Abstract Syntax Tree Visitation with Clang Tooling

23 October 2018

Martin Staley

CCS-7: Applied Computer Science

Introduction

As part of the development of our **FleCSI Static Analyzer**, we're gaining expertise in the capabilities of **Clang Tooling**.

FleCSI Static Analyzer is, at present, integrated into **Kitsune**. We'd like to split it out into its own application, however, as there's no fundamental reason for it to be tangled up with (or to tangle up) Kitsune.

I've put together a sort of **"Skeleton AST Visitor"** that I'll provide to the team as a **knowledge share**. ("Skeleton" because the code provides basic structure and control flow, not "skeleton" as relates to the upcoming Halloween.)

Goal: Provide a basic code with detailed comments, so that *you* can read it, understand it, and **make your own AST visitor**, if you wish, in minimal time.

Credit: Although it's largely rearranged and rewritten, this code originated from work done by Nick Moss.

Files

In one form or another (perhaps as just an emailed tarball), I'll provide y'all with at least the following files:

- **`visit.cc`** Complete AST Visitor code.
- **`runme`** Simple bash compilation script.
- **`one.hh`** Example input file.
- **`two.cc`** Another example input file.

Of course, you could pick any C++ file as example input.

The `visit.cc` code is only about 500 lines. About 300 are either comments or blank lines. The remaining ~200 lines are, we believe, lightweight and easy to understand. That's our goal here.

Compilation Script

Building `visit.cc` should be straightforward. I was able to do so on my office machine (Ubuntu Linux) with a simple bash script like this:

```
#!/bin/bash

export COMPILE=’
   g++ -pthread -fno-rtti -s’

export LIBRARIES=’
  -Wl,--start-group
  -lclangAnalysis
  -lclangAST
   ...
  -Wl,--end-group’

$COMPILE visit.cc -o visit $LIBRARIES
```

where “...” is, unfortunately, a list of about 70 libraries. (The usual linkage madness.) The list can probably be trimmed down a bit.

Compilation Script: Remarks

Clang is big. With **g++**, total compilation of `visit.cc` takes about **25 seconds** (10 compile, 15 link) on my modest office machine.

Clang is big. For me, the **-s** (strip symbols) option brought the executable's size down from about 450 MB (huge) to about **50 MB** (large).

On my machine, **clang++** worked in place of g++ with no other script changes. It's **much slower**, though: about 110 seconds for compilation.

You may need to fiddle with the **library list**. I tried unsuccessfully with scripts I found elsewhere, but eventually bit the bullet and included every library, in my `/usr/local/lib/`, that was related to `clang` or `llvm`. (I've trimmed my list down; it can probably be trimmed more.)

The **-fno-rtti** may not be necessary, but I kept getting link errors without it.

Code Outline

The `visit.cc` code is divided into the following sections.

- Documentation (yay!)
- Helper constructs (for printing; see remark below)
- class **`Visitor`**
- Definitions of Visitor's **`Visit*()`** functions
- class **`Consumer`**
- class **`Action`**
- class **`Factory`**
- class **`Database`**
- function **`visit()`**
- function **`main()`**

At present, our "example/FYI" code's purpose is simply to illustrate AST visitation by printing a transcript of what it does.

Classes

Our classes all derive from `clang` classes, and do as sketched below.

- **`class Visitor`** `:public clang::RecursiveASTVisitor<Visitor>`
  Encapsulates various `Visit*()` functions, for example `VisitTypeAliasDecl()`, that will be called by Clang's AST visitor when the requisite type of AST node is seen.
- **`class Consumer`** `:public clang::ASTConsumer`
  Makes a `Visitor` with which Clang will traverse the current translation unit's AST.
- **`class Action`** `:public clang::ASTFrontendAction`
  Gives Clang a Consumer, via its `CreateASTConsumer()` override.
- **`class Factory`** `:public clang::tooling::FrontendActionFactory`
  Gives Clang an Action, via its `create()` override.
- **`class Database`** `:public clang::tooling::CompilationDatabase`
  Contains a representation of "compilation steps" from which a series of translation units are produced.

Complexity Remark

A chain of events for visiting an AST involves the following sequence:

> Creating a **`Factory`**...
> which creates an **`Action`**...
> which creates a **`Consumer`**...
> which creates a **`Visitor`**...
> which contains callbacks for various types of AST nodes.

This seems, in our opinion, to be overly complicated. :-/

Factory's override looks like it can create just one Action, and Action's override looks like it can create just one Consumer. (Consumer can do multiple things.)

We don't know why someone can't just create a Consumer directly, and dispense with Factory and Action entirely. Or, perhaps this is somehow possible.

Control Flow

```
call main(argc,argv)
   call visit(argc,argv)
      make Factory
      make Database // Contains compilation commands
      make ArrayRef // Contains input file names
      make ClangTool(Database,ArrayRef)
      call ClangTool.run(Factory)
         for each file {
            // As determined by our Database's getCompileCommands(file) override...
            for each compile command { // Clang makes a translation unit for the given file
               get Factory's Action  // Via our create() override
               get Action's Consumer // Via our CreateASTConsumer() override
               call Consumer.HandleTranslationUnit() // <== Our override
                  make Visitor
                  call Visitor.TraverseDecl() // Applied to the translation unit
                     // Pursuant to what's in the AST, and
                     // to what Visit*() functions you provide...
                     Visitor . VisitCXXRecordDecl (...)
                     Visitor . VisitVarDecl (...)
                     Visitor . VisitCallExpr (...)
                     Visitor . VisitTypeAliasDecl (...)
                     Visitor . VisitContinueStmt (...)
                     // ...
            } // For each compile command
         } // For each file
```

visit()

```
bool visit(const int argc, const char *const *const argv)
{
   print("visit()");

   // Make a Factory
   Factory factory;

   // Make a Database
   Database db;

   // Make an ArrayRef<string> from the command-line arguments, which
   // we take to be the files we should examine with our AST visitor
   std::vector<std::string> vec;
   for (int a = 1;  a < argc;  ++a)
      vec.push_back(argv[a]);
   const clang::ArrayRef<std::string> files(vec);

   // Make a ClangTool, from the Database and the ArrayRef
   clang::tooling::ClangTool ctool(db,files);

   // Run the ClangTool, with the Factory
   const int status = ctool.run(&factory);
   printval(status);
   return status == 0;
}
```

Visitor

```
class Visitor : public clang::RecursiveASTVisitor< Visitor >
{
   // The following are initialized in the constructor. They aren't
   // actually used in our current example code; however, some or all
   // will be useful if you write real code in your Visit*() functions.
   clang::CompilerInstance &ci;
   clang::Sema             &sema;
   clang::ASTContext       &context;

public:

   Visitor(clang::CompilerInstance &);
  ~Visitor();

   // Visit*() functions.
   // For various types of AST nodes.
   // Note that these are *not* overrides.
   bool VisitCXXRecordDecl (const clang::CXXRecordDecl *const);
   bool VisitVarDecl       (const clang::VarDecl       *const);
   bool VisitCallExpr      (const clang::CallExpr      *const);
   bool VisitTypeAliasDecl (const clang::TypeAliasDecl *const);
   bool VisitContinueStmt  (const clang::ContinueStmt  *const);
   // ...
};
```

Consumer, Action, Factory

```
// Constructors and destructors are omitted, for brevity

class Consumer : public clang::ASTConsumer {
   clang::CompilerInstance &ci;
   void HandleTranslationUnit(clang::ASTContext &context) override
   {
      Visitor visitor(ci);
      visitor.TraverseDecl(context.getTranslationUnitDecl());
   }
};

class Action : public clang::ASTFrontendAction {
   std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
      clang::CompilerInstance &ci, llvm::StringRef file
   ) override {
      return std::unique_ptr<clang::ASTConsumer>(new Consumer(ci));
   }
};

class Factory : public clang::tooling::FrontendActionFactory {
   Action * create() override
   {
      return new Action();
   }
};
```

Database::getCompileCommands()

```
std::vector<clang::tooling::CompileCommand>
getCompileCommands(const llvm::StringRef file) const override
{
   print("Database::getCompileCommands()");
   printval(file.str());

   // vector<compilation commands>
   std::vector<clang::tooling::CompileCommand> commands;

   // A compilation command for the given file
   clang::tooling::CompileCommand c;
   c.Directory = ".";
   c.Filename  = file.str();
   c.CommandLine.push_back("clang++");
   c.CommandLine.push_back("-std=c++14");
   c.CommandLine.push_back("-DFOOBAR"); // if you wish
   c.CommandLine.push_back(file.str());
   commands.push_back(c);

   // Perhaps push additional compilation commands
   // ...

   return commands;
}
```

Example Input: one.hh

```
// VisitVarDecl (4)
int    i = 1;
int    j = 2;
float  f = 3.4;
double d = 5.6;

// foo, bar
#ifdef FOOBAR
   // VisitCXXRecordDecl
   struct foo
   {
   };

   // VisitCXXRecordDecl
   class bar
   {
   };
#endif
```

Example Input: two.cc

```
#include "one.hh"

// fun
int fun()
{
   // VisitVarDecl
   for (int i = 0;  i < 10;  ++i) {
      // VisitContinueStmt
      continue;
   }
   return 0;
}

// main
int main()
{
   // VisitTypeAliasDecl
   using Integer = int;

   // VisitVarDecl
   // VisitCallExpr
   int i = fun();
}
```