A Tour to LLVM IR

原文链接：https://zhuanlan.zhihu.com/p/66909226

内容概要

1. 什么是LLVM IR?如何得到IR?
2. LLVM编译的流程，IR文件之间的链接简介
3. C++ name mangling的用途，“extern C"作用的极简介绍
4. IR文件的布局
5. IR中函数定义的结构，什么是BB，什么是CFG
6. IR是一个强类型语言，如何用工具检查IR的合法性
7. 如何理解Language reference
8. 常见的terminator instruction介绍
9. 如何利用工具得到函数的CFG
10. 什么是SSA？SSA的好处和问题，以及如何解决这个问题

参考文献

1. what is tail reursion
2. make clang compile to ll
3. -cc1的含义
4. clang和clang++的区别
5. what is a linkage unit?
6. LLVM LanguageRef
7. extern "C"的作用
8. what is name mangling
9. what is static single assignment?
10. what is reaching definition?

- **LLVM IR** 是 **LLVM Intermediate Representation**，它是一种 *low-level languange，是一个像**RISC** 的指令集。*
- 然而可以很表达high-level的ideas，就是说high-level languange可以很干净地map到LLVM IR
- 这使得我们可以高效地进行代码优化

2. 如何得到IR?

我们先以尾递归的形式实现一个阶乘，再在`main`函数中调用中这个阶乘

```
// factorial.c

int factorial(int val, int total) {
  if(val==1) return total;
  return factorial(val-1, val * total);
}
```

```
// main.cpp

extern "C" int factorial(int);
int main(int argc, char** argv) {
    return factorial(2, 1) * 7 == 42;
}
```

> 注：这里的`extern "C"`是必要的，为了支持C++的函数重载和作用域的可见性的规则，编译器会对函数进行name mangling, 如果不加`extern "C"`，下文中生成的`main.ll`文件中`factorial`的函数名会被mangling成类似`_Z9factoriali`的样子，链接器便找不到要链接的函数。

LLVM IR有两种等价的格式，一种是`.bc`(Bitcode)文件，另一种是`.ll`文件，`.ll`文件是Human-readable的格式。 我们可以使用下面的命令得到这两种格式的IR文件

```
$ clang -S -emit-llvm factorial.c # factorial.ll
$ clang -c -emit-llvm factorial.c # factorial.bc
```

我们可以利用`grep`命令查看`clang`参数的含义

```
$ clang --help | grep -w -- -[Sc]
-c  Only run preprocess, compile, and assemble steps
-S  Only run preprocess and compilation steps
```

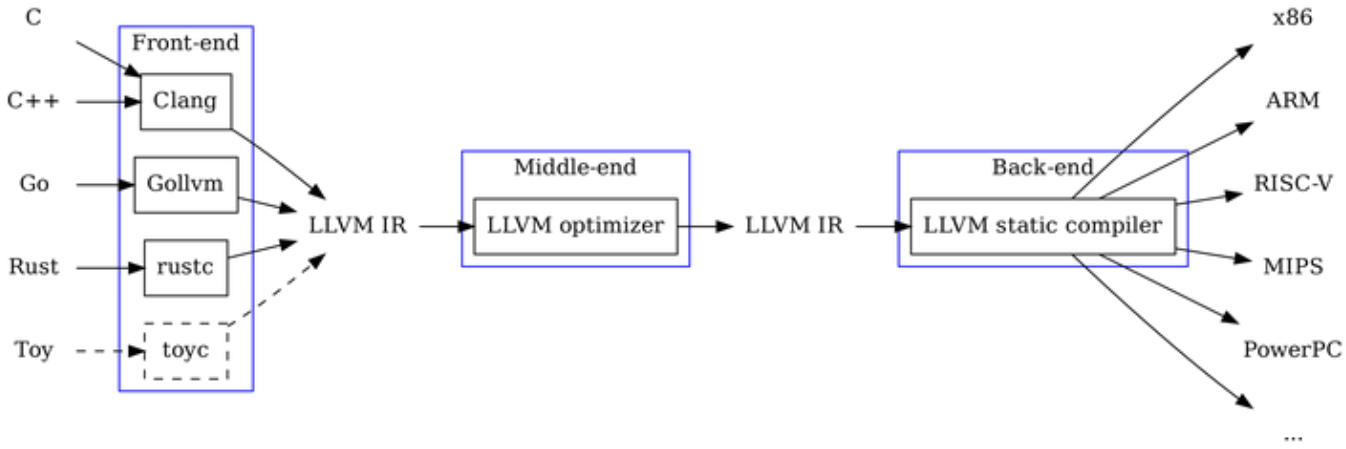
既然两种格式等价，自然就可以相互转换

```
$ llvm-as factorial.ll # factorial.bc
$ llvm-dis factorial.bc # factorial.ll
```

对于cpp文件，只需将`clang`命令换成`clang++`即可。

```
$ clang++ -S -emit-llvm main.cpp # main.ll
$ clang++ -c -emit-llvm main.cpp # main.bc
```

3. IR文件之间的链接以及将IR转为Target machine code



上图显示了llvm编译代码的一个pipleline, 其利用不同高级语言对应的前端（这里C/C++的前端都是clang）将其transform成LLVM IR，进行优化，链接后，再传给不同target的后端transform成target-specific的二进制代码。IR是LLVM的power所在，我们看下面这条command:

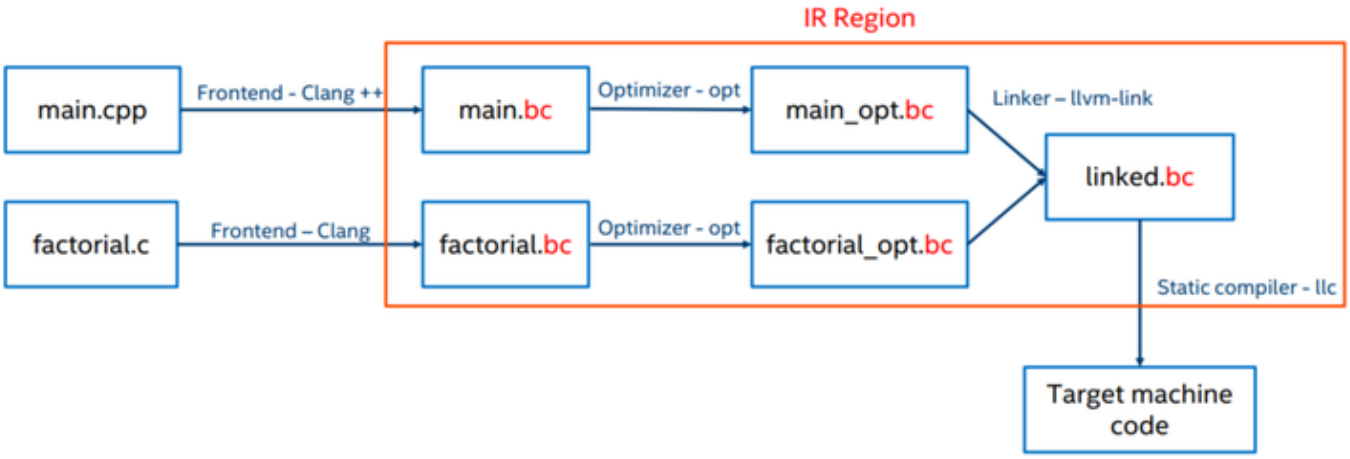
```
$ llvm-link factorial.bc main.bc -o linked.bc # lined.bc
```

`llvm-link`将两个IR文件链接起来了，值得注意的是`factorial.bc`是C转成的IR，而 `main.bc`是C++转成的IR，也就是说到了IR这个level，高级语言之间的差异消失了！它们之间可以相互链接（这里只是演示了C和C++的，其他语言的也可以链接）。

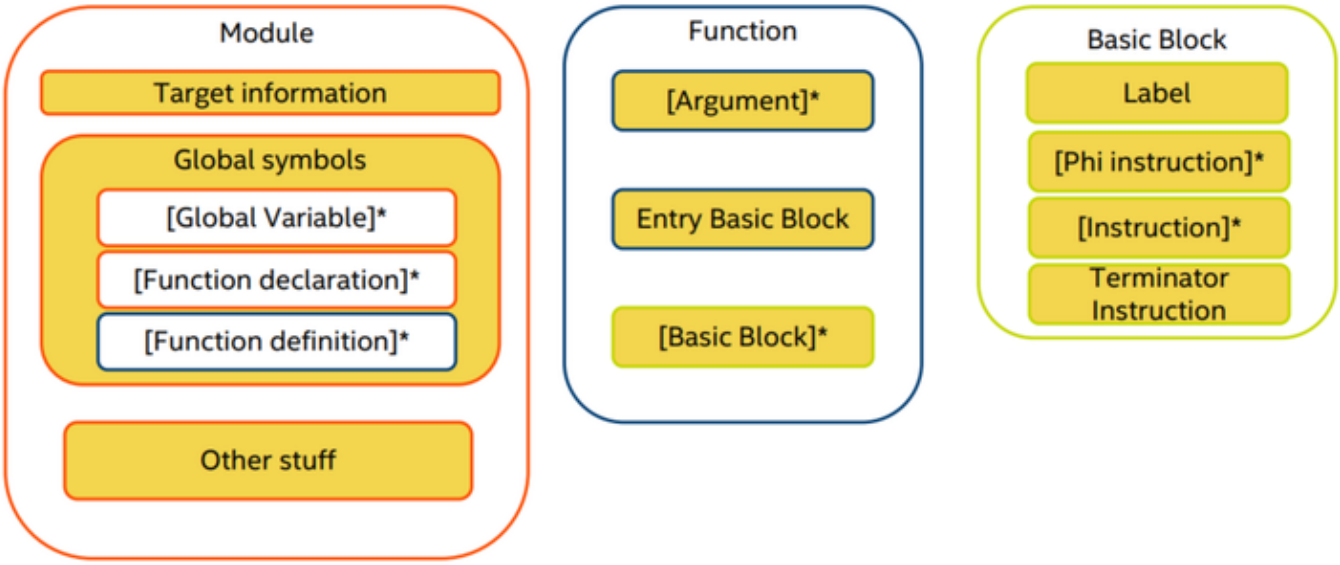
我们进一步可以将链接得到的IR转成target相关的code

```
llc --march=x86-64 linked.bc # linked.s
```

下图展示了完整的build过程



4. IR文件的布局



4.1 Target information

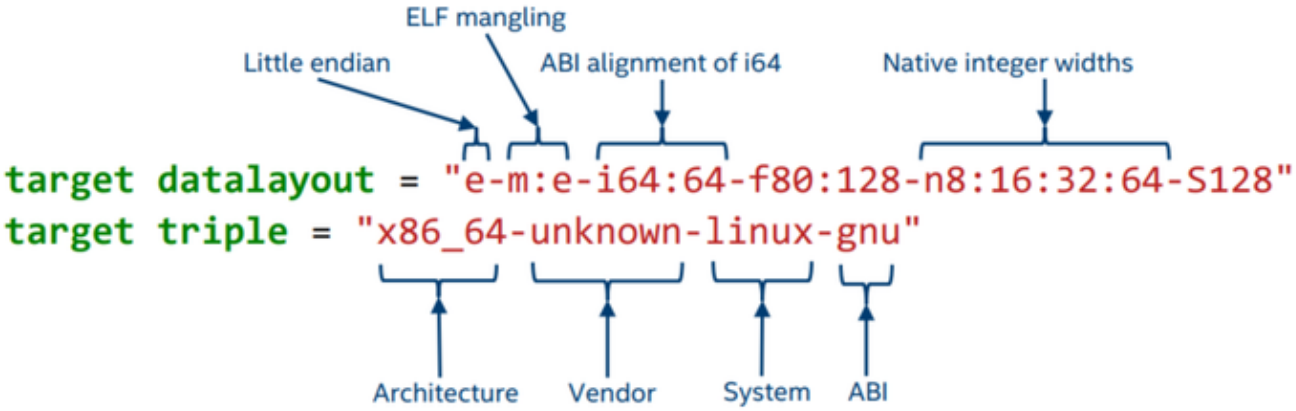
我们以`linked.ll`为例进行解析，文件的开头是

```
; ModuleID = 'linked.bc'

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

`;`后面的注释指明了module的标识，`source_filename`是表明这个module是从什么文件编译得到的（如果你打开`main.ll`会发现这里的值是`main.cpp`），如果该modules是通过链接得到的，这里的值就会是`llvm-link`。

Target information的主要结构如下：



4.2 函数定义的主要结构

我们看一下函数`factorial`的定义

```
; Function Attrs: noinline nounwind optnone uwtable

define dso_local i32 @factorial(i32 %val, i32 %total) #0 {
entry:
  %retval = alloca i32, align 4
  %val.addr = alloca i32, align 4
  %total.addr = alloca i32, align 4
  store i32 %val, i32* %val.addr, align 4
  store i32 %total, i32* %total.addr, align 4
  %0 = load i32, i32* %val.addr, align 4
  %cmp = icmp eq i32 %0, 1
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry

  %1 = load i32, i32* %total.addr, align 4
  store i32 %1, i32* %retval, align 4
  br label %return

if.end:                                           ; preds = %entry

  %2 = load i32, i32* %val.addr, align 4
  %sub = sub nsw i32 %2, 1
  %3 = load i32, i32* %val.addr, align 4
  %4 = load i32, i32* %total.addr, align 4
  %mul = mul nsw i32 %3, %4
  %call = call i32 @factorial(i32 %sub, i32 %mul)
  store i32 %call, i32* %retval, align 4
  br label %return

return:                                           ; preds = %if.end,
%if.then

  %5 = load i32, i32* %retval, align 4
  ret i32 %5
}
```

前面已经提到，`;`表示单行注释的开始。`define dso_local i32 @factorial(i32 %val) #0`表明开始定义一个函数，其中第一个`i32`是返回值类型，对应C语言中的`int`；`%factorial`是函数名；第二个`i32`是形参类型，`%val`是形参名。llvm中的标识符分为两种类型：全局的和局部的。全局的标识符包括函数名和全局变量，会加一个`@`前缀，局部的标识符会加一个`%`前缀。一般地，可用标识符对应的正则表达式为`[%@][-a-zA-Z$._][-a-zA-Z$._0-9]*`。

`dso_local`是一个Runtime Preemption说明符，表明该函数会在同一个链接单元（即该函数所在的文件以及包含的头文件）内解析符号。`#0`指出了该函数的attribute group。在文件的下面，你会找到类似这样的代码

```
attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false" "less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
```

因为attribute group可能很包含很多attribute且复用到多个函数，所以我们IR使用attribute group ID(即`#0`)的形式指明函数的attribute，这样既简洁又清晰。

在一对花括号里的就是函数体，函数体是由一系列basic blocks(BB)组成的，这些BB形成了函数的控制流图(Control Flow Graph, CFG)。每个BB都有一个label，label使得该BB有一个符号表的入口点，在函数`factorial`中，这些BB的label就是`entry`、`if.then`、`if.end`，BB总是以terminator instruction(e.g. `ret`、`br`、`callbr`)结尾的。

5. IR是一个强类型语言

看一下函数`main`的定义

```
; Function Attrs: noinline norecurse optnone uwtable

define dso_local i32 @main(i32 %argc, i8** %argv) #1 {
entry:
  %retval = alloca i32, align 4
  %argc.addr = alloca i32, align 4
  %argv.addr = alloca i8**, align 8
  store i32 0, i32* %retval, align 4
  store i32 %argc, i32* %argc.addr, align 4
  store i8** %argv, i8*** %argv.addr, align 8
  %call = call i32 @factorial(i32 2, i32 1)
  %mul = mul nsw i32 %call, 7
  %cmp = icmp eq i32 %mul, 42
  %conv = zext i1 %cmp to i32
  ret i32 %conv
}
```

LLVM的IR是一个强类型语言，每一条指令都显式地指出了实参的类型，例如`mul nsw i32 %call, 7`表明要将两个`i32`的数值相乘，`icmp eq i32 %mul, 42`表明要将两个i32的数据类型进行相等比较（这

里`%mul`是一个变量，而`mul`是一条指令，可以看出IR加前缀的好处）。此外，我们还很容易推断出返回值的类型，比如`i32`的数相乘的返回值就是`i32`类型，比较两个数值的相等关系的返回值就是`i1`类型。

强类型不但使得IR很human readable，也使得在优化IR时不需要考虑隐式类型转换的影响。在`main`函数的结尾，`zext i1 %cmp to i32`将`%cmp`从1位整数扩展成了32位的整数（即做了一个类型提升）。如果我们把最后两行用以下代码替代

那么这段IR就变成illegal的，检查IR是否合法可以使用`opt -verify <filename>`命令

```
$ opt -verify linked.ll
opt: linked.ll:45:11: error: '%cmp' defined with type 'i1' but expected
'i32'
  ret i32 %cmp
```

6. LangRef is your friend

在函数`main`的定义中，我们可以看到这样一条IR

```
%call = call i32 @factorial(i32 2)
```

对照着相应的C++代码我们很容易可以猜出每个符号的含义，但是每条指令可以有很多的变体，当我们不确定符号的含义的时候，LangRef为我们提供了参考

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

`[]`包围的表示可选参数（可以不写），`<>`包围的表示必选参数，选项用`|`分格开，表示只能写其中一个。

6. 常见的terminator instruction介绍

6.1 `ret`

语法

```
ret <type> <value>       ; Return a value from a non-void function

ret void                 ; Return from void function
```

概述

`ret`用来将控制流从callee返回给caller

Example

```
ret i32 5                       ; Return an integer value of 5

ret void                        ; Return from a void function

ret { i32, i8 } { i32 4, i8 2 } ; Return a struct of values 4 and 2
```

6.2 `br`

语法

```
br i1 <cond>, label <iftrue>, label <iffalse>
br label <dest>          ; Unconditional branch
```

概述

`br`用来将控制流转交给**当前**函数中的另一个BB。

Example

```
Test:
  %cond = icmp eq i32 %a, %b
  br i1 %cond, label %IfEqual, label %IfUnequal
IfEqual:
  ret i32 1
IfUnequal:
  ret i32 0
```

6.3 `switch`

语法

```
switch <intty> <value>, label <defaultdest> [ <intty> <val>, label <dest>
... ]
```

概述

`switch`根据一个整型变量的值，将控制流交给不同的BB。

Example

```
; Emulate a conditional br instruction

%Val = zext i1 %value to i32
switch i32 %Val, label %truedest [ i32 0, label %falsedest ]

; Emulate an unconditional br instruction

switch i32 0, label %dest [ ]
```

```
; Implement a jump table:

switch i32 %val, label %otherwise [ i32 0, label %onzero
                                    i32 1, label %onone
                                    i32 2, label %ontwo ]
```

6.4 `unreachable`

语法

概述

`unreachable`告诉optimizer控制流时到不了这块代码，就是说这块代码是dead code。

Example

在展示`unreachable`的用法的之前，我们先看一下`undef`的用法。`undef`表示一个未定义的值，只要是常量可以出现的位置，都可以使用`undef`。（此Example标题下的代码为伪代码）

```
%A = or %X, undef
%B = and %X, undef
```

`or`指令和`and`指令分别是执行按位或和按位与的操作，由于`undef`的值是未定义的，因此编译器可以随意假设它的值来对代码进行优化，譬如说假设undef的值都是0

可以假设undef的值是-1

也可以假设undef的两处值是不同的，譬如第一处是0，第二处是-1

为什么`undef`的值可以不同呢？这是因为`undef`对应的值是没有确定的生存期的，当我们需要一个undef的值的时候，编译器会从可用的寄存器中随意取一个值拿过来，因此并不能保证其值随时间变化具有一致性。下面我们可以看`unreachable`的例子了

```
%A = sdiv undef, %X
%B = sdiv %X, undef
```

`sdiv`指令是用来进行整数/向量的除法运算的，编译器可以假设`undef`的值是0，因为一个数除以0是未定义行为，因此编译器可以认为其是dead code，将其优化成

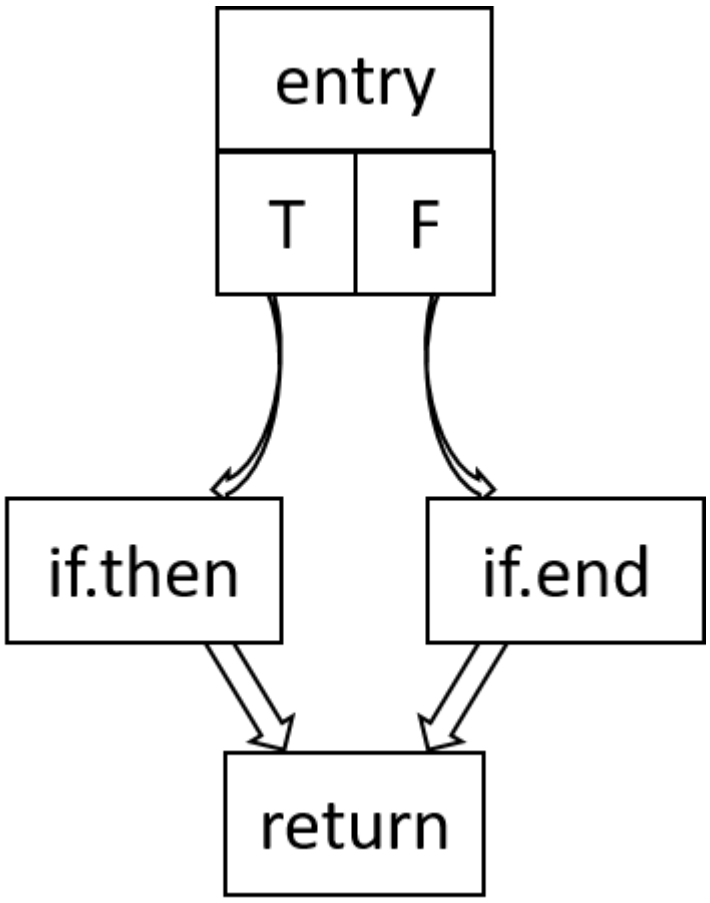
6. 控制流图（Control Flow Graph）

既然函数体是由一系列basic blocks(BB)组成的，并且BB形成了函数的控制流图,每个BB都有唯一的label,那么我们就可以label之间的跳转关系来表示整个函数的控制流图，llvm提供了`opt -analyze -dot-cfg-only <filename>`命令来帮助我们生成

```
$ opt -analyze -dot-cfg-only factorial.ll
$ vim .factorial.dot
digraph "CFG for 'factorial' function" {
        label="CFG for 'factorial' function";

        Node0x207ced0 [shape=record,label="{entry|{<s0>T|<s1>F}}"];
        Node0x207ced0:s0 -> Node0x207d7e0;
        Node0x207ced0:s1 -> Node0x207d8b0;
        Node0x207d7e0 [shape=record,label="{if.then}"];
        Node0x207d7e0 -> Node0x207da90;
        Node0x207d8b0 [shape=record,label="{if.end}"];
        Node0x207d8b0 -> Node0x207da90;
        Node0x207da90 [shape=record,label="{return}"];
}
```

把它画成图就非常清晰了



7. IR是静态单一赋值的（Static Single Assignment）

在IR中，每个变量都在使用前都必须先定义，且每个变量只能被赋值一次（如果套用C++的术语，就是说每个变量只能被初始化，不能被赋值），所以我们称IR是静态单一赋值的。举个例子的，假如你想返回`a*b+c`的值，你觉得可能可以这么写

```
%0 = mul i32 %a, %b
%0 = add i32 %0, %c
ret i32 %0
```

但是这里%0被赋值了两次，是不合法的，我们需要把它修改成这样

```
%0 = mul i32 %a, %b
%1 = add i32 %0, %c
ret i32 %1
```

7.1 SSA的好处

SSA可以简化编译器的优化过程，譬如说，考虑这段代码

```
d1: y := 1
d2: y := 2
d3: x := y
```

我们很容易可以看出第一次对`y`赋值是不必要的，在对`x`赋值时使用的`y`的值时第二次赋值的结果，但是编译器必须要经过一个定义可达性(Reaching definition)分析才能做出判断。编译器是怎么分析呢？首先我们先介绍几个概念：

***变量x的定义**是指一个会给x赋值或可能给x赋值的语句，譬如`d1`就是对`y`的一个定义*

当一个变量x有新的定义后 ，旧的的定义会被新的定义**kill**掉，譬如`d2`就kill掉了`d1`。

***一个定义d到达点p**是指存在一条d到p路径，在这条路径上，d没有被kill掉*

t1是t2的reaching definition是指存在一条t1到t2路径，沿着这条路径走就可以得到t1要赋值的变量的值，而不需要额外的信息。

按照上面的代码写法，编译器是很难判断`d3`的reaching definition的。因为`d3`的reaching definition可能是`d1`，也可能是`d2`，要搞清楚`d1`和`d2`谁kill了谁很麻烦。但是，如果我们的代码是SSA的，则代码就会长成这样

```
d1: y1 := 1
d2: y2 := 2
d3: x := y2
```

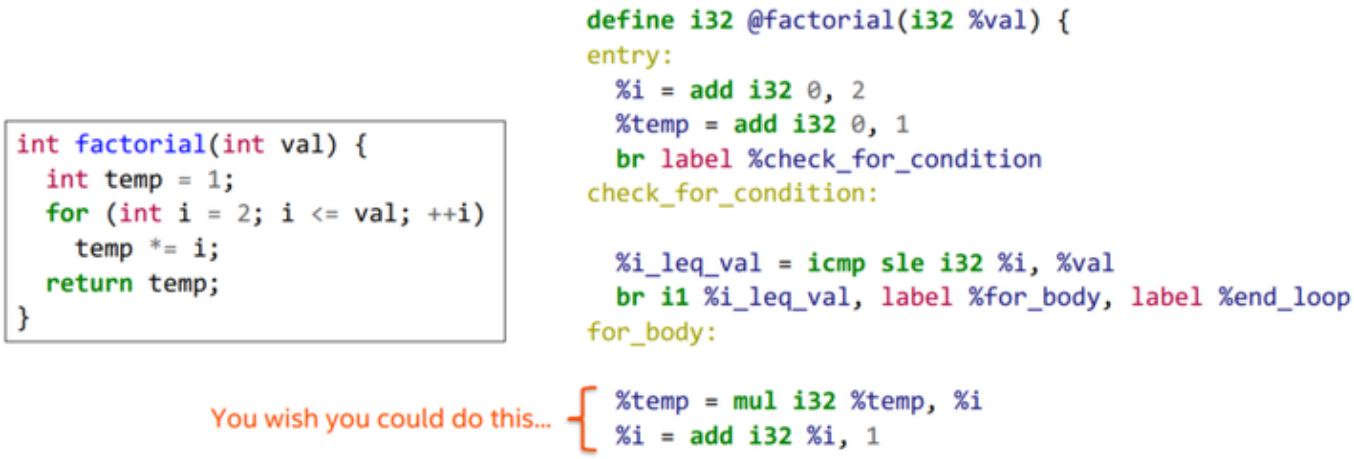
编译发现`x`是由`y2`赋值得到，而`y2`被赋值了2，且`x`和`y2`都只能被赋值一次，显然得到`x`的值的路径就是唯一确定的，`d2`就是`d3`的reaching definition。

7.3 SSA带来的问题

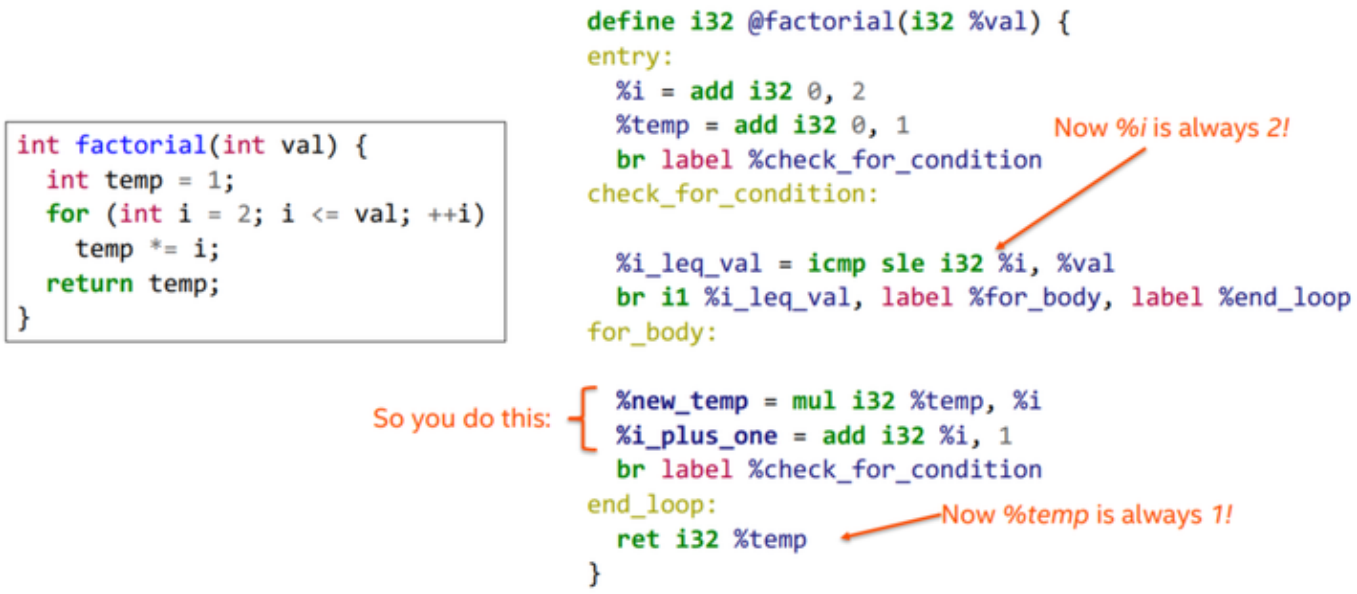
假设你想用IR写一个用循环实现的factorial函数

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

按照C语言的思路，我们可能大概想这样写



然而跑`opt -verify <filename>`命令我们就会发现`%temp`和`%i`被多次赋值了，这不合法。但是如果我们把第二处的`%temp`和`%i`换掉，改成这样



那返回值就会永远是1。

7.4 `phi`指令来救场

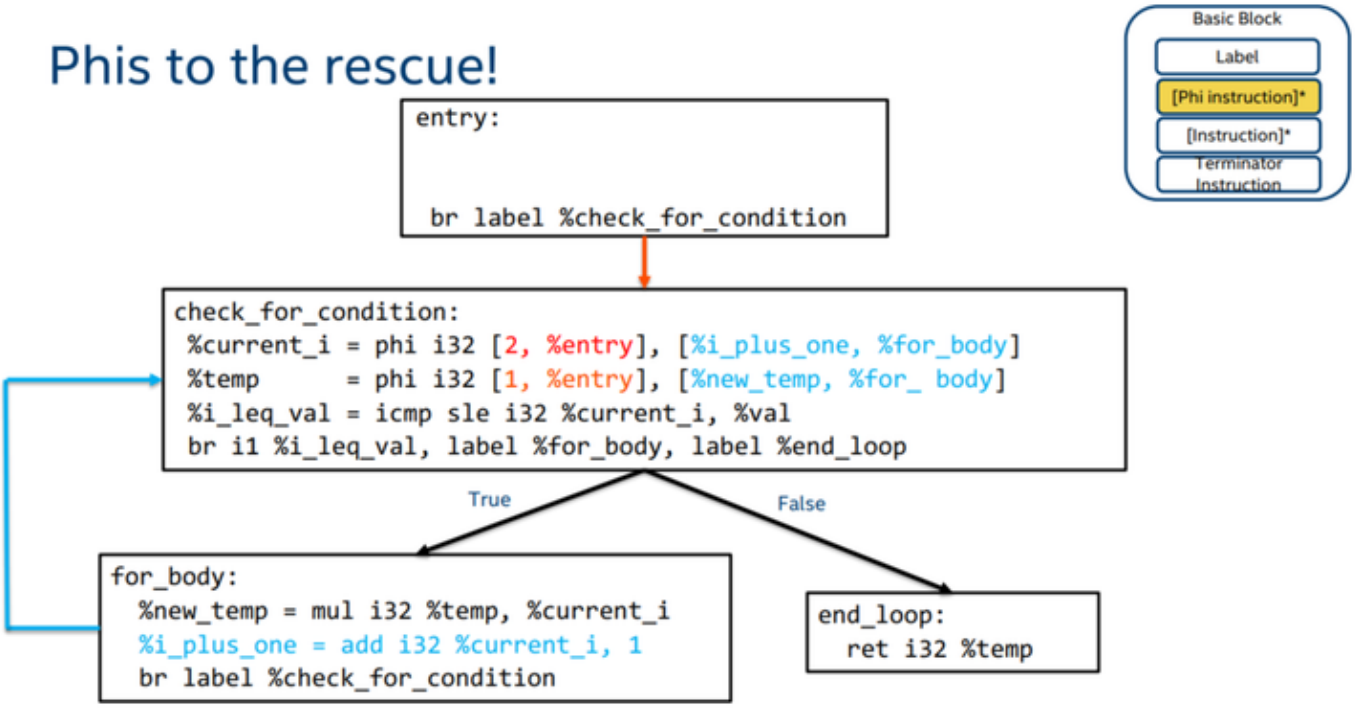
语法

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

概述

根据前一个执行的是哪一个BB来选择一个变量的值。

有了`phi`指令，我们就可以把代码改成这样



这样的话，每个变量就只被赋值一次，并且实现了循环递增的效果。

7.5 `alloca`指令来救场

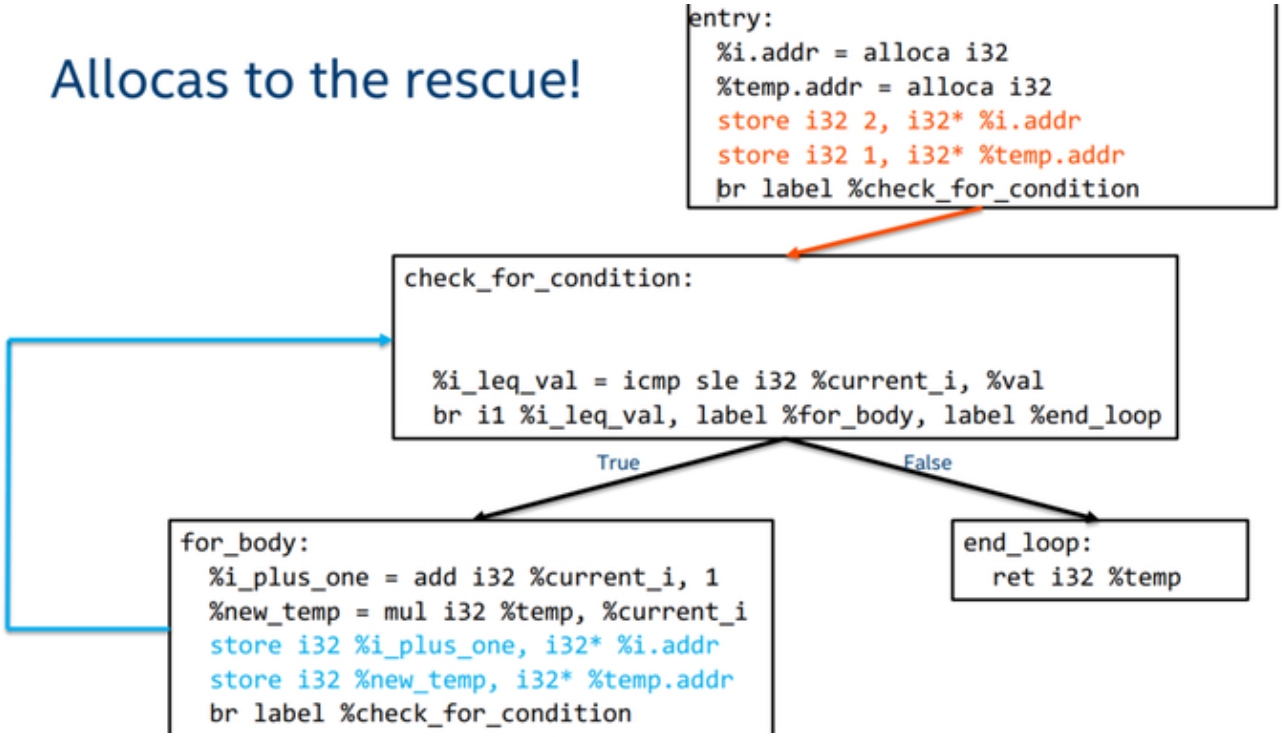
语法

```
<result> = alloca [inalloca] <type> [, <ty> <NumElements>] [, align
<alignment>] [, addrspace(<num>)]
```

概述

在当前执行的函数的栈帧上分配内存并返回一个指向这片内存的指针，当函数返回时内存会被自动释放（一般是改变栈指针）。

有了`alloca`指令，我们也可以通过使用指针的方式间接多次对变量赋值来骗过SSA检查



A Tour to LLVM IR（下）

原文链接：https://zhuanlan.zhihu.com/p/66909226

内容概要

1. IR的全局变量
2. IR中的Aggregate Types
3. `getelementptr`指令的使用

参考文献

1. LangRef

1. 全局变量

IR中的全局变量定义了一块在编译期分配的内存区域，其类型是一个指针，跟指令`alloca`的返回值用法一样。我们看一下一段使用全局变量简单的C代码对应的IR是什么样子

```
// a.c

static const int a=0;
const int b=1;
const int c=1;
int d=a+1;
```

```
; a.ll
```

```
@b = dso_local constant i32 1, align 4
@c = dso_local constant i32 1, align 4
@d = dso_local global i32 1, align 4
```

前面已经讲过`dso_local`是一个Runtime Preemption,表明该变量会在同一个链接单元内解析符号，`align 4`表示4字节对齐。`global`和`constant`关键字都可以用来定义一个全局变量，全局变量名必须有`@`前缀，因为全局变量会参与链接，所以除去前缀外，其名字会跟你用C语言定义时的相同。

因为我们定义变量`a`时使用了C语言的`static`关键字，也就是说`a`是local to file的，不参与链接，因此我们可以在生成的IR中可以看到，其被优化掉了。

```
// b.c

extern const int b;
extern const int c;
extern const int d;

int f() {
  return b*c+d;
}
```

```
; b.ll

@b = external dso_local constant i32, align 4
@c = external dso_local constant i32, align 4
@d = external dso_local constant i32, align 4

define dso_local i32 @f() #0 {
entry:
  %0 = load i32, i32* @b, align 4
  %1 = load i32, i32* @c, align 4
  %mul = mul nsw i32 %0, %1
  %2 = load i32, i32* @d, align 4
  %add = add nsw i32 %mul, %2
  ret i32 %add
}
```

从函数`f`的IR可以看到，全局变量其实是一个指针，在使用其时需要`load`指令（赋值时需要`store`指令）。那`gloal`和`constant`有什么区别呢？`constant`相比`global`，多赋予了全局变量一个const属性（对应C++的底层`const`的概念，表示指针指向的对象是一个常量）。

跟C/C++类似，IR中可以在定义全局变量时使用`global`，而在声明全局变量时使用`constant`，表示该变量在本文件内不改变其值。

我们可以使用`opt -S --globalopt <filename>`命令对全局变量进行优化

```
$ opt -S --globalopt a.ll -o a-opt.ll
@b = dso_local local_unnamed_addr constant i32 1, align 4
@c = dso_local local_unnamed_addr constant i32 1, align 4
@d = dso_local local_unnamed_addr global i32 1, align 4
```

可以看到优化过，全局变量前多了`local_unnamed_addr`的attribute, 该属性表明在这个module内，这个变量的地址是不重要的，只要关心它的值就好。有什么作用呢？譬如说这里`b`和`c`都是常量且等于1，又有`local_unnamed_addr`属性，编译器就可以把`b`和`c`合并成一个变量。

2. Aggregate Types

这里我们使用英文Aggregate Types主要是想跟C++的Aggregate Class区分开。IR的Aggregate Types包括数组和结构体。

2.1 数组

语法

```
[<elementnumber> x <elementtype>]
```

概述

跟C++的模板类`template<class T, std::size_t N > class array`类似，数组元素在内存中是连续分布的，元素个数必须是编译器常量，未被提供初始值的元素会被零初始化，只是下标的使用方式有点区别。

Example

```
@array = global [17 x i8] ; 17个i8都是0

%array2 = alloca [17 x i8] [i8 1, i8 2] ; 前两个是1、2，其余是0

%array3 = alloca [3 x [4 x i32]] ; 3行4列的i32数组

@array4 = global [2 x [3 x [4 x i16]]] ; 2x3x4的i16数组
```

2.2 结构体

语法

```
%T1 = type { <type list> }     ; Identified normal struct type

%T2 = type <{ <type list> }>   ; Identified packed struct type
```

概述

与C语言中的`struct`相同，不过IR提供了两种版本，normal版元素之间是由padding的，packed版没有。

Example

```
%struct1 = type { i32, i32, i32 } ; 一个i32的triple

%struct2 = type { float, i32 (i32) * } ; 一个pair，第一个元素是float，第二个元素是一个函数指针，该函数有一个i32的形参，返回一个i32

%struct3 = type <{ i8, i32 }> ; 一个packed的pair，大小为5字节
```

2.3 `getelementptr`指令（GEP）

我们可以使用`getelementptr`指令来获得指向数组的元素和指向结构体成员的指针。

语法

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

概述

第一个`ty`是第一个索引使用的基本类型，第二个`ty`表示其后的基址`ptrval`的类型，`inbounds`和`inrange`关键字的含义这里不讲，有兴趣可以去LangRef查阅。`<ty> <idx>`是第一组索引的类型和值，`<ty> <idx>`可以出现多次，其后出现的就是第二组、第三组等等索引的类型和值。要注意索引的类型和索引使用的基本类型是不一样的，索引的类型一般为`i32`或`i64`，而索引使用的基本类型确定的是增加索引值时指针的偏移量。

GEP的几个要点

理解第一个索引

1. 第一个索引不会改变返回的指针的类型，也就是说`ptrval`前面的`<ty>*`对应什么类型，返回就是什么类型
2. 第一个索引的偏移量的是由第一个索引的值和第一个`ty`指定的基本类型共同确定的。

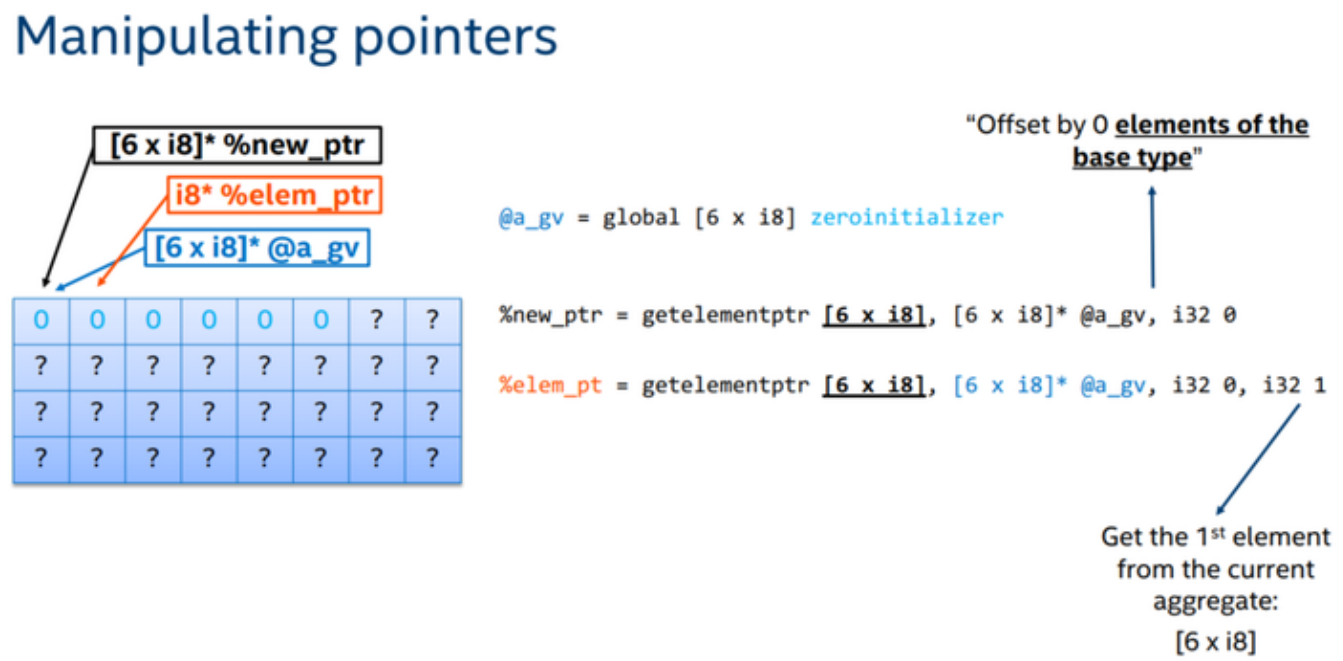
下面看个例子

上图中第一个索引所使用的基本类型是`[6 x i8]`，值是1，所以返回的值相对基址`@a_gv`前进了6个字节。由于只有一个索引，所以返回的指针也是`[6 x i8]*`类型。

理解后面的索引

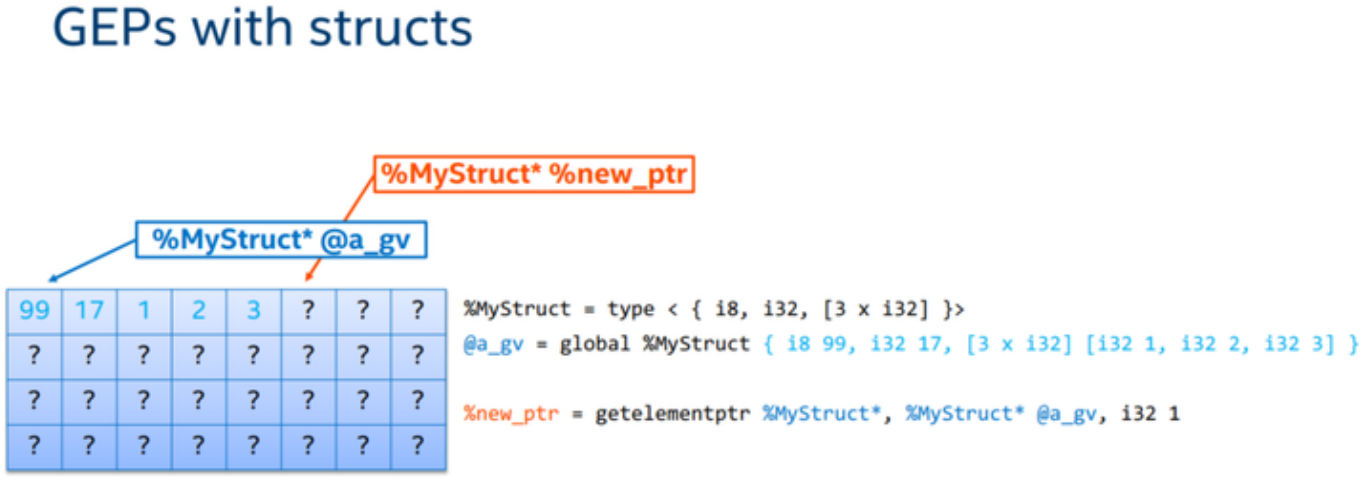
1. 后面的索引是在 Aggregate Types内进行索引
2. 每增加一个索引，就会使得该索引使用的基本类型和返回的指针的类型去掉一层

下面看个例子



我们看`%elem_ptr = getelementptr [6 x i8], [6 x i8]* @a_gv, i32 0, i32 0`这一句，第一个索引值是0，使用的基本类型`[6 x i8]`, 因此其使返回的指针先前进0 x 6 个字节，也就是不前进，第二个索引的值是1，使用的基本类型就是`i8`（`[6 x i8]`去掉左边的6），因此其使返回的指针前进一个字节，返回的指针类型为`i8*`（`[6 x i8]*`去掉左边的6）。

GEP如何作用于结构体



只有一个索引情况下，GEP作用于结构体与作用于数组的规则相同，`%new_ptr = getelementptr %MyStruct*, %MyStruct* @a_gv, i32 1`使得`%new_ptr`相对`@a_gv`偏移一个结构体`%MyStruct`的大小。

在有两个索引的情况下，第二个索引对返回指针的影响跟结构体的成员类型有关。譬如说在上图中，第二个索引值是1，那么返回的指针就会偏移到第二个成员，也就是偏移1个字节，由于第二个成员是`i32`类型，因此返回的指针是`i32*`。

如果结构体的本身也有Aggregate Type的成员，就会出现超过两个索引的情况。第三个索引将会进入这个Aggregate Type成员进行索引。譬如说上图中的第二个索引是2，指针先指向第三个成员，第三个成员是个数组。再看第三个索引是0，因此指针就指向该成员的第一个元素，指针类型也变成了`i32*`。

> 注：GEP作用于结构体时，其索引一定要是常量。GEP指令只是返回一个偏移后的指针，并没有访问内存。