# CSCD70 Compiler Optimization

## Tutorial #1 Introduction to LLVM

Bojian Zheng
bojian@cs.toronto.edu

Department of Computer Science, University of Toronto

**Acknowledgement**: Thanks to Professor Gennady Pekhimenko, Professor Nandita Vijaykumar and students from previous offerings of CSCD70.

# Abstract

In this tutorial, we will be discussing the following topics:

- C++ (P?)Review
- How to write an LLVM Analysis pass?

# C++ (P?)Review

# C++ (P?)Review

- "C with Classes" (i.e., Object-Oriented Programming)
- In today's tutorial, we will cover the following differences:
  - Pass by Reference: `void foo(int &a);`
  - Public Inheritance:

```
class Fox : public Animal {
};
```

  - Standard Template Library (STL)

# Pass by Reference

```
void foo(int &a);
```

- Why is there an & before variable `a`? **Pass by Reference**
- Review: What is the purpose of "Pass by Pointer"?
  - Modify the values and/or
  - Avoid the overhead of copying large objects.
- However, need to take the pain of indirect accesses:

  ```
  int *a; a = &...; ... = *a;
  ```

  ☞ References allow us to access the variables **as is**.
- Reference must be initialized by a variable. There is no null reference.

☞ `Example1-Cpp_PReview/VarReference.cpp`

# Public Inheritance

- C++ `class`: C `struct` with methods.

```
class A {
  void DoSomething();
};
```

- Similar to what you have seen in Java & Python, **public inheritance** allows us to leverage the members and methods of the base class:

```
class Fox extends Animal {
}   // Java
```

```
class Fox(Animal):
    pass   # Python
```

```
class Fox : public Animal {
};  // C++
```

# Public Inheritance

- Public inheritance can implement abstract methods of the base class.

```
class Animal {
  virtual void Run() = 0;  // Abstract Method
};
class Fox : public Animal {
  virtual void Run() override;
};
```

- Base class pointer/reference can point to any child class instances.

```
Fox fox = Fox();
Animal &animal = fox;
```

- **Dynamic casting** can be used to downcast points/references.

```
Fox &fox = dynamic_cast<Fox &>(animal);
```

☞ Example1-Cpp_PReview/PublicInheritance.cpp

# Standard Template Library (STL)

- One of the primary reasons of using C++.
- Provides easy access to common data structures. E.g.,
  - `vector` – `List`-like Data Structure

    ```
    vector<unsigned> a = {1, 2, 3, 4, 5};
    ```

  - `unordered_map` – `Dict`-like Data Structure

    ```
    unordered_map<string, unsigned> b = {
          {"Red", 0}, {"Green", 1}, {"Blue", 2}
        };
    ```

# Standard Template Library (STL)

- We traverse through STL containers using **iterators**:

```
vector<unsigned> a = {1, 2, 3, 4, 5};

for (auto iter = a.begin();
     iter != a.end(); ++iter) {
  // dereference the iterator just like pointer
  unsigned &a_elem = *iter;  // 1, 2, 3, 4, 5
}
```

- Dereferencing an iterator gives reference to the stored elements.

☞ Example1-Cpp_PReview/STL.cpp

# LLVM① Analysis③ Pass②

# LLVM

- State-of-the-Art Compiler Framework similar to GCC
- Modular and Well Documented

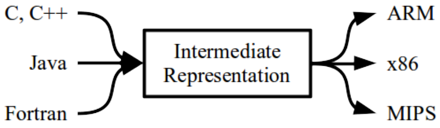
# Intermediate Representation

During the 1st lecture, we will cover numerous optimizations, e.g., Common Subexpression Elimination, Constant Propagation, ...

**Observation**

Many optimizations are **invariant** w.r.t. **frontend** and **backend**.

⇒ Want to have an **intermediate representation** that is **portable** to multiple programming languages and target machines and have **optimization passes** act on those representations.



# Intermediate Representation

**Intermediate Representation** (IR) has syntax and semantics similar to the assembly language that you are familiar with.

```c
int main()
{
    return 0;
}
```

Clang

$\Rightarrow$

```llvm
define i32 @main() ...
{
    ret i32 0
}
```

# Analysis and Transform Pass

We categorize optimization passes into **Analysis** and **Transform**.

- **Analysis** passes gather information about the program.
- **Transform** passes mutate the program.

Why such isolation exists?

- Better **Readability**
- Very frequently, multiple passes might require the **same** information.
  $\Rightarrow$ The isolation **avoids redundant analysis** (more later).

# How to write an LLVM Analysis Pass?

To answer this, we need to first understand the followings:

- **LLVM Module**: How is our program translated in LLVM?
- **Iterators**: How to traverse through the module?
- **Downcasting**: How to get more information out of the iterators?
- **LLVM Pass Interfaces**: What interfaces does LLVM provide us with?

# LLVM Module

**Your Program**

- Files
- Functions
- Basic Blocks
- Statements

**LLVM Module**

- `Module` list of `Functions` + Global Variables
- `Function` list of `BasicBlocks` + Arguments
- `BasicBlock` list of `Instructions`
- `Instruction` Opcode + Operands

# Iterators

```
Module &M = ...;
for (auto iter = M.begin(); iter != M.end(); ++iter) {
  Function &F = *iter;
  // do some stuff for each function
}
```

**Comment**

Similar syntax with STL container `vector`.

# Downcasting

Why do we need **Downcasting**?

- Suppose that we have an `Instruction`, how do we know whether this is an unary instruction? or a binary operator? or a branch instruction? etc.
- **Downcasting** helps us retrieve more information from the iterators.
- E.g.

```
Instruction *inst = ...
if (CallInst *call_inst = dyn_cast<CallInst>(inst))
  outs() << "I am a call instruction"
         << std::endl;
```

# LLVM Pass Interfaces

- LLVM provides multiple pass interfaces that target at different module levels (e.g., `FunctionPass`) and even more (e.g., `LoopPass`).
- How to use those interfaces? ⇒**Public Inheritance**

```
class ModulePass {
  virtual bool runOnModule(Module &M) = 0;
};

class MyModulePass : public ModulePass {
  bool runOnModule(Module &M) { for (iter = ... }
};
```

# Review

In this tutorial, we have discussed the following topics:

- C++ (P?)Review
  - Pass by Reference
  - Public Inheritance
  - Standard Template Library (STL)
- How to write an LLVM Analysis pass?
  - Intermediate Representation (IR) and Optimization Passes
  - Analysis vs. Transform
  - LLVM Module, Iterators, Downcasting, LLVM Pass Interfaces

☞ **Homework Assignment**: `FunctionInfo`