ACCESSED THROUGH POINTERS

Numeric types whose values are memory addressed

A pointer to a 32-bit integer has type `i32*`

A pointer to an 8-bit integer has type `i8*`

A pointer to an 8-bit integer has type `i8*`



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

Here, `%reg` has type `i32*` : a pointer type (a pointer that points at an `i32`)

Note, there is a “generic pointer” type that leaves the type being pointed to out

LLVM MEMORY: LOAD AND STORE

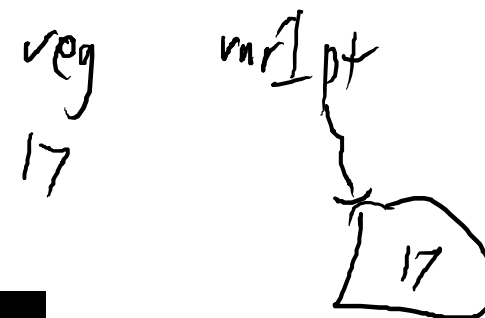
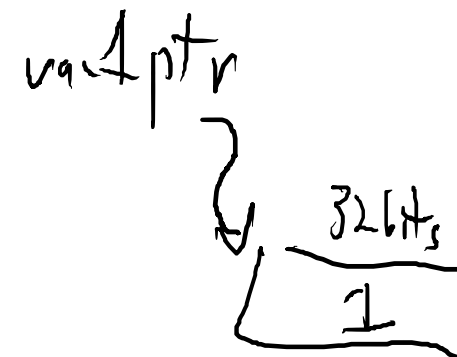
LLVM MEMORY

STORE

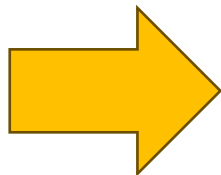
```
store <srcType> <srcOpd>, <dstType> <dstOpd>, align <align>
store    i32          1    , i32*    %var1ptr, align 4
```

LOAD

```
<dstOpd> = load <dstType>, <srcType> <srcOpd>, align <align>
%reg     = load    i32,          i32*    %var1ptr, align 4
```



```
int main(){
    int val;
    val = 12;
    return val;
}
```



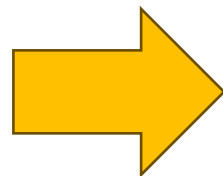
```
define i32 @main(){
    %valptr = alloca i32
    store i32 12, i32* %valptr
    %res = load i32, i32* %valptr
    ret i32 %res
}
```

return val

LLVM MEMORY: GLOBAL MEMORY EXAMPLE

LLVM MEMORY

```
int a = 2;  
int main(){  
    return a;  
}
```



```
@a = global i32 2, align 4  
define i32 @main() {  
    %reg = load i32, i32* @a, align 4  
    ret i32 %reg  
}
```

LLVM MEMORY: LOOK, NO SSA!

LLVM BITCODE

```
define i32 @main() {  
    %valptr = alloca i32  
    store i32 1, i32* %valptr  
    store i32 2, i32* %valptr  
    store i32 3, i32* %valptr  
    %res = load i32, i32* %valptr  
    ret i32 %res  
}
```



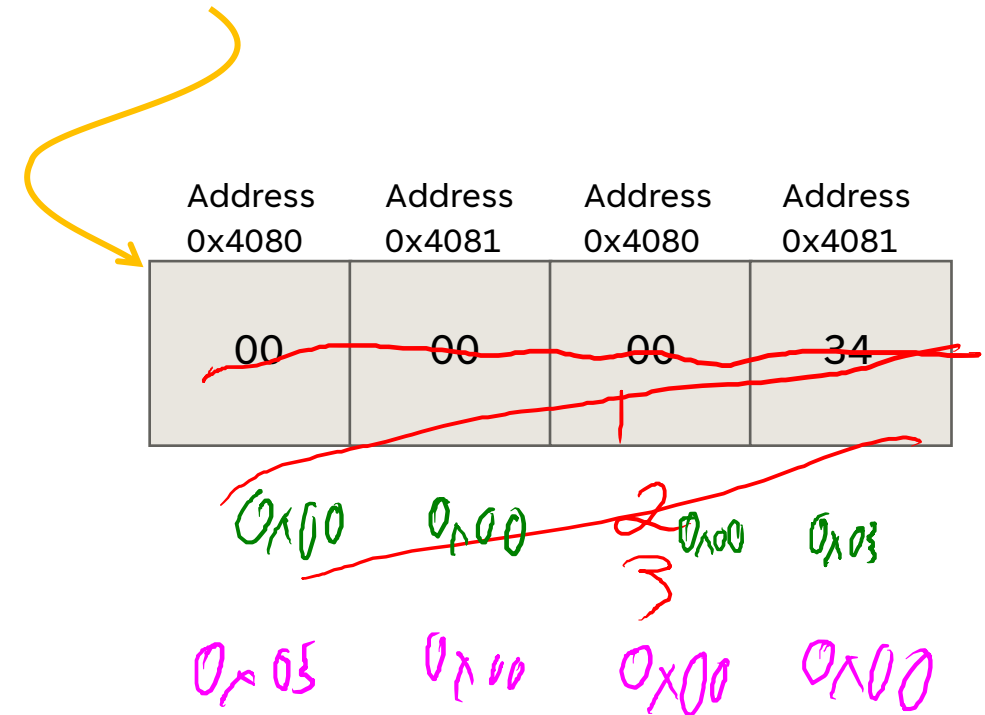
*The VALUE OF the register doesn't change
The VALUE AT the register is what changes!*

LLVM MEMORY: LOOK, NO SSA!

LLVM BITCODE

```
define i32 @main() {
  %valptr = alloca i32
  store i32 1, i32* %valptr
  store i32 2, i32* %valptr
  store i32 3, i32* %valptr
  %res = load i32, i32* %valptr
  ret i32 %res
}
```

%valptr: 0x4080



LLVM MEMORY: AGGREGATE TYPES

LLVM BITCODE

RECALL THAT BITCODE IS A TYPED LANGUAGE

Declare an aggregate type (think struct)

```
%Point = type { i32, i32 }
```

Allocate an aggregate type

```
%ptr = alloca %Point, align 4
```

Allocate an array

```
%arrayptr = alloca [8 x i32], align 16
```

Allocate a struct with an array in it

```
%ArrSize8 = type [8 x i32]
```

```
%struct = type {i32, %ArrSize8}
```

LLVM MEMORY: ACCESSING AGGREGATE MEMORY

LLVM BITCODE

AT THIS POINT, WE NEED TO DISCUSS HOW TO READ AN ARRAY INDEX OR FIELD

There is a powerful, but somewhat complicated instruction to do it, called `getelementptr` (GEP)

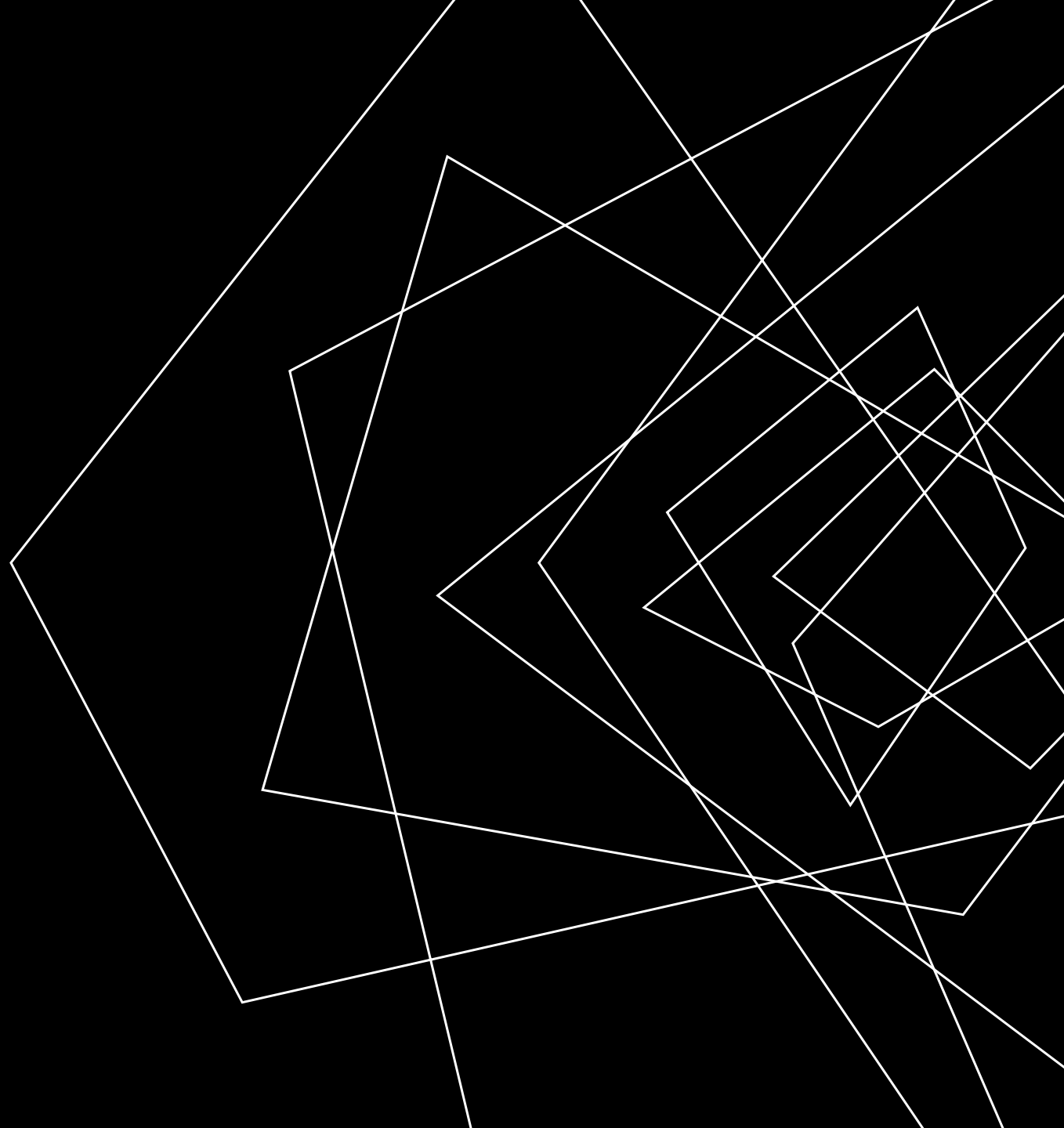
Information

Relationship Status:
It's Complicated

GEP never actually reads memory, it just computes what the offset from a base location would be

LECTURE OUTLINE

- LLVM Memory
- Load/Store
- The dreaded GEP



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

base
array index
Field traversal

```
<result> = getelementptr <tywork>, <tysrc> <src>, <idxtype> <siblingidx>, [<idxtype> <fieldidx>]+
```

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: `<tywork>`: Specify the type of the somethings
- Arg 2: `<tysrc> <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

`<result> = getelementptr <tywork>, base array index Field traversal`

`<tysrc> <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

$\text{<result> = getelementptr <ty_{work}>, \overset{\text{base}}{\text{<ty_{src}> <src>}, \overset{\text{array index}}{\text{<idxtype> <siblingidx>}, \overset{\text{Field traversal}}{[\text{<idxtype> <fieldidx>}] +}}$

7	7	7
---	---	---

```
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: PICTORIALY

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,
```

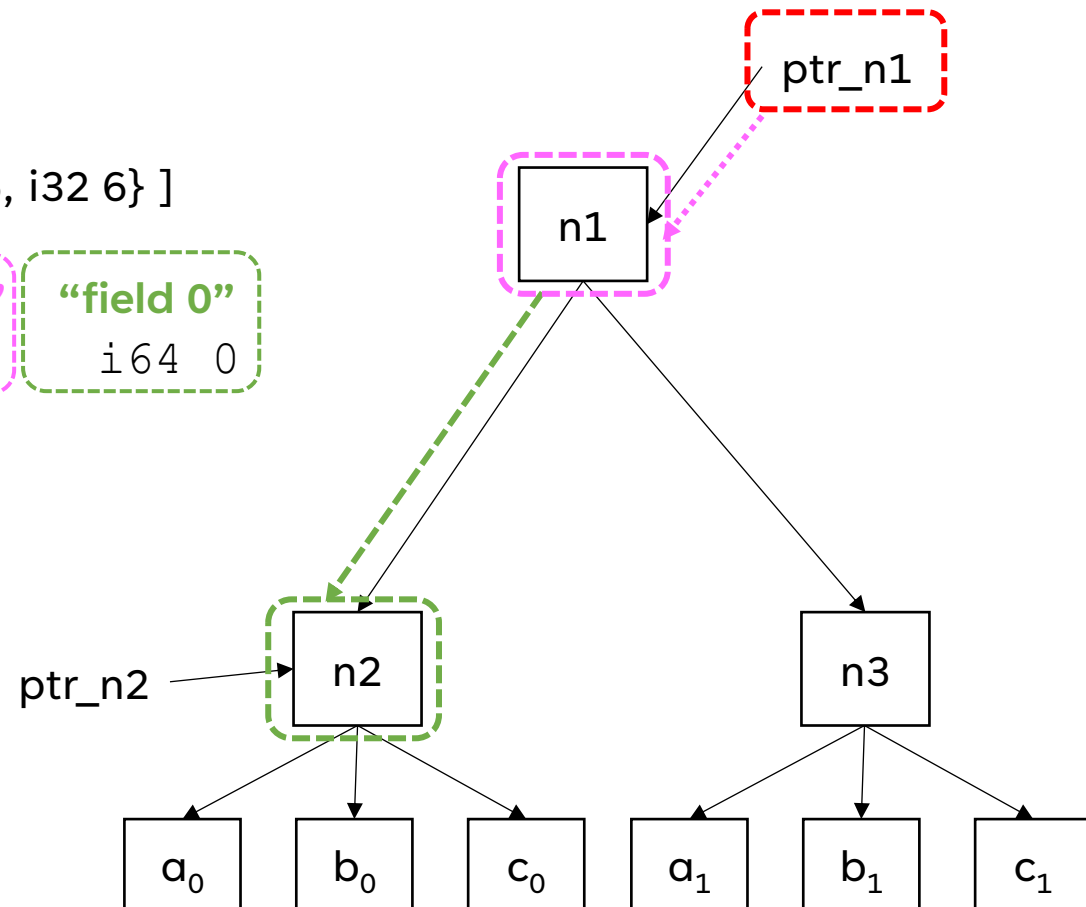
“base addr”

“sibling 0”

“field 0”

i64 0,

i64 0



GETELEMENTPTR: PICTORIALY

THE DREADED GEP

Can be helpful to walk through memory as a tree

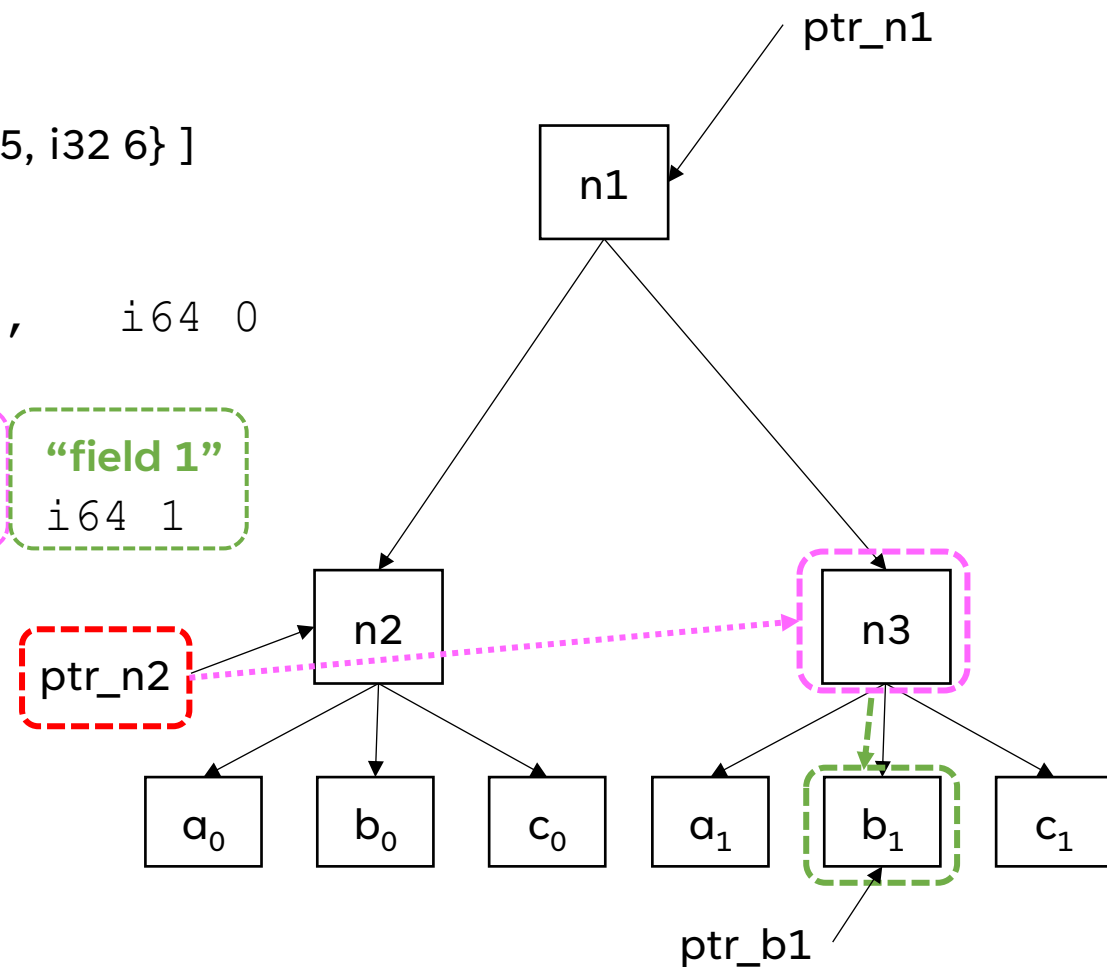
```
%T1 = type { i32, i32, i32 }
```

```
%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
```



GETELEMENTPTR: PICTORIALY

THE DREADED GEP

Can be helpful to walk through memory as a tree

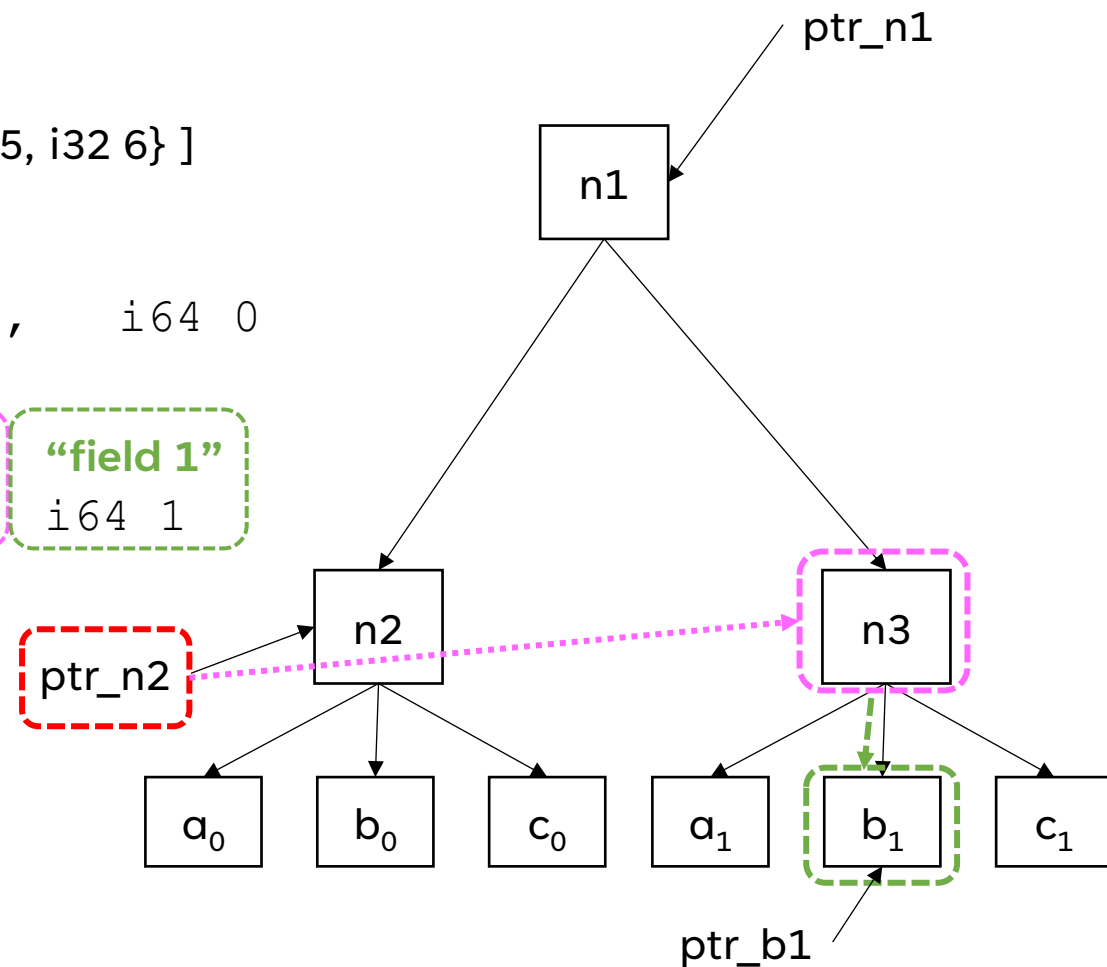
```
%T1 = type { i32, i32, i32 }
```

```
%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
```



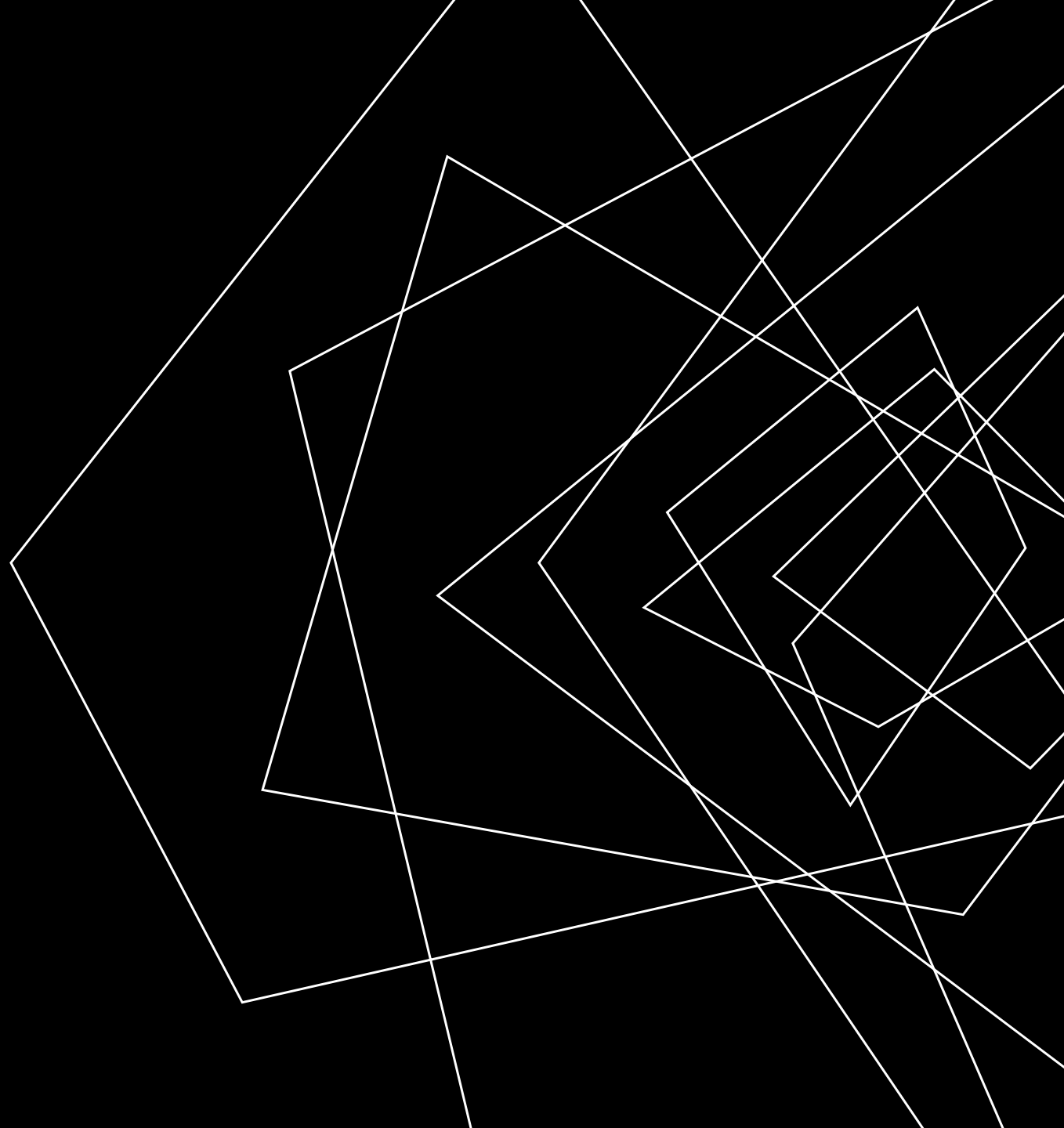
GETELEMENTPTR: YA GOTTA HANDLE C

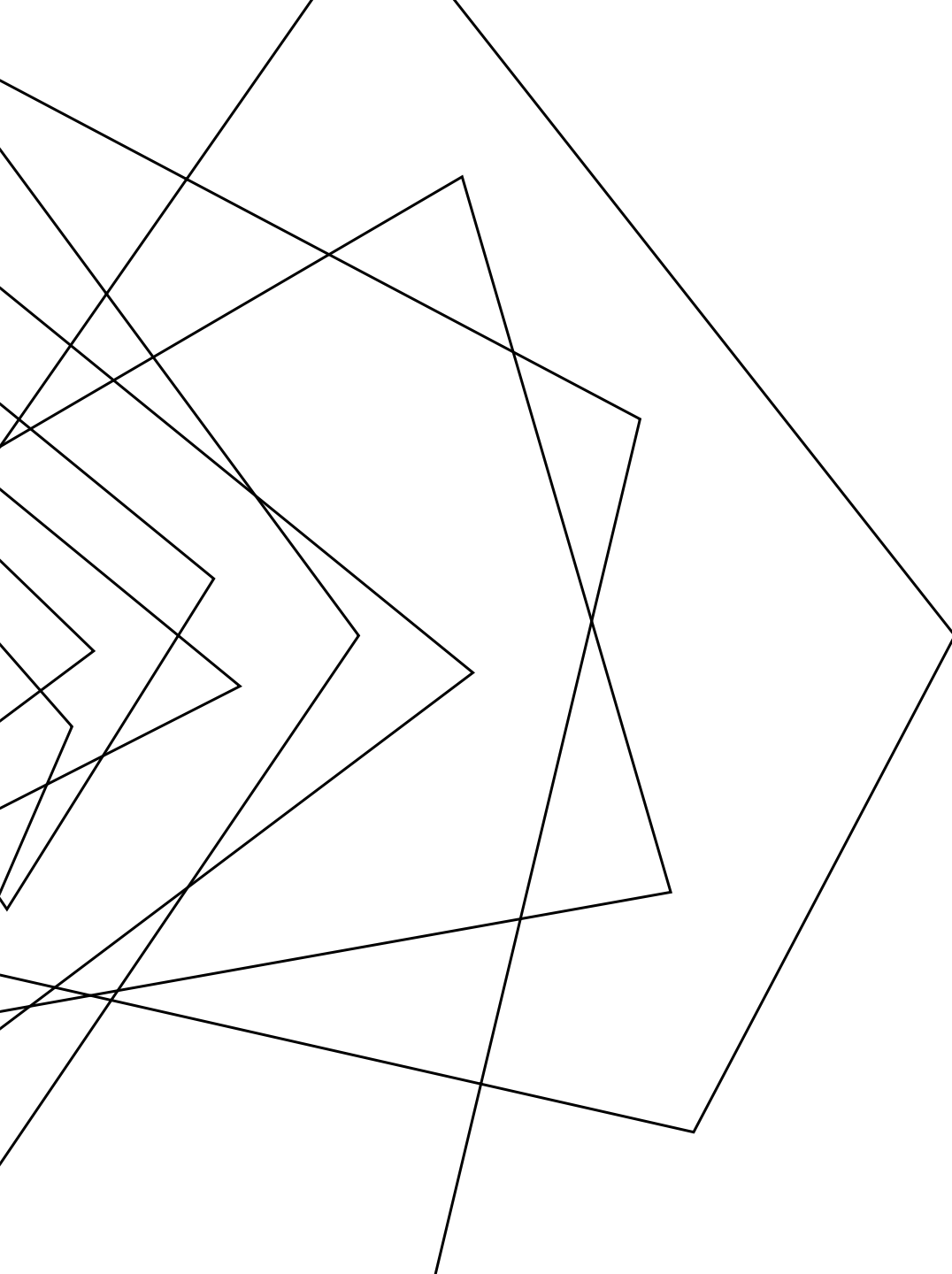
THE DREADED GEP

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';  
}
```

WRAP-UP





NEXT TIME

A COUPLE MORE BITCODE FEATURES

DESCRIBE THE FIRST PROJECT