Understanding the Clang AST

December 31, 2015

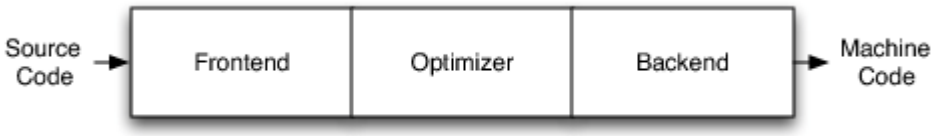
Clang is everywhere; It is the core of my favorite Vim plugin YouCompleteMe, it recently got supported by Microsoft Visual Studio, it is mentioned in numerous episodes of CppCast and it powers the excellent clang formatter. Naturally, I wanted to get a better understanding of how the clang front end works under the hood.

Table of Content

Clang Front End & AST

Clang is a *C language family front end* for LLVM. In compiler design, a front end takes care of the **analysis** part, which means breaking up the source code into pieces according to a grammatical structure. The result is an intermediate representation

which is transformed in a target program by the back end, called **synthesis**. Optionally, there is an intermediate step between the frond- and back end for **optimization**.



In concrete terms, the front end is responsible for parsing the source code, checking it for errors and turning the input code into an **Abstract Syntax Tree** (AST). The latter is a structured representation, which can be used for different purposes such as creating a symbol table, performing type checking and finally generating code. The AST is the part I'm mainly interested in, as it is clang's core, where all the interesting stuff happens.

From the clang documentation, I found the video below. Manuel Klimek gives an easy to follow explanation of the AST as well as its applications. I highly recommend watching this video!

ASTContext

The ASTContext keeps information about the AST of a translation unit that is not stored in its nodes. Examples are the identifier table and the source manager to name a few. It also forms the entry point to the AST by means of the `TranslationUnitDecl* getTranslationUnitDecl()` method.

Classes

The AST is built using three groups of core classes: declarations, statements and types. If you follow the links to the doxygen, you'll see that these three classes form the base of a range of specializations. However, it is important to note that they do not inherit from a common base class. As a result, there is no common interface for visiting all the nodes in the tree. Each node has dedicated *traversal methods* that allow you to navigate the tree. Further on I'll show how we can use visitors to do so efficiently, without needing a common API.

Example

Consider the if-statement in the code example below, which is represented by an `IfStmt` in the AST printed underneath. It consists of a conditional `Expr` (`BinaryOperator`) and two `Stmt` s, one for the then-case and one for the else-case respectively.

```
$ cat example.cpp
int f(int i) {
    if (i > 0) {
        return true;
    } else {
        return false;
    }
}

$ clang -Xclang -ast-dump -fsyntax-only example.cpp
[part of the AST left out for conciseness]
    `-IfStmt 0x2ac4638 <line:2:5, line:6:5>
      |-<<<NULL>>>
      |-BinaryOperator 0x2ac4540 <line:2:9, col:13> '_Bool' '>'
      | |-ImplicitCastExpr 0x2ac4528 <col:9> 'int' <LValueToRValue>
      | | `-DeclRefExpr 0x2ac44e0 <col:9> 'int' lvalue ParmVar 0x2ac4328 'i' 'int'
      | `-IntegerLiteral 0x2ac4508 <col:13> 'int' 0
      |-CompoundStmt 0x2ac45b0 <col:16, line:4:5>
      | `-ReturnStmt 0x2ac4598 <line:3:9, col:16>
      |   `-ImplicitCastExpr 0x2ac4580 <col:16> 'int' <IntegralCast>
      |     `-CXXBoolLiteralExpr 0x2ac4568 <col:16> '_Bool' true
      `-CompoundStmt 0x2ac4618 <line:4:12, line:6:5>
        `-ReturnStmt 0x2ac4600 <line:5:9, col:16>
          `-ImplicitCastExpr 0x2ac45e8 <col:16> 'int' <IntegralCast>
            `-CXXBoolLiteralExpr 0x2ac45d0 <col:16> '_Bool' false
```

Now how do we traverse this sub-tree? Easy, by calling one of `IfStmt` 's dedicated methods.

```
const Expr * getCond () const
const Stmt * getThen () const
const Stmt * getElse () const
```

Each node in the tree is a specific class with dedicated methods. Dumping the AST, as illustrated above, is the easiest way to find the appropriate class representation. Finding the right methods is just a matter of looking at the class reference.

Navigating Sources

The location of a token is represented by the SourceLocation class. Because this object is embedded in many of the AST nodes, it is required to be very small. This is achieved by working together with the SourceManager to encode the actual location in the source file.

AST Traversal

In order to manipulate the AST, we need to be able to traverse the syntax tree. Clang offers two abstractions to make our life easier: the [RecursiveASTVisitor] (http://clang.llvm.org/docs/RAVfront endAction.html) and ASTMatchers.

Let's consider 3 available approaches to a rudimentary use case where we want to find all calls to `doSomething` in a given source file `test.cpp` .

The first two use LibTooling, a library that enables the creation of a stand-alone tool. In addition to providing a nice C++ interface, it takes care of building up the actual syntax tree, saving you from having to reinvent the wheel. You have full control over the AST and you can even share code with Clang Plugins. However, as LLVM and Clang are fast-moving projects, you might have to change some API calls when a new version is released.

However, when a high-level and stable interface is important, Clang's C interface (LibClang), is the preferred choice. In the third example we explore AST traversal with **cursors**, which represent locations within the Abstract Syntax Tree.

Recursive AST Visitor

The Recursive AST Visitor enables you to traverse the nodes of Clang AST in a depth-first manner. We visit specific nodes by extending the class and implementing the desired `Visit*` method, i.e. `VisitCallExpr` for the example below. For a step-by-step tutorial on how to build your own visitor, have a look at this tutorial in Clang's docs.

```
#include "clang/AST/AST.h"
#include "clang/AST/ASTConsumer.h"
```

```
#include "clang/AST/ASTContext.h"
#include "clang/AST/RecursiveASTVisitor.h"
#include "clang/Driver/Options.h"
#include "clang/Frontend/ASTConsumers.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Rewrite/Core/Rewriter.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"

using namespace clang;
using namespace clang::tooling;

class FindNamedCallVisitor : public RecursiveASTVisitor<FindNamedCallVisitor> {
 public:
  explicit FindNamedCallVisitor(ASTContext *Context, std::string fName)
      : Context(Context), fName(fName) {}

  bool VisitCallExpr(CallExpr *CallExpression) {
    QualType q = CallExpression->getType();
    const Type *t = q.getTypePtrOrNull();

    if (t != NULL) {
      FunctionDecl *func = CallExpression->getDirectCallee();
      const std::string funcName = func->getNameInfo().getAsString();
      if (fName == funcName) {
        FullSourceLoc FullLocation =
            Context->getFullLoc(CallExpression->getLocStart());
        if (FullLocation.isValid())
          llvm::outs() << "Found call at "
                       << FullLocation.getSpellingLineNumber() << ":"
                       << FullLocation.getSpellingColumnNumber() << "\n";
      }
    }

    return true;
  }

 private:
  ASTContext *Context;
  std::string fName;
};

class FindNamedCallConsumer : public clang::ASTConsumer {
 public:
  explicit FindNamedCallConsumer(ASTContext *Context, std::string fName)
```

```
      : Visitor(Context, fName) {}

  virtual void HandleTranslationUnit(clang::ASTContext &Context) {
    Visitor.TraverseDecl(Context.getTranslationUnitDecl());
  }

 private:
  FindNamedCallVisitor Visitor;
};

class FindNamedCallAction : public clang::ASTFrontendAction {
 public:
  FindNamedCallAction() {}

  virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
      clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
    const std::string fName = "doSomething";
    return std::unique_ptr<clang::ASTConsumer>(
        new FindNamedCallConsumer(&Compiler.getASTContext(), fName));
  }
};

static llvm::cl::OptionCategory MyToolCategory("my-tool options");

int main(int argc, const char **argv) {
  const std::string fName = "doSomething";

  CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);
  ClangTool Tool(OptionsParser.getCompilations(),
                 OptionsParser.getSourcePathList());

  // run the Clang Tool, creating a new FrontendAction (explained below)
  int result = Tool.run(newFrontendActionFactory<FindNamedCallAction>().get());
  return result;
}
```

AST Matchers

The AST Matcher API provides a **Domain Specific Language** (DSL) for matching predicates on Clang's AST. Matching a call expression named *doSomething* looks like this:

```
callExpr(callee(functionDecl(hasName("doSomething"))))
```

The AST Matcher Reference explains you how to use the DSL to build matchers for the nodes you're interested in. A MatchCallback is attached that is executed when the predicate is matched. The LLVM documentation contains a comprehensive tutorial on getting started with matchers.

```
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"

#include "llvm/Support/CommandLine.h"

using namespace clang::ast_matchers;
using namespace clang::tooling;
using namespace clang;
using namespace llvm;

class MyPrinter : public MatchFinder::MatchCallback {
 public:
  virtual void run(const MatchFinder::MatchResult &Result) {
    ASTContext *Context = Result.Context;
    if (const CallExpr *E =
            Result.Nodes.getNodeAs<clang::CallExpr>("functions")) {
      FullSourceLoc FullLocation = Context->getFullLoc(E->getLocStart());
      if (FullLocation.isValid()) {
        llvm::outs() << "Found call at " << FullLocation.getSpellingLineNumber()
                     << ":" << FullLocation.getSpellingColumnNumber() << "\n";
      }
    }
  }
};

// Apply a custom category to all command-line options so that they are the
// only ones displayed.
static llvm::cl::OptionCategory MyToolCategory("my-tool options");

// CommonOptionsParser declares HelpMessage with a description of the common
// command-line options related to the compilation database and input files.
// It's nice to have this help message in all tools.
static cl::extrahelp CommonHelp(CommonOptionsParser::HelpMessage);

// A help message for this specific tool can be added afterwards.
static cl::extrahelp MoreHelp("\nMore help text...");
```

```
int main(int argc, const char **argv) {
  CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);
  ClangTool Tool(OptionsParser.getCompilations(),
                 OptionsParser.getSourcePathList());

  MyPrinter Printer;
  MatchFinder Finder;

  StatementMatcher functionMatcher =
      callExpr(callee(functionDecl(hasName("doSomething")))).bind("functions");

  Finder.addMatcher(functionMatcher, &Printer);

  return Tool.run(newFrontendActionFactory(&Finder).get());
}
```

In both cases we used the ASTContext to obtain the statement's location in the source file. Note that with AST matchers the context information is provided, while we were required to take care of this our self in the visitor approach.

The output for both programs is identical. The double dashes indicate that instead of using a compilation database we are passing (no) compiler options as arguments.

```
$ cat test.cpp
class Y {
public: void doSomething();
};

void z() { Y y; y.doSomething(); }

int doSomething(int i) {
    if (i == 0) return 0;
    return 1 + doSomething(i--);
}

int main() {
    return doSomething(2);
}

$ bin/examle test.cpp --
Found call at 5:17
Found call at 9:16
Found call at 13:12
```

Cursors

The third and last example uses Clang's C library `LibClang`. The core of this approach is the `visit` callback function but there's a tiny bit of boilerplate we need to get out of the way first. We start by creating an `index` which represents a set of translation units. Next we build a translation unit by parsing a given file, here the one specified as the first command-line argument `argv[1]`. The parsing function takes two parameters for an array of command line arguments that might be necessary for the file to compile, e.g. an include path or the specification of a sysroot. The `nullptr` and `0` argument indicate that we have no (zero) unsaved files in memory. With the final argument we tell the parser to skip function bodies in order to speed up parsing.

Once we have the translation unit we can visit each of the children in the AST with our `visitor` callback function. Using the `cursor` parameter we can check the kind of the visited node an extract information about its source location. We can ignore the two other parameters as we only need to consider the current node.

```
#include <clang-c/Index.h>
```

```
#include <iostream>

CXChildVisitResult visitor(CXCursor cursor, CXCursor, CXClientData) {
  CXCursorKind kind = clang_getCursorKind(cursor);

  // Consider functions and methods
  if (kind == CXCursorKind::CXCursor_FunctionDecl ||
      kind == CXCursorKind::CXCursor_CXXMethod) {
    auto cursorName = clang_getCursorDisplayName(cursor);

    // Print if function/method starts with doSomething
    auto cursorNameStr = std::string(clang_getCString(cursorName));
    if (cursorNameStr.find("doSomething") == 0) {
      // Get the source locatino
      CXSourceRange range = clang_getCursorExtent(cursor);
      CXSourceLocation location = clang_getRangeStart(range);

      CXFile file;
      unsigned line;
      unsigned column;
      clang_getFileLocation(location, &file, &line, &column, nullptr);

      auto fileName = clang_getFileName(file);

      std::cout << "Found call to " << clang_getCString(cursorName) << " at "
                << line << ":" << column << " in " << clang_getCString(fileName)
                << std::endl;

      clang_disposeString(fileName);
    }

    clang_disposeString(cursorName);
  }

  return CXChildVisit_Recurse;
}

int main(int argc, char **argv) {
  if (argc < 2) {
    return 1;
  }

  // Command line arguments required for parsing the TU
  constexpr const char *ARGUMENTS[] = {};

  // Create an index with excludeDeclsFromPCH = 1, displayDiagnostics = 0
```

```
  CXIndex index = clang_createIndex(1, 0);

  // Speed up parsing by skipping function bodies
  CXTranslationUnit translationUnit = clang_parseTranslationUnit(
      index, argv[1], ARGUMENTS, std::extent<decltype(ARGUMENTS)>::value,
      nullptr, 0, CXTranslationUnit_SkipFunctionBodies);

  // Visit all the nodes in the AST
  CXCursor cursor = clang_getTranslationUnitCursor(translationUnit);
  clang_visitChildren(cursor, visitor, 0);

  // Release memory
  clang_disposeTranslationUnit(translationUnit);
  clang_disposeIndex(index);

  return 0;
}
```

The code above illustrates that Clang's C interface makes walking the AST pretty straightforward. The high-level constructs together with its stable interface make `libclang` the approach of choice for many third-party applications interacting with Clang.

Building the Examples

To build the first two examples, you have two options. Either you build "in-tree" (with CMake) or you compile each file individually and link against Clang & LLVM. I will assume you built LLVM from source and installed it under `/opt/llvm`.

The former approach is how I got started initially, when I was writing this post. It is ver well documented in the tutorial on how to create a clang tool.

Alternatively, if you have Clang & LLVM installed, you can compile each example individually. Use `llvm-config` to specify the flags and object libaries to link against LLVM. This does not include the clang libraries, so don't forget to specify these explicitly.

```
clang++ example.cpp
        $(/opt/llvm/bin/llvm-config --cxxflags) \
        $(/opt/llvm/bin/llvm-config --ldflags --libs --system-libs) \
```

```
        -lclangAST \
        -lclangASTMatchers \
        -lclangAnalysis \
        -lclangBasic \
        -lclangDriver \
        -lclangEdit \
        -lclangFrontend \
        -lclangFrontendTool \
        -lclangLex \
        -lclangParse \
        -lclangSema \
        -lclangEdit \
        -lclangRewrite \
        -lclangRewriteFrontend \
        -lclangStaticAnalyzerFrontend \
        -lclangStaticAnalyzerCheckers \
        -lclangStaticAnalyzerCore \
        -lclangSerialization \
        -lclangToolingCore \
        -lclangTooling \
        -lclangFormat
```

Compiling the third example is more straightforward. As LibClang intends to be a sma
C API to Clang, it does not require you to link against LLVM. All you need to compile i
the command below:

```
clang++ example.cpp -std=c++11 -g -I/opt/llvm/include -L/opt/llvm/lib -lclang
```

Conclusion

When I started writing this blog post I had little to no knowledge of clang's internals. As part of LLVM it looked like a large and intimidating project. However, thanks to the availability of some excellent documentation, it was surprisingly easy to get an understanding of the basic concepts. I hope that the three examples can serve a starting point for anyone that want's to get dive into writing a Clang-based tool.

An heartfelt **thank you** to everyone that contributed to the LLVM project and its documentation!

Related

The latest episode of CppCast mentioned C++ Static Analysis using Clang which goes into more detail about using clang to build a static analyzer.

clang LLVM C++ AST compilers