


LLVM Frontends

Introduction to the various frontend languages and their integration with LLVM



Overview

LLVM (Low-Level Virtual Machine) is a compiler infrastructure framework that provides a set of reusable libraries and tools for building compilers. One of the key features of LLVM is its modular architecture, which allows it to support multiple programming languages through different frontend languages.



A frontend in the context of LLVM refers to the part of a compiler that analyzes and processes the source code written in a specific programming language. It performs tasks like lexing, parsing, semantic analysis, and type checking. The frontend translates the high-level source code into an intermediate representation (IR) that can be further optimized and transformed by LLVM.



LLVM supports various frontend languages

- C
- C++
- Objective-C
- Objective-C++
- RUST
- Swift
- Fortran



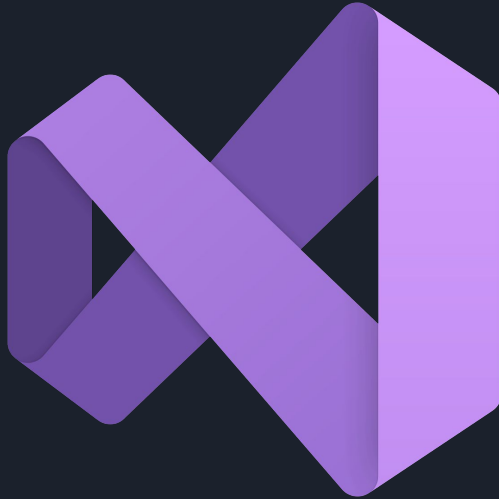
Clang is considered to be a production quality C, Objective-C, C++ and Objective-C++ compiler when targeting any target supported by LLVM. As example, Clang is used in production to build performance-critical software like Chrome or Firefox.





Why?

Fast compiles and low memory use



GCC & MSVC compatibility



rustc is the compiler for the Rust programming language, provided by the project itself.

The Rust programming language focuses on safety, performance, and concurrency. Rustc enforces strict memory safety guarantees through its ownership, borrowing, and lifetimes system. It also implements zero-cost abstractions, which means that high-level abstractions in Rust come with minimal runtime overhead. This makes Rust programs efficient and suitable for systems programming, web development, and other performance-critical applications.





Swift

Swiftc is the command-line compiler for the Swift programming language. It is used to translate Swift source code into executable binaries or object files that can be run on various platforms, including macOS, iOS, watchOS, and tvOS. The "c" in "swiftc" stands for "compiler," indicating its role in compiling Swift code.

The swiftc compiler takes Swift source code files (with .swift extension) as input and performs several steps to produce an executable or object file. These steps include lexical analysis, syntax parsing, semantic analysis, optimization, and code generation. The output of the compiler depends on the provided options, such as whether you want to generate an executable, a dynamic library, or an object file.

Flang is an open-source Fortran language compiler developed by the LLVM project. It aims to provide modern Fortran language support, including features from the latest Fortran standards such as Fortran 2018 and parts of Fortran 202x. Flang is designed to be compatible with existing Fortran codes while also providing performance optimizations through the LLVM infrastructure.

The Flang compiler leverages the LLVM Compiler Infrastructure, which is a widely used compiler framework that supports multiple programming languages. By utilizing LLVM, Flang can take advantage of its optimization capabilities, code generation, and other features.





```
#include <iostream>
```

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

```
#include <llvm/IR/Module.h>
```

```
#include <llvm/IR/IRBuilder.h>
```

```
#include <llvm/IR/Constants.h>
```

```
#include <llvm/ExecutionEngine/ExecutionEngine.h>
```

```
#include <llvm/ExecutionEngine/GenericValue.h>
```

```
#include <llvm/Support/TargetSelect.h>
```



```
// Initialize LLVM components
```

```
    llvm::InitializeNativeTarget();
```

```
    llvm::InitializeNativeTargetAsmPrinter();
```



```
// Create LLVM context, module, and builder
```

```
llvm::LLVMContext llvmContext;
```

```
llvm::Module llvmModule("MyModule", llvmContext);
```


```
llvm::IRBuilder<> builder(llvmContext);
```



// Define the function signature

```
llvm::FunctionType* funcType = llvm::FunctionType::get(builder.getInt32Ty(), { builder.getInt32Ty(), builder.getInt32Ty() }, false);
```

```
llvm::Function* sumFunc = llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, "sum", llvmModule);
```



```
// Create the entry basic block
```

```
    llvm::BasicBlock* entryBlock = llvm::BasicBlock::Create(llvmContext,  
    "entry", sumFunc);
```

```
    builder.SetInsertPoint(entryBlock);
```



```
// Get function arguments
```

```
llvm::Argument* arg1 = &(*sumFunc->args().begin());
```

```
llvm::Argument* arg2 = &(*(sumFunc->args().begin() + 1));
```



```
// Perform addition
```

```
llvm::Value* sum = builder.CreateAdd(arg1, arg2);
```

```
// Create the return instruction
```

```
builder.CreateRet(sum);
```

```
// Verify the module
```

```
llvm::verifyModule(llvmModule);
```




```
// Create the execution engine
```

```
    std::string error;
```

```
    llvm::ExecutionEngine* executionEngine =  
    llvm::EngineBuilder(std::unique_ptr<llvm::Module>(&llvmModule))  
        .setErrorStr(&error)  
        .create();
```



```
if (!executionEngine) {  
    std::cerr << "Failed to create ExecutionEngine: " << error << std::endl;  
    return 1;  
}
```



```
// JIT-compile the module
```

```
    executionEngine->finalizeObject();
```

```
// Get the function pointer
```

```
    auto sumFnPtr = (int (*)(int,  
int))executionEngine->getFunctionAddress("sum");
```



```
// Call the function
```

```
int result = sumFnPtr(5, 7);
```

```
std::cout << "Result: " << result << std::endl;
```



```
// Clean up
```

```
delete executionEngine;
```

Thank you so much

Sajjad Ranjbar Yazdi

