LLVM Tutorial

Seonyeong Heo

2020. 11. 17

Course Information

Introduction to LLVM IR

3 Chapters
8 Practices

Based on LLVM 11.0.0

Course Outline

- Basic Concept of LLVM IR
  - Goal: Learn the basic concept of LLVM IR
  - 2 Practices
  - Estimated Time: ~ 60 min
- IR Optimization - Analysis
  - Goal: Learn how to analyze LLVM IR
  - 4 Practices
  - Estimated Time: ~ 120 min
- IR Optimization – Transformation
  - Goal: Learn how to transform LLVM IR
  - 2 Practices
  - Estimated Time: ~ 60 min

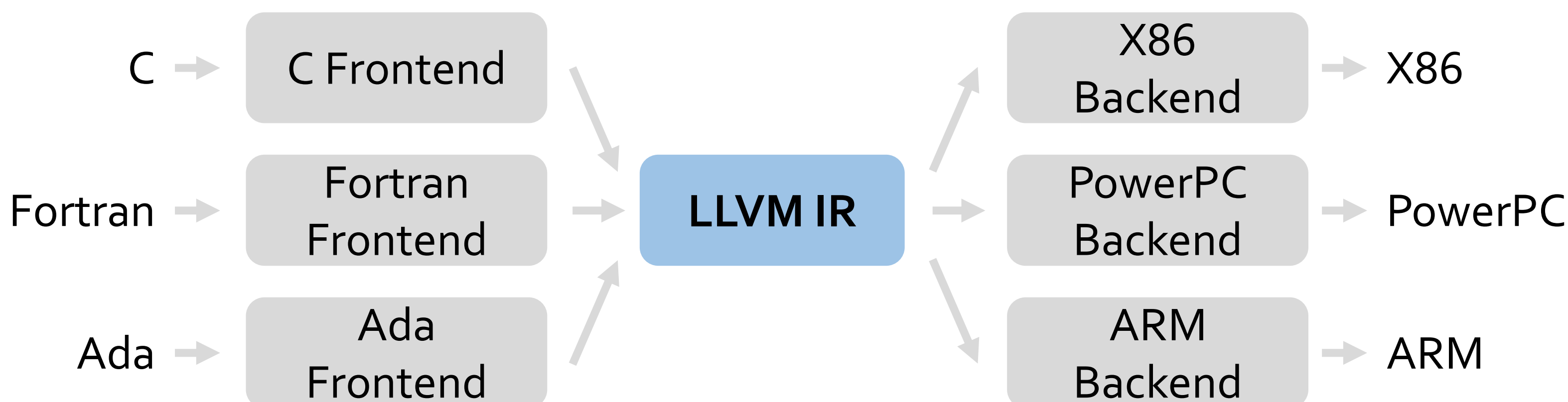
List of Practices

- Basic Concept of LLVM IR
  - Practice 1: First Compilation
  - Practice 2: Control Flow Graph
- IR Optimization – Analysis
  - Practice 3: First LLVM Pass
  - Practice 4: (Static) InstCount Pass
  - Practice 5: CallInstCount Pass
  - Practice 6: Loop Analysis Pass
- IR Optimization – Transformation
  - Practice 7: Insert inc Function
  - Practice 8: Dynamic CallCount

Basic Concept of LLVM IR

LLVM: Concept

- **L**ow **L**evel **V**irtual **M**achine
- Compiler Infrastructure
  - Source- and target-independent code generation



LLVM: Project Structure

- Primary Sub-projects in LLVM

Core Library
- Code optimization
- Code generation

Clang
- Native C/C++ Compiler
- clang ↔ **gcc**

LLDB
- Native Debugger
- lldb ↔ **gdb**

MLIR
- Multi-Level IR
- ***New from LLVM 10.0.0!***

LLVM: Project Structure

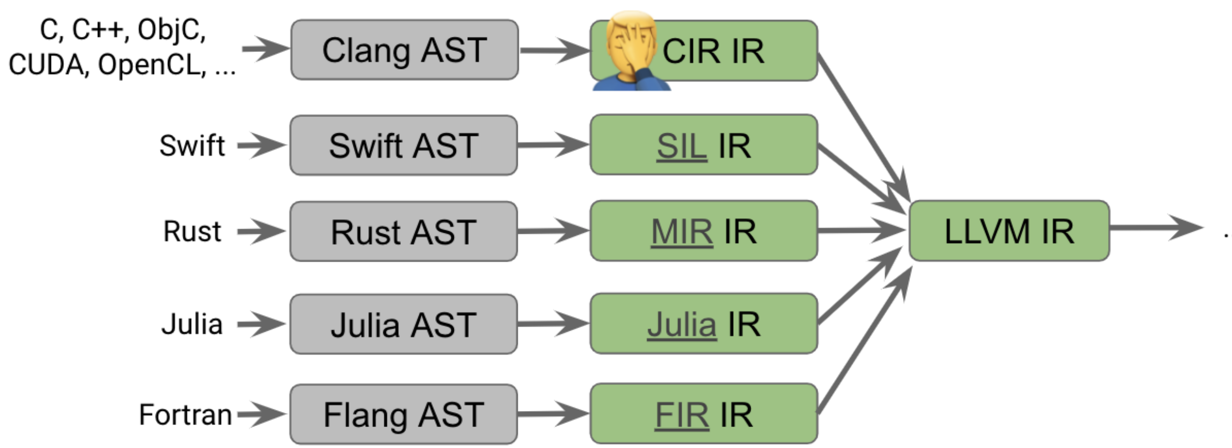
- Primary Sub-projects in LLVM

this tutorial

- **Core Library**
  - Code optimization
  - Code generation
- **Clang**
  - Native C/C++ Compiler
  - clang ↔ **gcc**
- **LLDB**
  - Native Debugger
  - lldb ↔ **gdb**
- **MLIR**
  - Multi-Level IR
  - ***New from LLVM 10.0.0!***

MLIR

- **M**ulti-**L**evel **IR**
- Modern languages need their own IRs



- Facilitate to create a new IR ("dialect")
- Now part of LLVM infrastructure

* Chris Lattner and Tatiana Shpeisman, MLIR: Multi-Level Intermediate Representation Compiler Infrastructure, CGO 2020 Keynote

LLVM IR: Concept

- IR = Intermediate Representation

High-level ⟷ Low-level

C language

```
int main()
{
  int a = 2;
  a = a + 1;
  return a;
}
```

LLVM IR

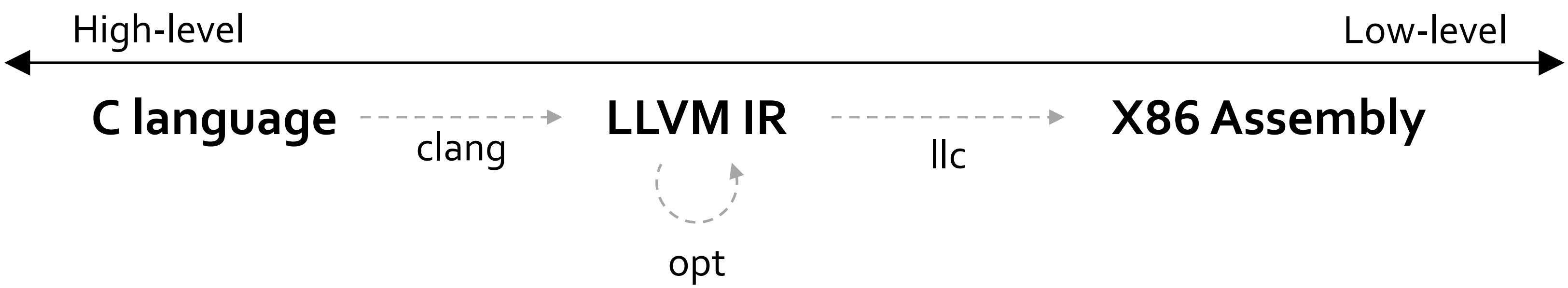
```
define i32 @main() {
entry:
  %retval = alloca i32
  %a = alloca i32
  store i32 0, i32* %retval
  store i32 2, i32* %a
  %0 = load i32, i32* %a
  %add = add nsw i32 %0, 1
  store i32 %add, i32* %a
  %1 = load i32, i32* %a
  ret i32 %1
}
```

X86 Assembly

```
main:
  .cfi_startproc
  pushq    %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset %rbp, -16
  movq     %rsp, %rbp
  .cfi_def_cfa_register %rbp
  movl     $0, -8(%rbp)
  movl     $2, -4(%rbp)
  movl     -4(%rbp), %eax
  addl     $1, %eax
  movl     %eax, -4(%rbp)
  movl     -4(%rbp), %eax
  popq     %rbp
  .cfi_def_cfa %rsp, 8
  retq
```

LLVM IR: Concept

- IR = Intermediate Representation



- Compilation Tools
  - clang: C program → LLVM IR
  - opt (or clang): LLVM IR → (optimized) LLVM IR
  - llc (or clang): LLVM IR → Machine code

LLVM IR: Concept

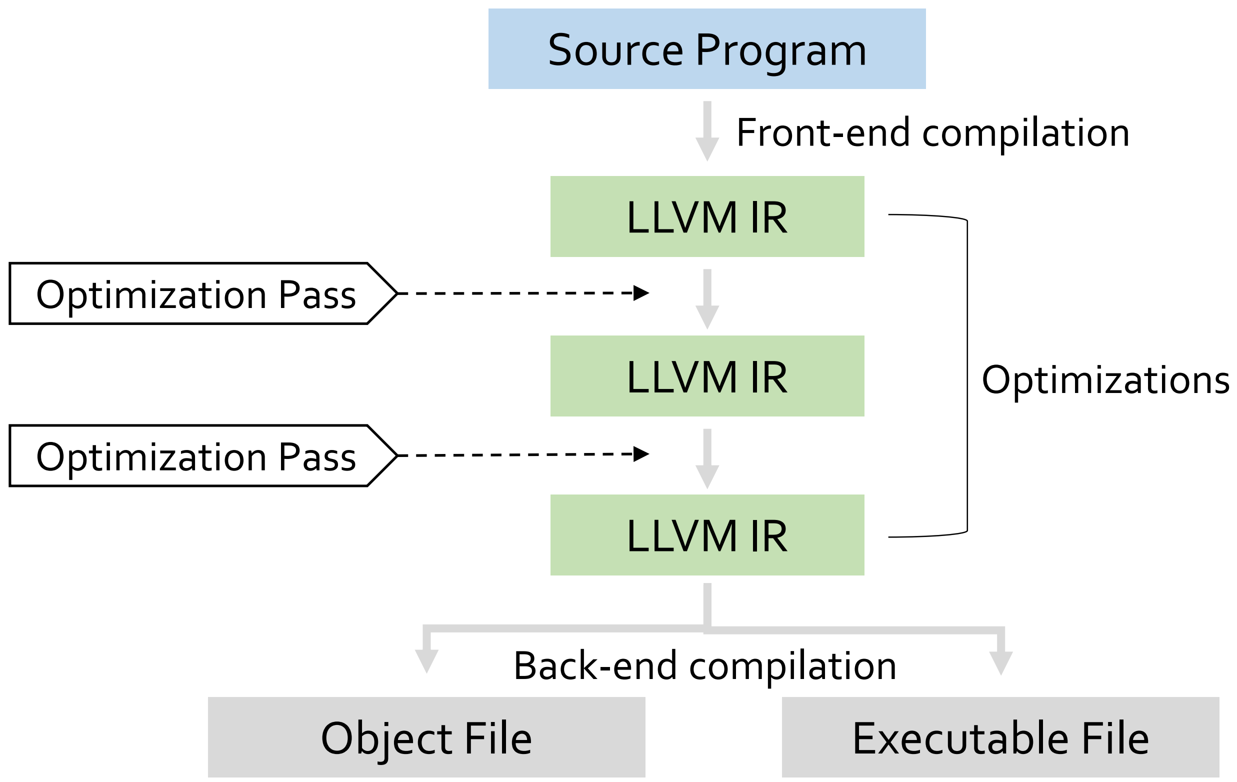
- File Extensions
  - *.bc: Bitcode (Not human-readable)
  - *.ll: Human-readable IR

- Convert bitcode to human-readable IR
  - Use **llvm-dis** = LLVM Disassembler
  - Example

```
$ llvm-dis source.bc
```

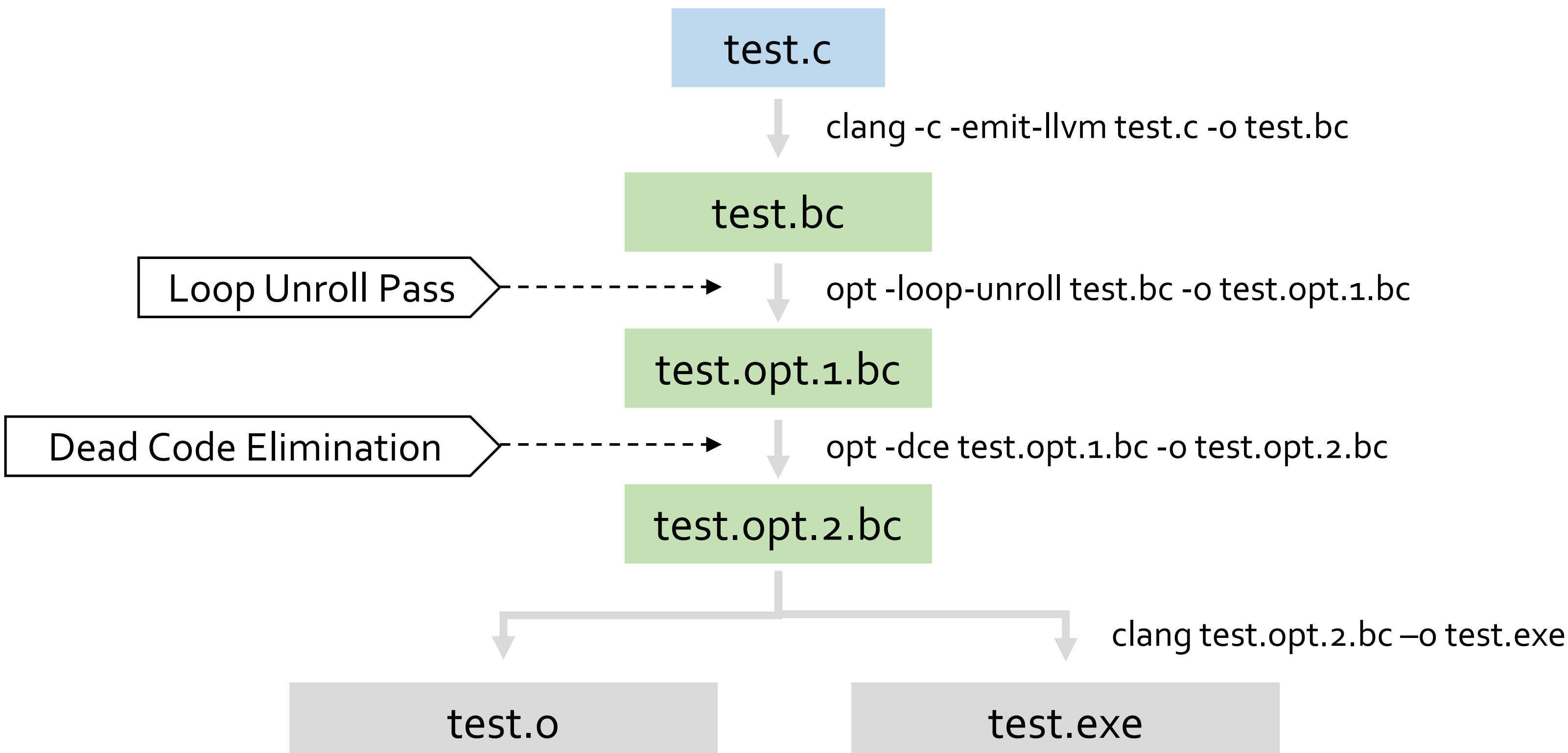
Compilation Process

- Diagram (General)



Compilation Process

- Diagram (Example)



Compilation Process

- Command Line (Example)
  - Front-end compilation

```
$ clang -c -emit-llvm test.c -o test.bc
```

  - Optimizations

```
$ opt -loop-unroll test.bc -o test.opt.1.bc
$ opt -dce test.opt.1.bc -o test.opt.2.bc
```

  - Back-end compilation

```
$ clang test.opt.2.bc -o test.exe
```

Built-in Optimizations

- LLVM provides various built-in passes
  - Try "**opt -help**" to see all the built-in passes

```
Optimizations available:
  -aa                                                   - Function Alias Analysis Results
  -aa-eval                                              - Exhaustive Alias Analysis Precision Evaluator
  -aarch64-a57-fp-load-balancing                        - AArch64 A57 FP Load-Balancing
  -aarch64-ccmp                                         - AArch64 CCMP Pass
  -aarch64-collect-loh                                  - AArch64 Collect Linker Optimization Hint (LOH)
  -aarch64-condopt                                      - AArch64 CondOpt Pass
  -aarch64-copyelim                                     - AArch64 redundant copy elimination pass
  -aarch64-dead-defs                                    - AArch64 Dead register definitions
  -aarch64-expand-pseudo                                - AArch64 pseudo instruction expansion pass
```

- LLVM applies different sets of passes according to the optimization level
  - Available optimization levels: O0, O1, O2, O3

Practice 1: First Compilation

- Goal
  - Learn how to generate and optimize LLVM IR

- Steps

1) Write a simple C program (test.c)
2) Generate test.bc from test.c
3) Optimize test.bc with any optimization pass (test.opt.bc)
4) Generate test.ll and test.opt.ll from test.bc and test.opt.bc
5) Generate test.exe from test.opt.bc

- Further Activity
  - Compare the compilation results with -O0 and -O3

LLVM IR: Example

- Example Code

C

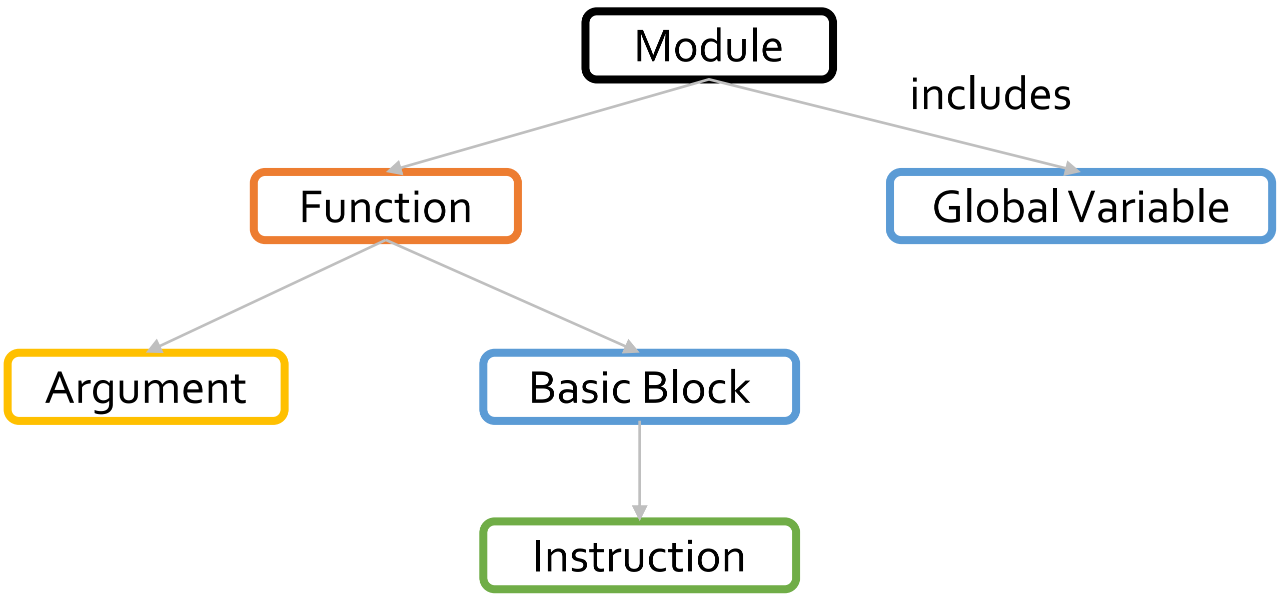
```c
int add (int a, int b) {
  return a+b;
}
```

LLVM IR

```llvm
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}
```

LLVM IR: Components

- Structure of a program
  - Has-a relationship of program components



LLVM IR: Components

- Example Code

C

```c
int add (int a, int b) {
  return a+b;
}
```

Function

LLVM IR

```llvm
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}
```

Basic Block

Argument

Instruction

LLVM IR: Features

- Strongly typed: No implicit type casting

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
}
```

LLVM IR: Features

- Single Static Assignment (SSA): No redefinition of value

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret
}
```

Not $0 nor $1

Single Static Assignment (SSA)

- Every value must be defined **only once**
  - In other words, every value has a **single** definition

```
%x = 1 + 2
%x = %x + 3
```

```
%x = 1 + 2
%y = %x + 3
```

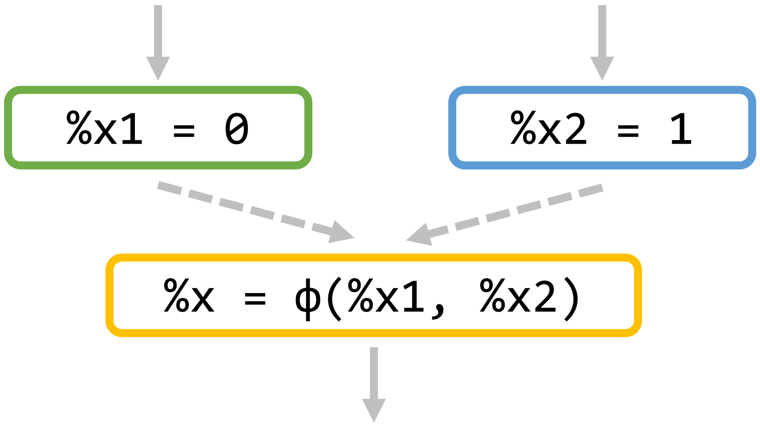
- Facilitate program analyses
  - Liveness Analysis: From **DEF** to last **USE**
  - Constant Propagation: If **DEF** is constant, then **USE** is also constant

Single Static Assignment (SSA)

- Phi(Φ) Node
  - Choose a variable according to the control flow
  - Require "remembering" previous basic block

```
if (k == 0)
  x = 0;
else
  x = 1;

printf(...,x);
```



LLVM IR: Type System

- Void type (void)

- First Class Types
  - Single Value Types
    - Integer Type
      - i***N***: ***N***-bit integer type
      - ex) i1, i8, i16, i32, i64, …

    - Floating-point Types
      - half (16-bit), float (32-bit), double (64-bit), fp128 (128-bit)
      - x86_fp80 (80-bit), ppc_fp128 (128-bit)

LLVM IR: Type System

- First Class Types
  - Pointer Type
    - Format: <type> *
    - ex) [4 x i32] *
  - Vector Type
    - Format: < <# of elements> x <element type> >
    - ex) <4 x i32>, <8 x float>

LLVM IR: Type System

- Aggregate Types
  - Array Type
    - Format: [<# of elements> x <element type>]
    - ex) [40 x i32], [3 x [4 x i32]]

  - Structure Type
    - Formats
      - *Normal* struct type: $T1 = type { <type list> }
      - *Packed* struct type: $T2 = type <{ <type list>}>
    - ex) { i32, i32, i32 }, <{ i8, i32 }>

LLVM IR: Type System

- Function Type
  - Format: <return type> (<parameter list>)
    - ex)
      - i32 (i32)
      - float (i16, i32*) *
      - i32 (i8*, ...)

Indicate variable argument
(e.g. printf)

LLVM IR: Instructions

- Binary Operations
  - add, fadd, sub, fsub…
- Memory Access and Addressing Operations
  - **alloca**: Allocate memory on the stack frame
    - ex) %ptr = alloca i32
  - load, store
  - **getelementptr:** Get the address of a subelement of an aggregate data structure
    - Pointer dereference
    - Structure member access
    - ex) %iptr = getelementptr [10 x i32], [10 x i32]* @arr, i16 0, i16 0

LLVM IR: Instructions

- Terminator Instructions
  - ret**:** Return control flow from a function
  - br: Transfer control flow to a different basic block
  - **invoke**: Transfer control flow to a function (exception handling)
- Other operations
  - icmp: Compare two integers
  - **phi**: Implement Φ node
  - call: Call a function
- Full reference at https://llvm.org/docs/LangRef.html

LLVM IR: Module

- Full LLVM IR Code
  - Includes target information, global variables, metadata, etc.

C Program

```c
int acc = 10;

int add (int a, int b) {
  return a + b + acc;
}
```

LLVM IR Code

```llvm
; ModuleID = 'test.bc'
source_filename = "test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@acc = dso_local global i32 10, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  %2 = load i32, i32* @acc, align 4
  %add1 = add nsw i32 %add, %2
  ret i32 %add1
}

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

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 8.0.0 (git@git.corelab.or.kr:corelab/clang.git 7973f6c2602b1e37f00a710ffa0c798a3f321e58) (git@git.corelab.or.kr:corelab/llvm.git c55bcb2f96806a3d9e5718497cede4665b27c8a4)"}
```

LLVM IR: Module

- Full LLVM IR Code (1/4)

Data Layout Description: Endianness, Alignment

```
; ModuleID = 'test.bc'
source_filename = "test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@acc = dso_local global i32 10, align 4

```

Global Variables: Start with @

Target Machine Description: <arch>-<vendor>-<os>-<env/abi>

LLVM IR: Module

- Full LLVM IR Code (2/4)

Attribute Group

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  st            i32* %b.addr, align 4
  %              i32* %a.addr, align 4
  %              i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  %2 = load i32, i32* @acc, align 4
  %add1 = add nsw i32 %add, %2
  ret i32 %add1
}

```

Local Value:
Start with %

LLVM IR: Module

- Full LLVM IR Code (3/4)

Attribute Group

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

LLVM IR: Module

- Full LLVM IR Code (4/4)

Named Metadata

```
!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 8.0.0 (git@git.corelab.or.kr:corelab
/clang.git 7973f6c2602b1e37f00a710ffa0c798a3f321e58) (git@g
it.corelab.or.kr:corelab/llvm.git c55bcb2f96806a3d9e5718497
cede4665b27c8a4)"}
```

(Unnamed) Metadata

How to Visualize LLVM IR Code

- LLVM provides several built-in printer passes
  - Control Flow Graph (CFG)
  - Call Graph
  - Dominance Tree

- Use the built-in printer pass to visualize LLVM IR Code
  - opt -dot-cfg
  - opt -dot-callgraph
  - opt -dot-dom

```
$ opt -dot-cfg test.bc -o test.bc
Writing 'cfg.add.dot'...
Writing 'cfg.main.dot’...
$ dot -Tpdf cfg.add.dot -o cfg.add.pdf
```

Practice 2: Control Flow Graph

- Goal
  - Learn how to obtain the control flow graph of a given IR code

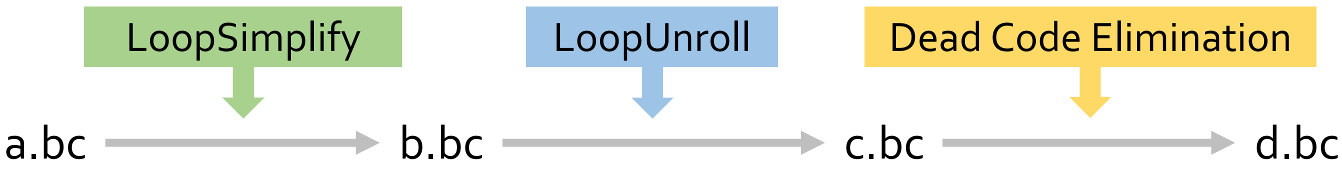
- Steps
  1) Write a simple matrix multiplication function (mm.c)
  2) Generate mm.bc from mm.c
  3) Apply the CFG printer pass to mm.bc
  4) Convert the dot file to a pdf file
  5) Check the pdf file

- Further Activity
  - Can you recognize a loop in the control flow graph?

IR Optimization
- Analysis

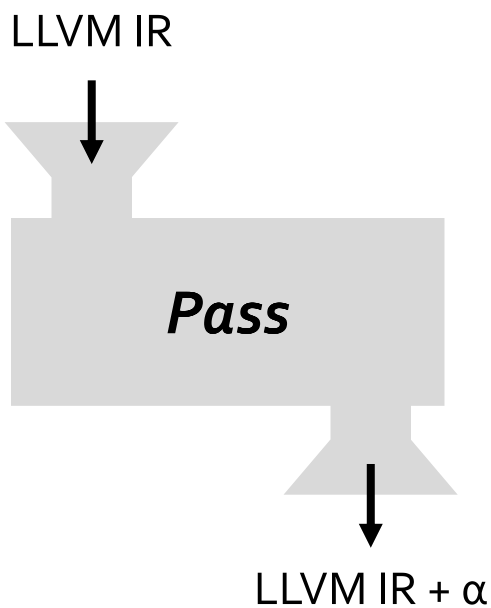
IR Optimization

- LLVM enables *modular optimizations* through the LLVM pass framework
  - **Pass**: Unit of optimization
- Each **LLVM pass** performs optimizations and transformations on LLVM IR
  - Example



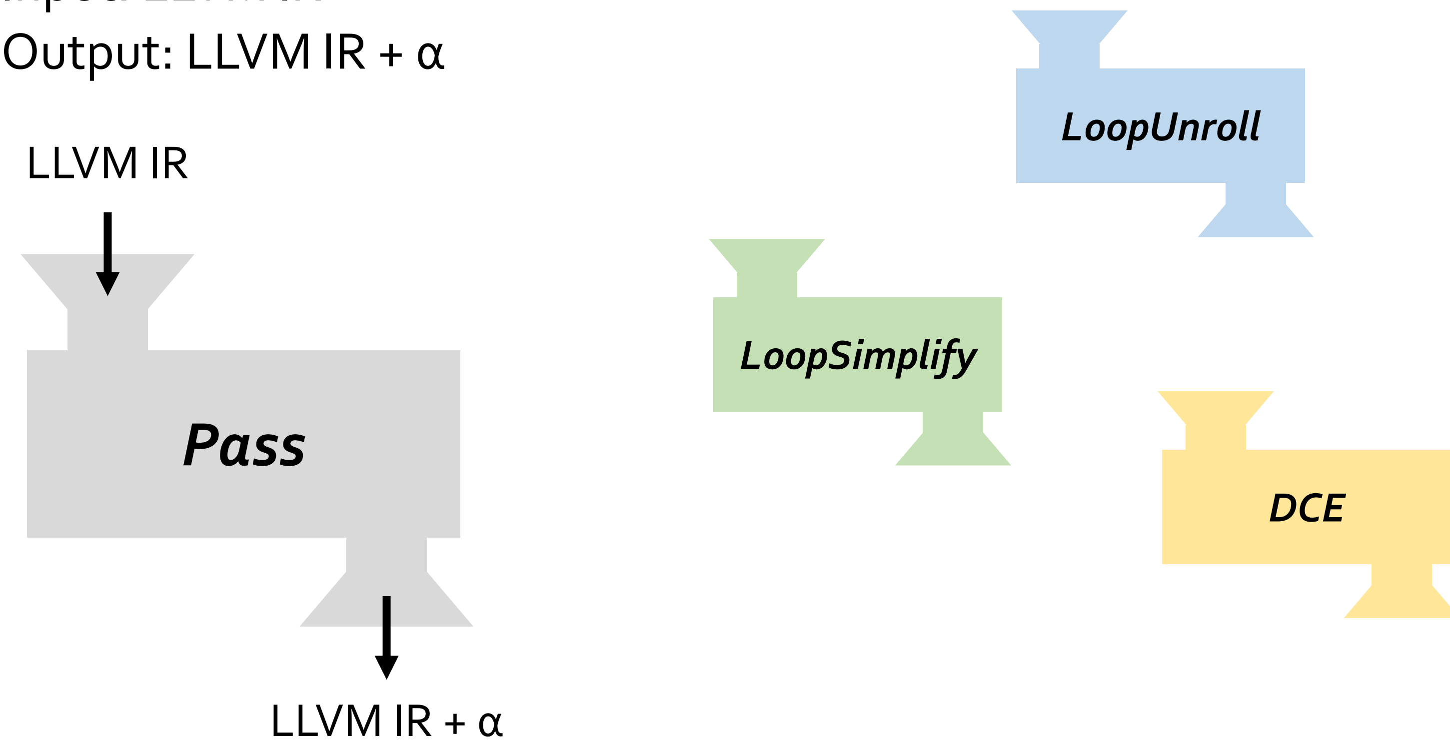
LLVM Pass: Concept

- Act like a filtering function
  - Input: LLVM IR
  - Output: LLVM IR + α



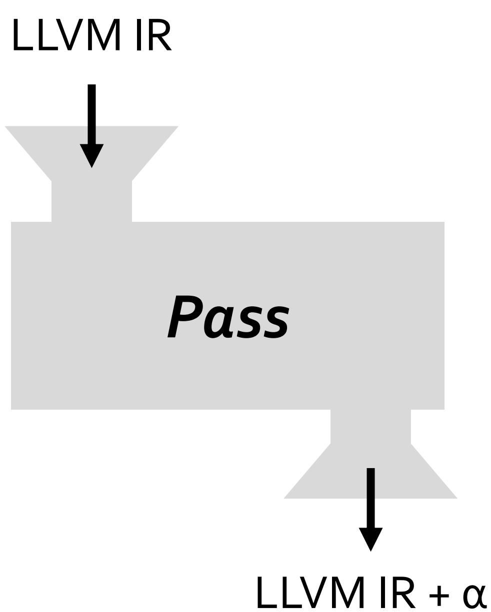
LLVM Pass: Concept

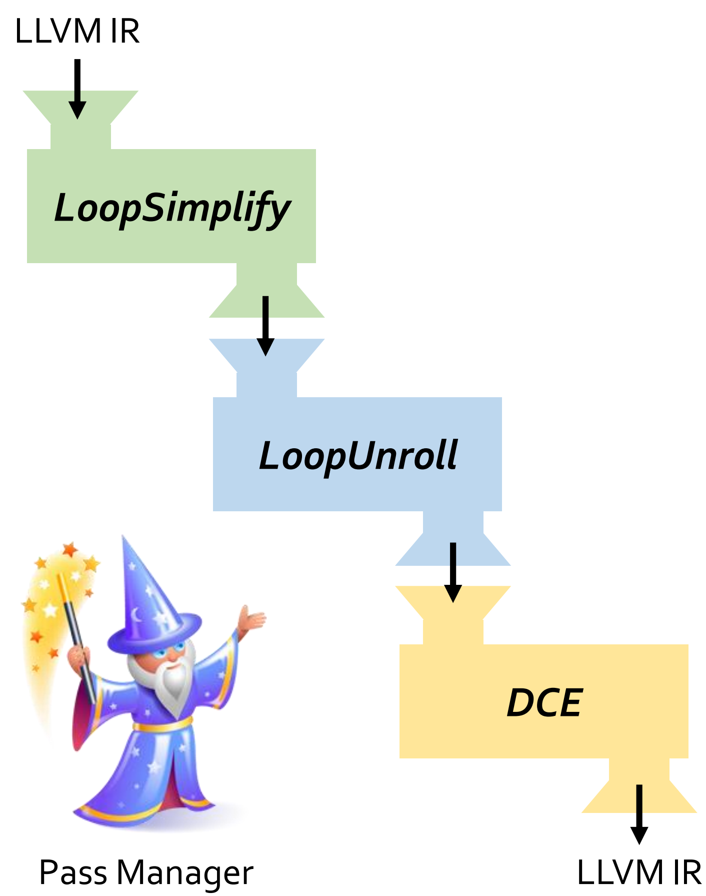
- Act like a filtering function
  - Input: LLVM IR
  - Output: LLVM IR + α



LLVM Pass: Concept

- Act like a filtering function
  - Input: LLVM IR
  - Output: LLVM IR + α





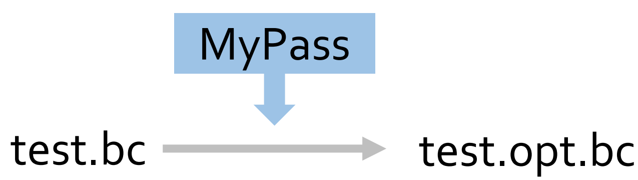
LLVM Pass: Implementation

- C++ class that inherits the "`Pass`" class in LLVM
  - Implement functionality by <u>overriding virtual methods</u>

    e.g. `runOnModule` or `runOnFunction`

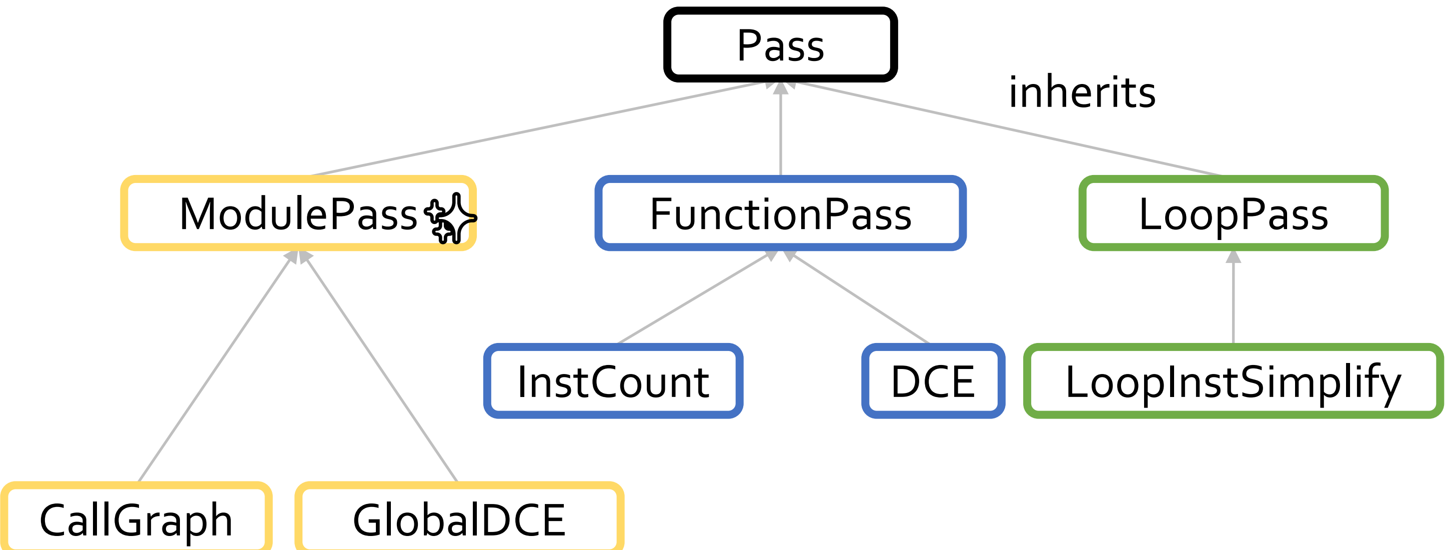
- Dynamically loaded at run-time
  - `opt -load PASS_LIBRARY_PATH -PASS_NAME`
  - example

```
$ opt -load ~/lib/MyPass.so -MyPass test.bc -o test.opt.bc
```



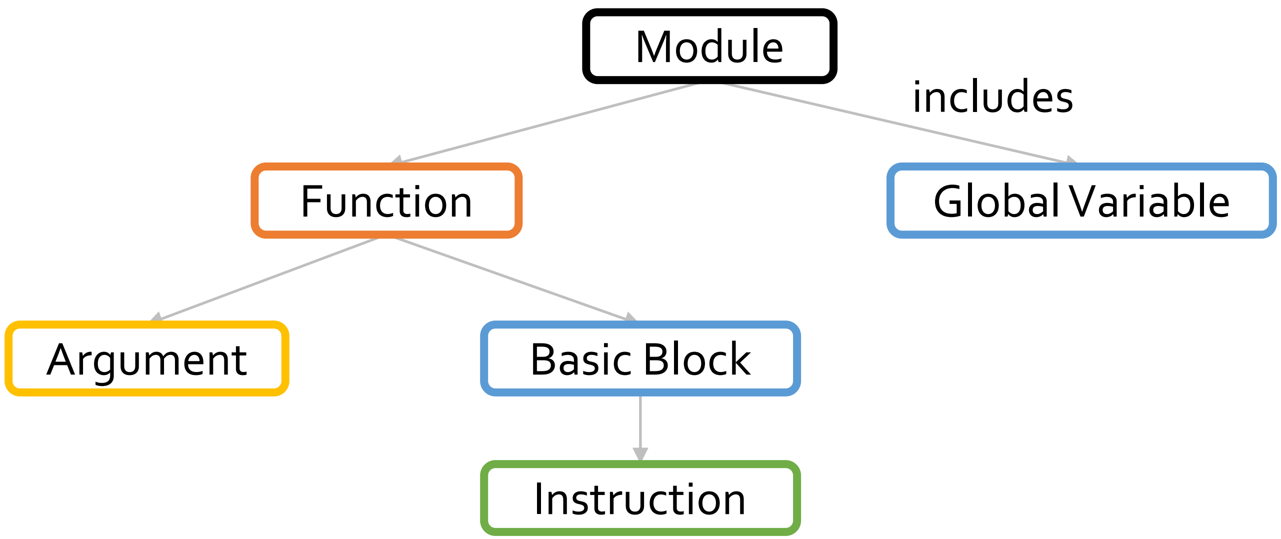
LLVM Pass: Implementation

- Is-A relationship of LLVM Pass classes
  - `xxxPass`: `xxx` is the unit of optimization



LLVM Pass: Implementation

- Program Components = C++ Objects
  - class Module, class Function, class GlobalVariable, …



LLVM Pass: Implementation

- Manipulate program components as C++ objects
  - class Module, class Function, class GlobalVariable, …

LLVM IR Code

LLVM Pass

```
; ModuleID = 'test.bc'
source_filename = "test.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@acc = dso_local global i32 10, align 4

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  %2 = load i32, i32* @acc, align 4
  %add1 = add nsw i32 %add, %2
  ret i32 %add1
}
```

GlobalVariable object

BasicBlock object

Function object

Module object

Skeleton Code: ModulePass

- Header File (`HelloModule.h`)

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

using namespace llvm;

namespace {
  struct HelloModule : public ModulePass {
    static char ID; // Pass identification, replacement for typeid
    HelloModule() : ModulePass(ID) {}

    bool runOnModule(Module &M) override;

    void getAnalysisUsage(AnalysisUsage &AU) const override;
  };
}
```

Skeleton Code: ModulePass

- Source File (`HelloModule.cpp`)

```
#include "HelloModule.h"

#define DEBUG_TYPE "hello"

bool HelloModule::runOnModule(Module &M) {
  return false;
}

void HelloModule::getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesAll();
}

char HelloModule::ID = 0;
static RegisterPass<HelloModule> X("helloModule", "Hello World Pass ");
```

source program

Do something to analyze or optimize code

Pass name

Skeleton Code: FunctionPass

- Header File (`HelloFunction.h`)

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

using namespace llvm;

namespace {
  struct HelloFunction : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    HelloFunction() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override;

    void getAnalysisUsage(AnalysisUsage &AU) const override;
  };
}
```

Skeleton Code: FunctionPass

- Header File (`HelloFunction.cpp`)

```
#include "HelloFunction.h"

#define DEBUG_TYPE "hello"

bool HelloFunction::runOnFunction(Function &F) {
  return false;
}

void HelloFunction::getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesAll();
}

char HelloFunction::ID = 0;
static RegisterPass<HelloFunction> Y("helloFunction", "Hello World Pass ");
```

function in source program

Do something to analyze or optimize code

How to Run LLVM Pass

Automatically generate compile options

1) Compile LLVM Passes

```
$ clang++ -c -fpic -fno-rtti `llvm-config --cppflags`
HelloModule.cpp -o HelloModule.o
```

2) Make a shared library with the LLVM passes

```
$ clang++ -shared HelloModule.o HelloFunction.o -o Hello.so
```

3) Run the LLVM Passes using opt

```
$ opt –load Hello.so -helloModule test.bc –o test.opt.bc
```

Practice 3: First LLVM Pass

- Goal
  - Learn how to write, compile and run passes

- Steps
  1) Implement a NamePrinter pass that inherits FunctionPass
  2) Print the names of functions in a module
     - Tip 1: To print a debug message, use
       - #include “llvm/Support/Debug.h”
       - dbgs() << “Message”;
     - Tip 2: To get a function name, use
       - F.getName()
  3) Compile and test the pass

IR Code Analysis

- Use the member functions of IR Classes!

```
bool HelloModule::runOnModule(Module &M) {
  return false;
}
```

- References
  - Doxygen
    - http://llvm.org/doxygen/
  - Existing LLVM Passes
    - Find the function in llvm/lib/Analysis or llvm/lib/Transforms

IR Code Analysis

- class `Module`
  - https://llvm.org/doxygen/classllvm_1_1Module.html

llvm::Module Class Reference

A **Module** instance is used to store all the information related to an LLVM module. More...

```
#include "llvm/IR/Module.h"
```

Function *	**getFunction** (**StringRef Name**) **const** Look up the specified function in the module symbol table. More...
GlobalVariable *	**getGlobalVariable** (**StringRef Name**) **const** Look up the specified global variable in the module symbol table. More...
const DataLayout &	**getDataLayout** () **const** Get the data layout for the module's target platform. More...

IR Code Analysis

- class `Function`
  - http://llvm.org/doxygen/classllvm_1_1Function.html

llvm::Function Class Reference

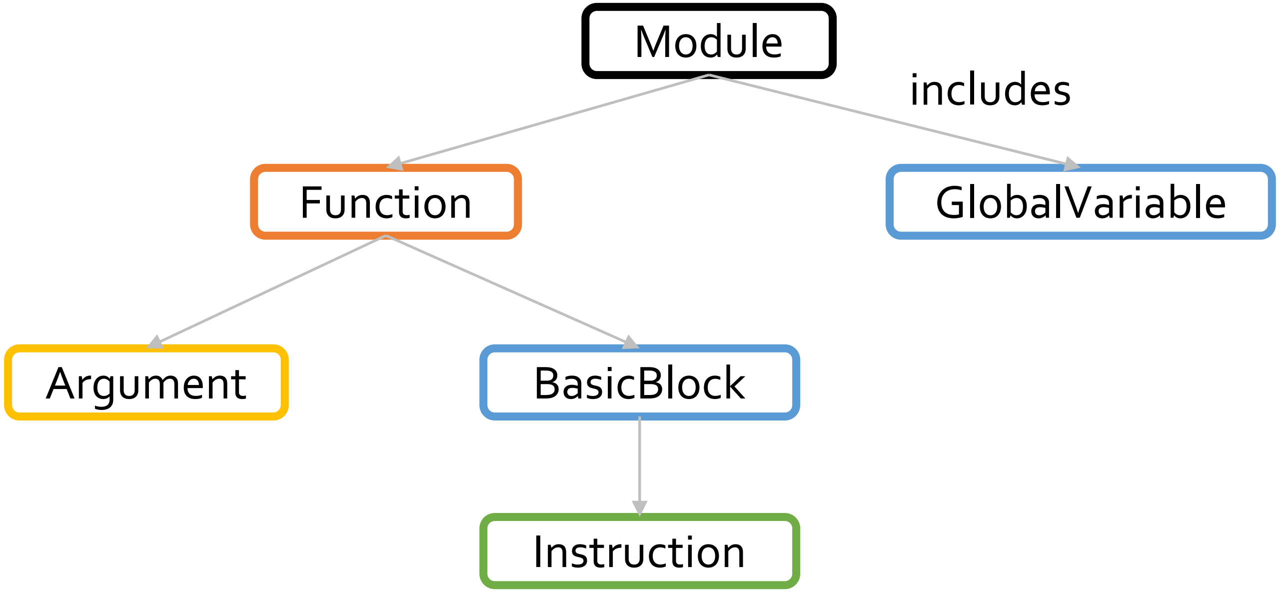
```
#include "llvm/IR/Function.h"
```

FunctionType *	getFunctionType () const Returns the FunctionType for me. More...
Type *	getReturnType () const Returns the type of the ret val. More...

BasicBlock &	getEntryBlock ()

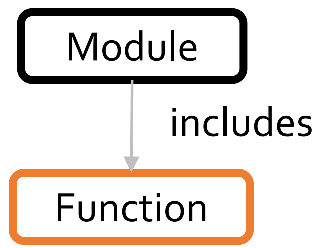
LLVM IR Classes

- Has-a relationship of LLVM IR classes



Useful Basic Iterators

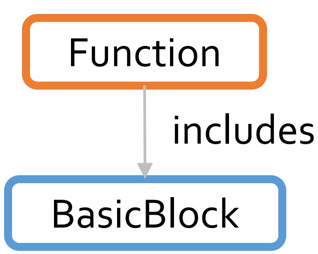
- class `Module`



using	iterator = FunctionListType::iterator The **Function** iterators. More...
using	const_iterator = FunctionListType::const_iterator The **Function** constant iterator. More...

iterator	begin ()
const_iterator	begin () const
iterator	end ()
const_iterator	end () const

- class `Function`

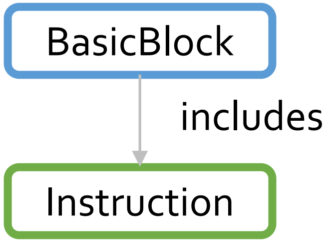


using	iterator = BasicBlockListType::iterator
using	const_iterator = BasicBlockListType::const_iterator

Useful Basic Iterators

- class `BasicBlock`

using	iterator = InstListType::iterator Instruction iterators... More...
using	const_iterator = InstListType::const_iterator



Useful Basic Iterators

- Example 1
  - Iterate through functions in the module
    1) For-each statement

```
for(Function &F : M) {
  // Do something with F
}
```

    2) Using iterators

```
Module::iterator Begin = M.begin();
Module::iterator End = M.end();
for (Module::iterator it = Begin; it != End; ++it) {
  Function &F = *it;
  // Do something with F
}
```

Useful Basic Iterators

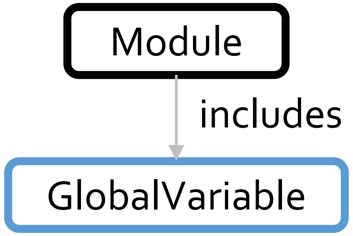
- Example 2
  - Iterate through instructions in the module

```
for(Function &F : M) {
  for(BasicBlock &BB : F) {
    for(Instruction &I : BB) {
      // Do something with I
    }
  }
}
```

Other Iterators

- class `Module`

using	global_iterator = GlobalListType::iterator The Global Variable iterator. More...
using	const_global_iterator = GlobalListType::const_iterator The Global Variable constant iterator. More...



Global Variable Iteration

global_iterator	global_begin ()
const_global_iterator	global_begin () const
global_iterator	global_end ()
const_global_iterator	global_end () const
bool	global_empty () const
iterator_range< global_iterator >	globals ()
iterator_range< const_global_iterator >	globals () const

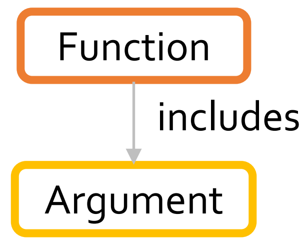
Other Iterators

- class `Function`

using	arg_iterator = Argument *
using	const_arg_iterator = const Argument *

Function Argument Iteration

arg_iterator	arg_begin ()
const_arg_iterator	arg_begin () const
arg_iterator	arg_end ()
const_arg_iterator	arg_end () const
iterator_range< arg_iterator >	args ()
iterator_range< const_arg_iterator >	args () const



Other Iterators

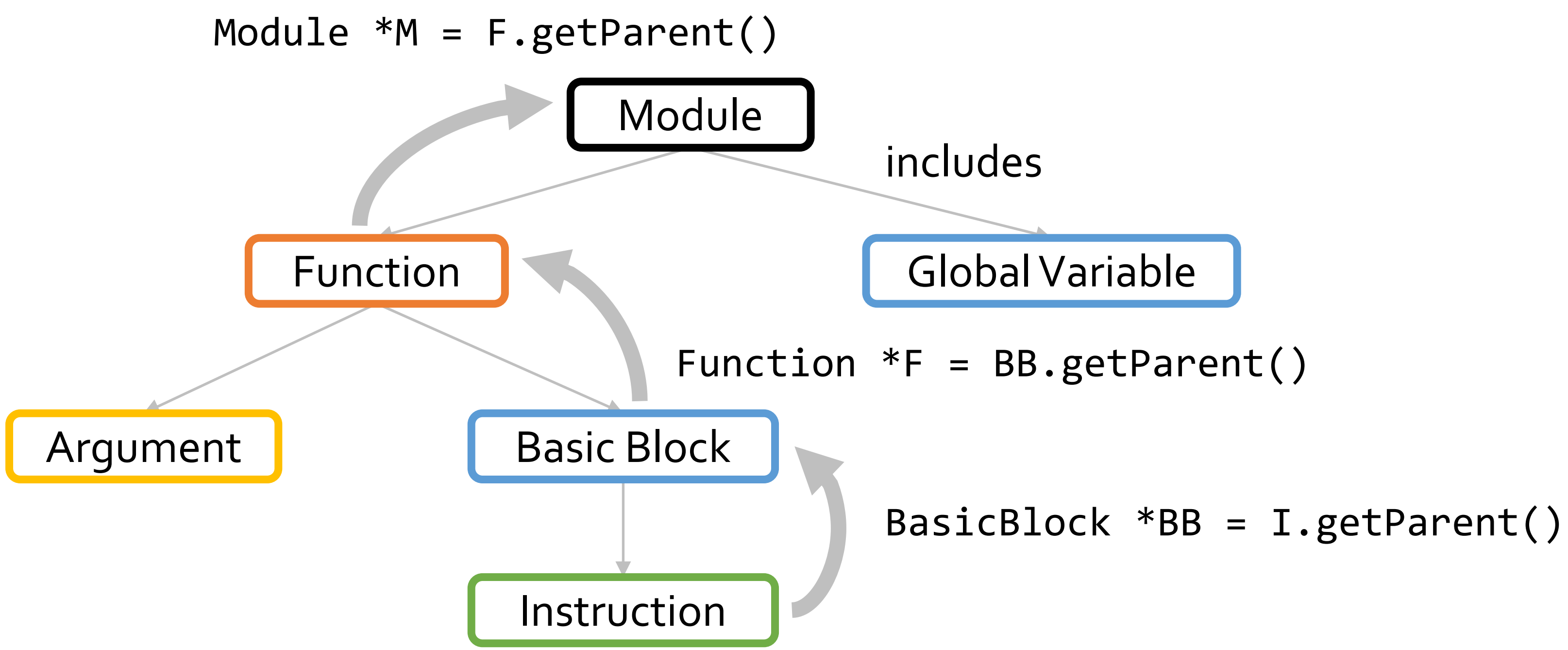
- Example 3
  - Iterate through the arguments of a function

```
for(Argument *Arg : F.args()) {
  // Do something with Arg
}
```

```
Function::arg_iterator Begin = F.arg_begin();
Function::arg_iterator End = F.arg_end();
for (Function::arg_iterator it = Begin; it != End; ++it) {
  Argument *Arg = *it;
  // Do something with Arg
}
```

Get Parent Instance

- Has-A relationship of LLVM IR classes



Practice 4: (Static) InstCount

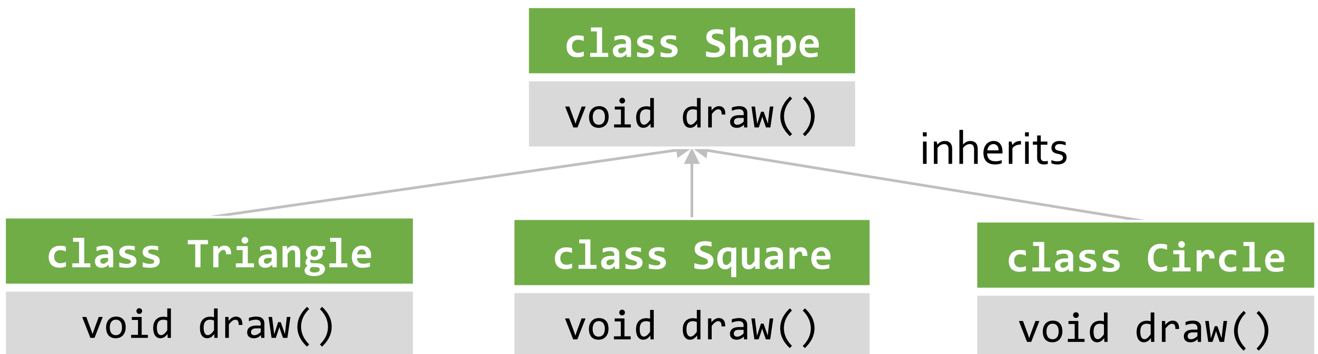
- Goal
  - Learn how to write static analysis pass

- Steps

1) Implement an InstCount pass that inherits `FunctionPass`
   - Count the number of instructions in a function
   - Print the function name and the number of instructions
2) Compile and test the pass

Dynamic Type Casting

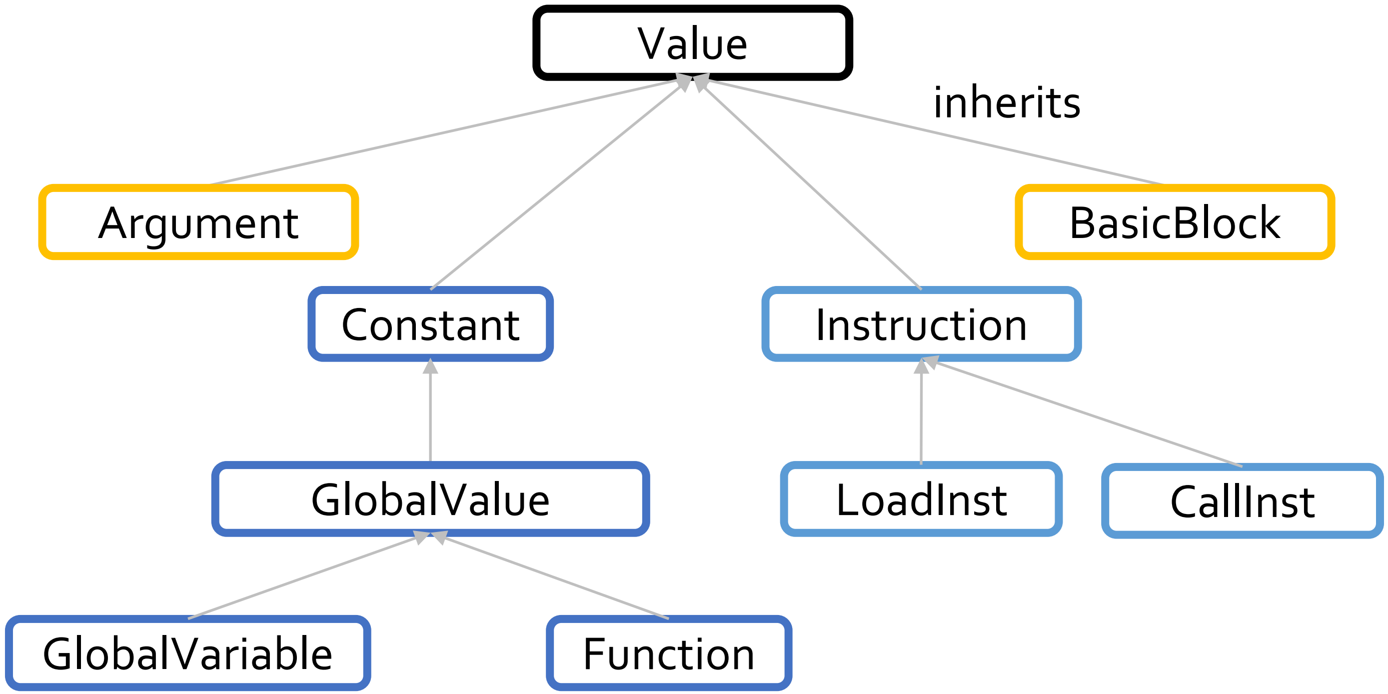
- **Polymorphism** in object-oriented programming
  - A super class type pointer can point to a subclass type instance




```
Shape *shape = nullptr;
if (arg == 3) shape = new Triangle();
else if (arg == 4) shape = new Square();
else shape = new Circle();
shape.draw();
```

LLVM IR Classes

- Is-A relationship of LLVM IR classes
  - Every object is a **Value**!



LLVM IR Classes

- There are various types of instructions

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 %a, i32 %b) #0 {
entry:
  %a.addr = alloca i32, align 4  ☜ alloca instruction
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4  ☜ store instruction
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4  ☜ load instruction
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1  ☜ add instruction
  %2 = load i32, i32* @acc, align 4
  %add1 = add nsw i32 %add, %2
  ret i32 %add1  ☜ return instruction
}
```

Dynamic Type Casting

- How can we distinguish the type of instructions?

```
for(Function &F : M) {
  for(BasicBlock &BB : F) {
    for(Instruction &I : BB) {
      // Do something with I
    }
  }
}
```

Dynamic Type Casting

- Special operators that support polymorphism
  - `isa<Type>`
    - Check the type of an instance that a pointer points
    - Input: Pointer
    - Output
      - True if the pointer points to an instance of "Type"
      - False if not

    - Example

```
if(isa<CallInst>(&I)) {
}
```

Dynamic Type Casting

- Special operators that support polymorphism
  - `dyn_cast<Type>`
    - Cast to the subclass type of a pointer
    - Input: Pointer
    - Output
      - Pointer of “Type” if the type casting is valid
      - Null pointer if not
    - Example

```
CallInst* CI = dyn_cast<CallInst>(&I);
```

```
if(CallInst* CI = dyn_cast<CallInst>(&I)){
}
```

Practice 5: CallInstCount Pass

- Goal
  - Understand runtime types

- Steps
  1) Implement a CallInstCount pass that inherits `FunctionPass`
     - Count the number of <u>call instructions</u> in a function
     - Print the function name and the number of call instructions
  2) Compile and test the pass

Interact with Other Passes

- Passes are **dependent** with each other
  - `opt --debug-pass=Structure` shows the dependence relations
  - Example

```
opt --debug-pass=Structure -reg2mem test.bc -o test.opt.bc
```

```
Pass Arguments:  -targetlibinfo -tti -targetpassconfig -break-crit-edges
-reg2mem -verify -write-bitcode
Target Library Information
Target Transform Information
Target Pass Configuration
  ModulePass Manager
    FunctionPass Manager
      Break critical edges in CFG
      Demote all values to stack slots
      Module Verifier
    Bitcode Writer
```

Interact with Other Passes

1) Include the header file of another pass

```
#include "llvm/Analysis/LoopInfo.h"
```

2) Call `addRequired` in `getAnalysisUsage`

```
void Hello::getAnalysisUsage(AnalysisUsage& AU) const {
  AU.addRequired< LoopInfoWrapperPass >();
  AU.setPreservesAll();
}
```

Interact with Other Passes

3) Bring the analysis result of the pass

```
LoopInfo &LI =
getAnalysis<LoopInfoWrapperPass>(F).getLoopInfo();
```

- Note
  - ModulePass brings FunctionPass

```
getAnalysis<LoopInfoWrapperPass>(F).getLoopInfo();
```

  - FunctionPass brings FunctionPass

```
getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
```

Practice 6: Loop Analysis Pass

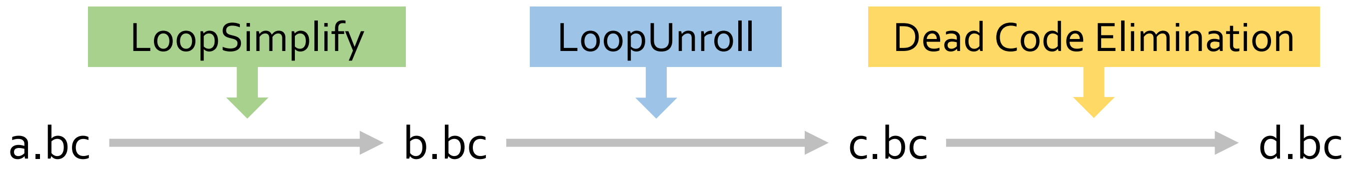
- Goal
  - Learn how to get analysis results from other passes

- Steps
  1) Get LoopInfo from LoopInfoWrapperPass
  2) Print the information about loops
     - The number of loops, the depth of a loop, …
     - Refer to http://llvm.org/doxygen/classllvm_1_1Loop.html
  3) Compile and run the pass

IR Optimization - Transformation

IR Optimization

- LLVM enables *modular optimizations* through the LLVM pass framework
  - **Pass**: Unit of optimization
- Each **LLVM pass** performs optimizations and transformations on LLVM IR
  - Example



IR Code Transformation

- Possible to:
  - Add or delete instructions in existing functions
  - Define new functions
  - Import external functions

- If source code is changed, return `true` at `runOnXXX`

```
bool Hello::runOnModule(Module &M) override {


  return false;
}
```

LLVM IRBuilder

- Help to create instructions and insert them into a basic block

- Include "`llvm/IR/IRBuilder.h`"

- Simple Declaration
  - `IRBuilder<> Builder(I)`
    - Insert instructions before I (Instruction*)
  - `IRBuilder<> Builder(BB)`
    - Insert instructions at the end of BB (BasicBlock*)

LLVM IRBuilder

- Available member functions
  - http://llvm.org/doxygen/classllvm_1_1IRBuilder.html

  - Memory Instructions

LoadInst * CreateLoad (Type *Ty, Value *Ptr, const Twine &Name="")

- `Type *Ty`: Pointer type
- `Value *Ptr`: Pointer to load

StoreInst * CreateStore (Value *Val, Value *Ptr, bool isVolatile=false)

- `Value *Val`: Value to store
- `Value *Ptr`: Pointer to store the value

LLVM IRBuilder

- Available member functions
  - Arithmetic Instructions

```
Value *   CreateAdd (Value *LHS, Value *RHS, const Twine &Name="",
                     bool HasNUW=false, bool HasNSW=false)

Value *   CreateSub (Value *LHS, Value *RHS, const Twine &Name="",
                     bool HasNUW=false, bool HasNSW=false)

Value *   CreateMul (Value *LHS, Value *RHS, const Twine &Name="",
                     bool HasNUW=false, bool HasNSW=false)
```

* NUW = No Unsigned Wrap, NSW = No Unsinged Wrap

LLVM IRBuilder

- Available member functions
  - Other Instructions

```
CallInst *   CreateCall (Value *Callee, ArrayRef< Value *> Args=None,
                         const Twine &Name="", MDNode *FPMathTag=nullptr)
```

ReturnInst *	CreateRetVoid () Create a 'ret void' instruction. More...
ReturnInst *	CreateRet (Value *V) Create a 'ret <val>' instruction. More...

LLVM IRBuilder

- Example
  - Insert a function call for every basic block

```
std::vector<Value*> args(0);
for (BasicBlock &BB : F) {
  IRBuilder<> Builder(BB->getTerminator());
  CallInst *newCallInst = Builder.CreateCall(MarkBBEnd, args, "");
}
```

LLVM IR: Types

- We need to specify **types** when inserting instructions
  - Because LLVM IR is strongly typed 😨

```
LoadInst *   CreateLoad (Type *Ty, Value *Ptr, const Twine &Name="")
```

- How can we do that?
  - How can we create a load instruction for the integer type?

- LLVM maintains IR types as static C++ objects

LLVM IR: Types

- How to get Type instances
  - Use static member functions of class Type

static IntegerType *	getInt1Ty (LLVMContext &C)
static IntegerType *	getInt8Ty (LLVMContext &C)
static IntegerType *	getInt16Ty (LLVMContext &C)
static IntegerType *	getInt32Ty (LLVMContext &C)
static IntegerType *	getInt64Ty (LLVMContext &C)
static IntegerType *	getInt128Ty (LLVMContext &C)

static Type *	getFloatTy (LLVMContext &C)
static Type *	getDoubleTy (LLVMContext &C)

LLVM IR: Types

- Type class member functions
  - Check a type: `isVoidTy`, `isHalfTy`, `isFloatTy`, …

bool	isHalfTy () const Return true if this is 'half', a 16-bit IEEE fp type. More...
bool	isFloatTy () const Return true if this is 'float', a 32-bit IEEE fp type. More...
bool	isDoubleTy () const Return true if this is 'double', a 64-bit IEEE fp type. More...

  - Get the point type of a type

PointerType *	getPointerTo (unsigned AddrSpace=0) const Return a pointer to the current type. More...

How to Create Function

- Multiple ways to create a function
  - `getOrInsertFunction` in `Module` class
    - Signature

FunctionCallee **getOrInsertFunction** (**StringRef Name**, **Type** *RetTy, ArgsTy... Args)
Same as above, but without the attributes. More...

    - Usage

```
FunctionCallee addFun = M.getOrInsertFunction(
  "add",
  Type::getInt64Ty(Context),
  Type::getInt64Ty(Context),
  Type::getInt64Ty(Context)
);
```

Return Type

Argument Types

How to Create Function

- Multiple ways to create a function
  - `getOrInsertFunction` in `Module` class
    - Signature

```
FunctionCallee getOrInsertFunction (StringRef Name, FunctionType *T)
```

    - Usage

```
std::vector<Type*> formals(2);
formals[0] = Type::getInt64Ty(Context);
formals[1] = Type::getInt64Ty(Context);

FunctionType *addFunType = FunctionType::get(
    Type::getInt64Ty(Context), formals, false);
FunctionCallee addFun = M.getOrInsertFunction(
  "add", addFunType);
```

Argument Types

Return Type

How to Create Function

- Multiple ways to create a function
  - Create In Function class
    - Signature

static Function * Create (FunctionType *Ty, LinkageTypes Linkage, const Twine &N, Module &M)
Creates a new function and attaches it to a module. More...

    - Usage

```
std::vector<Type*> formals(2);
formals[0] = Type::getInt64Ty(Context);
formals[1] = Type::getInt64Ty(Context);

FunctionType *addFunType = FunctionType::get(
    Type::getInt64Ty(Context), formals, false)
Function *addFun = Function::Create(
  addFunType, GlobalValue::InternalLinkage,
  "add", &M
);
```

How to Create BasicBlock

- `Create` in `BasicBlock` class
  - Signature

```
static BasicBlock *   Create (LLVMContext &Context, const Twine &Name="",
                              Function *Parent=nullptr, BasicBlock *InsertBefore=nullptr)
```

  - Usage

```
BasicBlock *entry = BasicBlock::Create(Context, "entry", addFun);
```

How to Create Constant

- Constant is also an object!
  - We need to create a constant object to insert a constant argument

- For example,

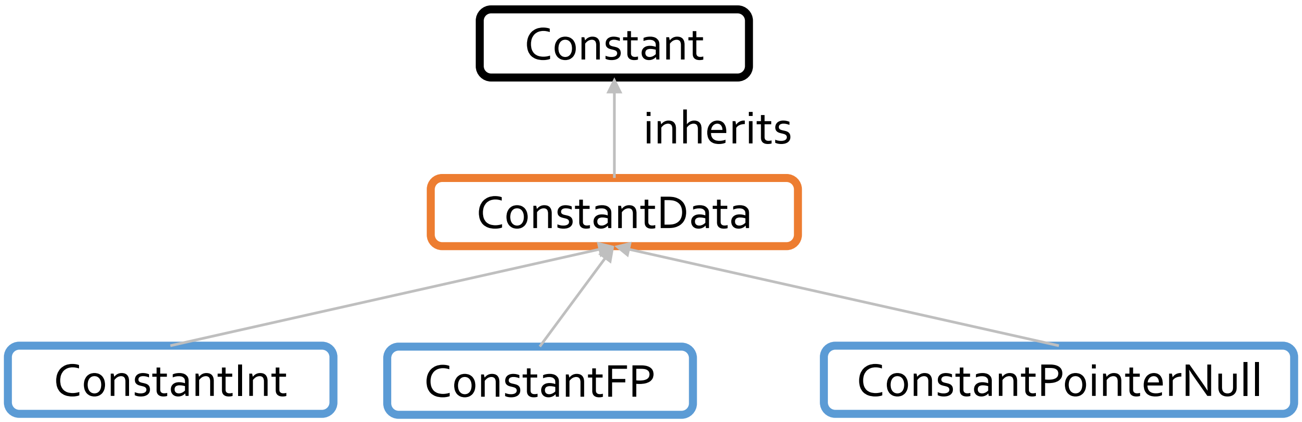
```
define dso_local i32 @addTen(i32 %a) #0 {
entry:
  %a.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %add = add nsw i32 %0, 10
  ret i32 %add
}
```

ConstantInt
Object

```
Value * CreateAdd (Value *LHS, Value *RHS, const Twine &Name="",
                   bool HasNUW=false, bool HasNSW=false)
```

How to Create Constant

- Is-A relationship of Constant classes



How to Create Constant

- `ConstantInt` class
  - Signature

```
static Constant *   get (Type *Ty, uint64_t V, bool isSigned=false)
```

  - Example

```
Constant *One = ConstantInt::get(Type::getInt32Ty(Context), 1);
```

- `ConstantFP` class
  - Signature

```
static Constant *   get (Type *Ty, double V)
```

  - Example

```
Constant *Zero = ConstantFP::get(Type::getFloatTy(Context), 0.0);
```

Practice 7: Insert inc Function

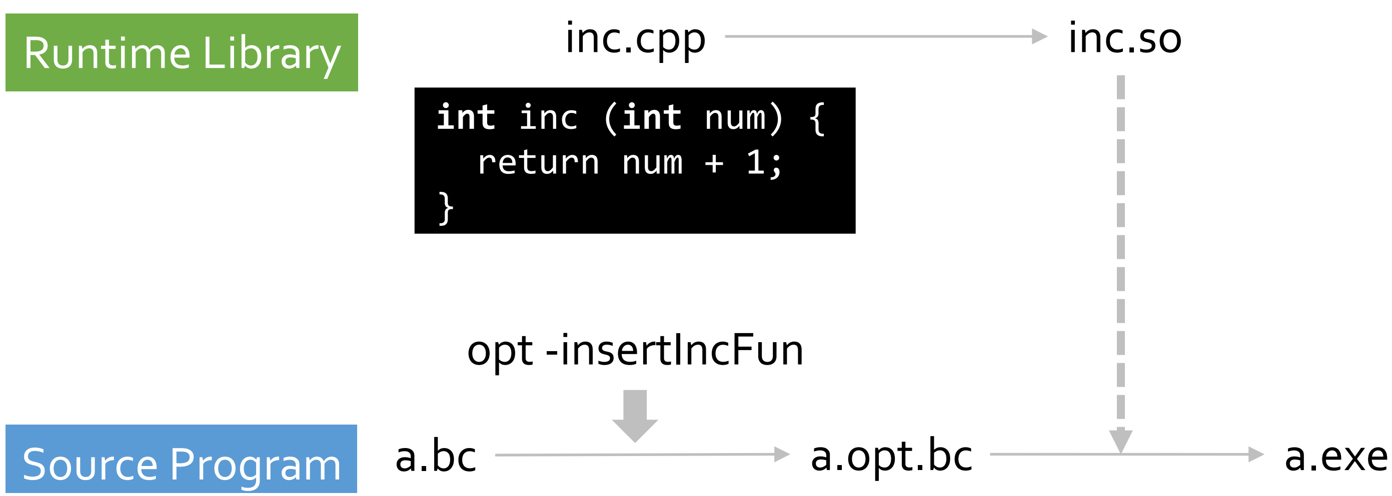
- Goal
  - Learn how to create a function and instructions

- Steps

1) Implement InsertIncFunction that inherits ModulePass
   - Create a function named "Inc"
     - **int** inc (**int** n) { return n + 1; }
   - Create a basic block for the Inc function
2) Fill the Inc function with instructions
3) Run the pass with `opt` on a sample program

How to Instrument Runtime Function Call

- It is time-consuming to implement functions in LLVM IR
- Implement functions in C/C++, then **link them as a library!**



How to Instrument Runtime Function Call

- Steps

1) Write a runtime function in C/C++
   - Use `extern "C"` for c++ functions to avoid name mangling

2) In a Pass, create a Function object for the runtime function with `getOrInsertFunction`
   - The function name and type in LLVM IR must match!
   - No need to fill the function (like an extern function)

3) Link the runtime function

How to Instrument Runtime Function Call

- Example: Load Tracer
  - Goal
    - Trace every load instruction in a program
  - Runtime Function
    - void traceLoadInstr(void *addr, InstID, instID)
      - Log the address and instruction ID of a load instruction
  - Need to insert the runtime function before every load instruction
    - With appropriate arguments

How to Instrument Runtime Function Call

- Example: Load Tracer
  - `LoadTracerRuntime.cpp`

```
extern "C"
void traceLoadInstr(void *addr, InstID instID) {
  // Do something
}
```

How to Instrument Runtime Function Call

- Example: Load Tracer
  - `LoadTracerPass.cpp`

```
traceLoadInstr = M.getOrInsertFunction(
    "traceLoadInstr",
    Type::getVoidTy(Context),
    Type::getInt64Ty(Context),  // Address
    Type::getInt64Ty(Context)); // Instruction ID
```

```
if(LoadInst *Load = dyn_cast<LoadInst>(&I)) {
  // Some Code
  actuals.resize(2);
  actuals[0] = CastedAddr;
  actuals[1] = ConstantInt::get(Type::getInt64Ty(Context), instID);
  CallInst::Create(traceLoadInstr, actuals, "", Load);
}
```

How to Instrument Runtime Function Call

- Example: Compilation Process
  - Compile the runtime code

```
$ clang++ -c –fpic LoadTracerRuntime.cpp -o LoadTracerRuntime.o
$ clang++ -shared -o LoadTracerRuntime.so LoadTracerRuntime.o
```

  - Run the pass that inserts runtime function calls

```
$ opt –load LoadTracer.so -traceload test.bc –o test.opt.bc
```

  - Link the runtime code

```
$ clang++ test.opt.bc -o test.exe -lLoadTracerRuntime
```

Practice 8: Dynamic CallCount

- Goal
  - Learn how to insert runtime function calls

- Steps

1) Implement a DynCallCount pass that inherits `ModulePass`
   - Insert countCall() before every CallInst instructions
   - Insert printResult() before every Ret instructions in the function `main`
     - Code at `practice/Runtime/callCount.c`
2) Compile and run the pass
3) Compile the runtime library
   - Type '`make`' at `pratice/Runtime`
4) Link the runtime library

Putting It All Together

Toy Project: Memory Profiler

- Implement a memory profiler that tracks
  - Memory allocations
  - Load instructions
  - Store instructions

- Steps
  - Define three runtime functions:
    - **void** traceMalloc(**void** *addr, **size_t** size)
    - **void** traceLoad(**void** *addr, **size_t** size)
    - **void** traceStore(**void** *addr, **size_t** size)
  - Insert the runtime functions before or after
    - Call instructions to malloc
    - Load instructions
    - Store instructions

Toy Project: Memory Profiler

- Hints
  1. You need to **define FunctionType** for runtime functions
     - **void*** in C = **i8*** in LLVM IR
     - How to get the i8* type
       - `Type::getInt8PtrTy(Context)`
  2. You need to **find out the size of load and store**
     - How to get the size of a type
       - `const DataLayout &Layout = M.getDataLayout();`
       - `TypeSize Size = Layout.getTypeSizeInBits(Type *)`
  - You need to **cast i32* to i8*** for load and store
    - How to cast a pointer value
      - `Builder.CreatePointerCast(Value *, Type *)`

Toy Project: Memory Profiler

- Example Output

Source Code

```
int main () {
    int *mat = (int *) malloc(MALLOC_SIZE);

    for (int i = 0; i < MALLOC_SIZE; i++) {
        mat[i] = rand();
    }

    for (int i = 0; i < MALLOC_SIZE; i++) {
        printf("%p\n", &mat[i]);
    }

    return 0;
}
```

After Applying Pass

```
clang test.opt.bc runtime.cpp -lstdc++ -o test.exe
./test.exe
[Store] Addr: 0x7ffc987c8e7c Size: 4
[Memory Allocation] Addr: 0x1472280 Size: 10
[Store] Addr: 0x7ffc987c8e70 Size: 8
[Store] Addr: 0x7ffc987c8e6c Size: 4
[Load] Addr: 0x7ffc987c8e6c Size: 4
[Load] Addr: 0x7ffc987c8e70 Size: 8
[Load] Addr: 0x7ffc987c8e6c Size: 4
[Store] Addr: 0x1472280 Size: 4
[Load] Addr: 0x7ffc987c8e6c Size: 4
[Store] Addr: 0x7ffc987c8e6c Size: 4
[Load] Addr: 0x7ffc987c8e6c Size: 4
```

Thank you!

Seonyeong Heo
seoheo@ethz.ch