LLVM* IR TUTORIAL - PHIS, GEPS AND OTHER THINGS, OH MY!

LLVM Developers Conference – Brussels 2019

Vince Bridgers

Felipe de Azevedo Piovezan

About this tutorial

Assume **no** previous Intermediate Representation (IR) knowledge.

But this is not a lecture about compiler theory!

After the tutorial, you should:

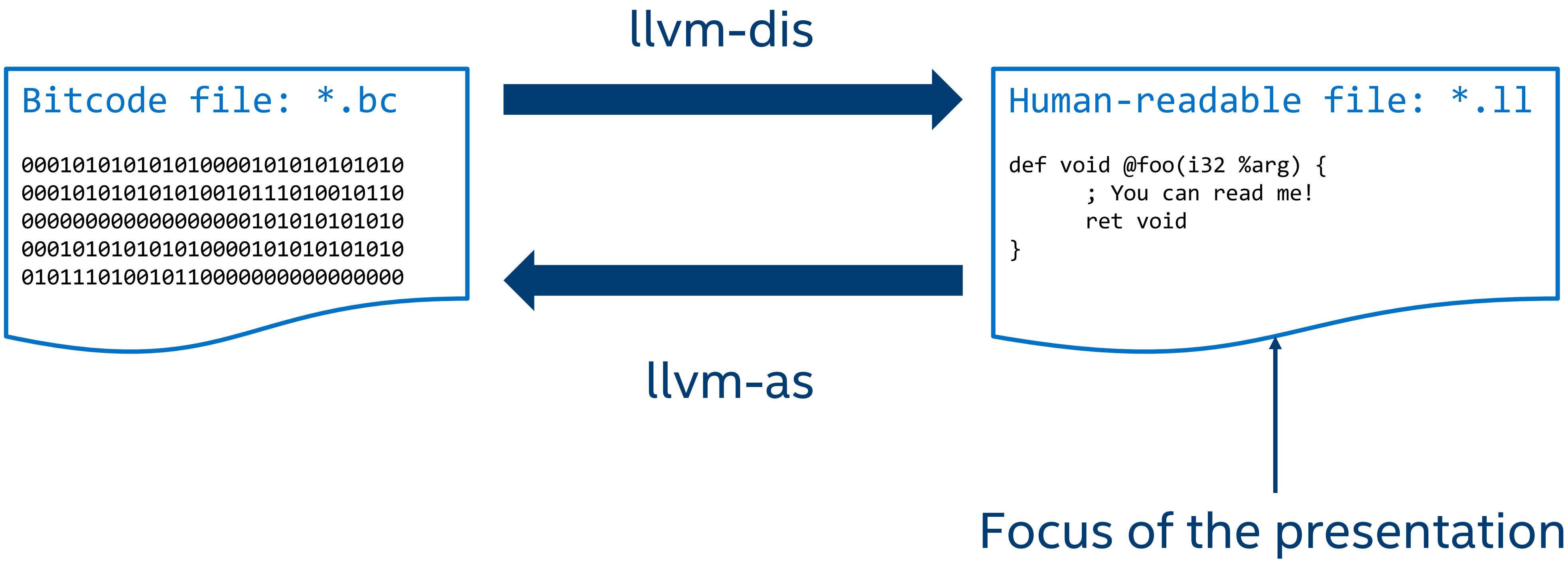
- Understand common LLVM tools.
- Be able to write simple IR.
- Be able to understand the language reference.
  - Use it to inspect compiler-generated IR.

What is the LLVM IR?

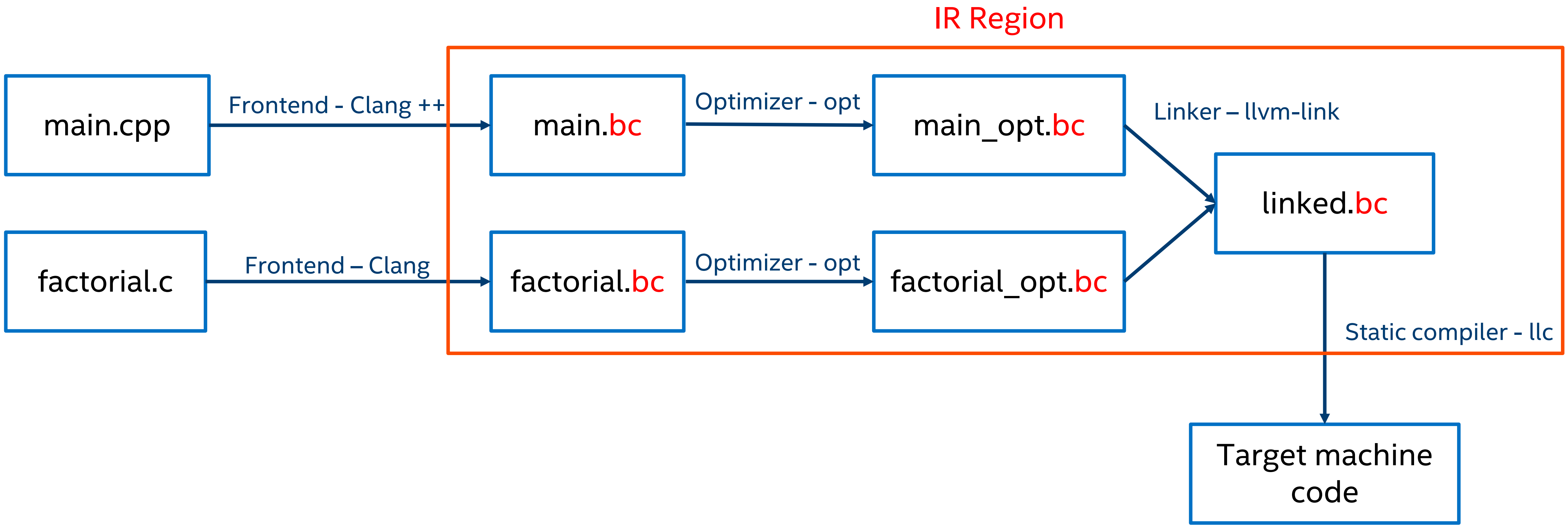
The LLVM **I**ntermediate **R**epresentation:

- Is a low level programming language
  - RISC-like instruction set
- … while being able to represent high-level ideas.
  - i.e. high-level languages can map cleanly to IR.
- Enables efficient code optimization

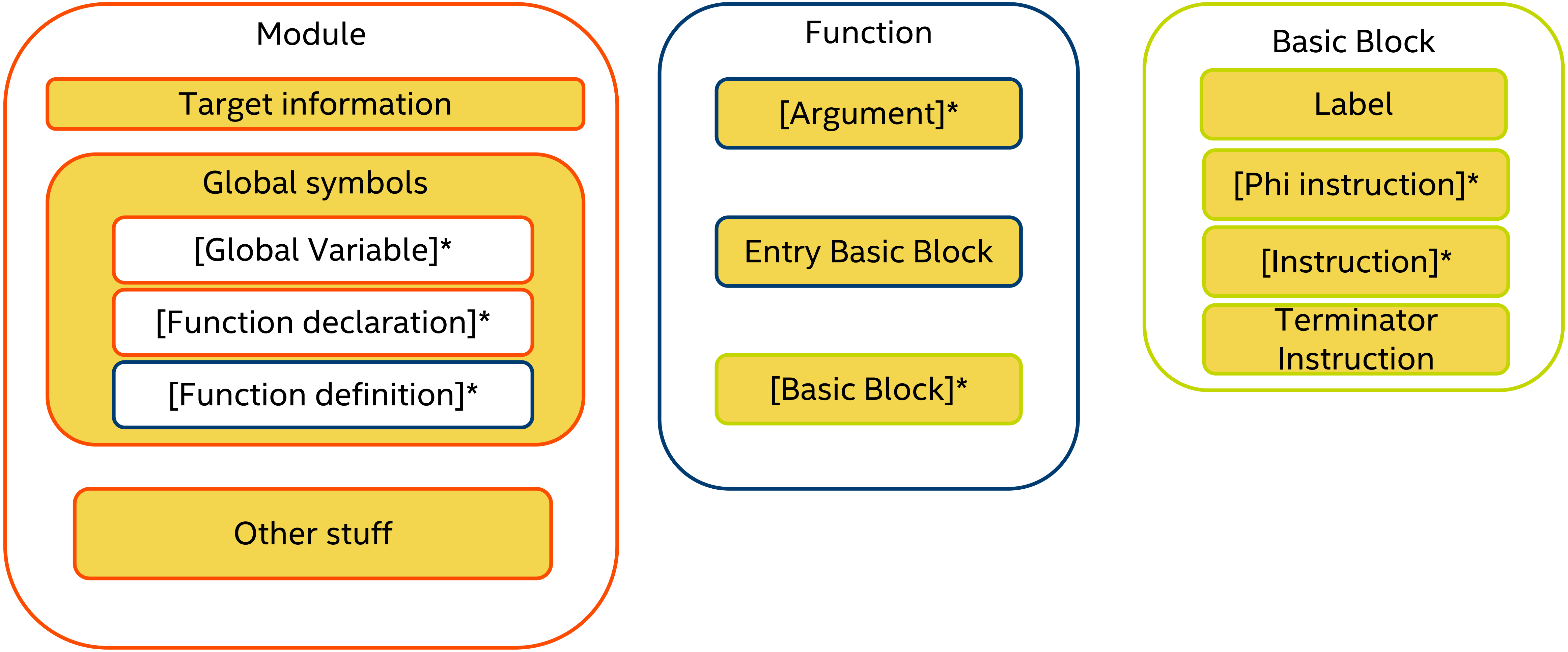
IR representation



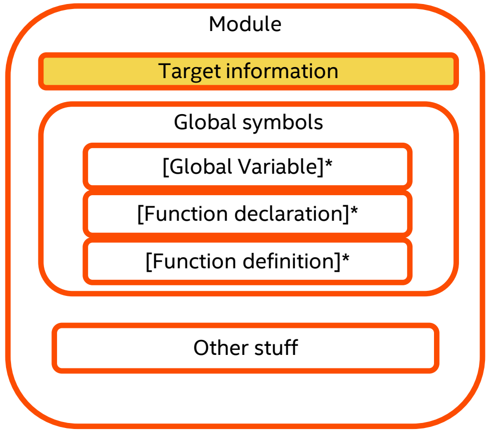
IR & the compilation process



Simplified IR layout



Target information



An IR module usually starts with a pair of strings describing the target:

ELF mangling

Little endian

ABI alignment of i64

Native integer widths

```
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
```

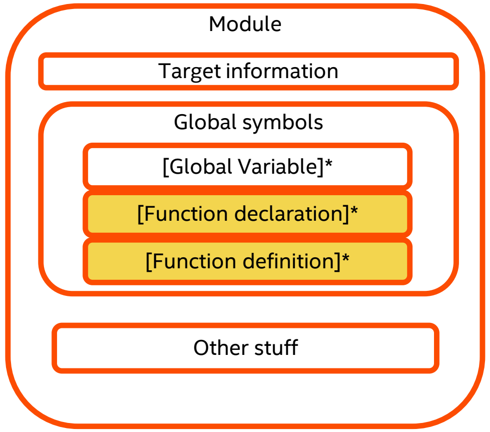
Architecture

Vendor

System

ABI

A basic main program



Hand-written IR for this program:

```c
int factorial(int val);

int main(int argc, char** argv)
{
  return factorial(2) * 7 == 42;
}
```

```llvm
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

% Virtual Registers %

Those are “local” variables.

Two flavors of names:

- Unnamed: %<number>

- Named: %<name>

“LLVM IR has infinite registers”

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

Types, types everywhere!

Very much a typed language.

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

Types, types everywhere!

Very much a typed language.

```
        i32            i32

       i32       i32         i8**       ) {
            i32            i32
           i32
               i32
                 i32      i32
      i32
}
```

Types, types everywhere!

The instructions explicitly dictate the types expected.

Easy to figure out argument types.

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

Types, types everywhere!

The instructions explicitly dictate the types expected.

Easy to figure out argument types.

Easy to figure out return types (mostly)

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

Types, types everywhere!

No implicit conversions!

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

Types, types everywhere!

No implicit conversions!

To check if this is valid IR:

opt –verify input.ll

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42

  ret i32 %3
}
```

```
opt: test.ll:8:11: error: '%3' defined with type 'i1' but expected 'i32'
```

The LangRef is your friend

Instructions often have **many** variants.

What else could a call instruction possibly need?

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

The LangRef is your friend

‘call’ Instruction

Syntax: `%1 = call i32 @factorial(i32 2)`

```
<result> = [tail | musttail | notail ] call [fast-math flags] [cconv] [ret attrs] [addrspace(<num>)]
           [<ty>|<fnty> <fnptrval>(<function args>) [fn attrs] [ operand bundles ]
```

Overview:

The ‘`call`’ instruction represents a simple function call.

Arguments:

This instruction requires several arguments:

The LangRef is your friend

Semantics:

The ‘`call`’ instruction is used to cause control flow to transfer to a specified function, with its incoming arguments bound to the specified values. Upon a ‘`ret`’ instruction in the called function, control flow continues with the instruction after the function call, and the return value of the function is bound to the result argument.

Example:

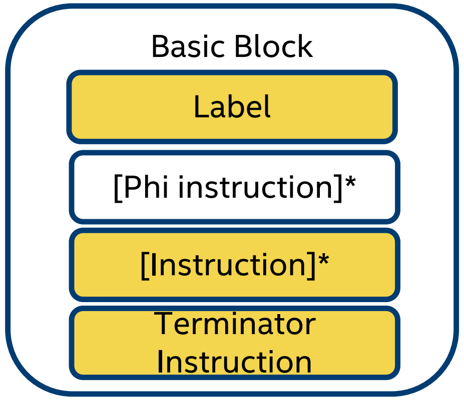
```
%retval = call i32 @test(i32 %argc)
call i32 (i8*, ...)* @printf(i8* %msg, i32 12, i8 42)        ; yields i32
%X = tail call i32 @foo()                                    ; yields i32
%Y = tail call fastcc i32 @foo()  ; yields i32
call void %foo(i8 97 signext)

%struct.A = type { i32, i8 }
%r = call %struct.A @foo()                        ; yields { i32, i8 }
%gr = extractvalue %struct.A %r, 0                ; yields i32
%gr1 = extractvalue %struct.A %r, 1               ; yields i8
%Z = call void @foo() noreturn                    ; indicates that %foo never returns normally
%ZZ = call zeroext i32 @bar()                     ; Return value is %zero extended
```

Recursive factorial

```c
// Precondition: val is non-negative.
int factorial(int val) {
  if (val == 0)
    return 1;
  return val * factorial(val - 1);
}
```

```llvm
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_case
base_case:
  ret i32 1
recursive_case:
  %1 = add i32 -1, %val
  %2 = call i32 @factorial(i32 %0)
  %3 = mul i32 %val, %1
  ret i32 %2
}
```

Basic Blocks

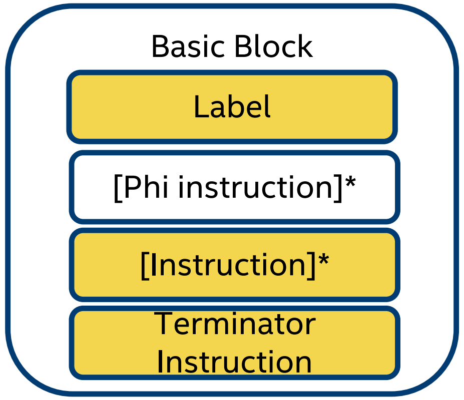


List of non-terminator instructions ending with a terminator instruction:

- **Branch - "br"**
- **Return - "ret"**
- Switch – "switch"
- Unreachable – "unreachable"
- Exception handling instructions

```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_case
base_case:
  ret i32 1
recursive_case:
  %1 = add i32 -1, %val
  %2 = call i32 @factorial(i32 %0)
  %3 = mul i32 %val, %1
  ret i32 %2
}
```

Basic Blocks



List of non-terminator instructions ending with a terminator instruction:

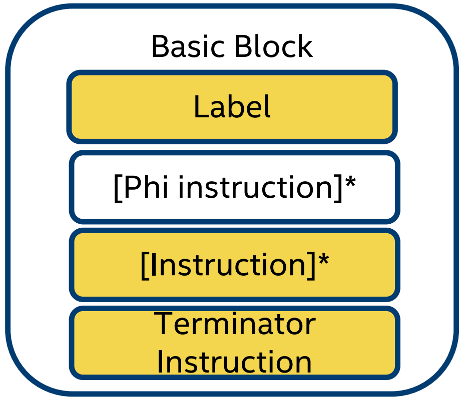
- Return - “ret”

Execution proceeds to:

- calling function

```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_case
base_case:
  ret i32 1
recursive_case:
  %1 = add i32 -1, %val
  %2 = call i32 @factorial(i32 %0)
  %3 = mul i32 %val, %1
  ret i32 %2
}
```

Basic Blocks



List of non-terminator instructions ending with a terminator instruction:

- Branch - “br”

Execution proceeds to:

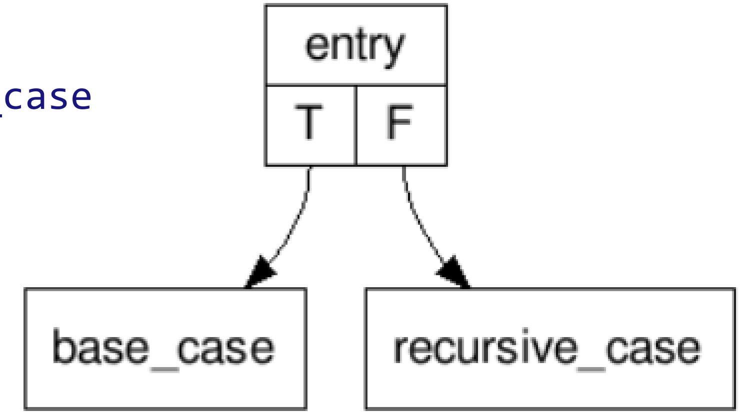
- another Basic Block
  - It’s **successor**!

```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_case
base_case:
  ret i32 1
recursive_case:
  %1 = add i32 -1, %val
  %2 = call i32 @factorial(i32 %0)
  %3 = mul i32 %val, %1
  ret i32 %2
}
```

Control Flow Graph (CFG)

```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
entry:
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_case
base_case:                                 ; preds = %entry
  ret i32 1
recursive_case:                            ; preds = %entry
  %0 = add i32 -1, %val
  %1 = call i32 @factorial(i32 %0)
  %2 = mul i32 %val, %1
  ret i32 %2
}
```

Automatically generated comments



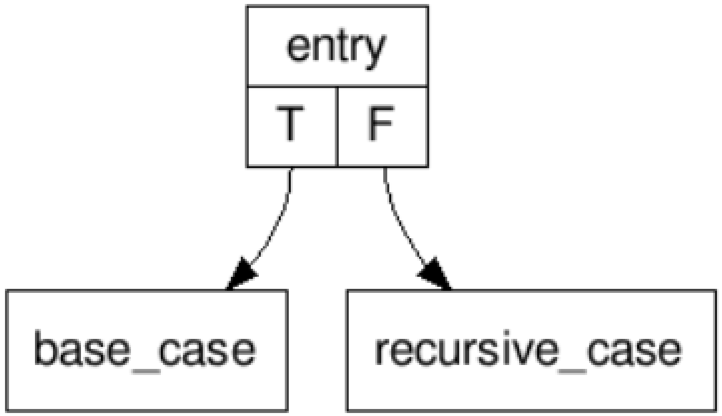
CFG for 'ir_factorial' function

Control Flow Graph (CFG)

The optimizer can generate the CFG in dot format:

opt –analyze –dot-cfg-**only** <input.ll>

-dot-cfg-only = Generate .dot files. Don’t include instructions.



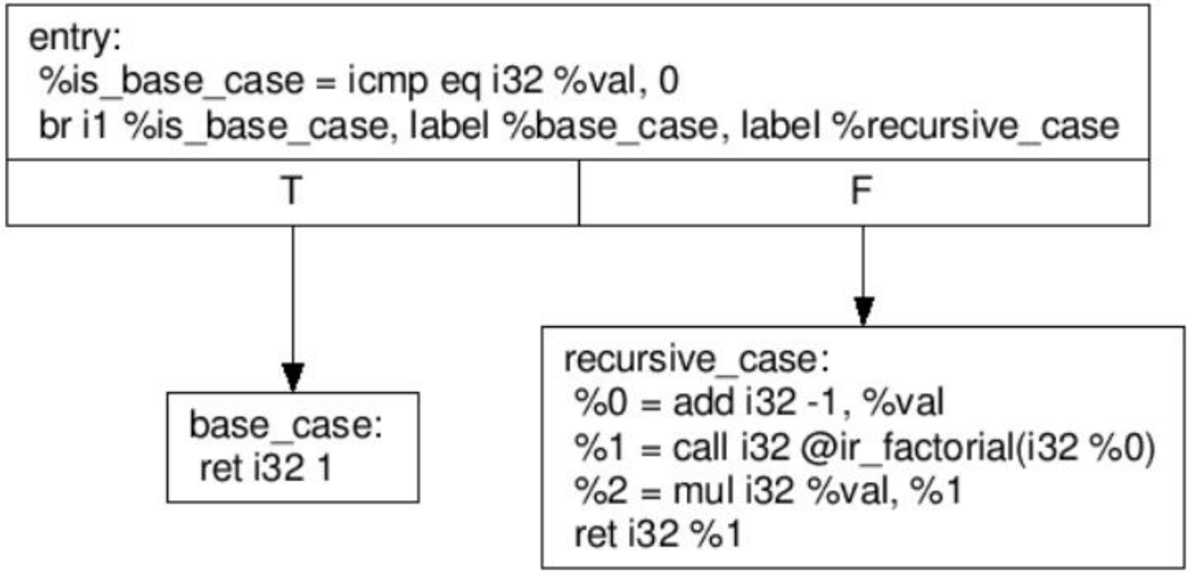
CFG for 'ir_factorial' function

Control Flow Graph (CFG)

The optimizer can generate the CFG in dot format:

opt –analyze –dot-cfg <input.ll>

-dot-cfg = Generate .dot files.



CFG for 'ir_factorial' function

Implicit labels

Every Basic Block has a label…

… even if it's not explicit

Implicit labels

Every Basic Block has a label…

… even if it's not explicit

```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
entry:
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_case
base_case:
  ret i32 1
recursive_case:
  %0 = add i32 -1, %val
  %1 = call i32 @factorial(i32 %0)
  %2 = mul i32 %val, %1
  ret i32 %2
}
```

Implicit labels

Every Basic Block has a label…

… even if it's not explicit

```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %0: ; Implicit label!
  %is_base_case = icmp eq i32 %val, 0
```

```
opt: ir_implementation.ll:11:3: error: instruction expected to be numbered '%1'
  %0 = add i32 -1, %val
  ^
```

```
recursive_case:
  %0 = add i32 -1, %val
  %1 = call i32 @factorial(i32 %0)
  %2 = mul i32 %val, %1
  ret i32 %2
}
```

Implicit labels

Every Basic Block has a label…

… even if it's not explicit

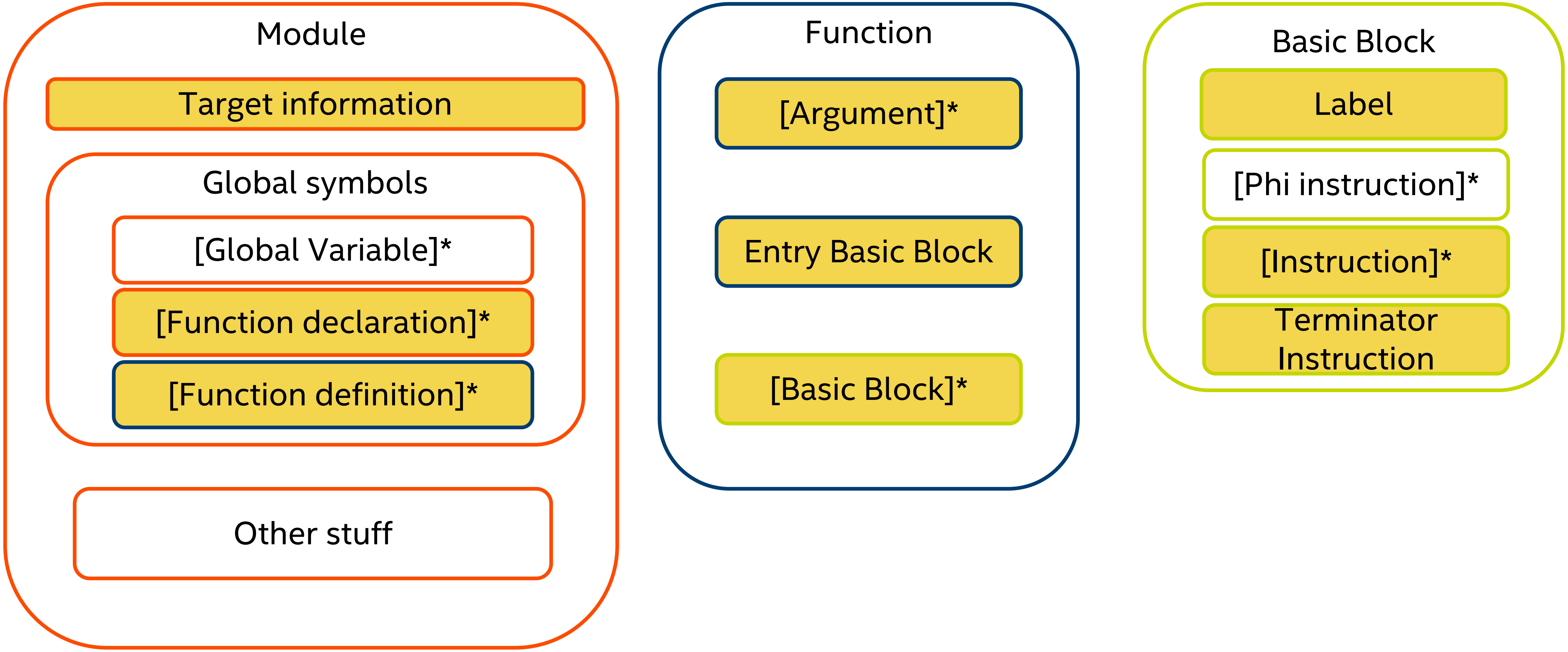
```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %0: ; Implicit label!
  %is_base_case = icmp eq i32 %val, 0
```

```
opt: ir_implementation.ll:11:3: error: instruction expected to be numbered '%1'
  %0 = add i32 -1, %val
  ^
```

```
recursive_case:
  %1 = add i32 -1, %val
  %2 = call i32 @factorial(i32 %1)
  %3 = mul i32 %val, %2
  ret i32 %3
}
```

And the same is true for function arguments!

Simplified IR layout



Iterative factorial

```
int factorial(int val) {
  int temp = 1;
  for (int i = 2; i <= val; ++i)
    temp *= i;
  return temp;
}
```

You wish you could do this...

```
define i32 @factorial(i32 %val) {
entry:
  %i = add i32 0, 2
  %temp = add i32 0, 1
  br label %check_for_condition
check_for_condition:

  %i_leq_val = icmp sle i32 %i, %val
  br i1 %i_leq_val, label %for_body, label %end_loop
for_body:

  %temp = mul i32 %temp, %i
  %i = add i32 %i, 1
  br label %check_for_condition
end_loop:
  ret i32 %temp
}
```

Iterative factorial

```c
int factorial(int val) {
  int temp = 1;
  for (int i = 2; i <= val; ++i)
    temp *= i;
  return temp;
}
```

```llvm
define i32 @factorial(i32 %val) {
entry:
  %i = add i32 0, 2
  %temp = add i32 0, 1
  br label %check_for_condition
check_for_condition:

  %i_leq_val = icmp sle i32 %i, %val
  br i1 %i_leq_val, label %for_body, label %end_loop
for_body:
```

You wish you could do this…

```llvm
  %temp = mul i32 %temp, %i
  %i = add i32 %i, 1
```

```
opt: test.ll:12:5: error: multiple definition of local value named 'temp'
    %temp = mul i32 %temp, %i
    ^
```

```llvm
}
```

Static Single Assignment (SSA)

Every variable is assigned *exactly* once.

Every variable is defined before it is used.

Iterative factorial

```c
int factorial(int val) {
  int temp = 1;
  for (int i = 2; i <= val; ++i)
    temp *= i;
  return temp;
}
```

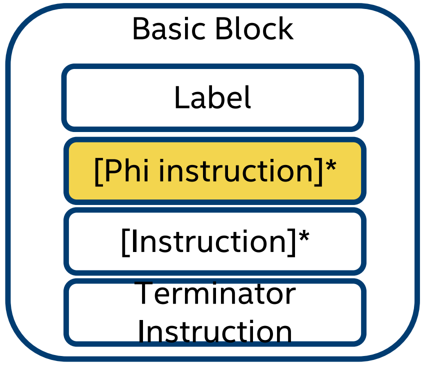
```llvm
define i32 @factorial(i32 %val) {
entry:
  %i = add i32 0, 2
  %temp = add i32 0, 1
  br label %check_for_condition
check_for_condition:

  %i_leq_val = icmp sle i32 %i, %val
  br i1 %i_leq_val, label %for_body, label %end_loop
for_body:

  %new_temp = mul i32 %temp, %i
  %i_plus_one = add i32 %i, 1
  br label %check_for_condition
end_loop:
  ret i32 %temp
}
```

Now *%i* is always *2!*

So you do this:

Now *%temp* is always *1!*

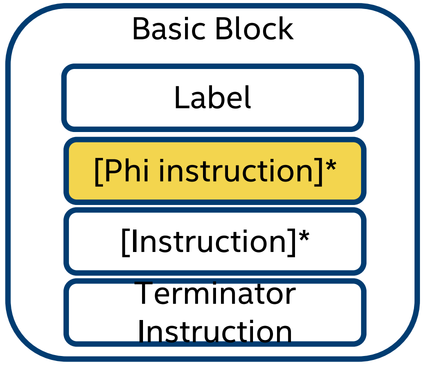
Phis to the rescue!




```
<result> = phi <ty> [                    ], [                    ] …
```

Select a value based on the **BasicBlock** that executed previously!

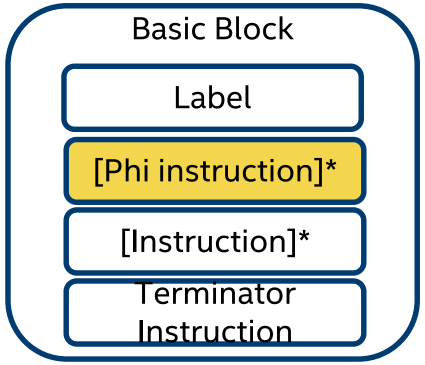
Phis to the rescue!




```
<result> = phi <ty> [<val0>, <label0>], [                    ] …
```

Select a value based on the **BasicBlock** that executed previously!

Phis to the rescue!




```
<result> = phi <ty> [<val0>, <label0>], [<val1>, <label1>] …
```

Select a value based on the **BasicBlock** that executed previously!

Phis to the rescue!

Basic Block
- Label
- [Phi instruction]*
- [Instruction]*
- Terminator Instruction

```
entry:
 %i = add i32 0, 2
 %temp = add i32 0, 1
 br label %check_for_condition
```

```
check_for_condition:


 %i_leq_val = icmp sle i32 %i, %val
 br i1 %i_leq_val, label %for_body, label %end_loop
```

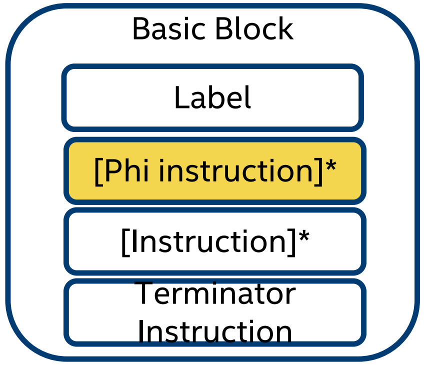
True

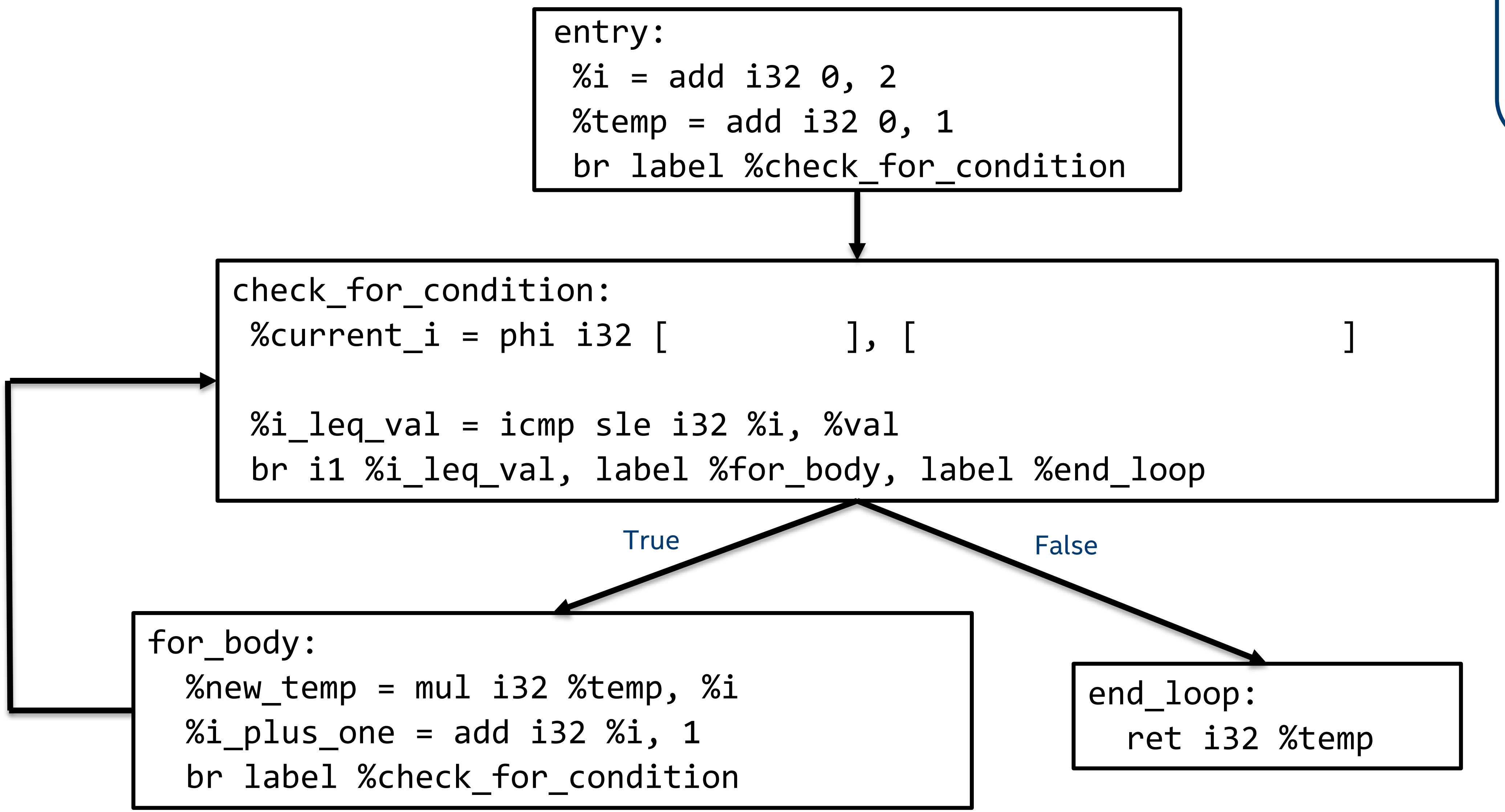
False

```
for_body:
  %new_temp = mul i32 %temp, %i
  %i_plus_one = add i32 %i, 1
  br label %check_for_condition
```

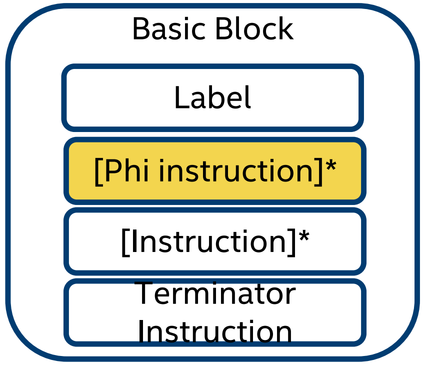
```
end_loop:
  ret i32 %temp
```

Phis to the rescue!





Phis to the rescue!




```
entry:
 %i = add i32 0, 2
 %temp = add i32 0, 1
 br label %check_for_condition
```

```
check_for_condition:
 %current_i = phi i32 [2, %entry], [                        ]

 %i_leq_val = icmp sle i32 %i, %val
 br i1 %i_leq_val, label %for_body, label %end_loop
```

True

False

```
for_body:
  %new_temp = mul i32 %temp, %i
  %i_plus_one = add i32 %i, 1
  br label %check_for_condition
```

```
end_loop:
  ret i32 %temp
```

Phis to the rescue!

Basic Block
- Label
- [Phi instruction]*
- [Instruction]*
- Terminator Instruction

```
entry:
 %i = add i32 0, 2
 %temp = add i32 0, 1
 br label %check_for_condition
```

```
check_for_condition:
 %current_i = phi i32 [2, %entry], [%i_plus_one, %for_body]

 %i_leq_val = icmp sle i32 %i, %val
 br i1 %i_leq_val, label %for_body, label %end_loop
```

True

```
for_body:
  %new_temp = mul i32 %temp, %i
  %i_plus_one = add i32 %i, 1
  br label %check_for_condition
```

False

```
end_loop:
  ret i32 %temp
```

Basic Block

- Label
- [Phi instruction]*
- [Instruction]*
- Terminator Instruction

Phis to the rescue!

```
entry:
 %i = add i32 0, 2
 %temp = add i32 0, 1
 br label %check_for_condition
```

```
check_for_condition:
 %current_i = phi i32 [2, %entry], [%i_plus_one, %for_body]

 %i_leq_val = icmp sle i32 %current_i, %val
 br i1 %i_leq_val, label %for_body, label %end_loop
```

True

False

```
for_body:
  %new_temp = mul i32 %temp, %current_i
  %i_plus_one = add i32 %current_i, 1
  br label %check_for_condition
```

```
end_loop:
  ret i32 %temp
```

Phis to the rescue!

Basic Block
- Label
- [Phi instruction]*
- [Instruction]*
- Terminator Instruction

```
entry:

 %temp = add i32 0, 1
 br label %check_for_condition
```

```
check_for_condition:
 %current_i = phi i32 [2, %entry], [%i_plus_one, %for_body]

 %i_leq_val = icmp sle i32 %current_i, %val
 br i1 %i_leq_val, label %for_body, label %end_loop
```

True

```
for_body:
  %new_temp = mul i32 %temp, %current_i
  %i_plus_one = add i32 %current_i, 1
  br label %check_for_condition
```

False

```
end_loop:
  ret i32 %temp
```

Phis to the rescue!

Basic Block

- Label
- [Phi instruction]*
- [Instruction]*
- Terminator Instruction

```
entry:

 br label %check_for_condition
```

```
check_for_condition:
 %current_i = phi i32 [2, %entry], [%i_plus_one, %for_body]
 %temp      = phi i32 [1, %entry], [%new_temp, %for_ body]
 %i_leq_val = icmp sle i32 %current_i, %val
 br i1 %i_leq_val, label %for_body, label %end_loop
```

True

False

```
for_body:
  %new_temp = mul i32 %temp, %current_i
  %i_plus_one = add i32 %current_i, 1
  br label %check_for_condition
```

```
end_loop:
  ret i32 %temp
```

Another way to cheat SSA

Frontend generates something different!

Gets around SSA restriction by writing to memory.

https://godbolt.org/z/Nlx6T5

(remember to untick the “hide comments” option!)

But the optimized code is similar to what we had (with O1):

https://godbolt.org/z/OirW9y

Another way to cheat SSA

Alloca instruction:

- You give it a type, it gives you a pointer to that type:
  - `%ptr = alloca i32             ; ptr is i32*`
  - `%ptr = alloca <any_type>      ; ptr is <any_type>*`
- Allocates memory on the stack frame of the executing function.
- Automatically released.
  - Akin to changing the stack pointer.
- Plays a big part in generating IR in SSA form.

Allocas to the rescue!

```
entry:
  %i.addr = alloca i32
  %temp.addr = alloca i32
  store i32 2, i32* %i.addr
  store i32 1, i32* %temp.addr
  br label %check_for_condition
```

```
check_for_condition:


  %i_leq_val = icmp sle i32 %current_i, %val
  br i1 %i_leq_val, label %for_body, label %end_loop
```

True

False

```
for_body:
  %i_plus_one = add i32 %current_i, 1
  %new_temp = mul i32 %temp, %current_i
  store i32 %i_plus_one, i32* %i.addr
  store i32 %new_temp, i32* %temp.addr
  br label %check_for_condition
```

```
end_loop:
  ret i32 %temp
```

Allocas to the rescue!

```
entry:
  %i.addr = alloca i32
  %temp.addr = alloca i32
  store i32 2, i32* %i.addr
  store i32 1, i32* %temp.addr
  br label %check_for_condition
```

```
check_for_condition:
  %current_i = load i32, i32* %i.addr
  %temp      = load i32, i32* %temp.addr
  %i_leq_val = icmp sle i32 %current_i, %val
  br i1 %i_leq_val, label %for_body, label %end_loop
```

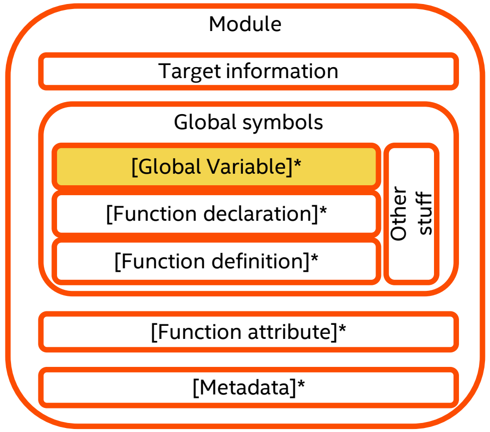
True

False

```
for_body:
  %i_plus_one = add i32 %current_i, 1
  %new_temp = mul i32 %temp, %current_i
  store i32 %i_plus_one, i32* %i.addr
  store i32 %new_temp, i32* %temp.addr
  br label %check_for_condition
```

```
end_loop:
  ret i32 %temp
```

Global variables



Allocas allocate memory for function scopes.

Global variables fill that role for the module in a static way.

- They are always pointers, like the values returned by Allocas.

Global variables

- Name prefixed with "@".
- Must have a type.
- Must be initialized
- Have the global keyword…
- … xor constant (never stored to!)
  – Not to be confused with C++ const

```
@gv =
@gv = i8
@gv = i8 42   ; Declarations excepted.
@gv = global i8 42
@gv = constant i8 42
```

Global variables

Are always pointers

Always **constant pointers!**

```
@gv = global i16 46
; …
; … Inside some function:
%load = load i16 , i16* @gv
store i16 0, i16* @gv
```

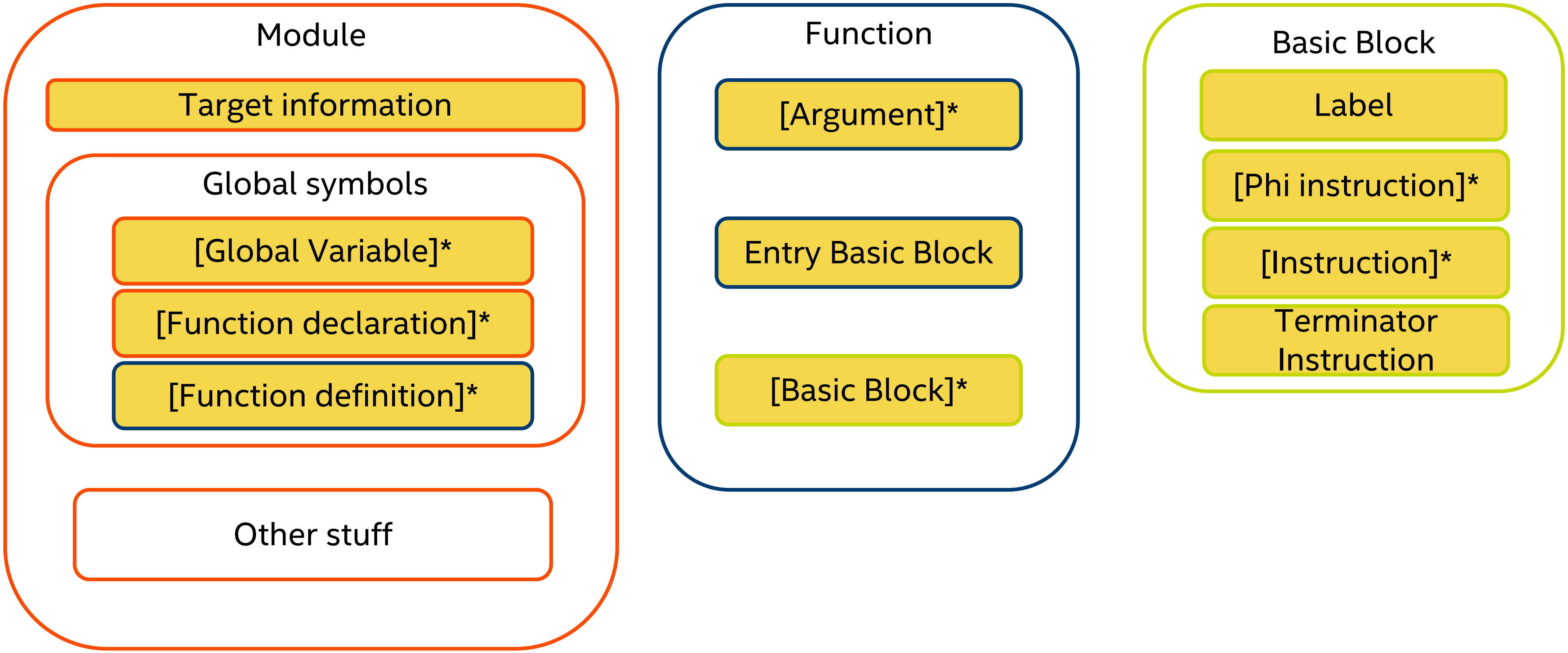
Global variables

Love qualifiers:

```
@<GlobalVarName> = [Linkage] [PreemptionSpecifier] [Visibility]
                   [DLLStorageClass] [ThreadLocal]
                   [(unnamed_addr|local_unnamed_addr)] [AddrSpace]
                   [ExternallyInitialized]
                   <global | constant> <Type> [<InitializerConstant>]
                   [, section "name"] [, comdat [($name)]]
                   [, align <Alignment>] (, !name !N)*
```

Check the language ref.

Simplified IR layout



TYPE SYSTEM AND GEPS!

LLVM’s type system

From the language reference:

- Void Type
- Function Type
- First Class Types
  - Single Value Types
    - Integer Type
    - Floating-Point Types
    - X86_mmx Type
    - Pointer Type
    - Vector Type
  - Label Type
  - Token Type
  - Metadata Type
  - Aggregate Types
    - Array Type
    - Structure Type
    - Opaque Structure Types

Aggregate types: arrays

Defined by:

- A constant size.
- An element type.
- [for GVs] an initializer

```
@array = global [17 x   ]
@array = global [17 x i8]
@array = global [17 x i8] zeroinitializer
```

Accessing arrays & manipulating pointers

The Get Element Pointer (GEP) instruction:

- Provides a way to calculate pointer offsets.
- Abstracts away details like:
  - Size of types
  - Padding inside structs
- Intuitive to use…
  … once you understand a few basic principles.

Manipulating pointers

The Get Element Pointer (GEP) instruction:

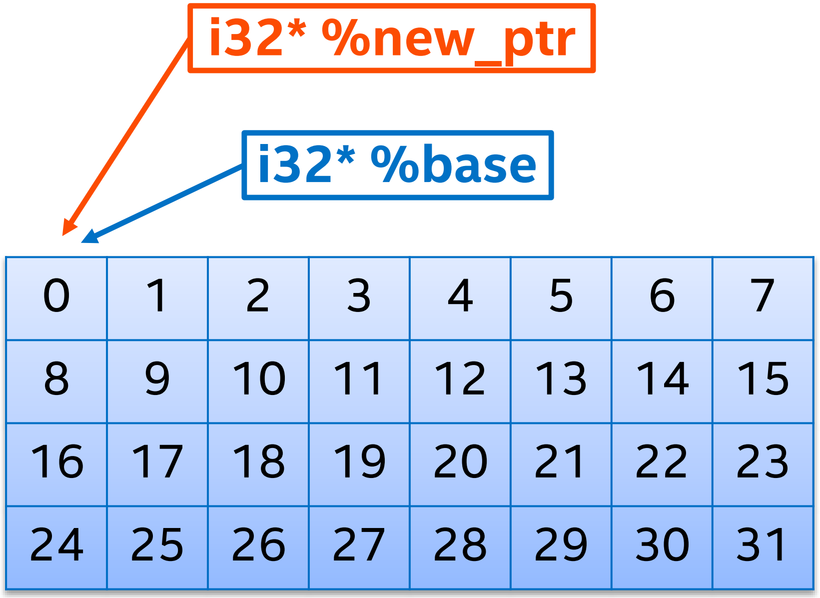
```
<result> = getelementptr <ty>, <ty>* <ptrval>,  [i32 <idx>]+
```

Base type used for the first index

Offsets – one per “dimension”

Base address to start from

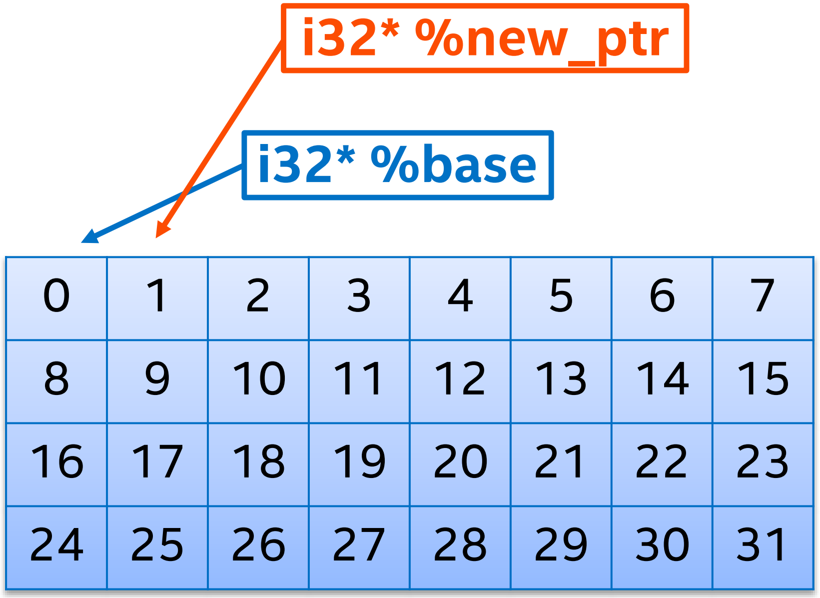
Manipulating pointers




```
%new_ptr = getelementptr i32, i32* %base, i32 0
```

"Offset by 0 **elements of the base type**"

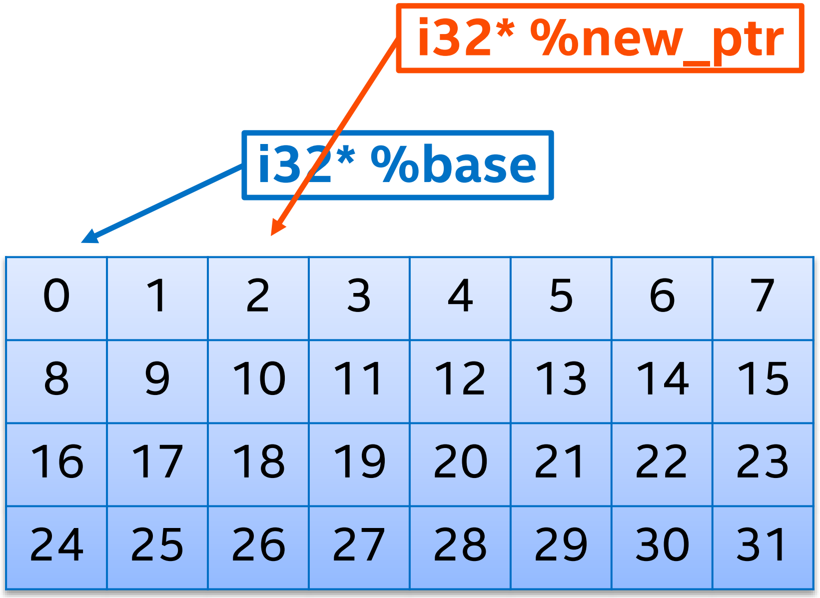
Manipulating pointers




```
%new_ptr = getelementptr i32, i32* %base, i32 1
```

"Offset by 1 **elements of the base type**"

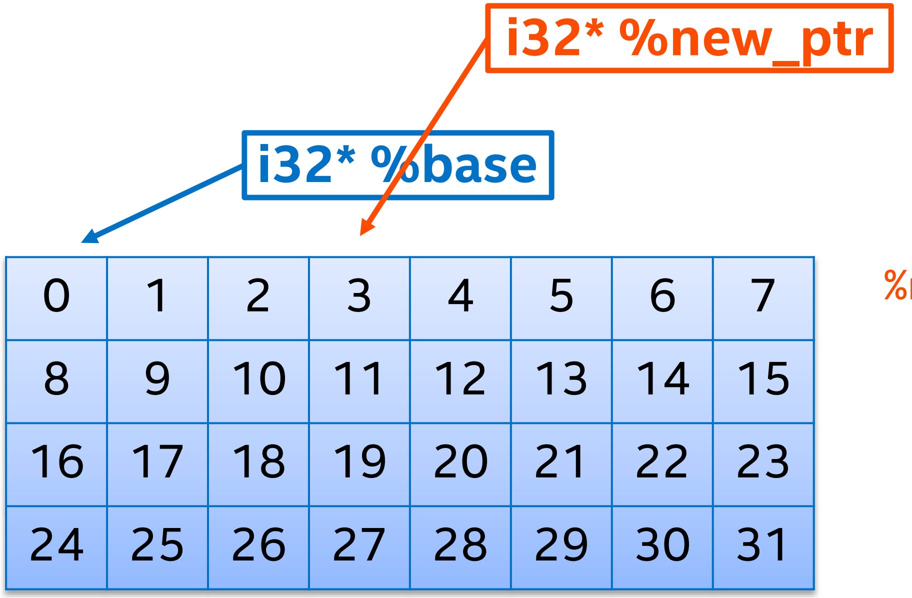
Manipulating pointers




```
%new_ptr = getelementptr i32, i32* %base, i32 2
```

"Offset by 2 **elements of the base type**"

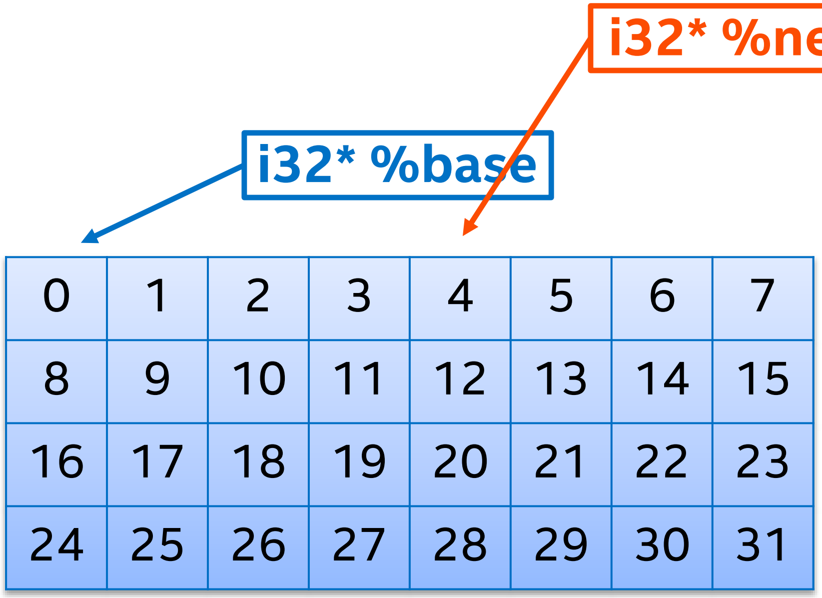
Manipulating pointers




```
%new_ptr = getelementptr i32, i32* %base, i32 3
```

"Offset by 3 **elements of the base type**"

Manipulating pointers




```
%new_ptr = getelementptr i32, i32* %base, i32 4
```

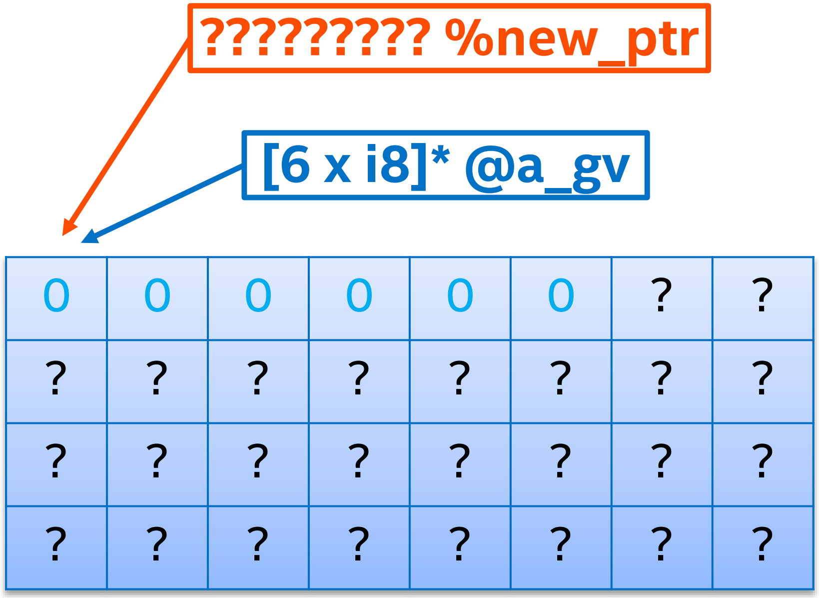
"Offset by 4 **elements of the base type**"

GEP fundamentals

1. Understand the first index:

- It does NOT change the resulting pointer type.
- It offsets by the **base type**.

Manipulating pointers

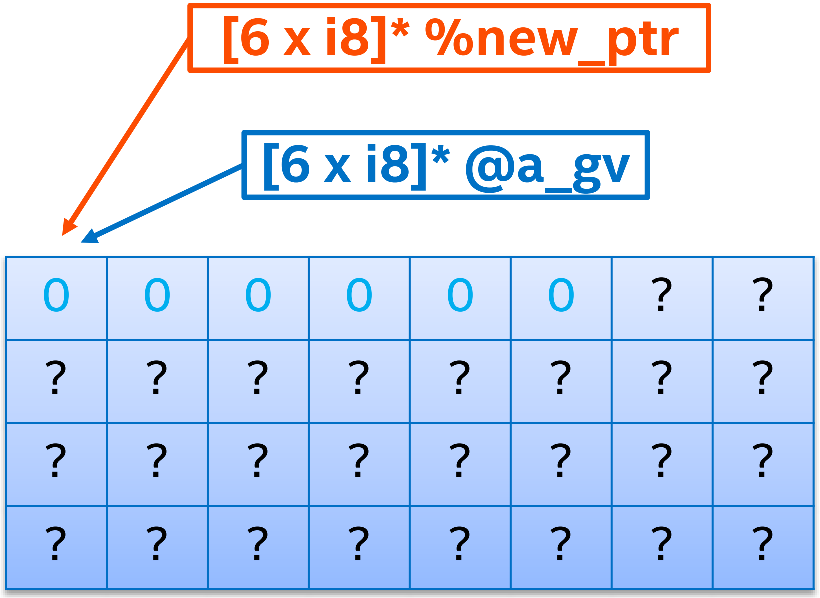



```
@a_gv = global [6 x i8] zeroinitializer
```

```
%new_ptr = getelementptr [6 x i8], [6 x i8]* @a_gv, i32 0
```

"Offset by 0 **elements of the base type**"

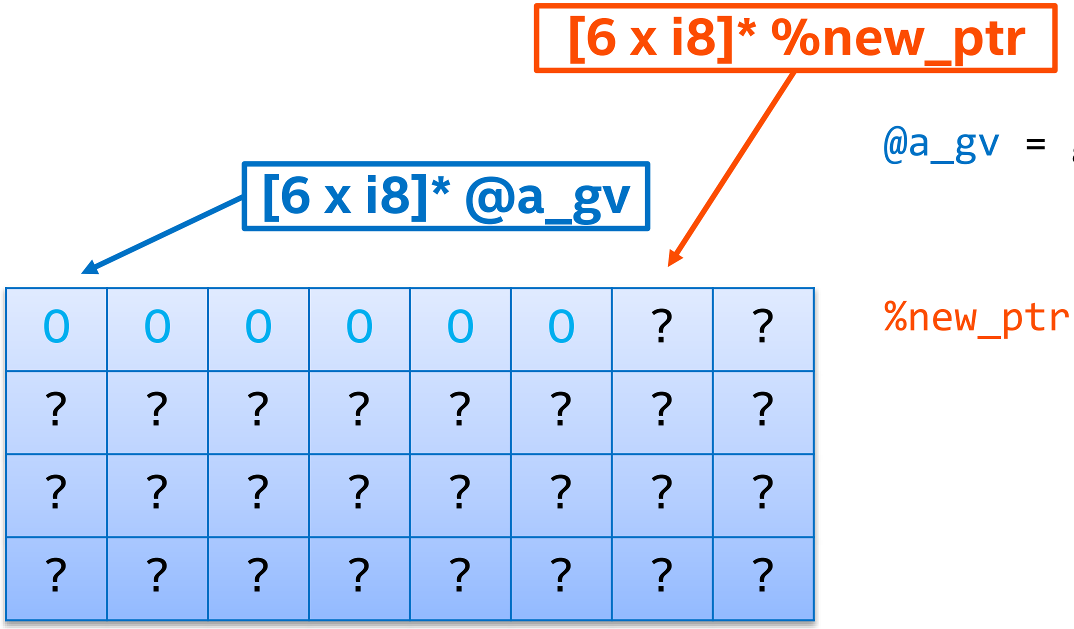
Manipulating pointers




```
@a_gv = global [6 x i8] zeroinitializer

%new_ptr = getelementptr [6 x i8], [6 x i8]* @a_gv, i32 0
```

"Offset by 0 **elements of the base type**"

Manipulating pointers




```
@a_gv = global [6 x i8] zeroinitializer

%new_ptr = getelementptr [6 x i8], [6 x i8]* @a_gv, i32 1
```

"Offset by 1 **elements of the base type**"

GEP fundamentals

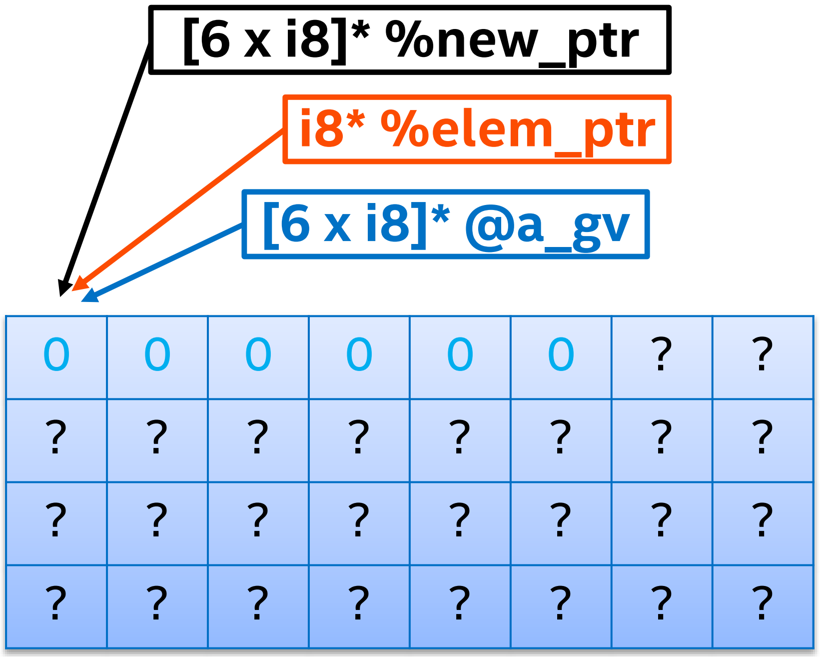
1. Understand the first index:

- It does NOT change the pointer type.
- It offsets by the **pointee type**.

2. Further indices:

- Offset inside **aggregate types.** **(and vectors)**
- Change the pointer type by removing one layer of “aggregation”.

Manipulating pointers



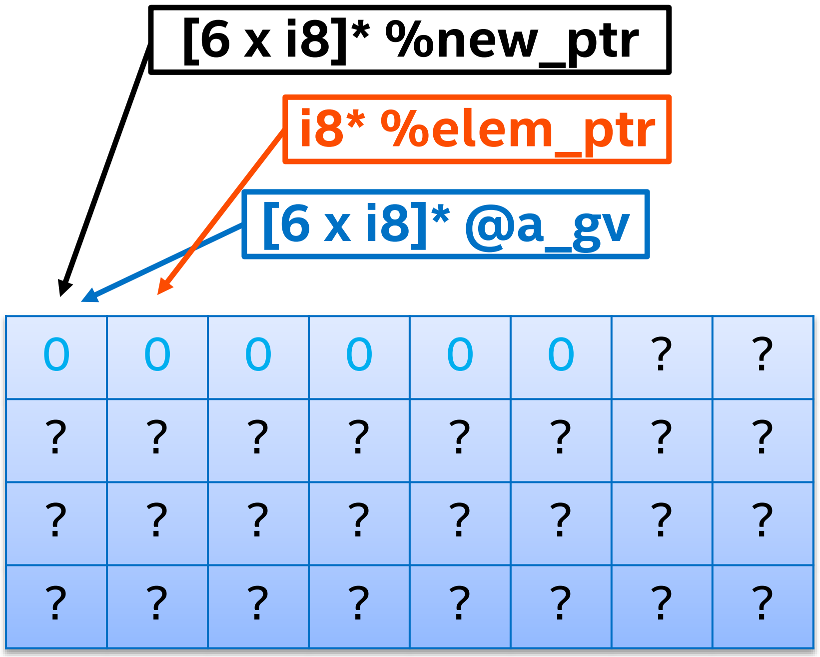
"Offset by 0 **elements of the base type**"

```
@a_gv = global [6 x i8] zeroinitializer

%new_ptr = getelementptr [6 x i8], [6 x i8]* @a_gv, i32 0

%elem_pt = getelementptr [6 x i8], [6 x i8]* @a_gv, i32 0, i32 0
```

Get the 0th element from the current aggregate:

[6 x i8]

Manipulating pointers



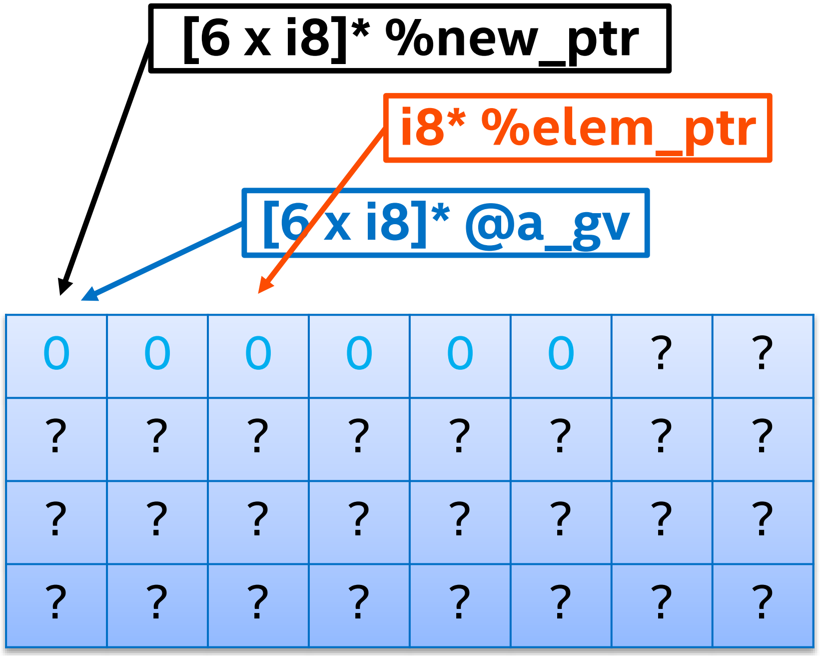
"Offset by 0 **elements of the base type**"

```
@a_gv = global [6 x i8] zeroinitializer

%new_ptr = getelementptr [6 x i8], [6 x i8]* @a_gv, i32 0

%elem_pt = getelementptr [6 x i8], [6 x i8]* @a_gv, i32 0, i32 1
```

Get the 1st element from the current aggregate:

[6 x i8]

Manipulating pointers



"Offset by 0 **elements of the base type**"

```
@a_gv = global [6 x i8] zeroinitializer

%new_ptr = getelementptr [6 x i8], [6 x i8]* @a_gv, i32 0

%elem_pt = getelementptr [6 x i8], [6 x i8]* @a_gv, i32 0, i32 2
```

Get the 2nd element from the current aggregate:

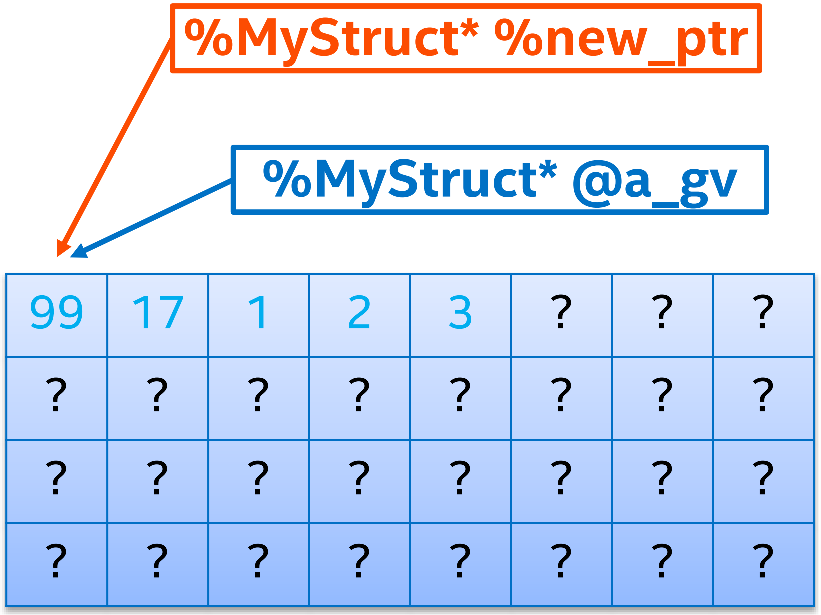
[6 x i8]

Aggregate types: structs

Defined by:

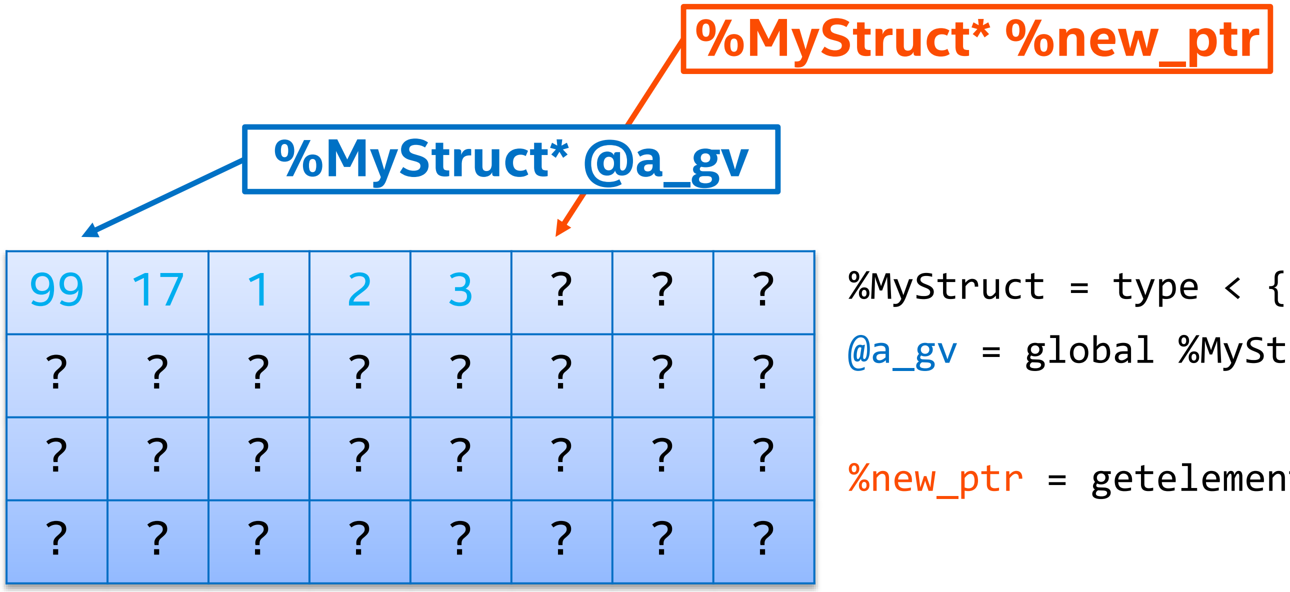
- A name: `%MyStruct =`
- Keyword “type”: `%MyStruct = type`
- A list of types: `%MyStruct = type { i8, i32, [3 x i32] }`

GEPs with structs




```
%MyStruct = type < { i8, i32, [3 x i32] }>
@a_gv = global %MyStruct { i8 99, i32 17, [3 x i32] [i32 1, i32 2, i32 3] }

%new_ptr = getelementptr %MyStruct*, %MyStruct* @a_gv, i32 0
```

GEPs with structs




```
%MyStruct = type < { i8, i32, [3 x i32] }>
@a_gv = global %MyStruct { i8 99, i32 17, [3 x i32] [i32 1, i32 2, i32 3] }

%new_ptr = getelementptr %MyStruct*, %MyStruct* @a_gv, i32 1
```

GEPs with structs

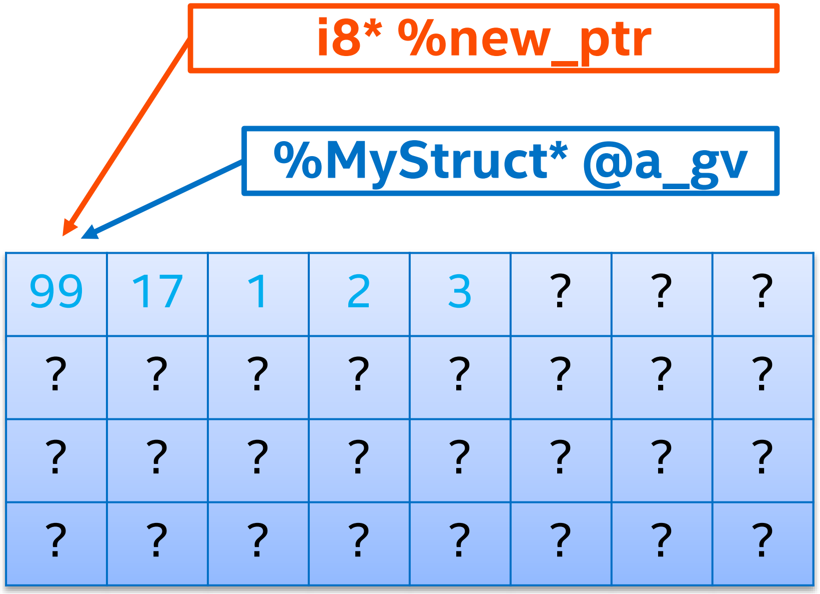
%MyStruct* %new_ptr

%MyStruct* @a_gv

99	17	1	2	3	?	?	?
?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?

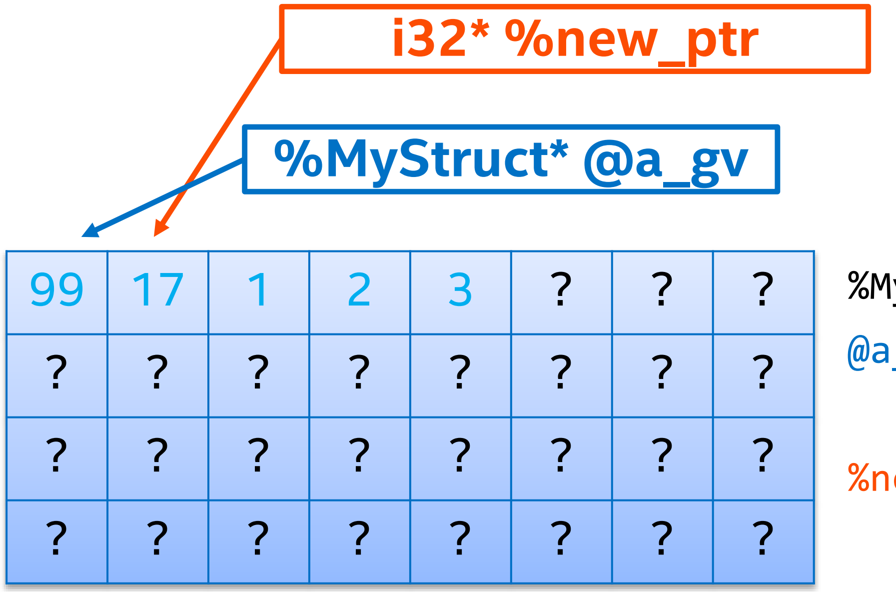
```
%MyStruct = type < { i8, i32, [3 x i32] }>
@a_gv = global %MyStruct { i8 99, i32 17, [3 x i32] [i32 1, i32 2, i32 3] }

%new_ptr = getelementptr %MyStruct*, %MyStruct* @a_gv, i32 0
```

GEPs with structs



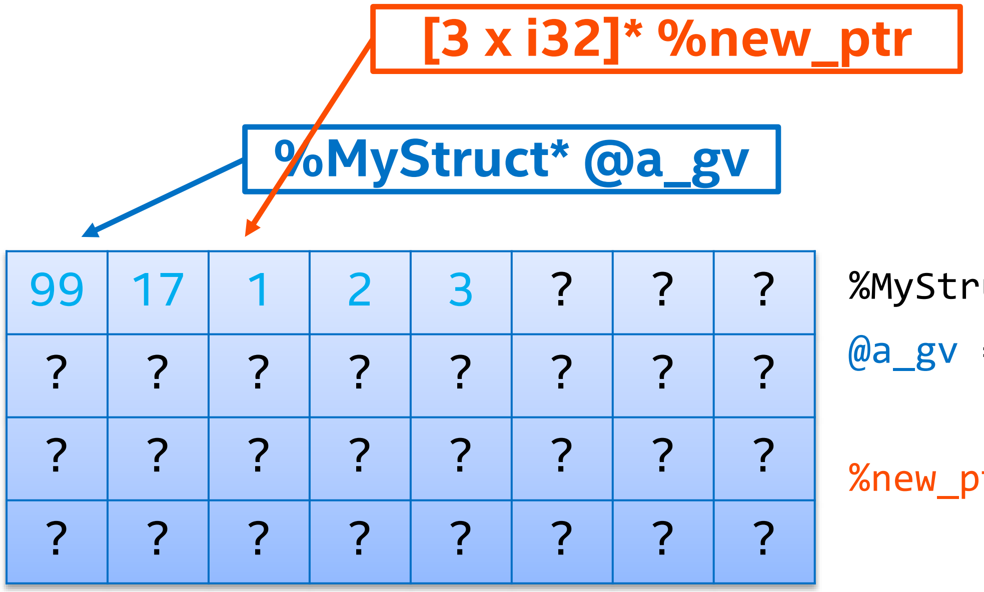

```
%MyStruct = type < { i8, i32, [3 x i32] }>
@a_gv = global %MyStruct { i8 99, i32 17, [3 x i32] [i32 1, i32 2, i32 3] }

%new_ptr = getelementptr %MyStruct*, %MyStruct* @a_gv, i32 0, i32 0
```

GEPs with structs



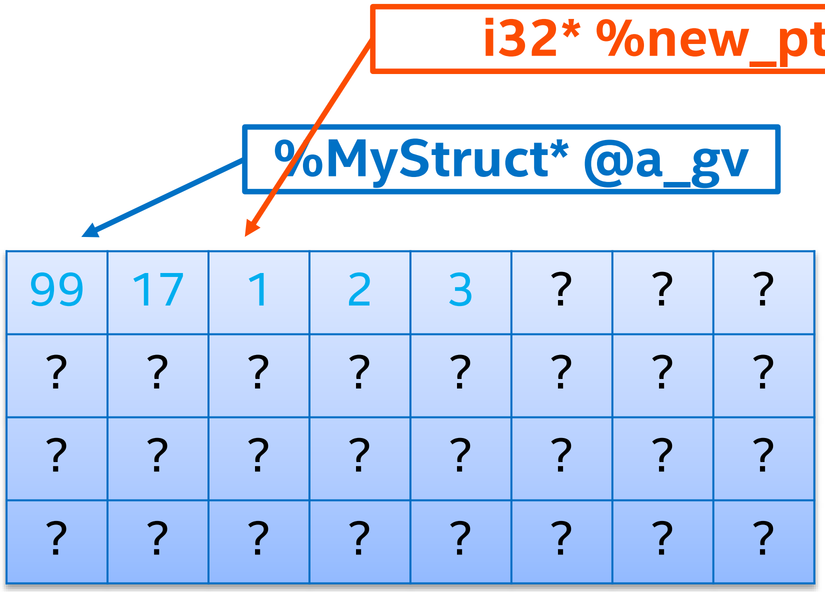

```
%MyStruct = type < { i8, i32, [3 x i32] }>
@a_gv = global %MyStruct { i8 99, i32 17, [3 x i32] [i32 1, i32 2, i32 3] }

%new_ptr = getelementptr %MyStruct*, %MyStruct* @a_gv, i32 0, i32 1
```

GEPs with structs



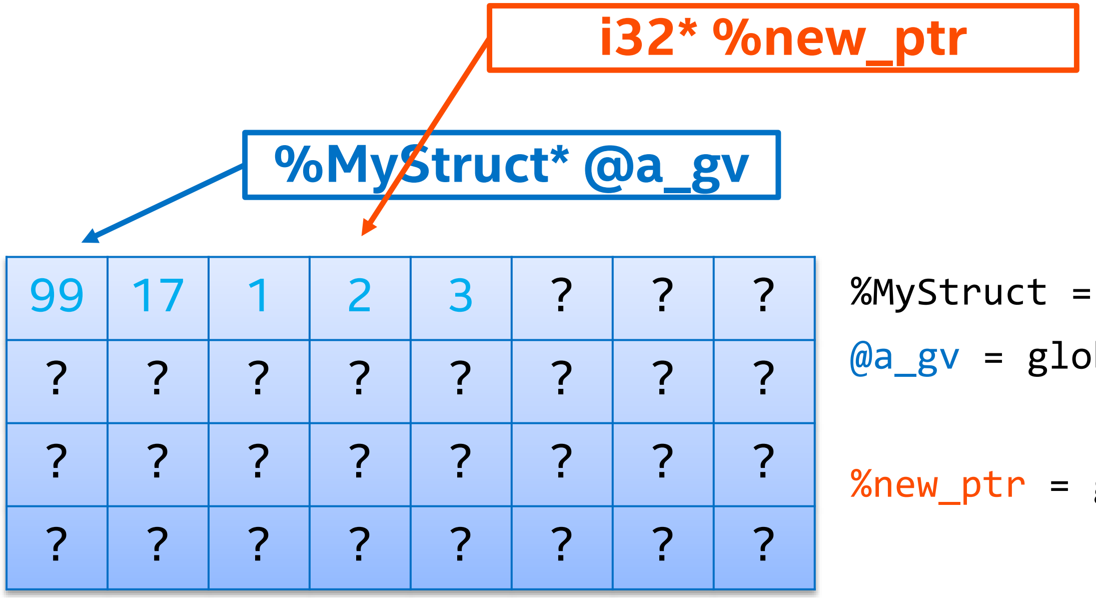

```
%MyStruct = type < { i8, i32, [3 x i32] }>
@a_gv = global %MyStruct { i8 99, i32 17, [3 x i32] [i32 1, i32 2, i32 3] }

%new_ptr = getelementptr %MyStruct*, %MyStruct* @a_gv, i32 0, i32 2
```

GEPs with structs




```
%MyStruct = type < { i8, i32, [3 x i32] }>
@a_gv = global %MyStruct { i8 99, i32 17, [3 x i32] [i32 1, i32 2, i32 3] }

%new_ptr = getelementptr %MyStruct*, %MyStruct* @a_gv, i32 0, i32 2, i32 0
```

GEPs with structs



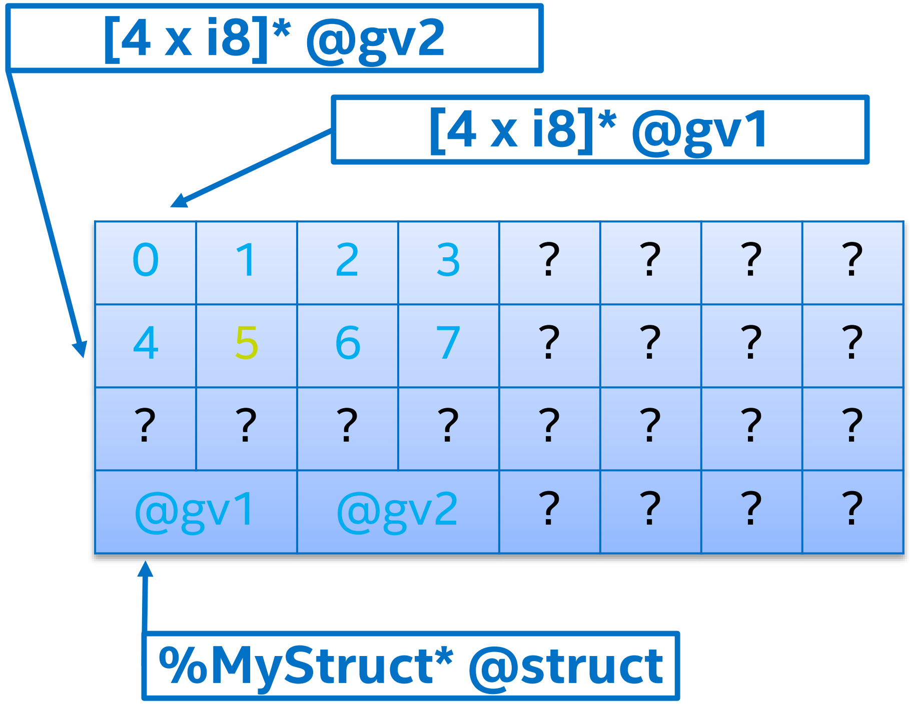

```
%MyStruct = type < { i8, i32, [3 x i32] }>
@a_gv = global %MyStruct { i8 99, i32 17, [3 x i32] [i32 1, i32 2, i32 3] }

%new_ptr = getelementptr %MyStruct*, %MyStruct* @a_gv, i32 0, i32 2, i32 1
```

GEP fundamentals

1. Understand the first index:
   - It does NOT change the pointer type.
   - It offsets by the **pointee type**.
2. Further indices:
   - Offset inside **aggregate types.**
   - Change the pointer type by removing one layer of “aggregation”.
3. Struct indices must be constants.

GEPs with structs

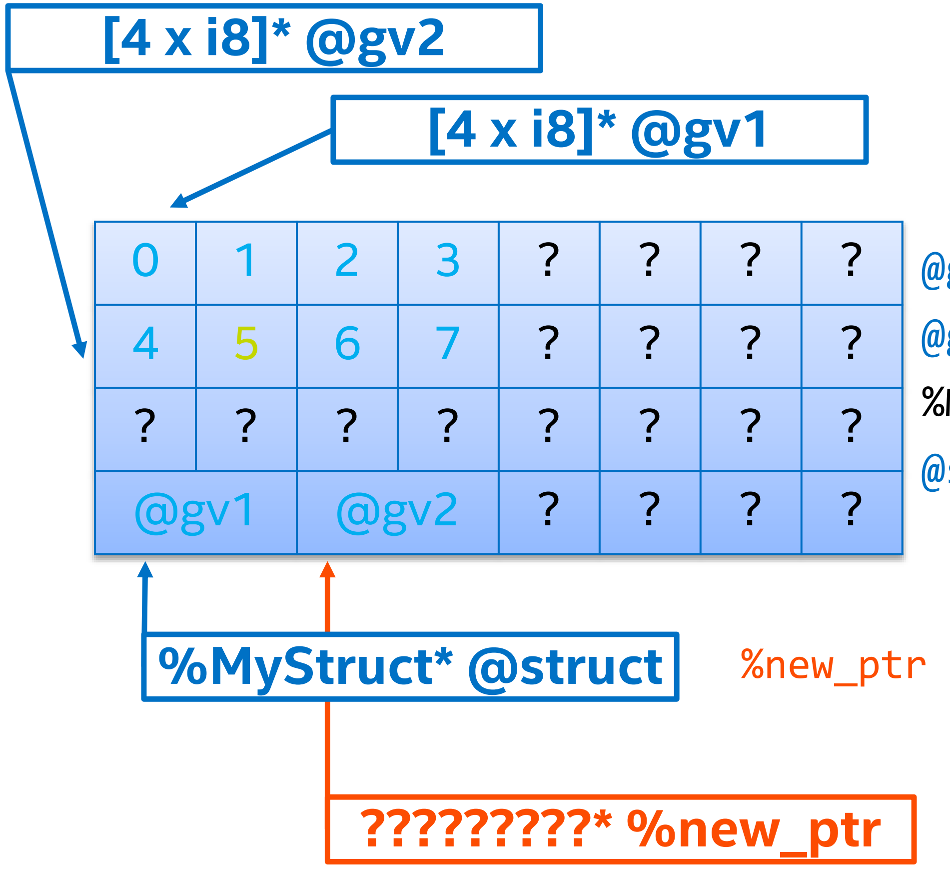
Goal: get an i8* to 2nd element of @gv2




```
@gv1 = global [4 x i8]  { [i8 0, i8 1, i8 2, i8 3] }
@gv2 = global [4 x i8]  { [i8 4, i8 5, i8 6, i8 7] }
%MyStruct = type { [4 x i8]*, [4 x i8]* }
@struct = global %MyStruct { [4 x i8]* @gv1,  [4 x i8]* @gv2 }
```

GEPs with structs

Goal: get an i8* to 2nd element of @gv2 **using @struct**

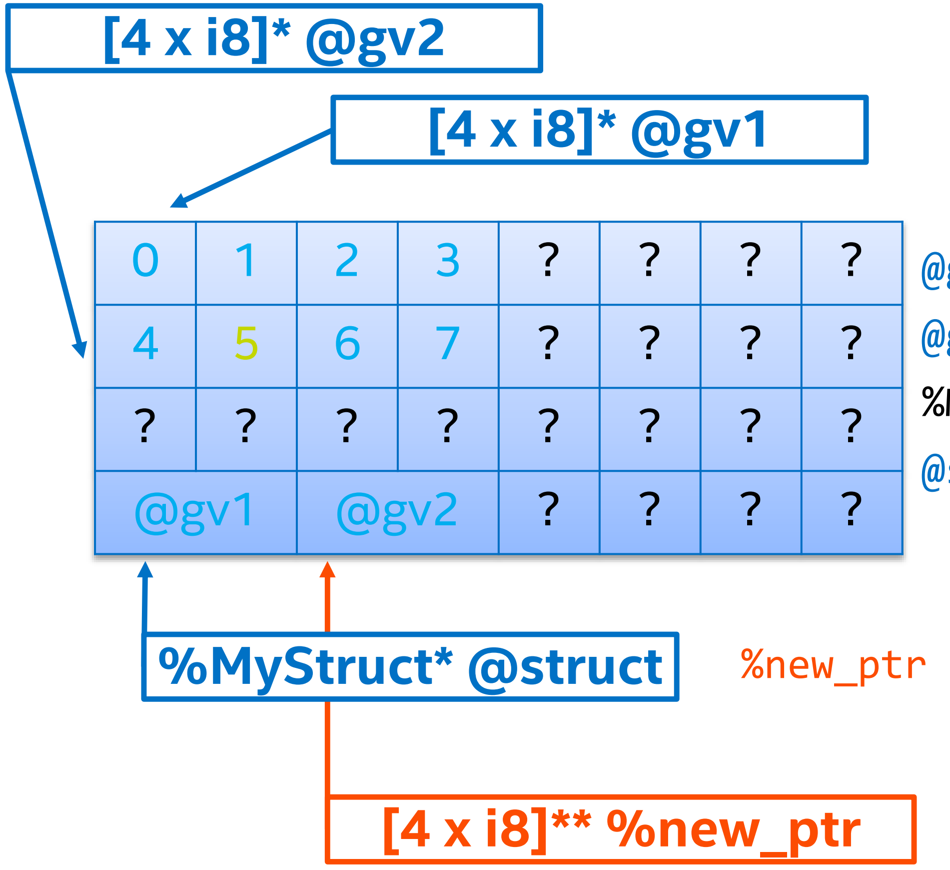



```
@gv1 = global [4 x i8]  { [i8 0, i8 1, i8 2, i8 3] }
@gv2 = global [4 x i8]  { [i8 4, i8 5, i8 6, i8 7] }
%MyStruct = type { [4 x i8]*, [4 x i8]* }
@struct = global %MyStruct { [4 x i8]* @gv1,  [4 x i8]* @gv2 }
```

```
%new_ptr = getelementptr %MyStruct, %MyStruct* @struct, i32 0, i32 1
```

GEPs with structs

Goal: get an i8* to 2nd element of @gv2 using @struct




```
@gv1 = global [4 x i8]  { [i8 0, i8 1, i8 2, i8 3] }
@gv2 = global [4 x i8]  { [i8 4, i8 5, i8 6, i8 7] }
%MyStruct = type { [4 x i8]*, [4 x i8]* }
@struct = global %MyStruct { [4 x i8]* @gv1,  [4 x i8]* @gv2 }
```

```
%new_ptr = getelementptr %MyStruct, %MyStruct* @struct, i32 0, i32 1
```

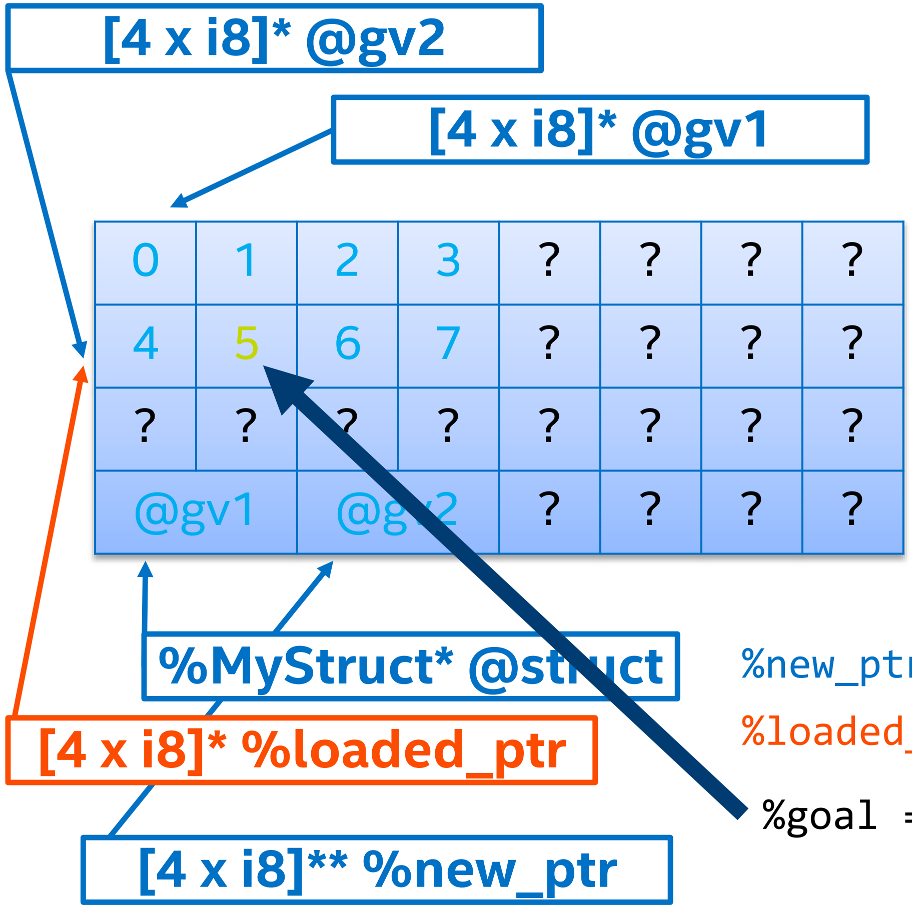
Quiz: Can we add an extra zero to this GEP?

GEP fundamentals

1. Understand the first index:
   - It does NOT change the pointer type.
   - It offsets by the **pointee type**.
2. Further indices:
   - Offset inside **aggregate types.**
   - Change the pointer type by removing one layer of “aggregation”.
3. Struct indices must be constants.
4. GEPs never load from memory!

GEPs with structs

Goal: get an i8* to 2nd element of @gv2 using @struct




```
@gv1 = global [4 x i8]  { [i8 0, i8 1, i8 2, i8 3] }
@gv2 = global [4 x i8]  { [i8 4, i8 5, i8 6, i8 7] }
%MyStruct = type { [4 x i8]*, [4 x i8]* }
@struct = global %MyStruct { [4 x i8]* @gv1,  [4 x i8]* @gv2 }
```

```
%new_ptr = getelementptr %MyStruct, %MyStruct* @struct, i32 0, i32 1
%loaded_ptr = load [4 x i8]*, [4 x i8]** %new_ptr
%goal = getelementptr [4 x i8]*, [4 x i8]* %loaded_ptr, i32 0, i32 1
```

Final remarks:

LLVM IR is constantly evolving.

Covered fundamental topics unlikely to change soon.

- There is a lot more to explore!

[Some] Topics not covered:

1. Constants
2. Constant expressions
3. Intrinsics
4. Exceptions
5. Debug information
6. Metadata
7. Attributes
8. Vector instructions

Final remarks:

LLVM IR is constantly evolving.

Covered fundamental topics unlikely to change soon.

- There is a lot more to explore!

Next steps:

- Learn how to manipulate IR using the LLVM library
- Look at the OPT code!

THANK YOU!

QUESTIONS?

Disclaimer

Intel, the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

***Other names and brands may be claimed as the property of others.**

© Intel Corporation