

# Abstract (Last updated 2/01/18)

Abstract: In this talk, Michael Shah (“Mike”) will be presenting an introduction to the LLVM Compiler Infrastructure. A discussion of what LLVM is, who is using it, and why you might be interested in using LLVM will be presented during the first part of the talk. The second part of the talk will show interactive examples, taking us through installation to the point where we build and run our first function pass. We will build on top of our first function pass, to begin outputting some program metrics about programs. Mike will also be presenting some steps on how to proceed further and what resources are available for working with LLVM.

## Materials:

- Please bring a laptop with LLVM 5.0 setup if you want to follow along
- Otherwise materials will be posted to [www.mshah.io](http://www.mshah.io)

## Resources:

- Downloading and setting up LLVM: <http://llvm.org/docs/GettingStarted.html#checkout>
- A really good introduction guide: <http://adriansampson.net/blog/llvm.html>

Contact: [mshah.475@gmail.com](mailto:mshah.475@gmail.com)

Twitter: @MichaelShah

# Terminology (Open in a second browser if you like)

- [LLVM](#) - The name of the project (not an acronym)
- [IR](#) - Intermediate representation (Human-readable, 3 address, assembly like representation)
- [Bitcode](#) (.bc) - LLVM binary format of the IR
- [JIT](#) - Just-In-Time Compiler
- [SSA](#) - Single Static Analysis

# Introduction to LLVM (Tutorial)



Mike Shah, Ph.D.

[@MichaelShah](https://twitter.com/MichaelShah) | [mshah.io](https://mshah.io)

February 4, 2018

60-75 Minutes for talk (plenty of time for questions)

# Demo Time! Right from the start!

- So you know what to pay attention to!
  - In case you (or maybe I) walked into the wrong room by accident!
  - (Or if you are deciding to commit to an hour long talk online in the *distant future*)
- For those attending this talk live
  - Take a moment to introduce yourself to someone next to you .
  
- demo1.sh - Print functions from program
- demo2.sh - Print out stats
- demo3.sh - Print out direct function callees
- demo4.sh - Instrument code

# Who Am I?

by Mike Shah

- Currently a lecturer at Northeastern University in Boston, Massachusetts. I teach courses in computer systems, computer graphics, and game engine development.
- My research is in performance tools using static/dynamic analysis and software visualization.
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of graphics, visualization, concurrency, and parallelism.
- [www.mshah.io](http://www.mshah.io)



# Who Am I?

by Mike Shah

- Currently a lecturer at Northeastern University in Boston, Massachusetts. I teach courses in computer systems, computer graphics, and game engine development.
- My research is in performance tools using static/dynamic analysis and software visualization.
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of graphics, visualization, concurrency, and parallelism.
- [www.mshah.io](http://www.mshah.io)



# Who Am I?

by Mike Shah

- Currently a lecturer at Northeastern University in Boston, Massachusetts. I teach courses in computer systems, computer graphics, and game engine development.
- My research is in performance tools using static/dynamic analysis and software visualization.
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of graphics, visualization, concurrency, and parallelism.
- [www.mshah.io](http://www.mshah.io)



# Who Am I?

by Mike Shah

- Currently a lecturer at Northeastern University in Boston, Massachusetts. I teach courses in computer systems, computer graphics, and game engine development.
- My research is in performance tools using static/dynamic analysis and software visualization.
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of graphics, visualization, concurrency, and parallelism.
- [www.mshah.io](http://www.mshah.io)



# This is an introduction to LLVM

We have some specific goals

1. Figure out what is LLVM
2. Understand how to obtain LLVM
  - a. (This can be a major bottleneck for students)
3. Do a little example with Clang
4. Understand how to produce the demos I have already shown



# Goals for Tomorrow

Because you'll be ready to think about more solutions

- Know some resources available to continue growing
- Know some projects to try in the future



# Goals for Tomorrow

Because you'll be ready to think about more solutions

- Know some resources available to continue growing
- Know some projects to try in the future
- Be able to run through these slides again with confidence and excitement!



Slides and code are at the following location

[www.mshah.io/fosdem18.html](http://www.mshah.io/fosdem18.html)



# What is LLVM

# LLVM

(Formerly known as Low Level Virtual Machine--but it's more!)

- Started at The University of Illinois in 2000.
- [Chris Lattner](#) is the lead architect
- Backed by companies like Apple, Google, Microsoft, Intel, and more!
- And of course--open source!



<http://nondot.org/sabre/>

The image shows the logos for Julia and Emscripten. On the left is the Julia logo, which consists of a stylized 'J' inside a gear-like circle, followed by the word 'julia' in a lowercase, sans-serif font. Below this is the Emscripten logo, which features a green square with a white lightning bolt and the word 'emscripten' in a stylized, outlined font.

A screenshot of the LLVM website. The browser address bar shows 'llvm.org/Projects/WithLLVM/WithLLVM'. The page has a sidebar with links like 'Download now!', 'Search this Site', and 'Useful Links'. The main content area has a 'Table of Contents' with a list of links to various LLVM projects and resources, including 'Terra Lang', 'CodaLab Studio', 'Forty Programming Language', 'SMACK Software Verifier', 'DiscoPop: A Parallelism Discovery Tool', 'Just-in-time Adaptive Decoder Engine (Jade)', 'The Crack Programming Language', 'Rubinius: a Ruby Implementation', 'MacRuby', 'pocl: Portable Computing Language', 'TIA-based Codegen Environment (TICP)', 'The IcedTea Version of Sun's OpenJDK', 'The Pure Programming Language Compiler', 'LDC: the LLVM-based D Compiler', 'How to Write Your Own Compiler', 'Register Allocation by Purple Solving', 'Fast Real-Time Signal Processing System', 'Adobe "Hydra" Language', 'Galyan Static Checker', 'Improvements on SSA-Based Register Allocation', 'LENS Project', 'Tidlet Compiler', 'Ascension Reconfigurable Processor Compiler', 'Scheme to LLVM Translator', and 'LLVM Visualizer Tool'.

# LLVM

(Formerly known as Low Level Virtual Machine--but it's more!)

- Started at
- Chris Lattner
- Backed by
- Intel, and
- And of course

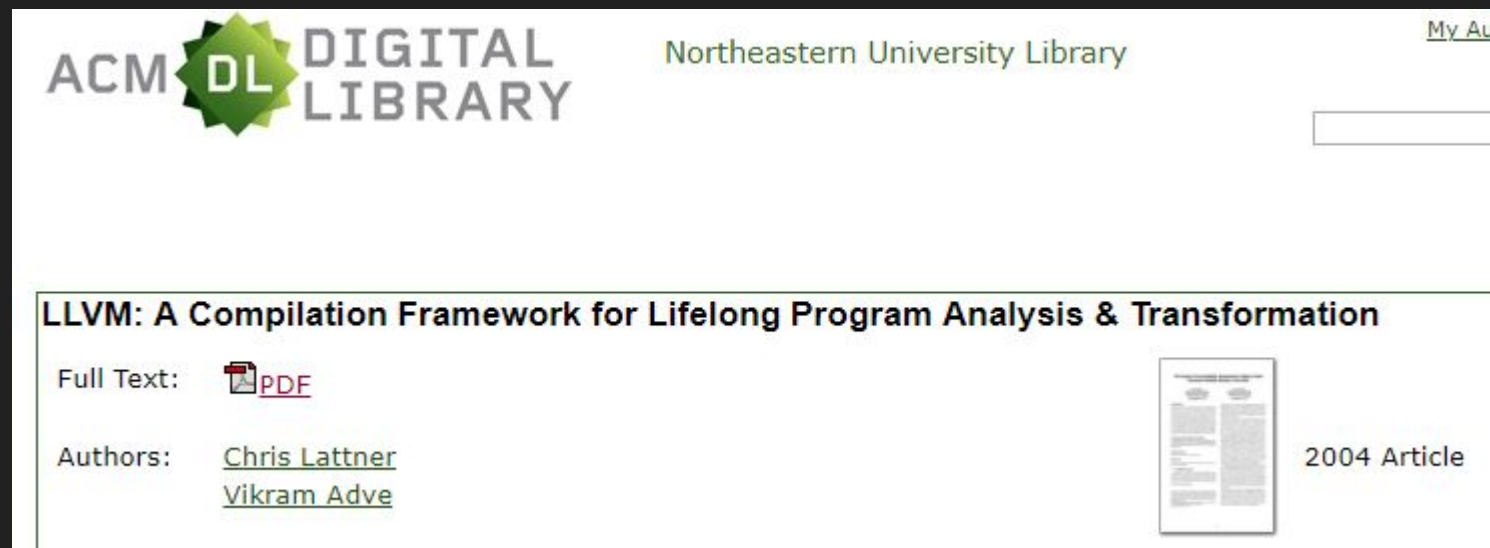
What is it that makes LLVM so great that programmers are paying attention to it?

<http://nondot.org/sabre/>

What is it that makes LLVM so great that programmers are paying attention to it?

# The Secret Recipe

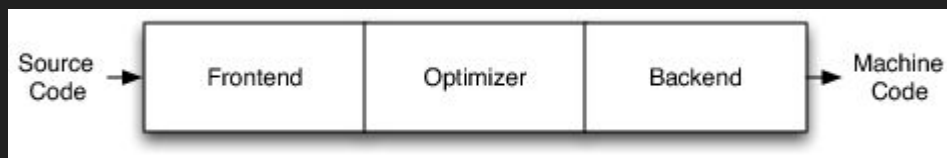
- The exact details are listed in the research paper:  
<https://dl.acm.org/citation.cfm?id=977673>



The screenshot shows the ACM Digital Library interface. At the top left is the 'ACM DL DIGITAL LIBRARY' logo. To its right is the text 'Northeastern University Library'. In the top right corner, there is a link 'My Auth' and a search input field. The main content area displays the title 'LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation'. Below the title, it says 'Full Text:' followed by a PDF icon and the text 'PDF'. Underneath, the authors are listed as 'Chris Lattner' and 'Vikram Adve'. To the right of the authors' names is a small thumbnail image of the paper's cover page. Further to the right, the text '2004 Article' is visible.

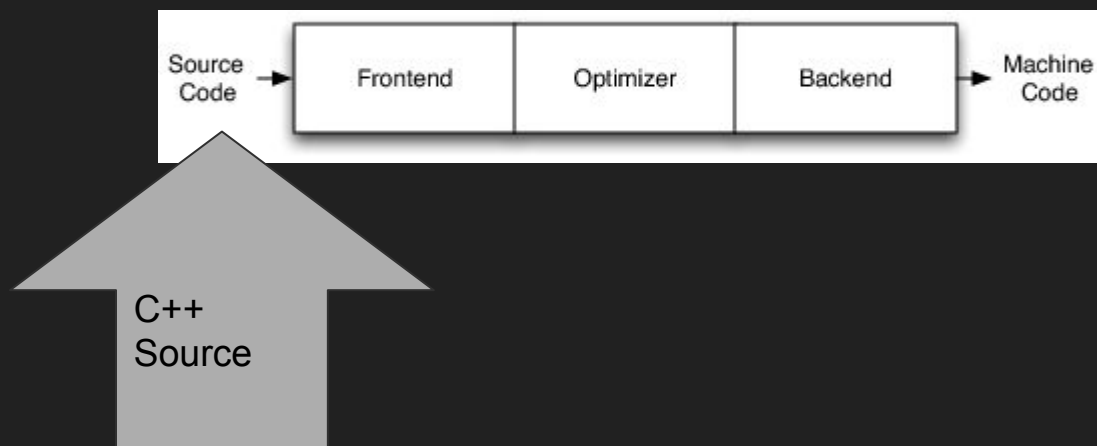
# Chris Lattner's big idea

- Lattner had been thinking about compilers while doing his graduate work.
- Job of the compiler:
  - Generate a high level language to machine code



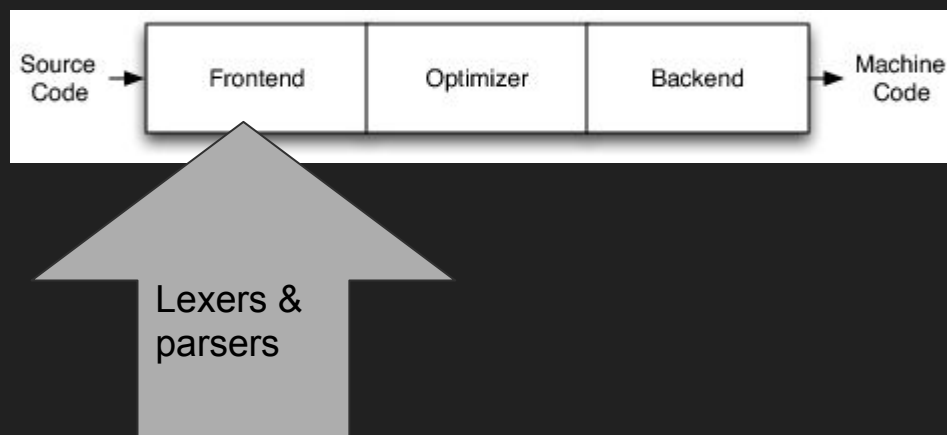
# Chris Lattner's big idea

- Lattner had been thinking about compilers while doing his graduate work.
- Job of the compiler:
  - Generate a high level language to machine code



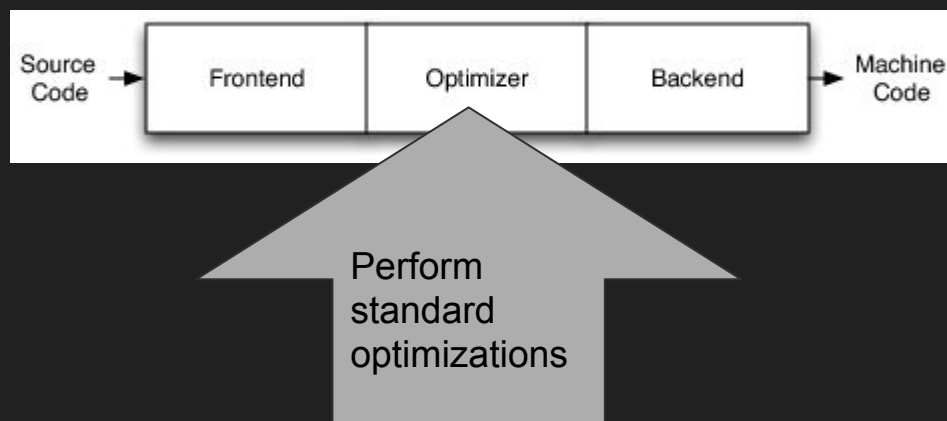
# Chris Lattner's big idea

- Lattner had been thinking about compilers while doing his graduate work.
- Job of the compiler:
  - Generate a high level language to machine code



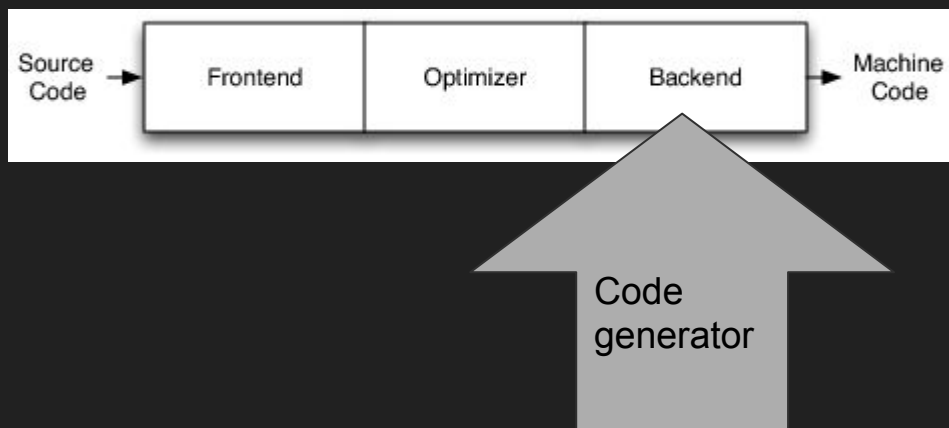
# Chris Lattner's big idea

- Lattner had been thinking about compilers while doing his graduate work.
- Job of the compiler:
  - Generate a high level language to machine code



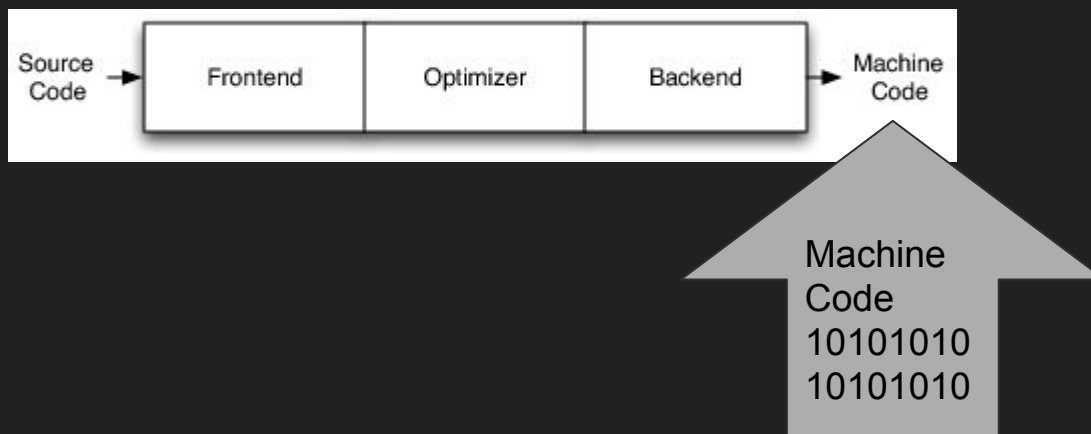
# Chris Lattner's big idea

- Lattner had been thinking about compilers while doing his graduate work.
- Job of the compiler:
  - Generate a high level language to machine code



# Chris Lattner's big idea

- Lattner had been thinking about compilers while doing his graduate work.
- Job of the compiler:
  - Generate a high level language to machine code



# The big idea | Around the year 2000

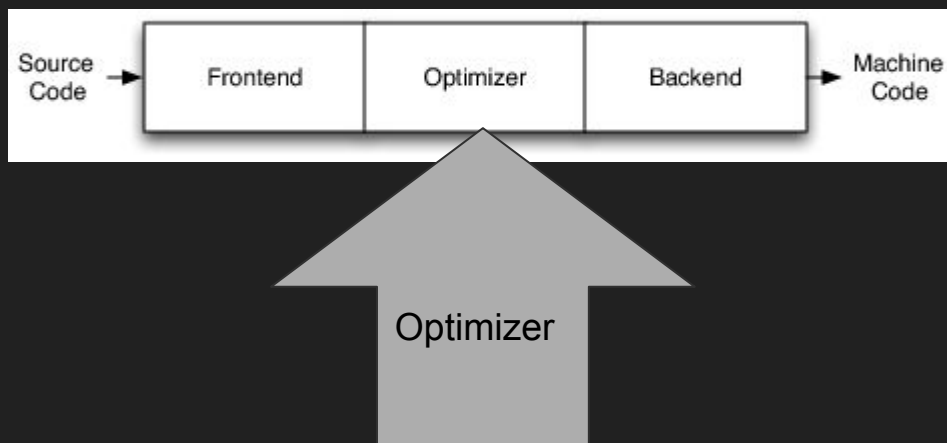
- JIT compilers were and continue to gain traction
  - A virtual machine compiles code online
  - This online compilation means performing optimizations over and over again
- So Lattner et al. big idea was to perform optimizations at compile-time that could do the heavy lifting.
  - Perhaps using some low level virtual machine

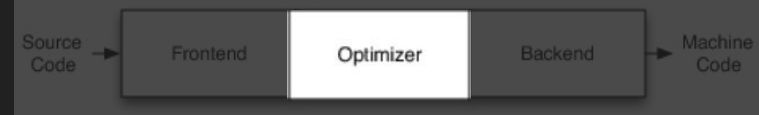
# The big idea | Around the year 2000

- JIT compilers were and continue to gain traction
  - A virtual machine compiles code online
  - This online compilation means performing optimizations over and over again
- So Lattner et al. big idea was to perform optimizations at compile-time that could do the heavy lifting.
  - Perhaps using some Low Level Virtual Machine

# The Optimizer

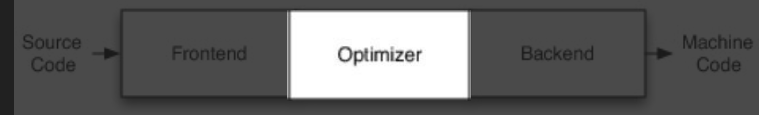
- So in the middle of our compiler pipeline, the optimizer (or optimization of code) is the focus.





# The optimization stage of compilers

- Typically programs are optimized by manipulating an intermediate representation (IR) of the high level language.
  - The intermediate representation (IR) is more 'regular' structurally
    - That means it is easier to analyze and manipulate.
      - (Just think about how many ways you can write and interpret the same program in a high-level language)



# The optimization stage of compilers

- Typically programs are optimized by manipulating an intermediate representation (IR) of the high level language.
  - The intermediate representation (IR) is more 'regular' structurally
    - That means it is easier to analyze and manipulate.
      - (Just think about how many ways you can write and interpret the same program in a high-level language)

## Example of what IR instructions look like

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

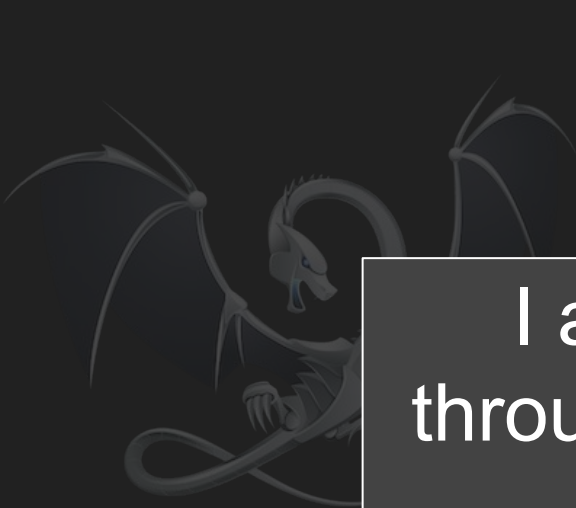


# How to get LLVM



# How to get LLVM

(And all the tools)



I am actually going to run  
through this section very quick!

Use it as a reference for how to  
setup and run examples from  
this slide deck



The LLVM project evolves at a  
good pace.

That is why you will want to  
know how to build from source  
to get the latest changes.

# Where the instructions always will be

- <http://llvm.org/docs/GettingStarted.html#checkout>

## Checkout LLVM from Subversion

If you have access to our Subversion repository, you can get a fresh copy of the entire source Subversion as follows:

- `cd where-you-want-llvm-to-live`
- Read-Only: `svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm`
- Read-Write: `svn co https://user@llvm.org/svn/llvm-project/llvm/trunk llvm`

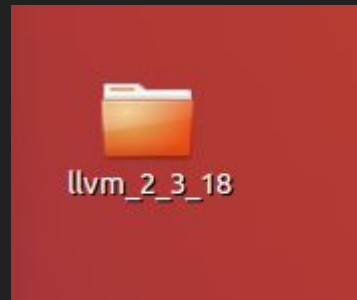


# Downloading LLVM 5.0

- For this talk, I am using and have tested the code with LLVM 5.0
- This tutorial is for an x86 based Ubuntu 16 machine
  - A similar process should work on Mac
    - (Windows users may need some different tools, I have not built LLVM on windows)
- Tools you will need
  - svn
  - Cmake
  - Make
  - A C compiler (Mine is GNU 5.4.0)

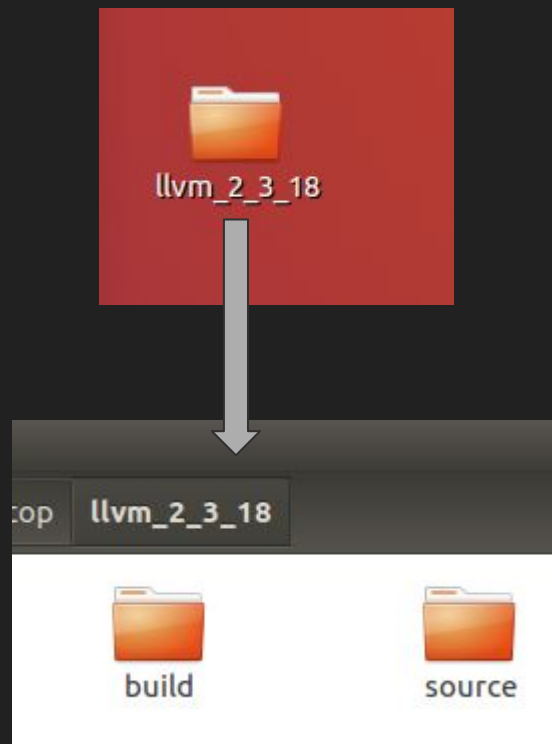
# Create a directory on your desktop

- I typically append a date to this directory



# Subdirectories

- Within the folder
  - A build directory where our compiled LLVM tools will go
    - (i.e. all the binaries)
  - A source directory where all of the LLVM source files live.





# From a Terminal

1. `svn co https://user@llvm.org/svn/llvm-project/llvm/tags/RELEASE_500/final llvm`
2. `cd llvm/tools`
3. `svn co http://llvm.org/svn/llvm-project/cfe/tags/RELEASE_500/final clang`
4. `cd clang/tools # (To be clear, you are now in llvm/tools/clang/tools)`
5. `svn co http://llvm.org/svn/llvm-project/clang-tools-extra/tags/RELEASE_500/final extra`
6. `cd ../../../../llvm/projects # (To be clear, you are now in llvm/projects)`
7. `svn co http://llvm.org/svn/llvm-project/compiler-rt/tags/RELEASE_500/final compiler-rt`
8. `cd ../../.. (You are now in your desktop directory)`
9. `mkdir build (if you have not already done so)`
10. `cd build (You are now in your build directory)`
11. `cmake -DLLVM_TARGETS_TO_BUILD="X86" -DLLVM_TARGET_ARCH=X86  
-DCMAKE_BUILD_TYPE="Release" -DLLVM_BUILD_EXAMPLES=1  
-DCLANG_BUILD_EXAMPLES=1 -G "Unix Makefiles" ../source/llvm/`
12. `'make -j 8' (from within the build directory to start the process)`



# From a Terminal

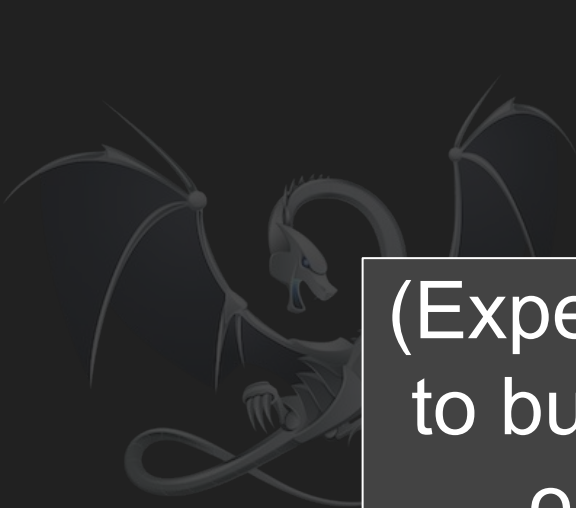
1. `svn co http://llvm.org/svn/llvm-project/llvm/trunk`
2. `cd llvm/tools`
3. `svn co http://llvm.org/svn/llvm-project/clang/trunk`
4. `cd clang/tools`
5. `svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk`
6. `cd ../../../../`
7. `svn co http://llvm.org/svn/llvm-project/llvm/trunk`
8. `cd ../../../../` (Y)
9. `mkdir build`
10. `cd build (Y)`
11. `cmake -DCLANG_BUILD_EXAMPLES=1 -DLLVM_BUILD_EXAMPLES=1 -DCMAKE_BUILD_TYPE="Release" -DCLANG_BUILD_EXAMPLES=1 -G "Unix Makefiles" ../source/llvm/`
12. `'make -j 8'` (from within the build directory to start the process)

Now get lunch/dinner/breakfast  
depending on speed of your cpu.

# How will we know it worked?

- Check your build/bin directory
- It should look something like this
- Note that for the examples, clang++, and other tools are referenced from here!
  - If your system already has clang++ installed from a package manager, it may have a different version!

```
mike@mike-Lenovo-ideapad-Y700-14ISK~/Desktop/llvm_2_3_18/build/bin
mike:build$ cd bin/
mike:bin$ ls
arcmt-test          find-all-symbols  llvm-mcmarkup
BrainF              HowToUseJIT        llvm-modextract
bugpoint            Kaleidoscope-Ch2   llvm-mt
BuildingAJIT-Ch1    Kaleidoscope-Ch3   llvm-nm
BuildingAJIT-Ch2    Kaleidoscope-Ch4   llvm-objdump
BuildingAJIT-Ch3    Kaleidoscope-Ch5   llvm-opt-report
BuildingAJIT-Ch4    Kaleidoscope-Ch6   llvm-pdbutil
BuildingAJIT-Ch5    Kaleidoscope-Ch7   llvm-PerfectShuffle
BuildingAJIT-Ch5-Server Kaleidoscope-Ch8   llvm-profdata
c-arcmt-test        llc                 llvm-ranlib
c-index-test        lli                 llvm-readelf
clang               lli-child-target   llvm-readobj
clang++             llvm-ar             llvm-rtdyld
clang-5.0            llvm-as             llvm-size
clang-apply-replacements llvm-bcanalyzer     llvm-split
clang-change-namespaces llvm-cat             llvm-stress
clang-check          llvm-config         llvm-strings
clang-cl             llvm-cov            llvm-symbolizer
clang-cpp            llvm-c-test         llvm-tblgen
clangd               llvm-cvtres         llvm-xray
clang-format         llvm-cxxdump        modularize
clang-import-test    llvm-cxxfilt        ModuleMaker
clang-include-fixer  llvm-diff           not
clang-interpreter    llvm-dis            obj2yaml
clang-move           llvm-dlltool        opt
clang-offload-bundler llvm-dsymutil       ParallelJIT
clang-query          llvm-dwarfdump      pp-trace
clang-rename         llvm-dwp            sancov
clang-reorder-fields llvm-extract         sanstats
clang-tblgen         llvm-lib            scan-build
clang-tidy           llvm-link           scan-view
count               llvm-lit            tool-template
diagtool            llvm-lto            verify-uselistorder
Fibonacci           llvm-lto2           yaml2obj
FileCheck            llvm-mc             yaml-bench
```



(Expect ~15-45 or more minutes  
to build from source depending  
on your cpu and internet  
connection)

Assumption: We all have a  
working LLVM at this point

# Our first example | Emitting LLVMs intermediate form

- We can output and actually look at LLVM's intermediate form.
- We are going to use the 'clang++' compiler
  - clang and clang++ are frontends for the C/C++ language.
  - The code they generate targets the LLVM intermediate form.
    - Let us try!

# Our first example | Emitting LLVMs intermediate form

- Here is some code we can use
  - hello.cpp

```
1 #include <stdio.h>
2
3 int main(){
4     printf("Bonjour!\n");
5     return 0;
6 }
```

## Compile and run

```
mike:examples$ ../../clang++ hello.cpp -o hello
mike:examples$ ./hello
Bonjour!
```

## Compile and run

```
mike:examples$ ../../clang++ hello.cpp -o hello
mike:examples$ ./hello
Bonjour!
```

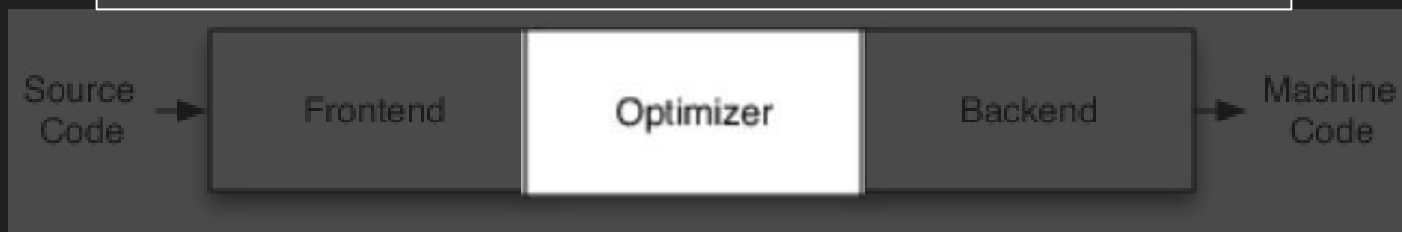
Again, make sure you are using the correct version of clang++ that we built!

```
mike:examples$ ../../clang++ --version
clang version 5.0.0 (tags/RELEASE_500/final 324176)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /home/mike/Desktop/llvm_2_3_18/build/bin/examples/../../
```

# Now we can use clang++ to emit LLVM IR

Our goal: Get an intermediate representation

Then we can talk more about this step:



Now we can use clang++ to emit LLVM IR

```
mike:examples$ clang++ -S -emit-llvm hello.cpp
```

## Now we can use clang++ to emit LLVM IR

```
mike:examples$ clang++ -S -emit-llvm hello.cpp
```

- Compiler arguments explained
  - -S -- only run preprocessor and compilation steps
  - -emit-llvm -- Use the LLVM Representation for assembler and object files

(Use clang++ -help to see options)

## Aside: Clang++, isn't this an LLVM talk?

- The news my friends is that LLVM has expanded since the early 2000s!
- LLVM is an umbrella of tools

LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, **LLVM** has grown to be an umbrella project consisting of a number of subprojects, many



# LLVM Tools - clang/clang++

1. clang - Clang is the frontend C/C++ compiler (llvm is the backend)
  - Likely you have heard or used Clang even if you did not know it!
2. llvm-as - Takes LLVM IR in assembly form and converts it to bitcode format.
3. llvm-dis - Converts bitcode to text readable llvm assembly
4. llvm-link - Links two or more llvm bitcode files into one file.
5. lli - Directly executes programs bit-code using JIT
6. llc - Static compiler that takes llvm input (assembly or bitcode) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
  - <http://llvm.org/docs/GettingStarted.html#llvm-tools>

What a second Mike!

So clang or perhaps other tools  
can work with this “LLVM”

☐ Yes

☐ No

What a second Mike!

So clang or perhaps other tools  
can work with this “LLVM”



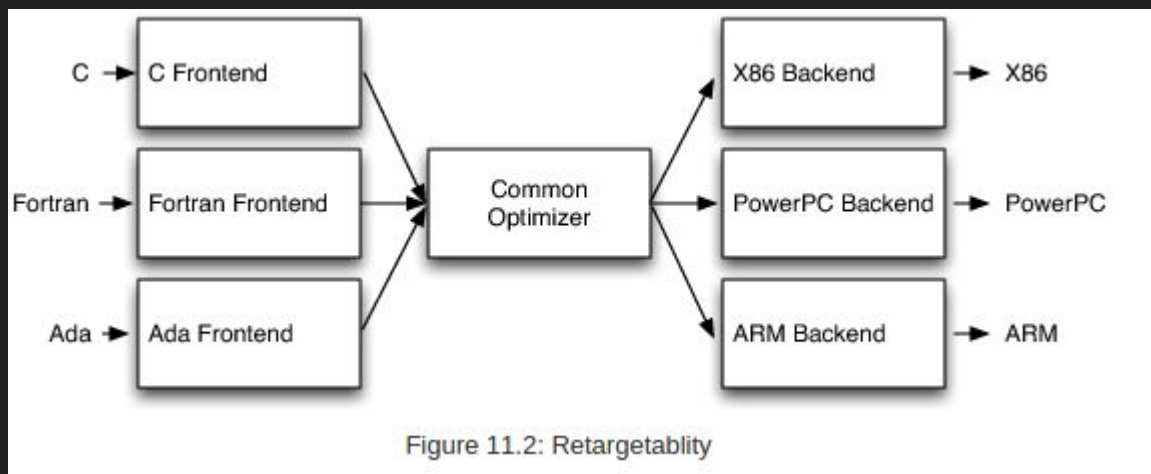
Yes



No

# Modularity

- A key feature is that language frontends can all target the same IR
- The optimizer can optimize that IR
- And the code generator can just the same target many other targets



# Modularity

- A key feature
- The optimizer
- And the code generator

Okay, now let us take a closer look at that IR

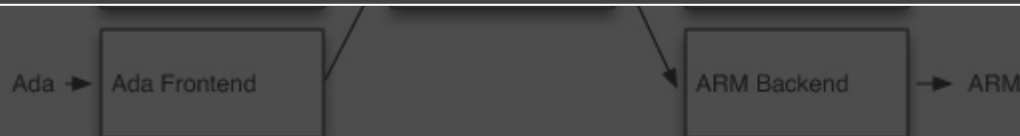


Figure 11.2: Retargetability

[Pop Quiz] What does this function do?

```
define i32 @add1(i32 %a, i32 %b) {  
entry:  
    %tmp1 = add i32 %a, %b  
    ret i32 %tmp1  
}
```

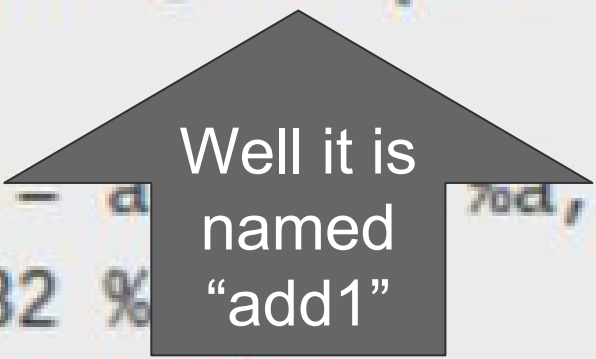
Guesses from  
the audience?

[Pop Quiz] What does this function do?

```
define i32 @add1(i32 %a, i32 %b) {  
entry:  
    %tmp1 = add i32 %a, %b  
    ret i32 %tmp1  
}
```

[Pop Quiz] What does this function do?


```
define i32 @add1(i32 %a, i32 %b) {  
entry:  
    %tmp1 = add i32 %a, %b  
    ret i32 %tmp1  
}
```



Well it is  
named  
"add1"

[Pop Quiz] What does this function do?

```
define i32 @add1(i32 %a, i32 %b) {  
entry:  
    %tmp1 = add i32 %a, %b  
    ret i32 %tmp1  
}
```



There are 2  
i32  
arguments

[Pop Quiz] What does this function do?

```
define i32 @add1(i32 %a, i32 %b) {  
entry:  
    %tmp1 = add i32 %a, %b  
    ret i32 %tmp1  
}
```



i32 = int


[Pop Quiz] What does this function do?

```
define i32 @add1(i32 %a, i32 %b) {
```

```
entry:
```

```
    tmp1 = add i32 %a, %b
```


```
    ret i32 %tmp1
```



Every  
function  
has a  
starting  
point

[Pop Quiz] What does this function do?

```
define i32 @add1(i32 %a, i32 %b) {  
entry:  
    %tmp1 = add i32 %a, %b  
    ret i32 %tmp1  
}
```



We store a  
result of an  
'add'  
operation

[Pop Quiz] What does this function do?

```
define i32 @add1(i32 %a, i32 %b) {  
entry:  
    %tmp1 = add i32 %a, %b  
    ret i32 %tmp1  
}
```



Then return the result as an int

[Pop Quiz] What does this function do?

If you can read assembly (or  
even C!) you can understand  
LLVM  
Intermediate Representation

# LLVM's Secret Sauce

# LLVM IR

- The LLVM IR can be targeted by many languages (we have discussed that)
  - It is fairly readable
  - It is also fairly writeable, considered a [first-class language](#)!
    - It is well-defined! (You have an alternative to targeting 'C' as your IR language :) )
- Other takeaways
  - The IR is strongly typed (e.g. i32 or even with pointers such as i32\*)
  - There are an infinite number of registers
    - You did not see a finite amount of registers like %rax, %rdx, %r15 if you are use to x86
    - Rather, anything that starts with '%' is a temporary register
    - IR uses Single Static Assignment ([SSA](#)) form.
      - Aides in program analysis and compiler optimizations
        - Constant Propagation
        - Dead Code Elimination
        - etc.

## (Quick Aside: SSA example from wikipedia)

[https://en.wikipedia.org/wiki/Static\\_single\\_assignment\\_form](https://en.wikipedia.org/wiki/Static_single_assignment_form)

```
y := 1
```

```
y := 2
```

```
x := y
```

Not  
SSA

```
y1 := 1
```

```
y2 := 2
```

```
x1 := y2
```

Uses  
SSA

# (Quick Aside: SSA example from wikipedia)

[https://en.wikipedia.org/wiki/Static\\_single\\_assignment\\_form](https://en.wikipedia.org/wiki/Static_single_assignment_form)

```
y := 1  
y := 2  
x := y
```

Not  
SSA

```
y1 := 1  
y2 := 2  
x1 := y2
```

Uses  
SSA

Quickly notice we  
can eliminate an  
extra variable

(Again, more examples from [AOSA](#) book from Lattner himself)

### 11.3. LLVM's Code Representation: LLVM IR

With the historical background and context out of the way, let's dive into LLVM: The most important aspect of its design is the LLVM Intermediate Representation (IR), which is the form it uses to represent code in the compiler. LLVM IR is designed to host mid-level analyses and transformations that you find in the optimizer section of a compiler. It was designed with many specific goals in mind, including supporting lightweight runtime optimizations, cross-function/interprocedural optimizations, whole program analysis, and aggressive restructuring transformations, etc. The most important aspect of it, though, is that it is itself defined as a first class language with well-defined semantics. To make this concrete, here is a simple example of a `.ll` file:

```
define i32 @add1(i32 %a, i32 %b) {
entry:
    %tmp1 = add i32 %a, %b
    ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
    %tmp1 = icmp eq i32 %a, 0
    br i1 %tmp1, label %done, label %recurse

recurse:
    %tmp2 = sub i32 %a, 1
    %tmp3 = add i32 %b, 1
    %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
    ret i32 %tmp4

done:
    ret i32 %b
}
```

# Using Clang++ and Generating IR

## Example 1 | hello.cpp

```
mike:examples$ clang++ -S -emit-llvm hello.cpp
```

- Returning to our example of ‘hello world’
- This command generated a .ll file (two lower-case L’s).
  - .ll files are the ‘textual’ form of LLVM’s IR.

```
mike:examples$ ls  
hello hello.cpp hello.ll
```

(Note ubuntu users: if the above failed, try adding `-fno-use-cxa-atexit` [link](#))

## And here it is:

```
1 ; ModuleID = 'hello.cpp'
2 target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
3 target triple = "x86_64-pc-linux-gnu"
4
5 @.str = private unnamed_addr constant [10 x i8] c"Bonjour!\0A\00", align 1
6
7 ; Function Attrs: norecurse uwtable
8 define i32 @main() #0 {
9     %1 = alloca i32, align 4
10    store i32 0, i32* %1, align 4
11    %2 = call i32 @i8*, ... @printf(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @.str, i
12    32 0, i32 0))
13    ret i32 0
14 }
15 declare i32 @printf(i8*, ...) #1
16
17 attributes #0 = { norecurse uwtable "disable-tail-calls"="false" "less-precise-fpmad"="false"
    "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "n
```

Audience,  
what stands  
out?

Pause -- Really take a second to look at the IR  
What jumps out at you in this snippet?

```
1 ; ModuleID = 'hello.cpp'
2 target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
3 target triple = "x86_64-pc-linux-gnu"
4
5 @.str = private unnamed_addr constant [10 x i8] c"Bonjour!\0A\00", align 1
6
7 ; Function Attrs: norecurse uwtable
8 define i32 @main() #0 {
9     %1 = alloca i32, align 4
10    store i32 0, i32* %1, align 4
11    %2 = call i32 @__printf(i8*, ...) @printf(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @.str, i
12    32 0, i32 0))
13    ret i32 0
14 }
15 declare i32 @__printf(i8*, ...) #1
16
17 attributes #0 = { norecurse uwtable "disable-tail-calls"="false" "less-precise-fpmad"="false" "n
    "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "n
```

# My Findings

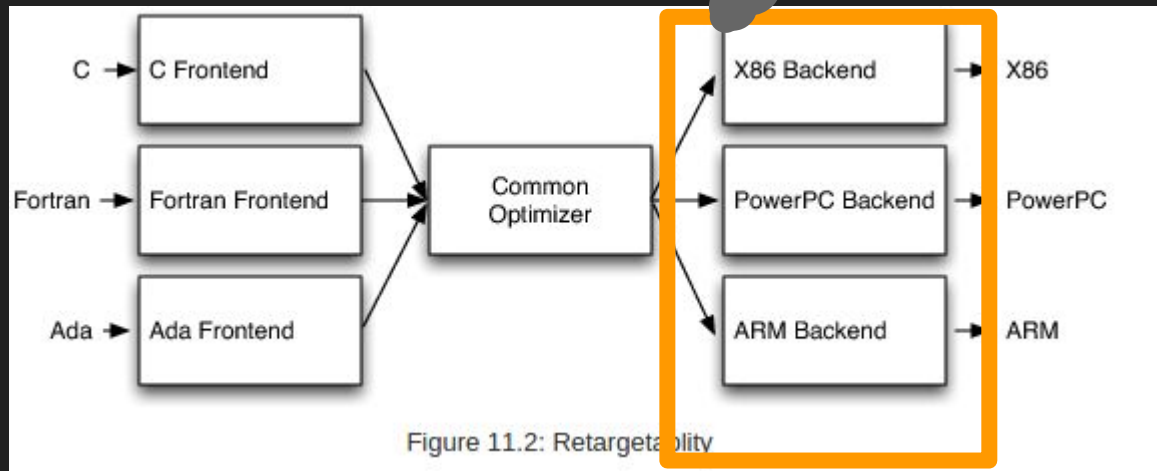
```
1 ; ModuleID = 'hello.cpp'
2 target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
3 target triple = "x86_64-pc-linux-gnu"
4
5 @.str = private unnamed_addr constant [10 x i8] c"Bonjour!\0A\00", align 1
6
7 ; Function Attrs: norecurse uwtable
8 define i32 @main() #0 {
9     %1 = alloca i32, align 4
10    store i32 0, i32* %1, align 4
11    %2 = call i32 @i8*, ... @printf(i8* getelementptr inbounds ([10 x i8], [10 x i8]* @.str, i
12    32 0, i32 0))
13    ret i32 0
14 }
15 declare i32 @printf(i8*, ...) #1
16
17 attributes #0 = { norecurse uwtable "disable-tail-calls"="false" "less-precise-fpmad"="false" "n
    "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fn-math"="false" "n
```

- Source filename
- [Data layout](#)
- [Target Triple](#)
- [Functions, Structure Types](#)
- Lots of % signs - These are registers (Remember the thing about SSA?)
- Other important things (not in this IR--[phi nodes](#))
- [Attributes](#)
- type information! Cool--better than assembly!
- Meta data (At the end with the "!")

# Targeting different backends

- Source filename
- [Data layout](#)
- [Target Triple](#)
- [Functions, Structure Types](#)
- Lots of % signs - These are registers
- Other important things (not in this IR--[phi nodes](#))
- [Attributes](#)
- type information! Cool--better than assembly!
- Meta data (At the end with the "!" )

Looks like good information to have for this stage (which we will not get to today)



# Targeting different backends

- Source files
- [Data layout](#)
- [Target Trip](#)
- [Functions,](#)
- Lots of % s registers
- Other impo this IR--phi
- [Attributes](#)
- type inform than assem
- Meta data (At the end with the "!" )

Are you enjoying the readability of IR yet?

Good news, machines like IR too



Figure 11.2: Retargetability

# LLVM Tools - lli

1. clang - Clang is the frontend C/C++ compiler (llvm is the backend)
  - Likely you have heard or used Clang even if you did not know it!
2. llvm-as - Takes LLVM IR in assembly form and converts it to bitcode format.
3. llvm-dis - Converts bitcode to text readable llvm assembly
4. llvm-link - Links two or more llvm bitcode files into one file.
5. **lli** - Directly executes programs bit-code using JIT
6. llc - Static compiler that takes llvm input (assembly or bitcode) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
  - <http://llvm.org/docs/GettingStarted.html#llvm-tools>

# The IR is very assembly like -- very readable!

- In fact the machine can read it, and the machine can directly execute the IR using it's Just-in-time (JIT compile for current architecture) execution engine.
- Let's do it now using `lli` ("L L I")
- What do you see?
  - Program should execute -- even though you did not see executable!
  - LLI can directly execute IR!

A terminal window with a dark background. The prompt is 'mike:examples\$'. The command entered is './../lli hello.ll'. The output of the command is 'Bonjour!' on the next line.

```
mike:examples$ ./../lli hello.ll
Bonjour!
```

- (If you're on Ubuntu 16.04--you may need an additional flag)
  - `../llvm_build/bin/clang++ -S -emit-llvm hello.cpp -fno-use-cxa-atexit`

# The IR is very assembly like -- very readable!

IR has a binary form called  
bitcode (.bc).  
Binary data will be more  
compact and thus to run through  
a JIT!

- In fact the IR is very assembly like -- very readable!
- Let's do it!
- What do we need?

- Program
- LLVM compiler

execute the IR  
in the JIT engine.

- (If you're on Ubuntu 16.04--you may need an additional flag)
  - `./../llvm_build/bin/clang++ -S -emit-llvm hello.cpp -fno-use-cxa-atexit`

# LLVM Tools - llvm-as

1. clang - Clang is the frontend C/C++ compiler (llvm is the backend)
  - Likely you have heard or used Clang even if you did not know it!
2. llvm-as - Takes LLVM IR in assembly form and converts it to bitcode format.
3. llvm-dis - Converts bitcode to text readable llvm assembly
4. llvm-link - Links two or more llvm bitcode files into one file.
5. lli - Directly executes programs bit-code using JIT
6. llc - Static compiler that takes llvm input (assembly or bitcode) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
  - <http://llvm.org/docs/GettingStarted.html#llvm-tools>

## Let's convert .ll to a .bc file | llvm-as

The llvm assembler converts the textual (or readable) IR to bitcode and now we have “hello.bc”.

```
nike:examples$ ../../llvm-as hello.ll
nike:examples$ ls
hello  hello.bc  hello.cpp  hello.ll
nike:examples$ vim hello.bc
```

Same result, as expected!

```
mike:examples$ ../../lli hello.bc  
Bonjour!
```

# lli executes bitcode (binary format of IR)

My claim is the JIT engine can execute more efficiently (Why?).

# lli executes bitcode (binary format of IR)

My claim is the JIT engine can execute more efficiently (Why?).

```
mike@mike-Lenovo-IdeaPad-Y700-14ISK ~/Desktop/llvm/examples
mike:examples$ head hello.bc
BC005b
0$LY0&00>-U000000
The immutab...
The main plan...
Although this p...
0!000000rory00p0uhx00w00rxzyh00q0s0r00600tiro 0!000000`00000000:h00;00000p00s(zh00(0p00
```

^binary representation of the textual .ll format we previously saw. A little more compressed, smaller file size.

lli executes bitcode (binary format of IR)

My claim is th

Eventually we may want the  
assembly for our target machine  
to build an executable

^binary repr  
compressed

little more

# LLVM Tools - llc

1. clang - Clang is the frontend C/C++ compiler (llvm is the backend)
  - Likely you have heard or used Clang even if you did not know it!
2. llvm-as - Takes LLVM IR in assembly form and converts it to bitcode format.
3. llvm-dis - Converts bitcode to text readable llvm assembly
4. llvm-link - Links two or more llvm bitcode files into one file.
5. lli - Directly executes programs bit-code using JIT
6. **llc** - Static compiler that takes llvm input (assembly or bitcode) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
  - <http://llvm.org/docs/GettingStarted.html#llvm-tools>

## The full circle -- compile our IR to assembly (.s file)

Run llc on our .bc file which creates an assembly file (hello.s)

```
mike:examples$ ../../llc hello.bc
mike:examples$ ls
hello  hello.bc_ hello.cpp  hello.ll  hello.s
```

# The full circle -- compile our IR to assembly (.s file)

Run llc on our .bc file which creates an assembly file (hello.s)

```
mike:examples$ ../../llc hello.bc
mike:examples$ ls
hello  hello.bc_  hello.cpp  hello.ll  hello.s
```

## hello.s

```
mike:examples$ cat hello.s
        .text
        .file   "hello.ll"
        .globl  main                # -- Begin function
        .p2align 4, 0x90
        .type   main,@function

main:
        .cfi_startproc              # @main
# BB#0:
        pusha    %rbp
```

# The full circle -- compile our IR to assembly (.s file)

A wide variety of targets are available for you to generate assembly code.

```
mike:examples$ ../../llc hello.bc -mcpu=help
Available CPUs for this target:

amdfam10      - Select the amdfam10 processor.
athlon        - Select the athlon processor.
athlon-4      - Select the athlon-4 processor.
athlon-fx     - Select the athlon-fx processor.
athlon-mp     - Select the athlon-mp processor.
athlon-tbird  - Select the athlon-tbird processor.
```

The full circle -- compile our IR to assembly (.s file)

At this point in the talk, we have played with IR and gotten familiar with some tools.

We have not utilized the optimizer, (i.e. Lattner's big idea)

# LLVM Tools - opt

1. clang - Clang is the frontend C/C++ compiler (llvm is the backend)
  - Likely you have heard or used Clang even if you did not know it!
2. llvm-as - Takes LLVM IR in assembly form and converts it to bitcode format.
3. llvm-dis - Converts bitcode to text readable llvm assembly
4. llvm-link - Links two or more llvm bitcode files into one file.
5. lli - Directly executes programs bit-code using JIT
6. llc - Static compiler that takes llvm input (assembly or bitcode) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
  - <http://llvm.org/docs/GettingStarted.html#llvm-tools>

# Lets run opt | ./../opt hello.ll --time-passes

```
mike:examples$ ./../opt hello.ll --time-passes
```

```
WARNING: You're attempting to print out a bitcode file.  
This is inadvisable as it may cause display problems. If  
you REALLY want to taste LLVM bitcode first-hand, you  
can force output with the '-f' option.
```

```
====  
... Pass execution timing report ...  
====
```

```
Total Execution Time: 0.0000 seconds (0.0000 wall clock)
```

---Wall Time---	--- Name ---
0.0000 (100.0%)	Module Verifier
0.0000 (100.0%)	Total

```
====  
LLVM IR Parsing  
====
```

```
Total Execution Time: 0.0000 seconds (0.0002 wall clock)
```

---Wall Time---	--- Name ---
0.0002 (100.0%)	Parse IR
0.0002 (100.0%)	Total

# Passes with 'opt'

- Opt is the 'optimizer'
- It works by making several passes through a module of code looking for opportunities to 'optimize' the code.
- There exists several ways to 'pass' through the code and gather information or make code changes.

```
mike:examples$ ../../opt hello.ll --time-passes
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the '-f' option.
```

```
=====
... Pass execution timing report ...
=====
Total Execution Time: 0.0000 seconds (0.0000 wall clock)

---Wall Time---  --- Name ---
0.0000 (100.0%)  Module Verifier
0.0000 (100.0%)  Total

=====
LLVM IR Parsing
=====
Total Execution Time: 0.0000 seconds (0.0002 wall clock)

---Wall Time---  --- Name ---
0.0002 (100.0%)  Parse IR
0.0002 (100.0%)  Total
```

# Passes with 'opt'

- Opt is the 'optimizer'
- It works by making several **passes** through a module of code looking for opportunities to 'optimize' the code.
- There exists several ways to 'pass' through the code and gather information or make code changes.

```
mike:examples$ ../../opt hello.ll --time-passes
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the '-f' option.
```

```
=====
                        ... Pass execution timing report ...
=====
Total Execution Time: 0.0000 seconds (0.0000 wall clock)

---Wall Time---  --- Name ---
0.0000 (100.0%)  Module Verifier
0.0000 (100.0%)  Total

=====
                        LLVM IR Parsing
=====
Total Execution Time: 0.0000 seconds (0.0002 wall clock)

---Wall Time---  --- Name ---
0.0002 (100.0%)  Parse IR
0.0002 (100.0%)  Total
```

# Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program.
    - i.e. A side effect occurs, which could invalidate other passes!

# Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program.
    - i.e. A side effect occurs, which could invalidate other passes!

# Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program.
    - i.e. A side effect occurs, which could invalidate other passes!

# Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program.
    - i.e. A side effect occurs, which could invalidate other passes!

# Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program.
    - i.e. A side effect occurs, which could invalidate other passes!

# Different Types of Passes in LLVM

- Levels of Granularity
  - Module Pass - Can think of this as a single source file
  - Call Graph Pass - Traverses a program bottom-up
  - Function Pass - Runs over individual functions
  - Basic Block Pass - Runs over individual basic blocks within a function
  - (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)
- Analysis Passes versus Transform pass
  - Analysis Pass - Computes information that other passes can use for debugging
  - Transform Pass - Mutates the program.
    - i.e. A side effect occurs, which could invalidate other passes!

## Different

- Levels of
  - Modu
  - Call C
  - Func
  - Basid
  - (Imm
- Analysis
  - Analy
  - Trans

## Our next task:

Learn how to analyze IR with passes. This can lead toward paths of:

1. Code optimization
2. Code understanding
3. etc.

# Goal - Print all of the Functions in a program

- What do we need? (Question for the audience)
- a.) [Module Pass](#) - Can think of this as a single source file
- b.) [Call Graph Pass](#) - Traverses a program bottom-up
- c.) [Function Pass](#) - Runs over individual functions
- d.) [Basic Block Pass](#) - Runs over individual basic blocks within a function
- e.) ([Immutable Pass](#), [Region Pass](#), [MachineFunctionPass](#) - Less important for today)

# Goal - Print all of the Functions in a

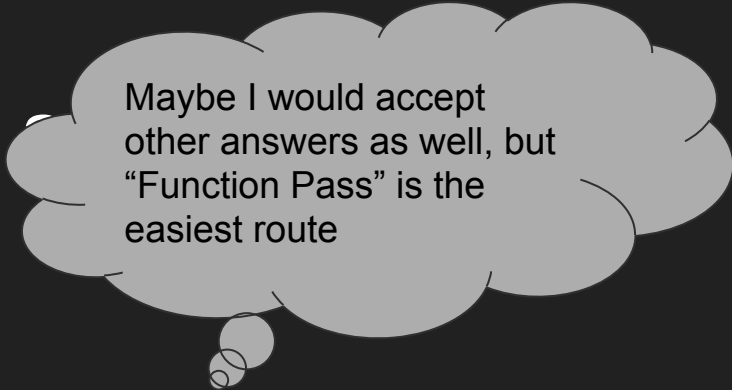
Guesses from  
the audience?

- What do we need? (Question for the audience)
- a.) [Module Pass](#) - Can think of this as a single source file
- b.) [Call Graph Pass](#) - Traverses a program bottom-up
- c.) [Function Pass](#) - Runs over individual functions
- d.) [Basic Block Pass](#) - Runs over individual basic blocks within a function
- e.) ([Immutable Pass](#), [Region Pass](#), [MachineFunctionPass](#) - Less important for today)

# Goal - Print all of the Functions in a program

- What do we need?
  - a.) Module Pass - Can think of this as a single source file
  - b.) Call Graph Pass - Traverses a program bottom-up
- c.) Function Pass - Runs over individual functions
  - d.) Basic Block Pass - Runs over individual basic blocks within a function
  - e.) (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)

# Goal - Print all of the Functions in



Maybe I would accept other answers as well, but “Function Pass” is the easiest route

- What do we need?
  - a.) Module Pass - Can think of this as a single source file
  - b.) Call Graph Pass - Traverses a program bottom-up
- c.) Function Pass - Runs over individual functions
  - d.) Basic Block Pass - Runs over individual basic blocks within a function
  - e.) (Immutable Pass, Region Pass, MachineFunctionPass - Less important for today)

# Writing Our First Function Pass

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
struct Hello : public FunctionPass {
    static char ID;
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(F) {
        errs() << "Hello: ";
        errs().write_escaped(F.getName()) << '\n';
        return false;
    }
}; // end of struct Hello
} // end of anonymous namespace

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass",
                             false /* Only looks at CFG */,
                             false /* Analysis Pass */);
```

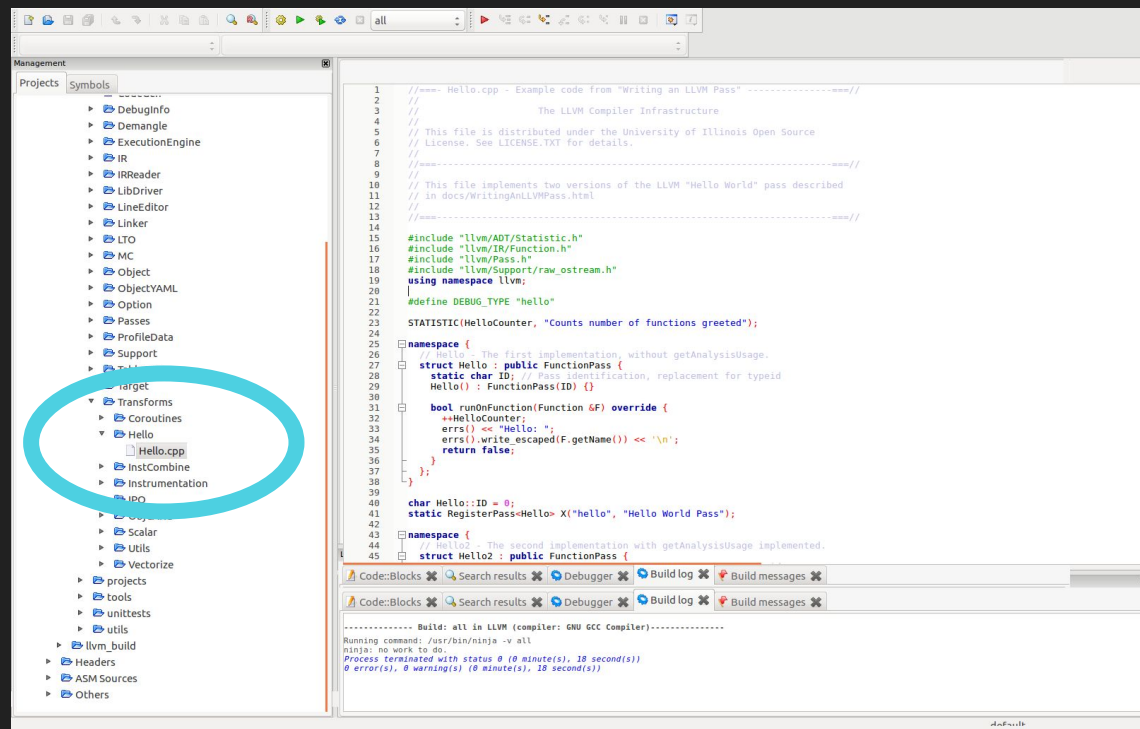
We will be working in: `llvm/lib/Transforms/Hello/Hello.cpp`

```
mike:Hello$ ls
CMakeLists.txt  Hello.cpp  Hello.exports
mike:Hello$ pwd
/home/mike/Desktop/llvm_2_3_18/source/llvm/lib/Transforms/Hello
```

- This is given to you when you download LLVM
  - (You can learn how to add more passes [here](#))

# (A visual if anyone setup Codeblocks)

This is given to you when you download LLVM (You can learn how to add more passes [here](#))



```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
```

```
using namespace llvm;
```

```
namespace {
struct Hello {
    static
    Hello()
```

```
    bool run(
        errs(
        errs(
        return
    }
}; // end
```

```
} // end
```

```
char Hello::ID = 0;
```

```
static RegisterPass<Hello> X("hello", "Hello World Pass",
                              false /* Only looks at CFG */,
                              false /* Analysis Pass */);
```

Okay, here is  
hello.cpp

It is a FunctionPass

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
```

```
using namespace llvm;
```

```
namespace
```

```
struct He
```

```
static
```

```
Hello()
```

```
bool ru
```

```
errs(
```

```
errs(
```

```
return
```

```
}
```

```
}; // end
```

```
} // end
```

```
char Hello::ID = 0;
```

```
static RegisterPass<Hello> X("hello", "Hello World Pass",
                             false /* Only looks at CFG */,
                             false /* Analysis Pass */);
```

(This code is included  
with LLVM)



```
bool runOnFunction(Function &F) override {
    errs() << "Hello: ";
    errs().write_escaped(F.getName()) << '\n';
    return false;
}
```

# Building our hello pass

- Navigate to the build directory
- In the lib/Transforms/Hello folder you'll find a make file
- type 'make'
- Any changes we have made will build.

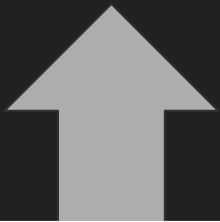
```
mike:Hello$ pwd
/home/mike/Desktop/llvm_2_3_18/build/lib/Transforms/Hello
mike:Hello$ make
[  0%] Built target LLVMHello_exports
[  0%] Built target obj.llvm-tblgen
[  0%] Built target LLVMDemangle
[ 66%] Built target LLVMSupport
[100%] Built target LLVMTableGen
[100%] Built target llvm-tblgen
[100%] Built target intrinsics_gen
[100%] Built target LLVMHello
```

Our pass is then compiled in build/lib/ as LLVMHello.so

```
libclangTidyMiscModule.a
libclangTidyModernizeModule.a
libclangTidyMPIModule.a
libclangTidyPerformanceModule.a
libclangTidyPlugin.a
libclangTidyReadabilityModule.a
libclangTidyUtils.a
libclangTooling.a
libclangToolingCore.a
libclangToolingRefactor.a
libDynamicLibraryLib.a
libfindAllSymbols.a
libgtest.a
libgtest_main.a
libLLVMAnalysis.a
libLLVMAsmParser.a
libLLVMAsmPrinter.a
libLLVMBinaryFormat.a
libLLVMBitReader.a
mike:lib$ pwd
/home/mike/Desktop/llvm_2_3_18/build/lib
LLVMHello.so
LTO
Makefile
MC
Object
ObjectYAML
Option
Passes
PrintFunctionNames.so
ProfileData
SampleAnalyzerPlugin.so
Support
TableGen
Target
Testing
ToolDrivers
Transforms
XRay
```

## Run our first pass with opt on hello.bc

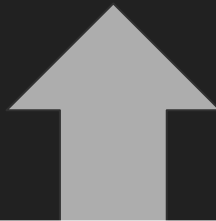
```
mike:examples$ ../../opt -load ../../lib/LLVMHello.so -hello < hello.bc
```



opt tool which  
we have used  
before

## Run our first pass with opt on hello.bc

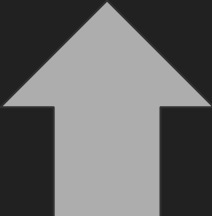
```
mike:examples$ ../../opt -load ../../lib/LLVMHello.so -hello < hello.bc
```



We load the  
library which  
contains our  
passes

## Run our first pass with opt on hello.bc

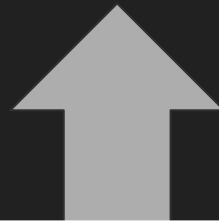
```
mike:examples$ ../../opt -load ../../lib/LLVMHello.so -hello < hello.bc
```



Path to our  
LLVMHello  
pass library

## Run our first pass with opt on hello.bc

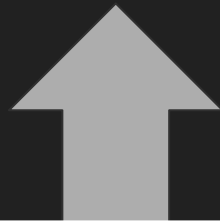
```
mike:examples$ ../../opt -load ../../lib/LLVMHello.so -hello < hello.bc
```



The particular  
function pass  
we want to run

## Run our first pass with opt on hello.bc

```
mike:examples$ ../../opt -load ../../lib/LLVMHello.so -hello < hello.bc
```



Our input file  
(.bc or .ll file)

# Run our first pass with opt on hello.bc

```
mike:examples$ ../../opt -load ../../lib/LLVMHello.so -hello < hello.bc > /dev/null  
Hello: main  
_
```

- Neat--we see all of the functions!
  - Or rather, we have one 'main' function in our program.

# Anatomy of a “Pass”

```
bool runOnFunction(Function &F) override {
    errs() << "Hello: ";
    errs().write_escaped(F.getName()) << '\n';
    return false;
}
```

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
struct Hello : public FunctionPass {
    static char ID;
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
        errs() << "Hello: ";
        errs().write_escaped(F.getName()) << '\n';
        return false;
    }
}; // end of struct Hello
} // end of anonymous namespace

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass",
                             false /* Only looks at CFG */,
                             false /* Analysis Pass */);
```

We are not  
'mutating code'  
so return false.

[illegible][illegible]

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
struct Hello : public FunctionPass {
    static char ID;
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
        errs() << "Hello: ";
        errs().write_escaped(F.getName()) << '\n';
        return false;
    }
}; // end of struct Hello
} // end of anonymous namespace
```

```
char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass",
                             false /* Only looks at CFG */,
                             false /* Analysis Pass */);
```

Register the pass. This is how the pass is built

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
struct Hello : public FunctionPass {
    static char ID;
    Hello() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
        errs() << "Hello: ";
        errs().write_escaped(F.getName()) << '\n';
        return false;
    }
}; // end of struct Hello
} // end of anonymous namespace
```

```
char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass",
                             false /* Only looks at CFG */,
                             false /* Analysis Pass */);
```

i.e. how I knew  
what to type in  
the comand line  
in our example

Congratulations on  
writing/running your  
first pass

LLVM is properly  
configured, on to  
more analysis

```
#include "llvm/Pass.h"
#include
#include
```

```
using nam
```

```
namespace
```

```
struct He
```

```
static
```

```
Hello()
```

```
bool ru
```

```
errs(
```

```
errs(
```

```
return
```

```
}
```

```
}; // end
```

```
} // end
```

```
char Hell
```

```
static Re
```

```
Pass",  
CFG */,
```

```
return / "Analysis Pass */);
```

# Static Analysis

**Goal of Static Analysis:** What information/bugs/performance errors can we uncover before we run the program.

**Pros:** Gives us full coverage of program

**Cons:** No real runtime data, overly conservative

## Our Second pass -- This time we collect some program stats

1. It will print the function name
2. It will count basic blocks and instruction counts.

## Our Second pass -- This time we collect some program stats

1. It will print the function name
2. It will count basic blocks and instruction counts.
3. We'll use this new sample source code -- or even better use one of your own!

```
1 #include <stdio.h>
2
3 void countDown(){
4     int x = 0;
5     while(x<10){
6         ++x;
7     }
8 }
9
10 int addFunc(int a, int b){
11     return a+b;
12 }
13
14
15 int main(){
16
17     printf("5+2=%i\n",addFunc(5,2));
18     countDown();
19
20     return 0;
21 }
```

# Compile and Test loops.cpp and use loops.ll on -hello pass

## 1. Compile program to IR

- a. `./../clang++ -S -emit-llvm loops.cpp`
- b. Test opt with our old pass (note we can just use the .ll version for this sample)
  - i. `./../opt -load ../../lib/LLVMHello.so -hello < loops.ll > /dev/null`

```
mike:examples$ ./../opt -load ../../lib/LLVMHello.so -hello < loops.ll > /dev/null
Hello: _Z9countDownv
Hello: _Z7addFuncii
Hello: main
```

# The Stats Pass source code

```
43 namespace {
44     // Hello2 - The second implementation with getAnalysisUsage implemented.
45     struct Hello2 : public FunctionPass {
46         static char ID; // Pass identification, replacement for typeid
47         Hello2() : FunctionPass(ID) {}
48
49         bool runOnFunction(Function &F) override {
50             unsigned int basicBlockCount = 0;
51             unsigned int instructionCount = 0;
52             for (BasicBlock &BB : F.getBasicBlockList()) {
53                 ++basicBlockCount;
54                 for (Instruction &I : BB) {
55                     ++instructionCount;
56                 }
57             }
58             errs() << "Hello2: basicBlockCount = " << basicBlockCount << "\n";
59             errs().write_ones();
60
61             return false;
62         }
63     }
64
65     // We don't modify the program, so we preserve all analyses.
66     void getAnalysisUsage(AnalysisUsage &AU) const override {
67         AU.setPreservesAll();
68     }
69 };
70 }
```

[michaeldshah.net/LLVM/Intro/hello.cpp](http://michaeldshah.net/LLVM/Intro/hello.cpp)

Okay, here is our second pass

It is a FunctionPass that collects stats

basicBlockCount  
instructionCount << "\n";

# The Stats Pass source code

```

43 namespace {
44     // Hello2 - The second implementation with getAnalysisUsage implemented.
45     struct Hello2 : public FunctionPass {
46         static char ID; // Pass identification, replacement for typeid
47         Hello2() : FunctionPass(ID) {}
48
49         bool runOnFunction(Function &F) override {
50             unsigned int basicBlockCount = 0;
51             unsigned int instructionCount = 0;
52             for(BasicBlock &bb : F){
53                 ++basicBlockCount;
54                 for(Instruction &i: bb){
55                     ++instructionCount;
56                 }
57             }
58             errs() << "Hello2: ";
59             errs().write_escaped(F.getName()) << "Basic Blocks: " << basicBlockCount
60             << "Instructions:" << instructionCount << "\n";
61
62             return false;
63         }
64
65         // We don't modify the program, so we preserve all analyses.
66         void getAnalysisUsage(AnalysisUsage &AU) const override {
67             AU.setPreservesAll();
68         }
69     };
70 }

```

Here is where we will accumulate the basic blocks and instructions within our function

# The Stats Pass source code

```

43 namespace {
44     // Hello2 - The second implementation with getAnalysisUsage implemented.
45     struct Hello2 : public FunctionPass {
46         static char ID; // Pass identification, replacement for typeid
47         Hello2() : FunctionPass(ID) {}
48
49         bool runOnFunction(Function &F) override {
50             unsigned int basicBlockCount =
51             unsigned int instructionCount = 0;
52             for(BasicBlock &bb : F){
53                 ++basicBlockCount;
54                 for(Instruction &i: bb){
55                     ++instructionCount;
56                 }
57             }
58             errs() << "Hello2: ";
59             errs().write_escaped(F.getName()) << "Basic Blocks: " << basicBlockCount
60             << "Instructions: " << instructionCount << "\n";
61
62             return false;
63         }
64
65         // We don't modify the program, so we preserve all analyses.
66         void getAnalysisUsage(AnalysisUsage &AU) const override {
67             AU.setPreservesAll();
68         }
69     };
70 }

```

Here notice, that within a function, we can iterate through its basic blocks, and every instruction within each basic block


# The Stats Pass source code

```

43 namespace {
44   // Hello2 - The second implementation with getAnalysisUsage implemented.
45   struct Hello2 : public FunctionPass {
46     static char ID; // Pass identification, replacement
47     Hello2() : FunctionPass(ID) {}
48
49     bool runOnFunction(Function &F) override {
50       unsigned int basicBlockCount = 0;
51       unsigned int instructionCount = 0;
52       for(BasicBlock &bb : F){
53         ++basicBlockCount;
54         for(Instruction &i: bb){
55           ++instructionCount;
56         }
57       }
58       errs() << "Hello2: ";
59       errs().write_escaped(F.getName()) << "Basic Blocks: " << basicBlockCount
60       << "Instructions:" << instructionCount << "\n";
61
62       return false;
63     }
64
65     // We don't modify the program, so we preserve all analyses.
66     void getAnalysisUsage(AnalysisUsage &AU) const override {
67       AU.setPreservesAll();
68     }
69   };
70 }

```

And finally we  
output this  
information



(Don't forget to save, and rebuild our pass)

```
mike:Hello$ pwd
/home/mike/Desktop/llvm_2_3_18/build/lib/Transforms/Hello
mike:Hello$ ls
CMakeFiles  cmake_install.cmake  LLVMHello.exports  Makefile
mike:Hello$ make
```

## Results of pass 2 (with loops.ll)

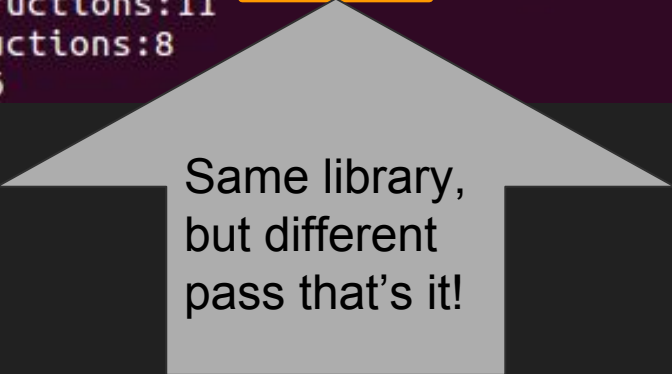
- `./../opt -load ./../..../lib/LLVMHello.so -hello2 < loops.ll > /dev/null`

```
mike:examples$ ./../opt -load ./../..../lib/LLVMHello.so -hello2 < loops.ll > /dev/null
Hello2: _Z9countDownv Basic Blocks: 4 Instructions:11
Hello2: _Z7addFuncii Basic Blocks: 1 Instructions:8
Hello2: main Basic Blocks: 1 Instructions:6
```

## Results of pass 2 (with loops.ll)

- `./../opt -load ./../lib/LLVMHello.so -hello2 < loops.ll > /dev/null`

```
mike:examples$ ./../opt -load ./../lib/LLVMHello.so -hello2 < loops.ll > /dev/null
Hello2: _Z9countDownv Basic Blocks: 4 Instructions:11
Hello2: _Z7addFuncii Basic Blocks: 1 Instructions:8
Hello2: main Basic Blocks: 1 Instructions:6
```



Same library,  
but different  
pass that's it!

## Results of pass 2 (with loops.ll)

- `./../opt -load ./../lib/LLVMHello.so -hello2 < loops.ll > /dev/null`

```
mike:examples$ ./../opt -load ./../lib/LLVMHello.so -hello2 < loops.ll > /dev/null
Hello2: _Z9countDownv Basic Blocks: 4 Instructions:11
Hello2: _Z7addFuncii Basic Blocks: 1 Instructions:8
Hello2: main Basic Blocks: 1 Instructions:6
```

Observe here, same pass runs on every function.

There is no “memory” here of previous runs. Need a data structure, analysis pass, or perhaps “module pass”

## Results of pass 2 (with loops.ll)

- `./../opt -load ./../lib/LLVMHello.so -hello2 < loops.ll > /dev/null`

```
mike:examples$ ./../opt -load ./../lib/LLVMHello.so -hello2 < loops.ll > /dev/null
Hello2: _Z9countDownv Basic Blocks: 4 Instructions:11
Hello2: _Z7addFuncii Basic Blocks: 1 Instructions:8
Hello2: main Basic Blocks: 1 Instructions:6
```

- Let's add more!
- What can we do with instruction information?

<http://llvm.org/docs/WritingAnLLVMPass.html>

## Writing an LLVM Pass

- Intro

- Quick

- 

- 

- 

- Pass

- 

- 

- The **CallGraphSCCPass** class

- The **doInitialization(CallGraph &)** method

- The **runOnSCC** method

- The **doFinalization(CallGraph &)** method

- The **FunctionPass** class

Here's homework for  
later!

I'm not pulling these  
ideas from nowhere!

```
82 namespace {
83     // Hello3 - The second implementation with getAnalysisUsage implemented.
84     struct Hello3 : public FunctionPass {
85         stat
86         Hell
87
88         bool
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105     }
106
107     // W
108     void getAnalysisUsage(AnalysisUsage &AU) const override {
109         AU.setPreservesAll();
110     }
111 };
112 }
113
114 char Hello3::ID = 0;
115 static RegisterPass<Hello3>
116 Z("hello3", "Hello World Pass (Get direct calls)");
117
```

Okay, here is our  
third pass

It is a FunctionPass  
that shows direct  
function calls

```

82 namespace {
83     // Hello3 - The second implementation with getAnalysisUsage implemented.
84     struct Hello3 : public FunctionPass {
85         static char ID; // Pass identification, replacement for typeid
86         Hello3() : FunctionPass(ID) {}
87
88         bool runOnFunction(Function &F) override {
89             for(BasicBlock &bb: F){
90                 for(Instruction &i: bb){
91                     // Find where callsite is of our instruction
92                     CallSite cs(&i);
93                     if(!cs.getInstruction()){
94                         continue;
95                     }
96                     Value *called = cs.getCalledValue()->stripPointerCasts();
97                     if(Function* f = dyn_cast<Function>(called)){
98                         errs() << "\tDirect call to function:" << f->getName()
99                             << " from " << F.getName();
100                     }
101                 }
102             }
103
104             return false;
105         }
106
107         // We don't modify the program, so we preserve all analyses.
108         void getAnalysisUsage(AnalysisUsage &AU) const override {
109             AU.setPreservesAll();
110         }
111     };
112 }
113
114 char Hello3::ID = 0;
115 static RegisterPass<Hello3>
116 Z("hello3", "Hello World Pass (Get direct calls)");

```

# Find Direct Calls

Added new header: `#include "llvm/IR/CallSite.h"`

```

88 bool runOnFunction(Function &F) override {
89     for(BasicBlock &bb: F){
90         for(Instruction &i: bb){
91             // Find where callsite is of our instruction
92             CallSite cs(&i);
93             if(!cs.getInstruction()){
94                 continue;
95             }
96             Value *called = cs.getCalledValue()->stripPointerCasts();
97             if(Function* f = dyn_cast<Function>(called)){
98                 errs() << "\tDirect Call to function:" << f->getName()
99                     << " from " << F.getName();
100             }
101         }
102     }
103
104     return false;
105 }
106
107 static RegisterPass<Hello3>
108 Z("hello3", "Hello World Pass (Get direct calls)");

```

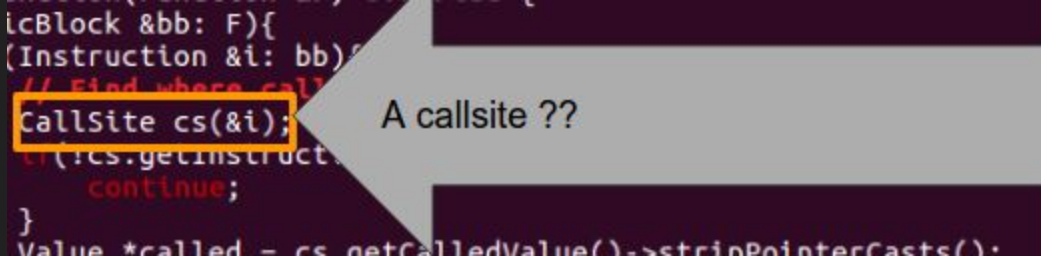
# Find Direct Calls

Added new header: `#include "llvm/IR/CallSite.h"`

```
88 bool runOnFunction(Function &F) override {
89     for(BasicBlock &bb: F){
90         for(Instruction &i: bb){
91             // Find where call
92             CallSite cs(&i);
93             if(!cs.getInstruction())
94                 continue;
95         }
96         Value *called = cs.getCalledValue()->stripPointerCasts();
97         if(Function* f = dyn_cast<Function>(called)){
98             errs() << "\tDirect Call to function:" << f->getName()
99                 << " from " << F.getName();
100         }
101     }
102 }
103
104 return false;
105 }
106
107 static RegisterPass<Hello3>
108 Z("hello3", "Hello World Pass (Get direct calls)");
```

A callsite ??

# LLVM Docs



- I do not actually know all of the LLVM commands by heart.
- As you start with LLVM, it is a good idea to keep the doxygen documentation open.
- “googling LLVM \_\_\_\_\_” will lead you to the correct page most often
  - [http://llvm.org/doxygen/classllvm\\_1\\_1CallSite.html](http://llvm.org/doxygen/classllvm_1_1CallSite.html)

**LLVM** 7.0.0svn

[Main Page](#) [Related Pages](#) [Modules](#) [Namespaces ▾](#) [Classes ▾](#) [Files ▾](#) [Examples](#)

[llvm](#) > [CallSite](#)

**llvm::CallSite Class Reference**

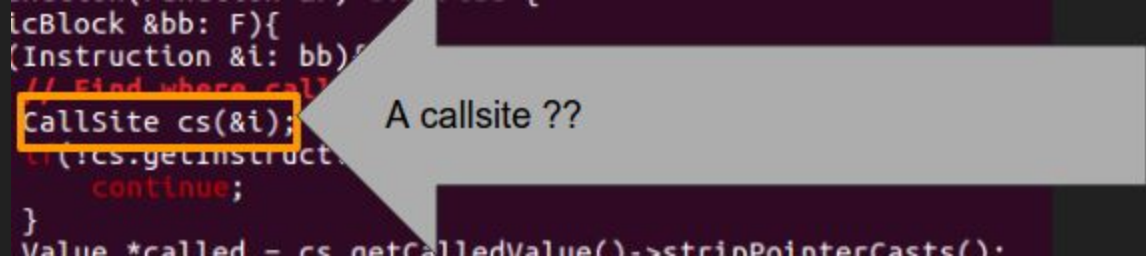
#include "llvm/IR/CallSite.h"

Inheritance diagram for llvm::CallSite:  

```
graph TD
    CallSite[llvm::CallSite] --> Function[Function]
    CallSite --> BasicBlock[BasicBlock]
```

< Function, BasicBlock,

# LLVM Docs



- From the documentation you can navigate to the appropriate function and even the source code

◆ **CallSite()** [5/6]

llvm::CallSite::CallSite ( **Instruction \* II** )

Definition at line **673** of file **CallSite.h**.

```
667  
668  
669 public:  
670   CallSite() = default;  
671   CallSite(CallSiteBase B) : CallSiteBase(B) {}  
672   CallSite(CallInst *CI) : CallSiteBase(CI) {}  
673   CallSite(InvokeInst *II) : CallSiteBase(II) {}  
674   explicit CallSite(Instruction *II) : CallSiteBase(II) {}  
675   explicit CallSite(Value *V) : CallSiteBase(V) {}  
676  
677   bool operator==(const CallSite &CS) const { return I == CS.I; }  
678   bool operator!=(const CallSite &CS) const { return I != CS.I; }  
679   bool operator<(const CallSite &CS) const {  
680     return getInstruction() < CS.getInstruction();  
681   }
```

# (Pssst! You have the source code as well)

Here is a sample grep

```
grep --include="*.cpp" -nr "getInstruction()" .
```

```
/llvm/lib/Transforms/IPO/ArgumentPromotion.cpp:320:      Call->replaceAllUsesWith(NewCS.getInstruction());  
/llvm/lib/Transforms/IPO/ArgumentPromotion.cpp:824:      if (CS.getInstruction() == nullptr || !CS.isCallee(&U))  
/llvm/lib/Transforms/IPO/ArgumentPromotion.cpp:827:      if (CS.getInstruction()->getParent()->getParent() == F)  
/llvm/lib/Transforms/IPO/ArgumentPromotion.cpp:1028:      Function *Caller = OldCS.getInstruction()->getParent()->getParent();  
/llvm/lib/Transforms/IPO/DeadArgumentElimination.cpp:163:      Instruction *Call = CS.getInstruction();  
/llvm/lib/Transforms/IPO/DeadArgumentElimination.cpp:187:      cast<CallInst>(NewCS.getInstruction())  
/llvm/lib/Transforms/IPO/DeadArgumentElimination.cpp:200:      Call->replaceAllUsesWith(NewCS.getInstruction());  
/llvm/lib/Transforms/IPO/DeadArgumentElimination.cpp:521:      const Instruction *TheCall = CS.getInstruction();
```

- Often times grepping through the source code gives you ideas of how to use instructions
- I myself do not pretend to be compare with the LLVM experts!

## (continued) Find Direct Calls

Added new header: `#include "Irm/IR/CallSite.h"`

```
88 bool runOnFunction(Function &F) override {
89     for(BasicBlock &bb: F){
90         for(Instruction &i: bb){
91             // Find where callsite is of
92             CallSite cs(&i);
93             if(!cs.getInstruction())
94                 continue;
95         }
96         Value *called = cs.getCalledValue()->stripPointerCasts();
97         if(Function* f = dyn_cast<Function>(called)){
98             errs() << "\tDirect Call to function:" << f->getName()
99                 << " from " << F.getName();
100         }
101     }
102 }
103
104 return false;
105 }
```

If our instruction is not a 'callable' (i.e. a function)

```
115 static RegisterPass<Hello3>
116 Z("hello3", "Hello World Pass (Get direct calls)");
```

# (continued) Find Direct Calls

Added new header: #include "FindDirectCalls.h"

Find out if our 'callee' is a direct function call (not a function pointer or anything)

```

88 bool runOnFunction(Function &F) override {
89     for(BasicBlock &bb: F){
90         for(Instruction &i: bb){
91             // Find where called
92             CallSite cs(&i);
93             if(!cs.getInstruction()){
94                 continue;
95             }
96             Value *called = cs.getCalledValue()->stripPointerCasts();
97             if(Function* f = dyn cast<Function>(called)){
98                 errs() << "\tDirect call to function:" << f->getName()
99                     << " from " << F.getName();
100             }
101         }
102     }
103     return false;
104 }
105
106 static RegisterPass<Hello3>
107 Z("hello3", "Hello World Pass (Get direct calls)");

```

# The Result!

```
mike:examples$ ../../opt -load ../../lib/LLVMHello.so -hello3 < loops.ll > /dev/null
Direct Call to function:_Z7addFuncii from main
Direct Call to function:printf from main
Direct Call to function:_Z9countDownv from main
mike:examples$
```

- Simple little function pass
- Now you can use this information to build a data structure
  - The function “F” is the caller, and “f” the callee.
  - Each of these forms an edge and could be put into a graph data structure.
  - Then output static graphs!

# Bonus Trick: Outputting graphs

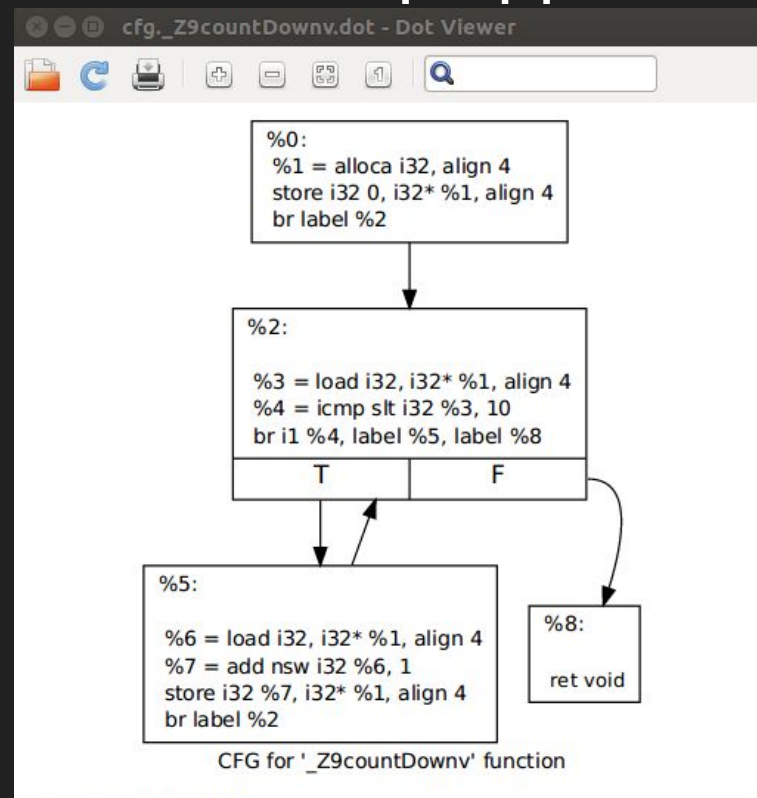
# LLVM actually provides a pass that can output control flow graphs

- Install a dot file viewer
  - `sudo apt install xdot (for linux)`
- Generate a dot file with
  - `./../opt -dot-cfg-only loops.ll > /dev/null`
- View dot file with
  - `xdot cfg._Z9countDownv.dot`

```
mike:examples$ ./../opt -dot-cfg-only loops.ll > /dev/null
Writing 'cfg._Z9countDownv.dot'...
Writing 'cfg._Z7addFunci.dot'...
Writing 'cfg.main.dot'...
```

# Here is the 'countdown function' from loops.pp

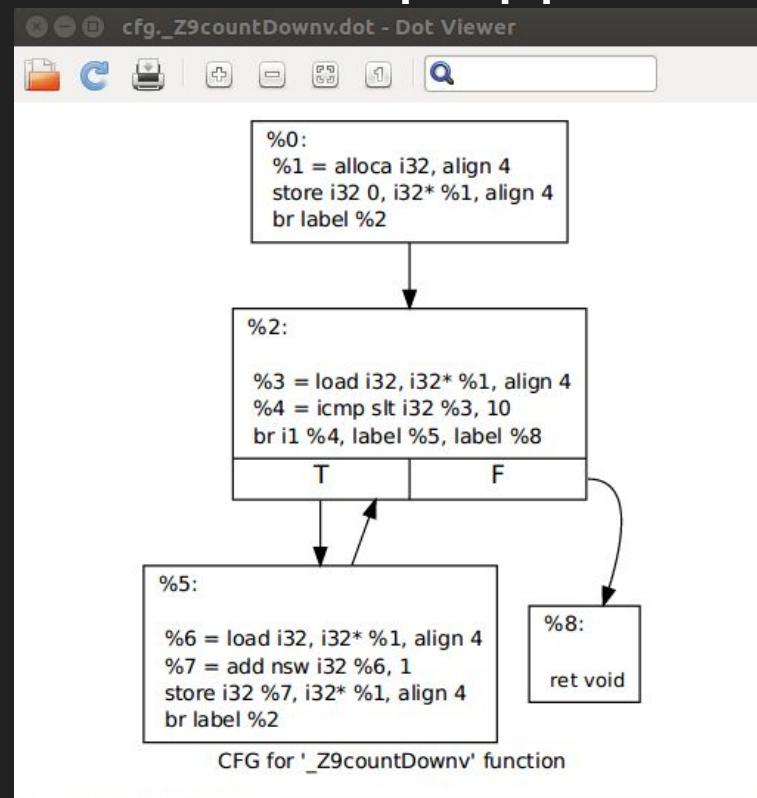
```
void countdown(){  
    int x = 0;  
    while(x<10){  
        ++x;  
    }  
}
```



# Here is the 'countdown function' from loops.pp

- You can slowly map each basic block from the visualization to the C++ code in this way.

```
void countdown(){  
    int x = 0;  
    while(x<10){  
        ++x;  
    }  
}
```



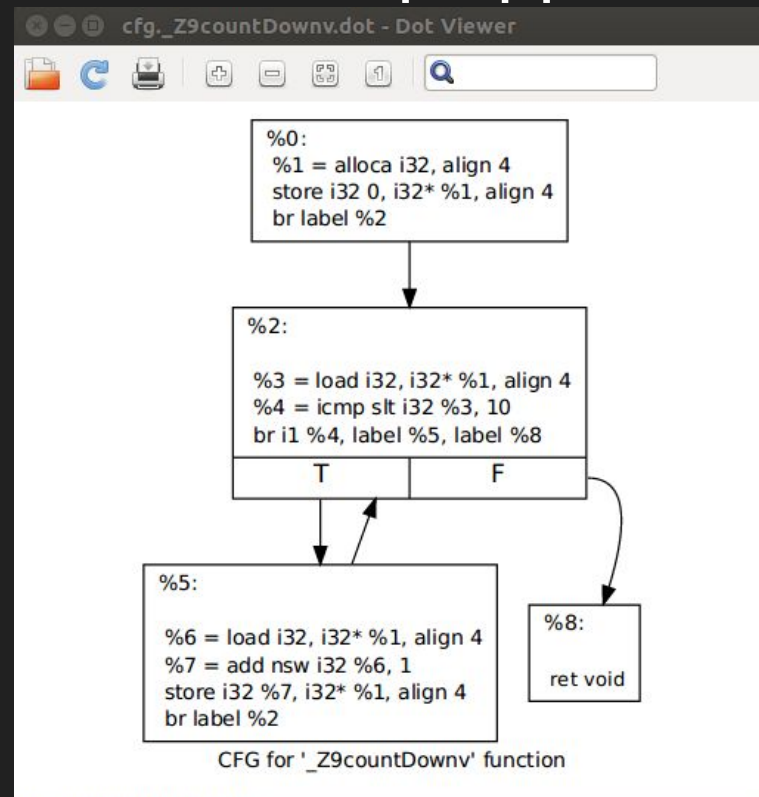
# Here is the 'countdown function' from loops.pp

- You can slowly map each basic block from the visualization or directly to the IR

```

8 ; Function Attrs: noinline nounwind optnone uwtable
9 define void @_Z9countDownv() #0 {
10     %1 = alloca i32, align 4
11     store i32 0, i32* %1, align 4
12     br label %2
13
14 ; <label>:2:                                ; preds = %5, %0
15     %3 = load i32, i32* %1, align 4
16     %4 = icmp slt i32 %3, 10
17     br i1 %4, label %5, label %8
18
19 ; <label>:5:                                ; preds = %2
20     %6 = load i32, i32* %1, align 4
21     %7 = add nsw i32 %6, 1
22     store i32 %7, i32* %1, align 4
23     br label %2
24
25 ; <label>:8:                                ; preds = %2
26     ret void
27 }

```



# Dynamic Analysis

**Goal of Dynamic Analysis:** What information/bugs/performance errors can we uncover when we run the program.

**Pros:** Gives us real values

**Cons:** Instrumentation effects results & Performance

Why use LLVM for this?

We can insert/inject code to monitor or change behavior of our code.

**Pros:** Gives us real values

**Cons:** Instrumentation effects results & Performance

# Adding in Functions (For Dynamic Analysis)

- Typically this is done in an ad-hoc fashion
  - Either spreading in 'printf' functions everywhere
  - Lots of #define #endif
- If we have our source code, we can inject code as needed.
  - No need to mess up or keep copies of various source versions.
- Fair warning, I am running through these examples fast, but you have the slides
  - (Lots of source code on slides ahead--I am breaking powerpoint rules!)

# Step 1:

Let's write some code that we want to instrument

# Step 1: Write a 'hook' or 'profiling code'

Let's write some code that we want to instrument

Here is a function '\_\_\_initMain' that will be inserted in our 'main' function and print a message

```
1 #include <stdio.h>
2
3 // This is the function that is
4 // called at the very start of the program.
5 // It will be called right after main.
6 // "dummyValue" does nothing except
7 // demonstrates how to pass arguments in our pass.
8 void ___initMain(int dummyValue){
9     printf("Hello, you are running an instrumented binary.\nPerformance may vary while running an instrumented binary.\n");
10    // Do more here..
11 }
```

# Step 1: Generate IR for hook

Now let's create the intermediate representation of our code.

Donzo. Finished. IR is ready

```
; Function Attrs: uwtable
define void @_Z10__initMaini(i32 %dummyValue) #0 {
    %1 = alloca i32, align 4
    store i32 %dummyValue, i32* %1, align 4
    %2 = call i32 @i8*, ... @printf(i8* getelementptr inbounds ([107 x i8], [107 x i8]* @.str, i32 0, i32 0))
    ret void
}
```

# Step 1: Generate IR for hook

Now let's create the intermediate representation of our code.

Donzo. Finished. IR is ready

```
; Function Attrs: uwtable
define void @_Z10__initMaini(i32 %dummyValue) #0 {
    %1 = alloca i32, align 4
    store i32 %dummyValue, %1, align 4
    %2 = call i32 @__getelementptr.inbounds ([107 x i8], [107 x i8], [107 x i8], @.str)
    ret void
}
```

This is our function name. Note it “looks weird”. It is a mangled function name.

## Step 2: Lets find the code we want to modify

How about our hello.cpp program. And we already have hello.ll from previous examples

This is the simplest program with one function

```
1 #include <stdio.h>
2
3 int main(){
4     printf("Bonjour!\n");
5     return 0;
6 }
```

## Now time for the Module pass

New headers needed: `#include "llvm/Module.h"`

Why?

- 1.) To show you a module pass
- 2.) It makes a little more sense (to me) to search functions in a module I want to instrument.

# The Module pass | Setup in 3 parts (in my code)

```
135     bool runOnModule(Module &M) override {
136         // Setup hooks
137         // Create a little stub function
138         setupHooks("_Z10__initMaini",M);
139
140         // Loop through all of our functions in the module
141         // This is where you could do something interesting like only
142         // modify a subset of the functions
143         // The key is not to modify instrumenting functions!
144         Module::FunctionListType &functions = M.getFunctionList();
145         for(Module::FunctionListType::iterator FI = functions.begin(), FE = functions.end(); FI !=
FE; ++FI){
146             // Ignore our instrumented function
147             if(FI->getName()=="_Z10__initMaini"){
148                 continue;
149             }
150
151             if(FI->getName()=="main"){
152                 InstrumentEnterFunction("_Z10__initMaini",*FI, M);
153             }
154         }
155         return true;
156     }
157
```

# The Module pass

## 1.) Create a “stub” function

```
135     bool runOnModule(Module &M) override {
136         // Setup hooks
137         // Create a little stub function
138         setupHooks("_Z10__initMaini",M);
139
140         // Loop through all of our functions
141         // This is where you could do something interesting like only
142         // modify a subset of the functions
143         // The key is not to modify instrumenting functions!
144         Module::FunctionListType &functions = M.getFunctionList();
145         for(Module::FunctionListType::iterator FI = functions.begin(), FE = functions.end(); FI !=
FE; ++FI){
146             // Ignore our instrumented function
147             if(FI->getName()=="_Z10__initMaini"){
148                 continue;
149             }
150
151             if(FI->getName()=="main"){
152                 InstrumentEnterFunction("_Z10__initMaini",*FI, M);
153             }
154         }
155         return true;
156     }
157 }
```

# The Module pass

```
135     bool runOnModule(Module &M) override {
136         // Setup hooks
137         // Create a little stub function
138         setupHooks("_Z10__initMaini",M);
139
140         // Loop through all of our functions
141         // This is where you could do something interesting like only
142         // modify a subset of the functions
143         // The key is not to modify instrumenting functions!
144         Module::FunctionListType &functions = M.getFunctionList();
145         for(Module::FunctionListType::iterator FI = functions.begin(), FE = functions.end(); FI !=
FE; ++FI){
146             // Ignore our instrumented function
147             if(FI->getName()=="_Z10__initMaini"){
148                 continue;
149             }
150
151             if(FI->getName()=="main"){
152                 InstrumentEnterFunction("_Z10__initMaini",*FI, M);
153             }
154         }
155         return true;
156     }
157 }
```

1.) Notice it is using the 'mangled' c++ function name

# The Module pass

```
135 bool runOnModule(Module &M) override {
136     // Setup hooks
137     // Create a little stub function
138     setupHooks("_Z10__initMaini",M);
139
140     // Loop through all of our functions in the module
141     // This is where you could do something interesting like only
142     // modify a subset of the functions
143     // The key is not to modify instrumenting functions!
144     Module::FunctionListType &functions = M.getFunctionList();
145     for(Module::FunctionListType::iterator FI = functions.begin();
FE; ++FI){
146         // Ignore our instrumented function
147         if(FI->getName()=="_Z10__initMaini"){
148             continue;
149         }
150
151         if(FI->getName()=="main"){
152             InstrumentEnterFunction("_Z10__initMaini",*FI, M);
153         }
154     }
155     return true;
156 }
157
```

2.) This next chunk of code iterates through a Module to look at all of the functions

# The Module pass

```
135 bool runOnModule(Module &M) override {
136     // Setup hooks
137     // Create a little stub function
138     setupHooks("_Z10__initMaini",M);
139
140     // Loop through all of our functions in the module
141     // This is where you could do something interesting like only
142     // modify a subset of the functions
143     // The key is not to modify instrumenting functions!
144     Module::FunctionListType &functions = M.getFunctionList();
145     for(Module::FunctionListType::iterator FI = functions.begin(), FE = functions.end(); FI !=
FE; ++FI){
146         // Ignore our instrumented function
147         if(FI->getName()=="_Z10__initMaini"){
148             continue;
149         }
150
151         if(FI->getName()=="main"){
152             InstrumentEntryFunction("_Z10__initMaini",*FI, M);
153         }
154     }
155     return true;
156 }
157
```

3.) I am modifying code, so I return true for this pass

# setupHooks()

This code creates “a placeholder” for our source program. I do not link in my instrumentation code until the very end.

```
165     void setupHooks(StringRef InstrumentingFunctionName, Module& M){
166         auto &Context = M.getContext();
167
168         Type* voidTy = Type::getVoidTy(Context);
169         Type* intTy = Type::getInt32Ty(Context);
170         // Specify the return value, arguments, and if there are variable numbers of arguments.
171         FunctionType* funcTy = FunctionType::get(voidTy, intTy, false);
172         Function::Create(funcTy, llvm::GlobalValue::ExternalLinkage)->setName(InstrumentingFunction
173     Name);
174     }
```

# setupHooks()

This code creates “a placeholder” for our source program. I do not link in my instrumentation code until the very end.

```
165     void setupHooks(StringRef InstrumentingFunctionName, Module& M){
166         auto &Context = M.getContext();
167
168         Type* voidTy = Type::getVoidTy(Context);
169         Type* intTy = Type::getInt32Ty(Context);
170         // Specify the return value, arguments, and if there are variable numbers of arguments.
171         FunctionType* funcTy = FunctionType::get(voidTy, intTy, false);
172         Function::Create(funcTy, llvm::GlobalValue::ExternalLinkage)->setName(InstrumentingFunction
173     Name);
174     }
```

The observation from  
setupHooks() is that I  
am building up a  
'function' that returns  
void and takes in one  
argument

# setupHooks()

This code creates “a placeholder” for our source program. I do not link in my instrumentation code until the very end.

```
165 void setupHooks(StringRef InstrumentingFunctionName, Module& M){
166     auto &Context = M.getContext();
167
168     Type* voidTy = Type::getVoidTy(Context);
169     Type* intTy = Type::getInt32Ty(Context);
170     // Specify the return value, arguments, and if there are variable numbers of arguments.
171     FunctionType* funcTy = FunctionType::get(voidTy, intTy, false);
172     Function::Create(funcTy, llvm::GlobalValue::ExternalLinkage)->setName(InstrumentingFunction
173     Name);
174 }
```

The observation from setupHooks() is that I am building up a ‘function’ that returns void and takes in one argument

Which is exactly the signature of \_\_initMain

```
1 #include <stdio.h>
2
3 // This is the function that is
4 // called at the very start of the program.
5 // It will be called right after main.
6 // "dummyValue" does nothing except
7 // demonstrates how to pass arguments in our pass.
8 void __initMain(int dummyValue){
9     printf("Hello, you are running an instrumented binary.\nPerformance may vary while running an instrumented binary.\n");
10    // Do more here.
11 }
```

# InstrumentEnterFunction

- Same idea from InstrumentEnterFunction
- I am building up a specific function to insert

```
175     void InstrumentEnterFunction(StringRef InstrumentingFunctionName, Function& FunctionToInstrument
, Module& M){
176         // Create the actual function
177         // If we have a function already, then the below is very useful
178         //
179         // FunctionType* funcTy = M.getFunction(InstrumentingFunctionName)->getFunctionType();
180         //
181         // However, we are hooking into a function that we will merge later, so we instead build ou
r function type
182         // Both methods will allow us to then modify the function arguments.
183         //
```

# InstrumentEnterF

- Same idea from InstrumentExit
- I am building up a

```
175 void InstrumentEnterFunction(FunctionToInstrument& f, Module& M){
176     // Create the act
177     // If we have a f
178     //
179     // FunctionType*
180     //
181     // However, we are
    r function type
182     // Both methods w
183     //
```

Why not do something more simple?

With this approach, I can push different values as parameters based on *whatever* I need to do.

```
nction& FunctionToInstrument
me)->getFunctionType();
ater, so we instead build ou
ents.
```

# Steps to running function pass number 4!

Get our source code setup by running our pass in.

```
./../opt -load ./../../lib/LLVMHello.so -hello4 -S < hello.ll > readyToBeHooked.ll
```

Link in our instrumentation

```
./../llvm-link readyToBeHooked.ll instrumentation.ll -S -o instrumentDemo.ll
```

# LLVM Tools - llvm-link

1. clang - Clang is the frontend C/C++ compiler (llvm is the backend)
  - Likely you have heard or used Clang even if you did not know it!
2. llvm-as - Takes LLVM IR in assembly form and converts it to bitcode format.
3. llvm-dis - Converts bitcode to text readable llvm assembly
4. **llvm-link - Links two or more llvm bitcode files into one file.**
5. lli - Directly executes programs bit-code using JIT
6. llc - Static compiler that takes llvm input (assembly or bitcode) and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
  - <http://llvm.org/docs/GettingStarted.html#llvm-tools>

## LLVM Tools - What do they do?

1. clang - Clang is the LLVM C/C++ compiler (frontend)
  - Likely you have this installed already
2. llvm-as - Takes LLVM IR and converts it to bitcode format.
3. llvm-dis - Converts bitcode to LLVM IR
4. llvm-link - Links two or more object files into an executable.
5. lli - Directly executes LLVM IR
6. llc - Static compiler that takes LLVM IR and generates assembly code
7. opt - LLVM analyzer and optimizer which runs certain optimizations and analysis on files
8. More
  - <http://llvm.org/docs/GettingStarted.html#llvm-tools>

Now that our files are merged, there is a declaration and a definition for our instrumentation!

# LLVM-Link

- Think of this like a ‘linker’ for IR code.
- Sometimes it is useful to link all of your code together, and then run your optimizations
  - We call this “whole program optimization”

```
./../llvm-link readyToBeHooked.ll instrumentation.ll -S -o instrumentDemo.ll
```

# Grand Finale!

Run our linked .ll file (using lli or compile to source)

```
mike:examples$ ../../lli instrumentDemo.ll  
Hello, you are running an instrumented binary.  
Performance may vary while running an instrumented binary.  
Bonjour!
```

# Grand Finale!

Run our linked .ll file (using lli or compile to source)

```
mike:examples$ ../../lli instrumentDemo.ll  
Hello, you are running an instrumented binary.  
Performance may vary while running an instrumented binary.  
Bonjour!
```



It works, we  
see our  
message  
before the  
“Bonjour” from  
hello.cpp!!

# Going Further (Challenges/Project Ideas)

Time permitting:

- Easy
  - Print out function arguments
  - Recover and print metadata and/or Profile Guided Optimization Data with functions
  - Write a python script that 'llvm-links' all of your .ll files together.
- Medium
  - Build both a control flow graph and call graph and output to .dot
  - Find Program attributes
    - Add an attribute for any function  $< 10$  instructions, and force it to inline
- Hard/Interesting?
  - Autovectorizing (Find patterns and Insert SIMD instructions)
  - Investigate the “sanitizer” projects. See if you can add interesting printouts.

# Resources

# Resources

- Online Resources

- The Documentation: <http://llvm.org/docs/>
- Developer Meetings: <http://llvm.org/devmtg/>
- Downloading and setting up LLVM: <http://llvm.org/docs/GettingStarted.html#checkout>
- An introductory guide: <http://adriansampson.net/blog/llvm.html>
- Weekly LLVM Newsletter: <http://llvmweekly.org/>
  - Developers Mailing List: <http://lists.llvm.org/mailman/listinfo/llvm-dev>
- IR Web interface: <http://ellcc.org/demo/index.cgi>
- LLVM Blog: <http://blog.llvm.org/>

- Useful Tools to Try

- Hexdump (hexdump -c some\_bitcode.bc)
- Meld - Tool for diff'ing and comparing files
- xdot or graphviz - View .dot files

- Other homework

- <https://cseweb.ucsd.edu/classes/sp14/cse231-a/proj1.html>

# More Guidance - Your LLVM Syllabus

- Feb, 5 -- Day 1 (or Today?):  
<https://www.youtube.com/watch?v=a5-WaD8VV38>
- Feb, 6 -- Day 2: [Official LLVM Youtube channel](#)
- Extend Program Analysis Knowledge:
  - Youtube series on Program Analysis (Some LLVM Lectures!)
    - <https://www.youtube.com/playlist?list=PLNC6lmslySCOPjY8lwKBtD2cqe-MMgIGM>

# Contributing to LLVM

<https://llvm.org/devmtg/2014-02/slides/ledru-how-to-contribute-to-llvm.pdf>

# How to contribute to LLVM, Clang, etc

# Conclusion

- LLVM is an exciting project with a lot of power
- LLVM or its related projects are likely the ‘right’ tool if you are working on programming languages, performance, or tool building
- If you are still not convinced, your takeaway can still be to look at the codebase, and see some great engineering with the C++ language.
- It’s big, but should not be scary
  - The difficulty that arises is that it is a lot of ‘new’ things
  - You can do it!

Thank You! **FOSDEM'18**

[@MichaelShah](#) | [www.mshah.io](http://www.mshah.io)

Feedback Form <https://tinyurl.com/fosdem18llvmintro>  
(Whether you watched this talk now or in the future!)

# Make sure we save output of opt

- Something new we are doing with this pass, is that it actually is modifying code.
- Occasionally you may see this message

```
mike:examples$ ../../opt -load ../../lib/LLVMHello.so -hello4 < instrumentDemoText.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the '-f' option.
```

- In our case, yes we do want to output the modified bitcode file, but this time to a new bitcode file.

# Some Gotcha's

- Having trouble with llvm-config?
  - Make sure your PATH variable is updated
  - `export PATH=/home/mike/Desktop/llvm/llvm_build/bin:$PATH`

## Courses Using LLVM

<https://www.cs.utexas.edu/users/lin/cs380c/prog1.pdf>

## Tour of LLVM Project

<https://blog.regehr.org/archives/1453> |  
<http://www.linux.org/threads/llvm-toolset.6644/>

# Useful debugging things

`dump()` command.

# Build your own LLVM language

<http://dev.stephendiehl.com/numpile/>

# LLVM Backend information

<https://jonathan2251.github.io/lbd/funccall.html>