The most crucial aspect is line 11 in fig. 4.2. It specifies different "kinds" of animals. One enum value is used for the horse (`AK_Horse`) and another for the sheep (`AK_Sheep`). Hence, the base class has some knowledge about its descendants.

```
21 class Horse : public Animal {
22 public:
23   Horse(int S) : Animal(AK_Horse), Speed(S){};
24
25   static bool classof(const Animal *A) { return A->getKind() == AK_Horse; }
26
27   int getSpeed() { return Speed; }
28
29 private:
30   int Speed;
31 };
32
33 class Sheep : public Animal {
34 public:
35   Sheep(int WM) : Animal(AK_Sheep), WoolMass(WM){};
36
37   static bool classof(const Animal *A) { return A->getKind() == AK_Sheep; }
38
39   int getWoolMass() { return WoolMass; }
40
41 private:
42   int WoolMass;
43 };
```

Figure 4.3: `clangbook::Horse` and `clangbook::Sheep` classes

The implementations for the `clangbook::Horse` and `clangbook::Sheep` classes can be found in fig. 4.3. Lines 25 and 37 are particularly important as they contain the `classof` static method implementation. This method is crucial for the cast operators in LLVM. A typical implementation might look like the following (simplified version):

```
template <typename To, typename From>
bool isa(const From *Val) {
  return To::classof(Val);
}
```

Figure 4.4: Simplified implementation for `llvm::isa<>`

The same mechanism can be applied to other cast operators.

Our next topic will discuss various types of containers that serve as more powerful alternatives to their corresponding STL counterparts.

4.2.2 Containers

The LLVM ADT (Abstract Data Type) library offers a set of containers. While some of them are unique to LLVM, others can be considered as replacements for containers from the standard C++ library (STL). We will explore some of the most popular classes provided by the ADT.

4.2.2.1 String operations

The primary class for working with strings in the standard C++ library is `std::string`. Although this class was designed to be universal, it has some performance-related issues. A significant issue concerns the copy operation. Since copying strings is a common operation in compilers, LLVM introduced a specialized class, `llvm::StringRef`, that handles this operation efficiently without using extra memory. This class is comparable to `std::string_view` from C++17 [for Standardization, 2017] and `std::span` from C++20 [for Standardization, 2020].

The `llvm::StringRef` class maintains a reference to data, which doesn't need to be null-terminated like traditional C/C++ strings. It essentially holds a pointer to a data block and the block's size, making the object's effective size 16 bytes. Because `llvm::StringRef` retains a reference rather than the actual data, it must be constructed from an existing data source. This class can be instantiated from basic string objects such as `const char*`, `std::string`, and `std::string_view`. The default constructor creates an empty object.

```
#include "llvm/ADT/StringRef.h"
...
llvm::StringRef StrRef("Hello, LLVM!");
// Efficient substring, no allocations
llvm::StringRef SubStr = StrRef.substr(0, 5);

llvm::outs() << "Original StringRef: " << StrRef.str() << "\n";
llvm::outs() << "Substring: " << SubStr.str() << "\n";
```

Figure 4.5: `llvm::StringRef` usage example

Typical usage example for `llvm::StringRef` is shown in fig. 4.5. The output for the code is below:

```
Original StringRef: Hello, LLVM!
Substring: Hello
```

Another class used for string manipulation in LLVM is `llvm::Twine`, which is particularly useful when concatenating several objects into one. A typical usage example for the class is shown in fig. 4.6

```
    #include "llvm/ADT/Twine.h"
    ...
    llvm::StringRef Part1("Hello, ");
    llvm::StringRef Part2("Twine!");
    llvm::Twine Twine = Part1 + Part2;  // Efficient concatenation

    // Convert twine to a string (actual allocation happens here)
    std::string TwineStr = Twine.str();
    llvm::outs() << "Twine result: " << TwineStr << "\n";
```

Figure 4.6: `llvm::Twine` usage example

The output for the code is below:

```
Twine result: Hello, Twine!
```

Another class that is widely used for string manipulations is `llvm::SmallString<>`. It represents a string that is stack-allocated up to a fixed size, but can also grow beyond this size, at which point it heap-allocates memory. This is a blend between the space efficiency of stack allocation and the flexibility of heap allocation.

The advantage of `llvm::SmallString<>` is that for many scenarios, especially in compiler tasks, strings tend to be small and fit within the stack-allocated space. This avoids the overhead of dynamic memory allocation. But in situations where a larger string is required, llvm::SmallString can still accommodate by transitioning to heap memory. Typical usage example is show in fig. 4.7

```
#include "llvm/ADT/SmallString.h"
...
// Stack allocate space for up to 20 characters.
llvm::SmallString<20> SmallStr;

// No heap allocation happens here.
SmallStr = "Hello, ";
SmallStr += "LLVM!";

llvm::outs() << "SmallString result: " << SmallStr << "\n";
```

Figure 4.7: `llvm::SmallString<>` usage example

Despite the fact that string manipulation is key in compiler tasks like text parsing, LLVM has many other helper classes. We'll explore its sequential containers next.

4.2.2.2 Sequential containers

LLVM recommends some optimized replacements for arrays and vectors from the standard library. The most notable are:

- `llvm::ArrayRef<>`: A helper class designed for interfaces that accept a sequential list of elements for read-only access. The `llvm::ArrayRef<>` class is akin to `llvm::StringRef<>` in that it does not own the underlying data but merely references it.

- `llvm::SmallVector<>`: An optimized vector for cases with a small size. It resembles `llvm::SmallString`, as discussed in Section 4.2.2.1, String operations. Notably, the size for the array isn't fixed, allowing the number of stored elements to grow. If the number of elements stays below `N` (the template argument), then there is no need for additional memory allocation.

Let's examine the `llvm::SmallVector<>` to better understand these containers, as shown in fig. 4.8.

```
1  llvm::SmallVector<int, 10> SmallVector;
2  for (int i = 0; i < 10; i++) {
3    SmallVector.push_back(i);
4  }
5  SmallVector.push_back(10);
```

Figure 4.8: `llvm::SmallVector<>` usage

The vector is initialized at line 5 with a chosen size of 10 (indicated by the second template argument). The container offers an API similar to `std::vector<>`, using the familiar `push_back` method to add new elements, as seen in fig. 4.8, lines 3 and 5.

The first 10 elements are added to the vector without any additional memory allocation (see fig. 4.8, lines 2-4). However, when the eleventh element is added at line 5, the array's size surpasses the pre-allocated space for 10 elements, triggering additional memory allocation. This container design efficiently minimizes memory allocation for small objects while maintaining the flexibility to accommodate larger sizes when necessary.

4.2.2.3 Map like containers

The standard library provides several containers for storing key-value data. The most common ones are `std::map<>` for general-purpose maps and `std::unordered_map<>` for hash maps. LLVM offers additional alternatives to these standard containers:

- `llvm::StringMap<>`: A map that uses strings as keys. Typically, this is more performance-optimized than the standard associative container `std::unordered_map<std::string, T>`. It is frequently used in situations where string keys are dominant, and performance is critical, as one might expect in a compiler infrastructure like LLVM. Unlike many other data structures in LLVM, `llvm::StringMap<>` does not store a copy of the string key. Instead, it keeps a reference to the string data, so it's crucial to ensure the string data outlives the map to prevent undefined behavior.
- `llvm::DenseMap<>`: This map is designed to be more memory- and time-efficient than `std::unordered_map<>` in most situations, though it comes with some additional constraints (e.g., keys and values having trivial destructors). It's especially beneficial when you have simple key-value types and require high-performance lookups.
- `llvm::SmallDenseMap<>`: This map is akin to `llvm::DenseMap<>` but is optimized for instances where the map size is typically small. It allocates from the stack for small maps and only resorts to heap allocation when the map exceeds a predefined size.
- `llvm::MapVector<>`: This container retains the insertion order, akin to Python's OrderedDict. It is implemented as a blend of `std::vector` and either `llvm::DenseMap` or `llvm::SmallDenseMap`.

It's noteworthy that these containers utilize a quadratically-probed hash table mechanism. This method is effective for hash collision resolution because the cache isn't recomputed during element lookups. This is crucial for performance-critical applications, such as compilers.

4.2.3 Smart pointers

Different smart pointers can be found in LLVM code. The `std::unique_ptr<>` and `std::shared_ptr<>` are the most popular ones. In addition, LLVM provides some supplementary classes to work with smart pointers. One of the most prominent among them is `llvm::IntrusiveRefCntPtr<>`. This smart pointer is designed to work with objects that support intrusive reference counting. Unlike `std::shared_ptr`, which maintains its own control block to manage the reference count, `IntrusiveRefCntPtr` expects the object to maintain its own reference count. This design can be more memory-efficient. A typical usage example is shown below:

```
class MyClass : public llvm::RefCountedBase<MyClass> {
// ...
};

llvm::IntrusiveRefCntPtr<MyClass> Ptr = new MyClass();
```

Figure 4.9: `llvm::IntrusiveRefCntPtr<>` usage example

As we can see, the smart pointer prominently employs the CRTP (Curiously Recurring Template Pattern) that was mentioned earlier in Section 3.2, AST traversal. The CRTP is essential for the `Release` operation when the reference count drops to 0 and the object must be deleted. The implementation is as follows:

```
template <class Derived> class RefCountedBase {
  // ...
  void Release() const {
    assert(RefCount > 0 && "Reference count is already zero.");
    if (--RefCount == 0)
      delete static_cast<const Derived *>(this);
  }
}
```

Figure 4.10: CRTP usage in `llvm::RefCountedBase<>`. The code was sourced from the `llvm/ADT/IntrusiveRefCntPtr.h` header.

Since `MyClass` in fig. 4.9 is derived from `RefCountedBase`, we can perform a cast on it in line 6 of fig. 4.10. This cast is feasible since the type to cast is known, given that it is provided as a template parameter.

We just finished with LLVM basic libraries and there is a time to move to Clang basic libraries. Clang is a compiler frontend and the most important operation are related to diagnostics. The diagnostic requires precise information about position location in the source code. Lets explore basic classes that Clang provides for these operations.

4.3 Clang basic libraries

We have just finished discussing LLVM's basic libraries, and now it's time to move on to Clang's basic libraries. Clang is a compiler frontend, and its most crucial operations are related to diagnostics. Diagnostics require precise information about position locations in the source code. Let's explore the basic classes that Clang provides for these operations.

4.3.1 SourceManager and SourceLocation

Clang, as a compiler, operates with text files (programs), and locating a specific place in the program is one of the most frequently requested operations. Let's look at a typical Clang error report. Consider a program from the previous chapter, as seen in fig. 3.20. Clang produces the following error message for the program:

```
llvm-project/install/bin/clang -fsyntax-only maxerr.cpp
maxerr.cpp:3:12: error: use of undeclared identifier 'ab'
    return ab;
           ^
1 error generated.
```

As you can see, the following information is required to display the message:

- File name: in our case, it's `maxerr.cpp` .
- Line in the file: in our case, it's 3 .
- Column in the file: in our case, it's 12 .

The data structure that stores this information should be as compact as possible because the compiler uses it frequently. Clang stores the required information in the `clang::SourceLocation` object.

This object is used often, so it should be small in size and quick to copy. We can check the size of the object using lldb. For instance, if we run Clang under the debugger, we can determine the size as follows:

```
(lldb) p sizeof(clang::SourceLocation)
(unsigned long) 4
(lldb)
```

That is, the information is encoded using a single `unsigned long` number. How is this possible? The number merely serves as an identifier for a position in the text file. An additional class is required to correctly extract and represent this information, which is `clang::SourceManager`. The SourceManager object contains all the details about a specific location. In Clang, managing source locations can be challenging due to the presence of macros, includes, and other preprocessing directives. Consequently, there are several ways to interpret a given source location. The primary ones are:

- **Spelling Location**: Refers to the location where something was actually spelled out in the source. If you have a source location pointing inside a macro body, the spelling location will give you the location in the source code where the contents of the macro are defined.
- **Expansion Location**: Refers to where a macro gets expanded. If you have a source location pointing inside a macro body, the expansion location will give you the location in the source code where the macro was used (expanded).

Let's look at a specific example.

```
1 #define BAR void bar()
2 int foo(int x);
3 BAR;
```

Figure 4.11: Example program to test different types of source locations

In fig. 4.11, we define two functions: `int foo()` at line 2 and `void bar()` at line 3. For the first function, both the spelling and expansion locations point to line 2. However, for the second function, the spelling location is at line 1, while the expansion location is at line 3.

Let's examine this with a test clang tool. We will use the test project from Section 3.3, Recursive AST Visitor and will replace some parts of the code here. First of all, we have to pass `clang::ASTContext` to our `Visitor` implementation. This is required because `clang::ASTContext` provides access to `clang::SourceManager`. We will replace line 11 in fig. 3.5 and pass ASTContext as follows:

```
10 CreateASTConsumer(clang::CompilerInstance &CI, clang::StringRef File) {
11   return std::make_unique<Consumer>(&CI.getASTContext());
```

The `Consumer` class (see fig. 3.6) will accept the argument and use it as a parameter for `Visitor`:

```
8  Consumer(clang::ASTContext *Context)
9      : V(std::make_unique<Visitor>(Context)) {}
```

The main changes are for the `Visitor` class, which is mostly rewritten as follows:

```
class Visitor : public clang::RecursiveASTVisitor<Visitor> {
public:
  explicit Visitor(clang::ASTContext *C) : Context(C) {}

  bool VisitFunctionDecl(const clang::FunctionDecl *FD) {
    clang::SourceManager &SM = Context->getSourceManager();
    clang::SourceLocation Loc = FD->getLocation();
    clang::SourceLocation ExpLoc = SM.getExpansionLoc(Loc);
    clang::SourceLocation SpellLoc = SM.getSpellingLoc(Loc);
    llvm::StringRef ExpFileName = SM.getFilename(ExpLoc);
    llvm::StringRef SpellFileName = SM.getFilename(SpellLoc);
    unsigned SpellLine = SM.getSpellingLineNumber(SpellLoc);
    unsigned ExpLine = SM.getExpansionLineNumber(ExpLoc);
    llvm::outs() << "Spelling : " << FD->getName() << " at " << SpellFileName
                 << ":" << SpellLine << "\n";
    llvm::outs() << "Expansion : " << FD->getName() << " at " << ExpFileName
                 << ":" << ExpLine << "\n";
    return true;
  }

private:
  clang::ASTContext *Context;
};
```

Figure 4.12: Visitor class implementation

If we compile and run the code, we will get the following output:

```
Spelling : foo at functions.hpp:2
Expansion : foo at functions.hpp:2
Spelling : bar at functions.hpp:1
Expansion : bar at functions.hpp:3
```

The `clang::SourceLocation` and `clang::SourceManager` are very powerful classes. In combination with other classes such as `clang::SourceRange` (a pair of two source locations that specify the beginning and end of a source range), they provide a great foundation for diagnostics used in Clang.

4.3.2 Diagnostics support

Clang's diagnostics subsystem is responsible for generating and reporting warnings, errors, and other messages. The main classes involved are:

- `DiagnosticsEngine`: Manages diagnostic IDs and options.
- `DiagnosticConsumer`: Abstract base class for diagnostic consumers.
- `DiagnosticIDs`: Handles the mapping between diagnostic flags and internal IDs.
- `DiagnosticInfo`: Represents a single diagnostic.

Here is a simple example to illustrate how you might emit a warning in Clang:

```
#include "llvm/Support/raw_ostream.h"

int main() {
  llvm::IntrusiveRefCntPtr<clang::DiagnosticOptions> DiagnosticOptions =
      new clang::DiagnosticOptions();
  clang::TextDiagnosticPrinter TextDiagnosticPrinter(
      llvm::errs(), DiagnosticOptions.get(), false);

  llvm::IntrusiveRefCntPtr<clang::DiagnosticIDs> DiagIDs =
      new clang::DiagnosticIDs();
  clang::DiagnosticsEngine DiagnosticsEngine(DiagIDs, DiagnosticOptions,
                                             &TextDiagnosticPrinter, false);

  // Emit a warning
  DiagnosticsEngine.Report(DiagnosticsEngine.getCustomDiagID(
      clang::DiagnosticsEngine::Warning, "This is a custom warning."));

  return 0;
}
```

Figure 4.13: Clang diagnostics example

The code will produce the following output

```
warning: This is a custom warning.
```

In this example, we first set up the `DiagnosticsEngine` with a `TextDiagnosticPrinter` as its `DiagnosticConsumer`. We then use the `Report` method of the `DiagnosticsEngine` to emit a custom warning. We will add a more realistic example later when we create our test project for the clang plugin in Section 4.5, Clang plugin project.

4.4 LLVM supporting tools

The LLVM project has its own tooling support. The most important LLVM tools are TableGen and LIT (LLVM Integrated Tester). We will look into them with examples from Clang code. These examples should help us understand the purpose of the tooling and how they can be used.

4.4.1 TableGen

TableGen is a domain-specific language (DSL) and associated tool used in the LLVM project for the purpose of describing and generating tables, particularly those that describe a target architecture. This is highly useful for compiler infrastructure, where one frequently needs to describe things like instruction sets, registers, and various other target-specific attributes in a structured manner.

TableGen is employed in various parts of the Clang compiler. It's primarily used where there's a need to generate large amounts of similar code. For instance, it can be used for supporting cast operations that necessitate extensive enum declarations in basic classes, or in the diagnostic subsystem where code generation is required to handle numerous similar diagnostic messages. We will examine how TableGen functions within the diagnostics system as an example.

We will begin with the `Diagnostic.td` file, which describes Clang's diagnostics. This file can be found at `clang/include/clang/Basic/Diagnostic.td`. Let's examine how diagnostic severity is defined:

```
1 // Define the diagnostic severities.
2 class Severity<string N> {
3 string Name = N;
4 }
5 def SEV_Ignored : Severity<"Ignored">;
6 def SEV_Remark : Severity<"Remark">;
7 def SEV_Warning : Severity<"Warning">;
8 def SEV_Error : Severity<"Error">;
9 def SEV_Fatal : Severity<"Fatal">;
```

Figure 4.14: Clang diagnostics severity definition in Diagnostic.td

In fig. 4.14, we define a class for severities (lines 2-4). Each severity is associated with a string that describes it. Lines 5-9 contain definitions for the different severities; for instance, the `Warning` severity is defined on line 7.

The severity is later used to define the `Diagnostic` class, with the `Warning` diagnostic being defined as a descendant of this class:

```
1 // All diagnostics emitted by the compiler are an indirect subclass of this.
2 class Diagnostic<string summary, DiagClass DC, Severity defaultmapping> {
3   ...
4 }
5 ...
6 class Warning<string str>    : Diagnostic<str, CLASS_WARNING, SEV_Warning>;
```

Using the `Warning` class definition, different instances of the class can be defined. For example, below is an instance that defines an unused parameter warning located in `DiagnosticSemaKinds.td`:

```
1 def warn_unused_result : Warning<
2   "ignoring return value of function declared with %0 attribute">,
3   InGroup<UnusedResult>;
```

The tool `clang-tblgen` will generate the corresponding `DiagnosticSemaKinds.inc` file:

```
DIAG(warn_unused_result, CLASS_WARNING, (unsigned)diag::Severity::Warning,
  ↪  "ignoring return value of function declared with %0 attribute", 933,
  ↪  SFINAE_Suppress, false, false, true, false, 35)
```

This file retains all the necessary information about the diagnostic. This information can be retrieved from the Clang source code using different definitions of the `DIAG` macro.

For instance, the code below leverages the TableGen-generated code to extract diagnostic descriptions, as found in `clang/lib/Basic/DiagnosticIDs.cpp`:

```
  const StaticDiagInfoDescriptionStringTable StaticDiagInfoDescriptions = {
#define DIAG(ENUM, CLASS, DEFAULT_SEVERITY, DESC, GROUP, SFINAE, NOWERROR,
  ↪  \
               SHOWINSYSHEADER, SHOWINSYSMACRO, DEFERRABLE, CATEGORY)
  ↪  \
  DESC,
// clang-format off
...
#include "clang/Basic/DiagnosticSemaKinds.inc"
...
// clang-format on
#undef DIAG
};
```

The code will expand to

```
  const StaticDiagInfoDescriptionStringTable StaticDiagInfoDescriptions = {
    ...
    "ignoring return value of function declared with %0 attribute",
    ...
  };
```

The provided example demonstrates how TableGen can be used to generate code in Clang and how it can simplify Clang development. The diagnostic subsystem is not the only area where TableGen is utilized; it is also widely used in other parts of Clang. For instance, the macros used in various types of AST visitors also rely on the code generated by TableGen, see Section 3.2.2, Visitor implementation.

4.4.2 LLVM test framework

LLVM uses several testing frameworks for different types of testing. The primary ones are LIT (LLVM Integrated Tester) and Google Test (GTest). Both LIT and GTest play significant roles in Clang's testing infrastructure:

- LIT is primarily used for testing the behavior of the Clang toolchain as a whole, with a focus on its code compilation capabilities and the diagnostics it produces.
- GTest is utilized for unit tests, targeting specific components of the codebase, primarily utility libraries and internal data structures.

These tests are crucial for maintaining the quality and stability of the Clang project.

> *We will not delve into GTest, as this testing framework is commonly used outside LLVM and isn't part of LLVM itself. For more information about GTest, please visit its official page: https://github.com/google/googletest*

Our focus will be on the LLVM Integrated Tester, commonly referred to as LIT. LIT is LLVM's own test framework and is heavily used for testing the various tools and libraries in LLVM, including the Clang compiler. LIT is designed to be lightweight and is tailored for the needs of compiler testing. It's commonly used for running tests that are essentially shell scripts, often with checks for specific patterns in the output. A typical LIT test may consist of a source code file along with a set of "RUN" commands that specify how to compile, link, or otherwise process the file, and what output to expect.

The RUN commands often use FileCheck, another utility in the LLVM project, to check the output against expected patterns. In Clang, LIT tests are often used for testing frontend features like parsing, semantic analysis, code generation, diagnostics, etc. These tests typically look like source code files with embedded comments to indicate how to run the test and what to expect.

Consider the following example from `clang/test/Sema/attr-unknown.c`

```
1 // RUN: %clang_cc1 -fsyntax-only -verify -Wattributes %s
2
3 int x __attribute__((foobar)); // expected-warning {{unknown attribute
  ↪  'foobar' ignored}}
4 void z(void) __attribute__((bogusattr)); // expected-warning {{unknown
  ↪  attribute 'bogusattr' ignored}}
```

Figure 4.15: LIT test for clang warnings about unknown attributes

The example is a typical C source file that can be processed by Clang. LIT's behavior is controlled by comments within the source text. The first comment (on line 1) specifies how the test should be executed. As

indicated, `clang` should be started with some additional arguments: `-fsyntax-only` and `-verify`. There are also substitutions that begin with the '%' symbol. The most important of these is '%s', which is replaced by the source file's name. LIT will also examine comments beginning with `expected-warning` and ensure that the warnings produced by Clang's output match the expected values.

The test can be run as follows

```
$ ./build/bin/llvm-lit ./clang/test/Sema/attr-unknown.c
...
-- Testing: 1 tests, 1 workers --
PASS: Clang :: Sema/attr-unknown.c (1 of 1)

Testing Time: 0.06s
  Passed: 1
```

We run `llvm-lit` from the build folder because the tool is not included in the installation procedure. We can obtain more details about LIT setup and its invocation once we create our test clang plugin project and configure LIT tests for it.

4.5 Clang plugin project

The goal of the test project is to create a clang-plugin that will estimate class complexity. Specifically, a class is deemed complex if the number of its methods exceeds a certain threshold. We will leverage all the knowledge we have acquired thus far for this project. This will include the use of a recursive visitor and Clang diagnostics. Additionally, we will create a LIT test for our project. Developing the plugin will necessitate a unique build configuration for LLVM, which will be our initial step.

4.5.1 Environment setup

The plugin will be created as a shared object, and our LLVM installation should be built with support for shared libraries (see Section 1.3.1, Configuration with CMake):

```
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug -DCMAKE_INSTALL_PREFIX=../install
 ↪  -DLLVM_TARGETS_TO_BUILD="X86" -DLLVM_ENABLE_PROJECTS="clang"
 ↪  -DLLVM_USE_SPLIT_DWARF=ON -DBUILD_SHARED_LIBS=ON ../llvm
```

As can be seen, only the clang project is enabled.

The next step involves building and installing clang. This can be achieved with the following command:

```
$ ninja install
```

As soon as we are done with the clang build and installation, we can proceed with the `CMakeLists.txt` file for our project.

4.5.2 CMake build configuration for plugin

We will use fig. 3.13 as the foundation for our plugin build configuration. We will change the project name to `classchecker`, and `ClassComplexityChecker.cpp` will serve as our primary source file. The main portion of the file is displayed in fig. 4.16. As can be observed, we will construct a shared library (lines 18-20) rather than an executable, as in our previous test projects. Another modification is in line 12, where we set up a config parameter for the LLVM build folder. This parameter is necessary to locate the LIT executable, which is not included in the standard installation process, as mentioned earlier in Section 4.4.2, LLVM test framework. Some additional modifications need to be made to support LIT test invocations, but we will discuss the details later in Section 4.5.8, LIT tests for clang plugin (see fig. 4.25).

```
8  message(STATUS "$LLVM_HOME found: $ENV{LLVM_HOME}")
9  set(LLVM_HOME $ENV{LLVM_HOME} CACHE PATH "Root of LLVM installation")
10 set(LLVM_LIB ${LLVM_HOME}/lib)
11 set(LLVM_DIR ${LLVM_LIB}/cmake/llvm)
12 set(LLVM_BUILD $ENV{LLVM_BUILD} CACHE PATH "Root of LLVM build")
13 find_package(LLVM REQUIRED CONFIG)
14 include_directories(${LLVM_INCLUDE_DIRS})
15 link_directories(${LLVM_LIBRARY_DIRS})
16
17 # Add the plugin's shared library target
18 add_library(classchecker MODULE
19   ClassChecker.cpp
20 )
21 set_target_properties(classchecker PROPERTIES COMPILE_FLAGS "-fno-rtti")
22 target_link_libraries(classchecker
23   LLVMSupport
24   clangAST
25   clangBasic
26   clangFrontend
27   clangTooling
28 )
```

Figure 4.16: CMakeLists.txt file for class complexity plugin

After completing the build configuration, we can start writing the primary code for the plugin. The first component we'll create is a recursive visitor class named `ClassVisitor`.

4.5.3 Recursive visitor class

Our visitor class is located in the file `ClassVisitor.hpp` (see fig. 4.17). This is a recursive visitor that handles `clang::CXXRecordDecl`, which are the AST nodes for C++ class declarations.

We calculate the number of methods in lines 13-16 and emit diagnostics in lines 19-25 if the threshold is exceeded.

```
#include "clang/AST/ASTContext.h"
#include "clang/AST/RecursiveASTVisitor.h"

namespace clangbook {
namespace classchecker {
class ClassVisitor : public clang::RecursiveASTVisitor<ClassVisitor> {
public:
  explicit ClassVisitor(clang::ASTContext *C, int T)
      : Context(C), Threshold(T) {}

  bool VisitCXXRecordDecl(clang::CXXRecordDecl *Declaration) {
    if (Declaration->isThisDeclarationADefinition()) {
      int MethodCount = 0;
      for (const auto *M : Declaration->methods()) {
        MethodCount++;
      }

      if (MethodCount > Threshold) {
        clang::DiagnosticsEngine &D = Context->getDiagnostics();
        unsigned DiagID =
            D.getCustomDiagID(clang::DiagnosticsEngine::Warning,
                              "class %0 is too complex: method count = %1");
        clang::DiagnosticBuilder DiagBuilder =
            D.Report(Declaration->getLocation(), DiagID);
        DiagBuilder << Declaration->getName() << MethodCount;
      }
    }
    return true;
  }

private:
  clang::ASTContext *Context;
  int Threshold;
};
} // namespace classchecker
} // namespace clangbook
```

Figure 4.17: Source code for ClassVisitor.hpp

It's worth noting the diagnostic calls. The diagnostic message is constructed in lines 20-22. Our diagnostic message accepts two parameters: the class name and the number of methods for the class. These parameters are encoded with the placeholders '%1' and '%2' in line 22. The actual values for these parameters are passed in line 25, where the diagnostic message is constructed using the `DiagBuild` object. This object is an instance of the `clang::DiagnosticBuilder` class, which implements the Resource Acquisition Is Initialization (RAII) pattern. It emits the actual diagnostics upon its destruction.

> *In C++, the RAII principle is a common idiom used to manage resource lifetimes by tying them to the lifetime of an object. When an object goes out of scope, its destructor is automatically called, and this provides an opportunity to release the resource that the object holds.*

The `ClassVisitor` is created within an AST consumer class, which will be our next topic.

4.5.4 Plugin AST consumer class

The AST consumer class is implemented in `ClassConsumer.hpp` and represents the standard AST consumer, as seen in our AST visitor test projects (refer to fig. 3.6). The code is presented in fig. 4.19.

```
1  #include "clang/AST/ASTConsumer.h"
2
3  #include "ClassVisitor.hpp"
4
5  namespace clangbook {
6  namespace classchecker {
7  class ClassConsumer : public clang::ASTConsumer {
8  public:
9    explicit ClassConsumer(clang::ASTContext *Context, int Threshold)
10       : Visitor(Context, Threshold) {}
11
12   virtual void HandleTranslationUnit(clang::ASTContext &Context) {
13     Visitor.TraverseDecl(Context.getTranslationUnitDecl());
14   }
15
16 private:
17   ClassVisitor Visitor;
18 };
19 } // namespace classchecker
20 } // namespace clangbook
```

Figure 4.18: Source code for ClassConsumer.hpp

The consumer must be created from a special AST action class, which we will discuss next.

4.5.5 Plugin AST action class

The code for the AST action is shown in fig. 4.19. Several important parts can be observed:

- Line 7: We inherit our `ClassAction` from `clang::PluginASTAction`.
- Lines 10-13: We instantiate a `ClassConsumer` and utilize the `MethodCountThreshold`, which is derived from an optional plugin argument.
- Lines 15-25: We process the optional `threshold` argument for our plugin.

```cpp
#include "ClassConsumer.hpp"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendAction.h"

namespace clangbook {
namespace classchecker {
class ClassAction : public clang::PluginASTAction {
protected:
  std::unique_ptr<clang::ASTConsumer>
  CreateASTConsumer(clang::CompilerInstance &CI, llvm::StringRef) {
    return std::make_unique<ClassConsumer>(&CI.getASTContext(),
                                           MethodCountThreshold);
  }

  bool ParseArgs(const clang::CompilerInstance &CI,
                 const std::vector<std::string> &args) {
    for (const auto &arg : args) {
      if (arg.substr(0, 9) == "threshold") {
        auto valueStr = arg.substr(10); // Get the substring after
        ↪  "threshold="
        MethodCountThreshold = std::stoi(valueStr);
        return true;
      }
    }
    return true;
  }
  ActionType getActionType() { return AddAfterMainAction; }

private:
  int MethodCountThreshold = 5; // default value
};
} // namespace classchecker
} // namespace clangbook
```

Figure 4.19: Source code for ClassAction.hpp

We are almost done and ready to initialize our plugin.

4.5.6 Plugin code

```
#include "clang/Frontend/FrontendPluginRegistry.h"

#include "ClassAction.hpp"

static
    clang::FrontendPluginRegistry::Add<clangbook::classchecker::ClassAction>
    X("classchecker", "Checks the complexity of C++ classes");
```

Figure 4.20: Source code for ClassChecker.cpp

Our plugin registration is carried out in the `ClassChecker.cpp` file, shown in fig. 4.20. As we can observe, the majority of the initialization is hidden by helper classes, and we only need to pass our implementation to `lang::FrontendPluginRegistry::Add`.

Now we are ready to build and test our clang plugin.

4.5.7 Build and run plugin code

We need to specify a path to the installation folder for our llvm project. The rest of the procedure is the standard one that we have previously used.

```
export LLVM_HOME=<...>/llvm-project/install
mkdir build
cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug ..
ninja classchecker
```

The build artifacts will be located in the `build` folder. We can then run our plugin on a test file as follows:

```
$LLVM_HOME/bin/clang -fplugin=./build/libclasschecker.so -fsyntax-only <filepath>
```

where `<filepath>` is the file we want to compile. For example, if we use a test file named `test.cpp` that defines a class with 3 methods (see fig. 4.21), we will not receive any warnings.

```
class Simple {
public:
  void func1() {}
  void func2() {}
  void func3() {}
};
```

Figure 4.21: Test for the clang plugin

However, if we specify a smaller threshold, we will receive a warning for the file:

```
$LLVM_HOME/bin/clang -fplugin=./build/libclasschecker.so -fsyntax-only \
-fplugin-arg-classchecker-threshold=2 test.cpp
test.cpp:1:7: warning: class Simple is too complex: method count = 3
class Simple {
      ^
1 warning generated.
```

It's now time to create a LIT test for our plugin.

4.5.8 LIT tests for clang plugin

We'll begin with a description of the project organization. We'll adopt the common pattern used in the clang source code and place our tests in the `test` folder. This folder will contain the following files:

- `lit.site.cfg.py.in` : This is the main configuration file, a CMake config file. It replaces patterns marked as '@...@' with corresponding values defined during the CMake configuration. Additionally, this file loads `lit.cfg.py` .
- `lit.cfg.py` : This serves as the primary configuration file for LIT tests.
- `simple_test.cpp` : This is our LIT test file.

The basic workflow is as follows: CMake takes `lit.site.cfg.py.in` as a template and generates the corresponding `lit.site.cfg.py` in the `build/test` folder. This file is then utilized by LIT tests as a seed to execute the tests.

4.5.8.1 LIT config files

There are two configuration files for LIT tests. The first one is shown in fig. 4.22.

```
config.ClassComplexityChecker_obj_root = "@CMAKE_CURRENT_BINARY_DIR@"
config.ClassComplexityChecker_src_root = "@CMAKE_CURRENT_SOURCE_DIR@"
config.ClangBinary = "@LLVM_HOME@/bin/clang"
config.FileCheck = "@FILECHECK_COMMAND@"

lit_config.load_config(
        config, os.path.join(config.ClassComplexityChecker_src_root,
        ↪  "test/lit.cfg.py"))
```

Figure 4.22: lit.site.cfg.py.in file

This file is a CMake template that will be converted into a Python script. The most crucial part is shown in lines 6-7, where the main LIT config is loaded. It is sourced from the main source tree and is not copied to the `build` folder.

The subsequent configuration is displayed in fig. 4.23. It is a Python script containing the primary configuration for LIT tests.

```
# lit.cfg.py
import lit.formats

config.name = 'classchecker'
config.test_format = lit.formats.ShTest(True)
config.suffixes = ['.cpp']
config.test_source_root = os.path.dirname(__file__)

config.substitutions.append(('%clang-binary', config.ClangBinary))
config.substitutions.append(('%path-to-plugin',
  ↪  os.path.join(config.ClassComplexityChecker_obj_root,
  ↪  'libclasschecker.so')))
config.substitutions.append(('%file-check-binary', config.FileCheck))
```

Figure 4.23: lit.cfg.py file

Lines 4-7 define the fundamental configuration; for example, line 6 determines which files should be utilized for tests. All files with the '.cpp' extension in the `test` folder will be employed as LIT tests.

Lines 9-11 detail the substitutions that will be employed in the LIT tests. These include the path to the clang binary (line 9), the path to the shared library with the plugin (line 10), and the path to the `FileCheck` utility (line 11).

We have defined only one basic LIT test, `simple_test.cpp` , as shown in fig. 4.24.

```
1  // RUN: %clang-binary -fplugin=%path-to-plugin -fsyntax-only %s 2>&1 |
   ↪  %file-check-binary %s
2
3  class Simple {
4  public:
5    void func1() {}
6    void func2() {}
7  };
8
9  // CHECK: :[[@LINE+1]]:{{[0-9]+}}: warning: class Complex is too complex:
   ↪  method count = 6
10 class Complex {
11 public:
12   void func1() {}
13   void func2() {}
14   void func3() {}
15   void func4() {}
16   void func5() {}
17   void func6() {}
18 };
```

Figure 4.24: simple_test.cpp file

The use of substitutions can be observed in line 1, where paths to the clang binary, the plugin shared library, and the `FileCheck` utility are referenced. Special patterns recognized by the utility are used in line 9.

The final piece of the puzzle is the CMake configuration. This will set up the required variables for substitutions in `lit.site.cfg.py.in` and also define a custom target to run the LIT tests.

4.5.8.2 CMake configuration for LIT tests

The `CMakeLists.txt` file requires some adjustments to support LIT tests. The necessary changes are displayed in fig. 4.25.

```
31  find_program(LIT_COMMAND llvm-lit PATH ${LLVM_BUILD}/bin)
32  find_program(FILECHECK_COMMAND FileCheck ${LLVM_BUILD}/bin)
33  if(LIT_COMMAND AND FILECHECK_COMMAND)
34    message(STATUS "$LIT_COMMAND found: ${LIT_COMMAND}")
35    message(STATUS "$FILECHECK_COMMAND found: ${FILECHECK_COMMAND}")
36
37    # Point to our custom lit.cfg.py
38    set(LIT_CONFIG_FILE "${CMAKE_CURRENT_SOURCE_DIR}/test/lit.cfg.py")
39
40    # Configure lit.site.cfg.py using current settings
41    configure_file("${CMAKE_CURRENT_SOURCE_DIR}/test/lit.site.cfg.py.in"
42                   "${CMAKE_CURRENT_BINARY_DIR}/test/lit.site.cfg.py"
43                   @ONLY)
44
45    # Add a custom target to run tests with lit
46    add_custom_target(check-classchecker
47                      COMMAND ${LIT_COMMAND} -v
                          ↪  ${CMAKE_CURRENT_BINARY_DIR}/test
48                      COMMENT "Running lit tests for classchecker clang
                          ↪  plugin"
49                      USES_TERMINAL)
50  else()
51    message(FATAL_ERROR "It was not possible to find the LIT executables at
        ↪  ${LLVM_BUILD}/bin")
52  endif()
```

Figure 4.25: LIT tests configuration at CMakeLists.txt

In lines 31 and 32, we search for the necessary utilities `llvm-lit` and `FileCheck`. It's worth noting that they rely on the `${LLVM_BUILD}` environment variable, which we also verify in line 12 of the config (see fig. 4.16). The steps in lines 41-43 are essential for generating `lit.site.cfg.py` from the provided template file `lit.site.cfg.py.in`. Lastly, we establish a custom target to execute the LIT tests in

lines 46-49.

Now we are ready to start the LIT tests.

4.5.8.3 Run LIT tests

To initiate the lit tests, we must set an environment variable that points to the build folder, compile the project, and then execute the custom target `check-classchecker`. Here's how this can be done:

```
export LLVM_BUILD=<...>/llvm-project/build
export LLVM_HOME=<...>/llvm-project/install
rm -rf build
mkdir build
cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug ..
ninja classchecker
ninja check-classchecker
```

Upon executing these commands, you may observe the following output:

```
...
[2/2] Linking CXX shared module libclasschecker.so
[0/1] Running lit tests for classchecker clang plugin
-- Testing: 1 tests, 1 workers --
PASS: classchecker :: simple_test.cpp (1 of 1)

Testing Time: 0.12s
Passed: 1
```

With this, we conclude our first comprehensive project, which encompasses a practical clang plugin that can be tailored via supplemental plugin arguments. Additionally, it includes the respective tests that can be executed to verify its functionality.

4.6 Summary

In this chapter, we became familiar with the basic classes from the LLVM ADT library. We gained knowledge of clang diagnostics and the test frameworks used in LLVM for various types of testing. Using this knowledge, we created a simple clang plugin that detects complex classes and issues a warning about their complexity.

4.7 Further reading

- LLVM Coding Standards[Community, 2023b]: https://llvm.org/docs/CodingStandards.html
- LLVM Programmer's Manual[Community, 2023e]: https://llvm.org/docs/ProgrammersManual.html
- "Clang" CFE Internals Manual[Community, 2023b]: https://clang.llvm.org/docs/InternalsManual.html
- How to set up LLVM-style RTTI for your class hierarchy [Community, 2023f]: https://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html

Part 2: Clang tools

You can find some info about different clang tools here. We will start with linters that are based on clang-tidy, continue with some advanced code analysis techniques (CFG and live time analysis). The next chapter will be about different refactoring tools such as clang-format. The last chapter will be about IDE support. We are going to investigate how VSCode can be extended with language server provided by LLVM (clangd).

5

clang-tidy linter framework

There is an introduction to clang-tidy - the clang-based linter framework that uses AST to find anti-patterns in the C/C++/Objective-C code. First, we will start with a clang-tidy description, what kind of checks it has and how we can use them. Later we will investigate the clang-tidy architecture and how we can create our custom lint check.

5.1 clang-tidy overview and usage examples

TBD

5.2 clang-tidy internal design

TBD

5.3 Custom clang-tidy check

TBD

5.4 Results in the case of compilation errors

As we noticed in Section 3.6, Processing AST in the case of errors, compilation errors might produce strange results for lint checks and there is an example.

6

Advanced code analysis

he clang-tidy checks from the previous chapter can be considered based on an advanced matching provided by AST. However, it might not be enough when you want to detect some complex problems, such as lifetime issues. We will introduce advanced code analysis tools based on CFG (control flow graph). The clang static analyser is an excellent example of such devices, but we also have some CFG integration into clang-tidy. The chapter will start with typical usage examples and continue with implementation details. We will finish with our custom check that uses advanced techniques.

6.1 Usage cases

TBD

6.2 CFG and life time analysis

TBD

6.3 Custom CFG check

TBD

7

Refactoring tools

The chapter is about code refactoring tools. For example, suppose you want to modify a set of files according to a new code style or want to detect and fix a specific problem in your project. The code modification tools can help here. We will start with a typical usage scenario and finish with our custom code modification tool.

7.1 Code modification and clang-format

TBD

7.2 Custom code modification tool

Idea: we are going to replace public vars in a class with custom get/set methods.

TBD