FREE eBook

LEARNING

llvm

Free unaffiliated eBook created from
Stack Overflow contributors.

#llvm

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: llvm

It is an unofficial and free llvm ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official llvm.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with llvm

Remarks

This section provides an overview of what llvm is, and why a developer might want to use it.

It should also mention any large subjects within llvm, and link out to the related topics. Since the Documentation for llvm is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

It is always recommend to go to the LLVM official website and follow the installation guides depending on your OS.

If you are working on posix then in short you have to add one of the official LLVM package repositories. For example if you work on Ubuntu Xenial (16.04) you add a `deb` and `deb-src` entry to your `/etc/apt/sources.list` file:

```
$ sudo su
$ echo deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-4.0 main \ >>
/etc/apt/sources.list
$ echo deb-src http://apt.llvm.org/xenial/ llvm-toolchain-xenial-4.0 main \ >>
/etc/apt/sources.list
```

and once you do that the installation is as simple as calling

```
$ sudo apt update
$ sudo apt install clang-X
```

where `X` is the version you are looking for (4.0 is current at the time of writing this post).

Note that clang is a C/C++ compiler writen over LLVM (and actually is self hosted now) and comes together with all LLVM libraries. Once you do that you can go to any turorial and start coding.

If you wish you can install LLVM libraries manually. For that you just have to `apt install llvm-Y` where `Y` is a library you are looking for. However I do recommend compiling LLVM using projects with clang.

Once you do that you should have `llvm-config` tool. It is very useful to get compiler flags needed for correct LLVM project compilation. So the first test that it worked would be by calling

```
$ llvm-config-4.0 --cxxflags --libs engine
-I/usr/lib/llvm-4.0/include -std=c++0x -gsplit-dwarf -Wl,-fuse-ld=gold -fPIC -fvisibility-
inlines-hidden -Wall -W -Wno-unused-parameter -Wwrite-strings -Wcast-qual -Wno-missing-field-
initializers -pedantic -Wno-long-long -Wno-maybe-uninitialized -Wdelete-non-virtual-dtor -Wno-
```

```
comment -Werror=date-time -std=c++11 -ffunction-sections -fdata-sections -O2 -g -DNDEBUG  -
fno-exceptions -D_GNU_SOURCE -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS -
D__STDC_LIMIT_MACROS
-lLLVM-4.0
```

You may get a different set of flags, do not worry about it. As long as it doesn't fail with `command not found` you should be fine.

Next step is to test the actual LLVM library itself. So lets create a simple `llvmtest.cpp` file:

```
#include <iostream>
#include "llvm/IR/LLVMContext.h"

int main() {
    llvm::LLVMContext context;
    std::cout << &context << std::endl;
    return 0;
};
```

Note that I use `std::cout` so that we actually use the `context` variable (so the compiler won't remove it during the compilation phase). Now compile the file with

```
$ clang++-4.0 -o llvmtest `llvm-config-4.0 --cxxflags --libs engine` llvmtest.cpp
```

and test it

```
$ ./llvmtest
0x7ffd85500970
```

Congratulations! You are ready to use LLVM.

Compilation of a simple function in llvm 4.0

So what we will try to do is to compile a following function

```
int sum(int a, int b) {
    return a + b + 2;
}
```

on the fly. And here's the entire `.cpp` example:

```
#include <iostream>

#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/Verifier.h"
#include "llvm/ExecutionEngine/ExecutionEngine.h"
#include "llvm/ExecutionEngine/SectionMemoryManager.h"
#include "llvm/ExecutionEngine/Orc/CompileUtils.h"
#include "llvm/Support/TargetSelect.h"
```

```
// Optimizations
#include "llvm/Transforms/Scalar.h"
#include "llvm/Analysis/BasicAliasAnalysis.h"

using namespace llvm;


llvm::Function* createSumFunction(Module* module) {
    /* Builds the following function:

    int sum(int a, int b) {
        int sum1 = 1 + 1;
        int sum2 = sum1 + a;
        int result = sum2 + b;
        return result;
    }
    */

    LLVMContext &context = module->getContext();
    IRBuilder<> builder(context);

    // Define function's signature
    std::vector<Type*> Integers(2, builder.getInt32Ty());
    auto *funcType = FunctionType::get(builder.getInt32Ty(), Integers, false);

    // create the function "sum" and bind it to the module with ExternalLinkage,
    // so we can retrieve it later
    auto *fooFunc = Function::Create(
        funcType, Function::ExternalLinkage, "sum", module
    );

    // Define the entry block and fill it with an appropriate code
    auto *entry = BasicBlock::Create(context, "entry", fooFunc);
    builder.SetInsertPoint(entry);

    // Add constant to itself, to visualize constant folding
    Value *constant = ConstantInt::get(builder.getInt32Ty(), 0x1);
    auto *sum1 = builder.CreateAdd(constant, constant, "sum1");

    // Retrieve arguments and proceed with further adding...
    auto args = fooFunc->arg_begin();
    Value *arg1 = &(*args);
    args = std::next(args);
    Value *arg2 = &(*args);
    auto *sum2 = builder.CreateAdd(sum1, arg1, "sum2");
    auto *result = builder.CreateAdd(sum2, arg2, "result");

    // ...and return
    builder.CreateRet(result);

    // Verify at the end
    verifyFunction(*fooFunc);
    return fooFunc;
};

int main(int argc, char* argv[]) {
    // Initilaze native target
    llvm::TargetOptions Opts;
    InitializeNativeTarget();
    InitializeNativeTargetAsmPrinter();
```

```
    LLVMContext context;
    auto myModule = make_unique<Module>("My First JIT", context);
    auto* module = myModule.get();

    std::unique_ptr<llvm::RTDyldMemoryManager> MemMgr(new llvm::SectionMemoryManager());

    // Create JIT engine
    llvm::EngineBuilder factory(std::move(myModule));
    factory.setEngineKind(llvm::EngineKind::JIT);
    factory.setTargetOptions(Opts);
    factory.setMCJITMemoryManager(std::move(MemMgr));
    auto executionEngine = std::unique_ptr<llvm::ExecutionEngine>(factory.create());
    module->setDataLayout(executionEngine->getDataLayout());

    // Create optimizations, not necessary, whole block can be ommited.
    // auto fpm = llvm::make_unique<legacy::FunctionPassManager>(module);
    // fpm->add(llvm::createBasicAAWrapperPass());
    // fpm->add(llvm::createPromoteMemoryToRegisterPass());
    // fpm->add(llvm::createInstructionCombiningPass());
    // fpm->add(llvm::createReassociatePass());
    // fpm->add(llvm::createNewGVNPass());
    // fpm->add(llvm::createCFGSimplificationPass());
    // fpm->doInitialization();

    auto* func = createSumFunction(module);  // create function
    executionEngine->finalizeObject();       // compile the module
    module->dump();                          // print the compiled code

    // Get raw pointer
    auto* raw_ptr = executionEngine->getPointerToFunction(func);
    auto* func_ptr = (int(*)(int, int))raw_ptr;

    // Execute
    int arg1 = 5;
    int arg2 = 7;
    int result = func_ptr(arg1, arg2);
    std::cout << arg1 << " + " << arg2 << " + 1 + 1 = " << result << std::endl;

    return 0;
}
```

It should work fine when compiled with **clang++-4.0** with following flags:

```
$ llvm-config-4.0 --cxxflags --libs core
```

Read Getting started with llvm online: https://riptutorial.com/llvm/topic/3390/getting-started-with-llvm

Credits

S. No	Chapters	Contributors
1	Getting started with llvm	Ben Steffan, Community, freakish