

ONR Project

System Transparent Application Refactoring

Developer Manual

By Konstantin Rebrov <krebrov@mail.csuchico.edu>

Contents

• Introduction	Page 1
• Prerequisite Knowledge	Page 2
◦ C strings	Page 2
◦ X Macros	Page 3
◦ Inheritance and Polymorphism	Page 6
• Overview of Refactorings	Page 9
• Important parts of Clang/LLVM API to know	Page 9
• How to Write Refactorings	Page 14
◦ AST Based Refactorings	Page 15
◦ Static Analysis Based Refactorings	Page 22
• Structure of the software	Page 27
• Guidance for future development	Page 34
• Useful resources	Page 36
• Index	Page 37

Introduction

This STAR Tool (System of Transparent Application Refactoring) is a source to source **compiler tool** which translates C code representing various models from the form of a CPU-targeted program into the form of an FPGA-targeted program. What does the term **compiler tool** mean? It is simply a kind of software which uses the API of a compiler. We are all familiar with GCC, a proprietary C/C++ compiler. **Clang/LLVM** is also a C/C++ compiler, but unlike GCC, it is an open source compiler which has it's own API. If we want to give our applications, such as compiler tools, the power of modern compilers, we need to use the APIs of these compilers. Using an API means that we don't have to reinvent the wheel, we can use preexisting functionality. In the case of writing a compiler tool, it means that we don't have to go through the arduous process of lexing and parsing the C/C++ source code, instead devoting our time to algorithm development.

The downside is that we have to know the ins and outs of the API that we're using. In case of the Clang/LLVM API, there are a lot of details to know. Since I started this project, I spent a lot of time learning the details. Information about Clang/LLVM API is largely decentralized and scattered throughout the nooks and crannies of the internet. This document is an attempt to gather the knowledge into one place. It is a manual that I wrote, explaining the most relevant parts of Clang/LLVM API for this project.

Prerequisite Knowledge

In order to work on this project and be able to read my code, you need to have a solid foundation in C++ programming. Here I will mark a list of things that you should know well before going to work on this project. Later I will expand on some of these topics to provide my own important information. I wrote my own explanations about topics marked with a star * at the end.

The list of topics that you need to know for this project are although not an exhaustive list:

- Signed and unsigned data types
- Scope and lifetime
- Namespaces
- Pointers vs references
- Smart Pointers
- Auto variables
- Makefile
- Preprocessor, Compiler, and Linker
- Standard Algorithms and data structures
- C Strings*
- C++ Standard Library
- Conversion Constructors
- Inheritance and Polymorphism*
- X Macros*
- Exception Catching

C strings

Adjacent string literals, even on the different lines, are stored in the memory concatenated into one big C string. This is useful if you want to write a very large text message and it doesn't all fit on one line. The first string literal evaluates as a pointer to the implicit C string in the read only section of the program's memory.

Example:

```
const char* text = "hello " "privet "
                  "nihao";
```

text → hello privet nihao\0

X Macros

X Macros are a powerful technique for generic programming. If there are multiple statements, or groups of statements that are identical except for only some parts, then they can be combined together, removing the need to write unnecessary blocks of code.

They are used for generating arbitrary repetitive code constructs for shrinking source code size without compromising code's functionality. More generalized code is easier to maintain, easier to change, and more readable.

Just like using loops we introduce a level of abstraction allowing us to write more compact and generic code, so X Macros is an analogous level of abstraction. Unlike C++ templates, X Macros are more powerful in that they allow you to generalize variable names, C Strings, operators, data types and all kinds of code. C++ templates only allow you to generalize types. The main part of this system is the **definition list**, which contains as arguments any pieces of code that are then inserted into the macro's implementation. Usually the definition list is a collection of records that are related in some way, and these records are used in multiple places. If you need to modify, delete, or add new records, you don't need to go into all the different places where these records are used in the code. You just change the definition list.

```
#define DAYS_OF_WEEK \  
    X(monday) \  
    X(tuesday) \  
    X(wednesday) \  
    X(thursday) \  
    X(friday)
```

The last line does not have a \ at the end.

Then the second part of this system is the **implementation** of the X Macro itself, the location in the source code where the records of the definition list are to be inserted.

```
#define X(dayOfWeek) \  
    cout << #dayOfWeek << " = " << dayOfWeek << endl;  
    DAYS_OF_WEEK  
#undef X
```

The syntax of the implementation list is to be understood, not memorized.

<https://www.youtube.com/watch?v=0haKL2eR41A>

The preprocessor performs a series of transformations on this code. First the DAY_OF_WEEK is unfolded into what it is defined as.

```
#define X(dayOfWeek) \  
    cout << #dayOfWeek << " = " << dayOfWeek << endl;  
    X(monday)  
    X(tuesday)  
    X(wednesday)  
    X(thursday)  
    X(friday)  
#undef X
```

Then at each location where the X macro is called (lines 3-7) the argument is passed into the macro and a substitution is performed, replacing dayOfWeek with the actual argument, transforming the code as shown below. Lastly the X macro is undefined because it is common practice to redefine it as a later time.

```
cout << "monday" << " = " << monday << endl;  
cout << "tuesday" << " = " << tuesday<< endl;  
cout << "wednesday" << " = " << wednesday<< endl;  
cout << "thursday" << " = " << thursday<< endl;  
cout << "friday" << " = " << friday<< endl;
```

This approach allows arbitrary code constructs to be built, especially if they are highly repetitive.

There are some very important places in the code of this project where the X macros are used. This is the definition list:

```
// Define the list of tool specific command line options.
#define OPTIONS_LIST \
    X(RunRemoveMemcpy, "remove-memcpy", "This option turns on replacement of memcpy().") \
    X(RunRemoveMemset, "remove-memset", "This option turns on replacement of memset().") \
    X(RunMakeStatic, "make-static", "This option turns all dynamic memory allocations " \
        "into stack ones, gets rid of calloc() and free().") \
    X(RunRemovePointer, "remove-pointer", "This option turns on removal of the global pointer.") \
    X(RunRemoveHypot, "remove-hypot", "This option turns on replacement of hypot().") \
    X(RunRemoveVariables, "remove-variables", "This option removes unreferenced variables.") \
    X(RunRemoveAssignment, "remove-assignment", "This option removes unreferenced assignments.") \
    X(RunRemoveInitialize, "remove-initialize", "This option removes the initialize function.") \
    X(RunRemoveIndirectRecursion, "remove-indirect-recursion", "This option removes any indirec\
        "recursion in the step function by splitting it into major and minor step functions.")
```

It specifies all the tool specific **command line options**. The first field in each record is the name of the bool variable to be used in the code. The second field is the command line argument that the user types in the command prompt. The third field is the help message for that command. If you need to add/remove command line arguments, you don't change have to all the places in the code where those command line arguments are used. You just edit the definition list.

The first implementation declares a bunch of boolean variables that turn on various refactorings. For more information about them please read *Getting Started with LLVM Core Libraries* p. 273

```
// Options to turn on various refactorings are optional.
opt<bool> RunAll("all", desc("This options turns on all supported refactorings."));
#define X(VariableName, command_argument, description) \
opt<bool> VariableName(command_argument, desc(description));
OPTIONS_LIST
#undef X
```

The generated code will be something like:

```
opt<bool> RunRemoveMemcpy("remove-memcpy", desc("This option turns on\
replacement of memcpy()."));
```

The second implementation generates an extrahelp message. The text which is passed into the constructor defines the message which is printed when the user provides the command `refactoring_tool --help` at the command prompt. This is just a single string which is dynamically generated based on two programming techniques that you just learned. Those are automatic concatenation of adjacent string literals, and the power of X Macros to dynamically generate arbitrary code. Inside the string literals is an implementation of an X Macro, which constructs adjacent string literals based on the definition list given above. Finally, all those string literals are automatically concatenated and placed into the program's memory, and the completed single large string is passed into the constructor.

```

// Define an additional help message to be printed.
extrahelp CommonHelp(
    "\n"
    "  --" "debug" "\t" "This option enables diagnostic output." "\n"
    "  --" "all" "\t" "This options turns on all supported refactorings." "\n"
    #define X(VariableName, command_argument, description) \
    "  --" command_argument "\t" description "\n"
    OPTIONS_LIST
    #undef X
    "\n"
    "\nArguments above mentioned in [ ] are optional (not required).\n"
    "<build-path> should be specified if specific compiler options\n"
    "are not provided on the command line.\n"
);

```

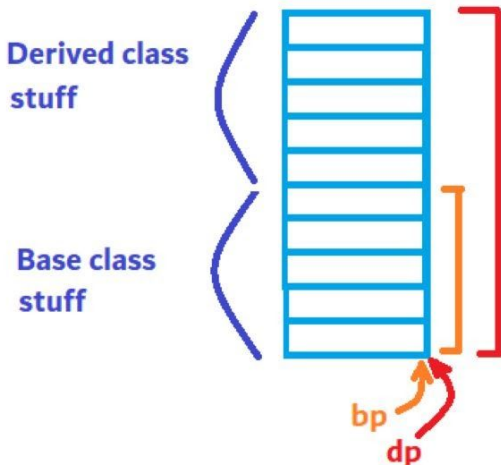
Inheritance and Polymorphism

Since Clang/LLVM API is an Object Oriented API, it uses Inheritance and Polymorphism very heavily. I assume that you already know about Inheritance and Polymorphism in Object Oriented C++ Programming, but I will anyway write my own information that I think would be especially helpful to know for this project.

My explanation of what `dynamic_cast` does and what is it for.

The pointer to the object points to it's first byte, the start of the object. Then we get the `sizeof(object)` bytes that actually comprise that object in memory.

The way inheritance works in C++ is that the Derived class literally builds on top of the Base class. The stuff of the Base class is at the start of the object at lower memory addresses near to where the pointer is pointing to. An instance of the Derived class has both the stuff in the Base class, and the Derived class stuff piled on top of it. We can say that the Derived class extends the Base class.



`Base* bp` is a pointer to an instance of the Base class. Although it may as well be pointing to a Derived object hiding in the memory, it doesn't know that by itself. The `bp` can only see `sizeof(Base)` bytes into the memory from its position, not any farther. It can only access the Base class stuff.

We need to `dynamic_cast` the `bp` to obtain a `Derived* dp`, which can see `sizeof(Derived)` bytes into the memory. So it can now get the Derived class stuff of the object which it points to.

So basically a `Base*` can point to an object of a Derived type but it can't use it fully. This means both member variables and methods. We need to `dynamic_cast` that pointer to a `Derived*`, then we can use that object as an instance of the Derived class type.

A dynamic cast on a pointer type:

- If it succeeds, it returns that dynamically casted pointer.
- If it fails, it returns `nullptr`

This is a common pattern in the code.

```
Base* bp;
if (Derived* dp = dynamic_cast<Derived*>(bp) ) {
    // use the Derived object to which dp points
} else { // bp points to a Base object
    // use the Base object to which bp points
}
```

If `bp` points to a Derived object, then the cast will initialize `dp` to point to the Derived object to which `bp` points. In that case it is safe for the code inside the `if` to use Derived operations. If the cast fails, then `dp` is set to `nullptr`. So the pointer `bp` does not point to a Derived object. In this case, the `else` clause does processing appropriate to Base instead.

It is worth noting that we defined `dp` inside the condition. By defining the variable in a condition, we do the cast and corresponding check as a single operation. Moreover, the pointer `dp` is not accessible outside the `if`. If the cast fails, the bad Derived pointer is not available for us to “accidentally” use it in subsequent code where we might forget to check whether the cast succeeded.

Performing a `dynamic_cast` in a condition ensures that the base and test of it's result are done in a single expression.

Here is another example:

```
//
// Iterate over all instructions within a BasicBlock
//
BasicBlock::iterator it;
BasicBlock::iterator ie;
for (it = BB->begin(), ie = BB->end(); it != ie; ++it) {
    Instruction* I = *it;
    if (BranchInst* BI = dyn_cast<BranchInst>(I)) {
        // Do something with branch instruction BI
    }
}
```

`dyn_cast` vs. `dynamic_cast` in C++

`dyn_cast` is part of the LLVM API, works just like `dynamic_cast`, however one difference is that the class doesn't require a v-table like `dynamic_cast`.

Actually it is `llvm::dyn_cast_or_null` which is equivalent to `dynamic_cast`.

These will produce a null pointer if passed a null pointer, whereas `llvm::dyn_cast` will bail.

LLVM Programmer's Manual

`dyn_cast` vs `isa`

`dyn_cast` returns a `Derived*` if the `Base*` is pointing to a Derived object, and `nullptr` otherwise. `isa` returns true or false instead. Use `dyn_cast` when you want to get a pointer to the Derived object, and use `isa` when you just want to check if the Base object is in fact a Derived object.

Overview of Refactorings

A **refactoring** is defined as some kind of transformation to the input code, which may be applied to multiple places. There are several concepts that we need to know in order to understand what refactorings are. Before performing any refactoring we need to know several things: what is the subset of the code text that we want to replace, and what do we want to replace it with. In order to know these things, we first need some way of analyzing the source code text.

There are two types of analyses that I have used so far in this project. The first kind of analyses are based on the structure of the source code text, finding certain coding constructs (such as for loops) and refactoring them. In order to do that, you need to compile the source code into an **Abstract Syntax Tree (AST)**, which is a data structure describing the actual structure of the source code. It's very similar to a Javascript DOM tree, which specifies the structure of a webpage. Similarly, an AST specifies a hierarchical structure of a source code, and that is how computers know what the source code looks like.

The second kind of analyses are based on the functionality of the code, based on the code's behavior during the running time. In order to do this you would need to run the Clang Static Analyzer over the code, and find code constructs that might generate bugs under certain specific paths of execution.

The first step in any refactoring is finding the source code text that we want to replace. We can either get this information from the AST or from the Static Analyzer. Getting this information from the AST entails constructing matchers that will search for a specific type of problematic coding construct. Getting this information from the Static Analyzer entails telling the Static Analyzer what checker to use when symbolically executing the code, in order to identify code constructs which violate the runtime constraints placed by that checker.

Important parts of Clang/LLVM API to know

Clang/LLVM is a very large API with a lot of classes and functions, and furthermore it's official documentation can be confusing to read. In order to save you time I have created a list of the most important classes and functions of Clang/LLVM API that you need to know. I have put an emphasis on specifically what you need to know for using these classes in your code. Any additional information you can obtain by reading the official documentation later. My explanations are more practical than theoretical.

There are two parts of the Clang/LLVM API. The first part is what I like to call "the actual API". It is just a collection of classes and functions that you can use to give your application the power of a modern compiler. These perform various actions. They are the interface from your code to the Clang Compiler. The second part is just a collection of classes which are "building blocks of AST". Unlike the first part, which is more like a "conventional" API that you use to give your code additional functionality (such as OpenCV, SDL, System Calls), this is just a bunch of node classes which make up the AST data structure. They can be thought of just classes as data types.

In this chapter I am writing only about the first part of Clang/LLVM API, which are facilities that you invoke for them to do something for you.

The important parts of Clang/LLVM API that you need to know in order to integrate the Compiler into your own code are:

- `CompilationDatabase`
- `RefactoringTool`
- `ClangTool`
- `FrontendAction`
- `FrontendActionFactory`

CompilationDatabase

Tools that use the C++ Abstract Syntax Tree need information how to parse a translation unit. In order to identify refactorings we first need to generate an AST equivalent to the input source code. In order to generate an AST we need to compile the input source code.

Compilers need to be passed options on the command line in order to parse the source code properly. The thing is that different options (such as `-std=`) can make the Compiler compile the code differently, generating potentially different AST. So the command line options directly affect the AST that we analyze in order to perform refactorings. For example:

```
-Wall -Werror -std=c++17 -O2 -g -D MACRO
```

Because the compiler needs to know which flags the code was compiled with, we need some way of passing this information to automatic source code transformation tools, otherwise we would not be able to compile the code correctly and generate the AST.

We use the class `CompilationDatabase` for this. The name implies that it is a “database” (a place where data is stored) of the “compilations” (instructions on how to compile the source code files). So this class basically contains inside of it all the information about the settings that you need in order to compile the code exactly as the user intended it.

`CommonOptionsParser` class is responsible for parsing the command line arguments and figuring out which options the user specified. In order to do that it needs to have known what the command line arguments are.

```
CommonOptionsParser OptionsParser(argc, argv, StarToolCategory);  
CompilationDatabase& Compilations = OptionsParser.getCompilations();
```

Then after `CommonOptionsParser` is done parsing the command line arguments, the `CompilationDatabase` can be retrieved from it.

RefactoringTool

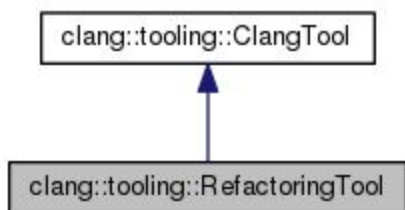
A **RefactoringTool** class is the core of this application. Everything stands on top of it. The RefactoringTool is what actually runs the Clang compiler, compiles your input code, generates an AST (Abstract Syntax Tree) for it, identifies locations in the source code that should be refactored, and then actually applies your refactorings, and writes the changes out to the file on disk. It does all of this seamlessly under the hood, so for the most part you don't need to know how it works. You don't need to worry about parsing and lexing the file, input and parsing, buffering, or file permissions.

The RefactoringTool can be thought of as an engine in a car. It is very powerful, and it makes the car go, but without any of the other parts, it is useless. That's why we have all of these other classes and components, to help the RefactoringTool do it's job.

NOTE: Sometimes an application (such as the STAR Tool) which uses Clang/LLVM APIs is called a Refactoring Tool. This is because such applications use the RefactoringTool as their "engine". Context will help you identify which one I mean.

It is worth noting that RefactoringTool class is for analyzing the source code on the level of it's AST, and applying AST-based refactorings. Anyway, the take away point is that RefactoringTool performs refactorings based on the structure of the source code only. If you want to perform refactorings based on other features of the source code, such as it's functionality (runtime state), then a slightly different method is used.

Another very important point is that RefactoringTool is a derived class from ClangTool. Whereas RefactoringTool is a refactoring specific version of ClangTool, specifically designed to apply refactorings to the source code, **ClangTool** is a more generic kind of tool. In this context a tool means a class which does something to the source code.



Because RefactoringTool is a derived class from ClangTool, it inherits all of ClangTool's public member functions, and additionally it has it's own ones on top of that as well. The most important functions from RefactoringTool to know are:

- The constructor
- `getReplacements()`
- `runAndSave()`

The constructor just takes the `CompilationDatabase` and a list of source files, so that it knows how to compile your code when it builds up the AST for it.

`getReplacements()` returns a pointer to a `std::map` of strings and `Replacements` objects, which should be passed into constructor of a call back class (more about this later).

`runAndSave()` compiles the input source code (using the compilations settings and source files passed into the constructor), creates the AST, applies all generated refactorings, and immediately saves the results into the disk. This function takes a `FrontendActionFactory ...` as a parameter (more about this later).

`RefactoringTool` also inherits some methods from its parent class the `ClangTool`, which can also be used.

ClangTool

`ClangTool` is the base class out of which `RefactoringTool` is derived. It is for performing more generic actions onto the input source code than just doing refactorings, which is what `RefactoringTool` does. `ClangTool` can be described as a utility to run a `FrontendAction` over a set of files.

`ClangTool`'s constructor also takes in a list of the `Compilations` and `SourcePaths`.

There are many methods that `ClangTool` has which can also be used by `RefactoringTool` (and you can read about them in your spare time), but I would like to make an emphasis on the method `run()`. This method runs an action over all files specified in the command line. It is `ClangTool`'s counterpart to `RefactoringTool`'s `runAndSave()` method. `run()` takes in a `ToolAction*` as a parameter, although a `FrontendActionFactory` can also be used. This is because `FrontendActionFactory` is derived from `ToolAction`. In my code of the STAR project I am using `ClangTool` as a way to invoke Clang's Static Analyzer programmatically. So in this case `run()` function runs the Static Analyzer over all files specified in the command line. It fills up the `StaticAnalysisDiagnosticConsumer` with the information about all occurrences of dead code assignments.

FrontendAction and FrontendActionFactory

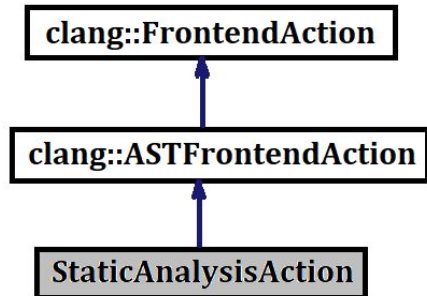
In order to understand what these two classes are and what is their relationship, we must first understand what is a factory design pattern. Basically it is a class whose sole purpose is to create new instances of another class. That class is called as a "factory", which creates "products" of another class.

`FrontendAction` is a "product" class of the "factory" class `FrontendActionFactory`. `FrontendAction` is an abstract base class for actions which can be performed by the frontend.

`ASTFrontendAction` is a class derived out of `FrontendAction` which is for performing AST consumer-based frontend actions. My class `StaticAnalysisAction`, which is used for

implementing invoking the Clang Static Analyzer programmatically (more on that later), is a class derived out of `ASTFrontendAction`.

The inheritance diagram is like this:

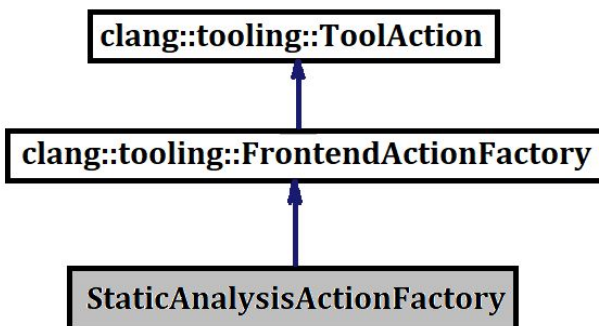


It is important to understand that a `FrontendAction` class performs an action on a single file. That action could be matching the AST of a file, or running the Static Analyzer over the file. We can change what this action is by creating a derived class inherited from that class.

The most important method of `FrontendAction` is `CreateASTConsumer()`. It creates and returns a new instance of `ASTConsumer` class. So `FrontendAction` class itself acts as a “factory” for the `ASTConsumer` class, which is the “product”. `CreateASTConsumer()` is overridden in the derived class `StaticAnalysisAction`.

The class `FrontendActionFactory` is the “factory” for the `FrontendAction` class. The `FrontendActionFactory` creates instances of class `FrontendAction` for each translation unit that there is in order to process it. The `FrontendActionFactory` is a class derived out of `ToolAction`, and my `StaticAnalysisActionFactory` is in turn derived out of `FrontendActionFactory`.

The inheritance diagram is like this:



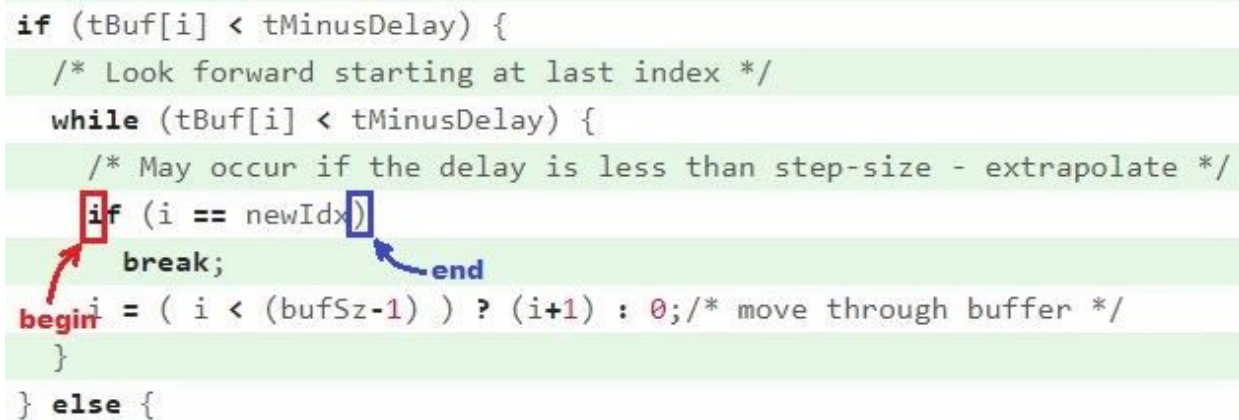
Important methods of `FrontendActionFactory` to know are:

- create()
- runInvocation()

create() function creates and returns a new instance of a FrontendAction. runInvocation() invokes the compiler with a FrontendAction created by create().

SourceLocation

The SourceLocation is exactly what it sounds like. It is an index (Location) of some text in the source code. It is a wrapper class around a pointer to a specific character in the source code text.



```

if (tBuf[i] < tMinusDelay) {
    /* Look forward starting at last index */
    while (tBuf[i] < tMinusDelay) {
        /* May occur if the delay is less than step-size - extrapolate */
        if (i == newIdx)
            break;
        i = ( i < (bufSz-1) ) ? (i+1) : 0; /* move through buffer */
    }
} else {

```

Every AST Node class has two methods defined getBeginLoc() and getEndLoc() returning the corresponding SourceLocations. These SourceLocations can span across multiple lines also. A newline is represented as a '\n' character in the input source code text.

The source code text is represented as one big long continuous array of characters. You can get the actual pointer into the char from the SourceLocation:

```
const char* loc = SM->getCharacterData(loc_start);
```

How to Write Refactorings

The STAR tool, as a High Level Synthesis Translation Tool, needs to identify all possible problematic pieces of code, construct functionally equivalent refactorings for that code, and then implement the refactorings into the file.

There are two kinds of refactorings that should be performed, resulting from two different kinds of code problems. The first kind of refactorings are those based on the Abstract Syntax Tree (AST) of the input source code. These refactorings do analysis of the structure of the code

constructs as they are seen at the time of compilation and their relationships with each other. AST based refactorings are supported natively by the Clang/LLVM API. There is an API, a collection of classes and functions provided specifically for performing these kinds of refactorings. I will be describing this already provided API in this document.

The second kind of refactorings are those based on Static Analysis of the input source code. These refactorings are more sophisticated in that they use the power of Clang's Static Analyzer to do their work. This is a process of symbolic execution of source code in order to find bugs during the running time by simulating all possible code execution paths that may be taken when the program is running. How exactly it does this is beyond the scope of this document. Instead I will talk about the code required to run the Static Analysis Based Refactorings. It is much more challenging to implement and perform these kinds of refactorings. The Clang API by itself does not support invoking the Static Analyzer in code, as it is a task which is not commonly done. In order to do that I created my own small API for achieving such tasks. So here I am describing my own API for invoking Clang Static Analyzer programmatically.

AST Based Refactorings

As briefly explained above, AST Based Refactorings utilize the RefactoringTool class to parse and lex the source code, and perform analyses on the generated Abstract Syntax Tree. In this section I will describe in more detail what code makes up an AST Based Refactoring.

The first step is to create an instance of RefactoringTool in order to run a single round of refactorings.

```
RefactoringTool tool(Compilations, SourcePaths);
```

In a single round of refactorings, all the input source code files are parsed, and an AST is built. Then the installed matchers are run over the AST, and whenever a certain kind of code construct is matched, a special handler function is called to deal with that match and create the corresponding refactoring for it. Then these refactorings are applied to the file, and all changes are saved into the disk. For more information, please read the chapter "Structure of the Application".

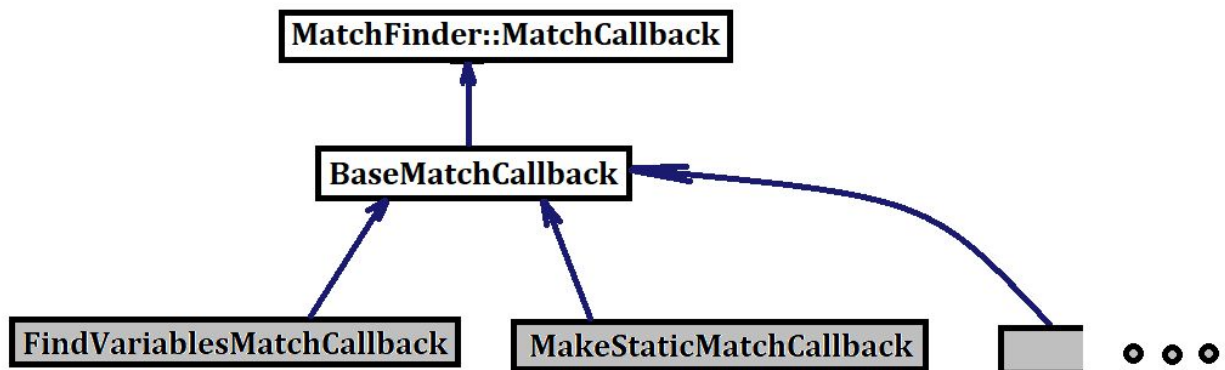
The next step is to create a MatchFinder.

A MatchFinder is simply a class whose job is just to find matches in the AST. In order to find certain code constructs inside of the input source files, we need to create matchers. We add the matchers to the MatchFinder, which will then use these matchers (things that do the matching) to find matches (or matched code constructs).

Then we need to create a Callback class. Just as it sounds like, this is a class which implements the Callback design pattern. This design pattern means that a collection of your

created code, such as function or class, is passed as an argument to the library's code. That library code is expected to *call back* (execute) the argument at a given time.

MatchFinder::MatchCallback is the base class of all the Callback classes. All the Callback classes are indirectly inherited from it, through the class BaseMatchCallback, which should all your custom Callback classes should be directly derived from.



A MatchCallback is a relatively simple class. It has one useful method,

```
virtual void run (const MatchResult &Result)=0
```

The Callback class adds matchers to the MatchFinder, and whenever a certain piece of code gets matched, it triggers the matcher, and that piece of code is sent to the run() method of the corresponding Callback class which created that activated matcher. The matched code is sent wrapped inside a container class called MatchResult.

Every time a match is found, the MatchFinder will invoke the registered MatchCallback with a MatchResult containing information about the match. That is why it is called a Callback class, because it is a wrapper class for a function or method that gets *called back* whenever a line of code gets matched in the input AST.

Looking more closely at the signature of run() function, we see that there is `virtual` before it and `=0` after it. This means that the method is a pure virtual function. A pure virtual function does not have an implementation. It defines an interface only (a description of what the function should be like). Any class that inherits from the MatchCallback absolutely has to provide it's own implementation of the run() function according to that interface. An additional side effect is that a class with a pure virtual function cannot be instantiated. You cannot create objects of a class with pure virtual function, not having an implementation. So you cannot create instances of MatchCallback class, and any class that inherits from it must provide an implementation of run() function in order to be instantiated.

For more information about Interfaces in C++ (Pure Virtual Functions), please watch this short video:

<https://www.youtube.com/watch?v=UWAdd13EfM8>

Although you should not have your custom Callback class be inherited from MatchCallback directly. It should be inherited from the class BaseMatchCallback, which I created. This class has some useful features that facilitate development of AST based refactorings.

Creating the Callback class

The Callback classes are responsible for

- Adding source code text matchers into the MatchFinder.
- Handling matched code constructs that get called back into their run() functions.
- Creating source code text replacements and adding them to the Replacements vector of the RefactoringTool.

The simplest Callback class used in the project is RemoveHypotMatchCallback. We will be studying it as an example of what any Callback class should do. Please open that file right now and look at the code as you are reading the book.

The first important thing here is the constructor.

```
explicit RemoveHypotMatchCallback(map<string, Replacements> * replacements)
    : BaseMatchCallback(), replacements(replacements) {}
```

After the `:` there is an initializer list. First you should initialize the base class always! Then initialize the replacements variable. It is just a `std::map<string, Replacements>*` *pointer to a C++ map data structure*, which can be thought of as a list of source code text replacements.

The actual data structure that the pointer is pointing to, is owned by the RefactoringTool, because the RefactoringTool takes each replacement, and applies it to the input source code. A pointer to this list is given to the CalBack class through it's constructor. The Callback class adds replacements to the list. That is how RefactoringTool knows what replacements to apply into the input files.

The next important method that all Callback classes should have is getASTmatchers(). The MatchFinder is passed in by a non-const reference, so that it can add matches to the original MatchFinder in the main() function.

```
void RemoveHypotMatchCallback::getASTmatchers(MatchFinder& mf)
{
    StatementMatcher hypot_matcher = callExpr(
```

```

        callee(functionDecl(hasName("hypot")))
    ).bind("hypot_call");

    mf.addMatcher(hypot_matcher, this);
}

```

This method constructs one or more either `StatementMatcher` or `DeclarationMatcher` specifying what source code text to match in the AST. Each matcher is made by building up a hierarchy of the AST matcher classes. It is a form of declarative programming paradigm. The most important point is the `.bind("name")` at the end. This name is used to uniquely identify the instance of the matched source code text among other matches. For more information about how to construct matchers I recommend the text `AST_Matcher_Reference.pdf` or you can just open any Callback class I wrote and study how I constructed the matchers.

We add matchers to the `MatchFinder` using the member method `addMatcher()`. The matcher is the first parameter. The second parameter is a pointer to the `CallBack` class which will refactor the code that is matched by that particular matcher.

When a matched code construct is detected, it will *call back* the `run()` method of the `CallBack` class, passing the result in as a parameter.

The first step is to get the `SourceManager` from the result. The `SourceManager`, as it's name implies, is the manager of the source code text. All interactions into the source code text including reading and writing the source code text happen through the `SourceManager`.

```
SM = result.SourceManager;
```

Then the AST Node representing the matched statement is gotten from the result.

```

if (const CallExpr* hypot_call =
    result.Nodes.getNodeAs<CallExpr>("hypot_call")) {

```

The assignment inside the `if ()` is intentional. The right hand side will return a valid pointer to the matched node in the AST, if the current matched result is a `CallExpr` with the name "hypot_call". It will return a `nullptr` otherwise. It asks, was a `CallExpr` named "hypot_call" matched? This is how you determine what was the code construct matched by the `MatchFinder`. `MatchFinder` returns you a result, it could be anything, and you use such chains of `if ()` statements to find out what was the matched code construct. It is very important to have the name be the same as the `.bind("name")`, and also the template data type and the pointer data type should be the same as the data type that you are matching in the `getASTmatchers()`.

NOTE: Each particular Callback class will only receive results from the matchers that it defined and passed to the MatchFinder by itself. It will not receive results from matchers defined in other Callback classes, so it is safe to re-use names.

After determining the kind of matched code construct in the run() method, you obtain a pointer to that node in the AST. Nodes in the AST generated by Clang are immutable. Because you cannot modify them, you must get string representations of that matched code construct, manipulate the strings for creating replacements, and then add those replacements as strings to the `map<string, Replacements>* replacements;`

There is no specific algorithm for constructing the replacements. You have to create your own heuristics for identifying the different parts of the AST node that you want to refactor, and construct appropriate expressions.

The first step to creating such heuristics is identifying different parts of the AST node. Clang's AST classes have multiple member methods to extract subexpressions out of them. AST nodes representing larger code constructs, such as if () statements, have methods returning pointers to AST nodes representing their condition, if block, and else block, for example. We just have to read the Doxygen documentation for the Clang's AST classes. In this example, the 0th (first) argument of the CallExpr is returned.

```
// Get the first argument, which is named x in the man page,
// both as an expression and as a string.
const Expr* x = hypot_call->getArg(0);
SourceLocation string_x_begin = x->getBeginLoc();
SourceLocation string_x_end = getCharOffsetLoc(string_x_begin, ',', true);
string string_x = getAsString(string_x_begin, string_x_end);
```

Then you have to get a string representation of that AST element. The BaseMatchCallback class defines a set of functions that make it easier for you to get string representations of AST elements:

- `string getAsString(const astElement* element)`
Returns a string representation of the given AST node.
- `string getAsString(const SourceLocation& loc_start, const SourceLocation& loc_end)`
Returns a string representation of the text in between these two locations in the source code.
- `void outputSource(const astElement* element, raw_ostream& output, string extraString = "")`
Prints the source code text of the given AST node to the provided output stream (`llvm::outs()` or `llvm::errs()`), appending an optional extra string at the end.

- `void outputSource(const SourceLocation& loc_start, const SourceLocation& loc_end, raw_ostream& output, string extraString = "")`
Prints the text in between these two locations in the source code to the provided output stream (`llvm::outs()` or `llvm::errs()`), appending an optional extra string at the end.
- `SourceLocation getCharOffsetLoc(const SourceLocation& loc_start, char character, bool forwards)`
Returns the `SourceLocation` of a character which is offset either forwards or backwards from the given `SourceLocation`.

After obtaining strings for the desired subexpressions, we construct a replacement string. It is literally just a string that will be overwritten into the input source code by the automated refactorings.

```
string replacement = "sqrt( (" + string_x + ") * (" + string_x +
    ") + (" + string_y + ") * (" + string_y + ") )";
```

In order to implement this string replacement, we need to identify the range of characters in the input source code text which should be replaced, and construct a `Replacement` object, giving it that character range and the string replacement. That character range doesn't have to be the same length as the string replacement. The LLVM API automatically scales it. Providing an empty string as the replacement, deletes the source code text highlighted by the character range from the file.

```
SourceLocation startLoc = hypot_call->getBeginLoc();
SourceLocation endLoc = hypot_call->getEndLoc();
CharSourceRange range = CharSourceRange::getTokenRange(startLoc,
    endLoc);
Replacement hypot_call_rep(*SM, range, replacement);
```

After creating the `Replacement` object, it should be added to the. That expressions if () statement's condition tries to perform the addition. It is important to know that this expression returns false if it added the replacement object successfully, and true if an error occurred. So inside the if() condition you should put an error message letting the user know what happened.

If the replacement was successful, then a notification message should be printed on the screen depending on if the user enabled debugging output. Also the number of individual refactorings of that kind should be incremented for displaying in the final output.

```

Replacement hypot_call_rep(*SM, range, replacement);
if (Error err = (*replacements)[hypot_call_rep.getFilePath()].add(hypot_call_rep)) {
    outputSource(hypot_call, errs());
    errs() << "ERROR: Error adding replacement that replaces hypot() with a manual ca
    errs() << "\n\n";
    return;
}
if (print_debug_output) {
    outputSource(hypot_call, outs());
    outs() << "replaced with:\n" << replacement << '\n';
    outs() << "\n\n";
}
++num_hypot_replacements;

```

This code can be copied and pasted into new Callback classes. The only thing that needs to be changed is the name of the Replacement. It is worth noting that the name of the replacement should be in these two places in the if () statement.

Going back to the main() function.

After adding the matchers to the MatchFinder, we use it to create a new FrontendActionFactory which you then pass into the RefactoringTool::runAndSave() to run those matchers that we put into the MatchFinder, over the AST of the translation unit.

```

MatchFinder mfl;
tool1.runAndSave(newFrontendActionFactory(&mfl).get());

```

This RefactoringTool::runAndSave() is the actual library code mentioned above which runs the MatchFinder over the AST, and whenever a matching code construct is detected, it automatically calls the run() method of the corresponding Callback class, which constructs a replacement for it, and adds it into the Replacements vector of the RefactoringTool.

```

Callback_class callback(refactoring_tool.getReplacements());

```

That code causes the replacements to be applied on the input source code, and saved onto the disk (it performs the replacements in place and saves the file).

The order of the matches is a pre-order traversal of the AST, and applying the matchers in order they were added to the MatchFinder.

newFrontendActionFactory returns a std::unique_ptr, and calling .get() returns the actual FrontendActionFactory* this is a pointer to a dynamic object. RefactoringTool::runAndSave()

takes a `FrontendActionFactory*` as a parameter. The `std::unique_ptr` ensures that the dynamic object will be properly deleted as soon as it goes out of scope.

Static Analysis Based Refactorings

As I mentioned above, I am using `ClangTool` as a way to invoke Clang's Static Analyzer programmatically. `ClangTool` runs a `FrontendAction` over a set of files, and that action is to run the Static Analyzer over these files. There are several steps in this process:

1. Setup the necessary settings and boilerplate code in order to run the Static Analyzer.
2. Run the Static Analyzer over the given input files, and as you do this, keep track of all the problematic code constructs that were detected, as a pair of `SourceLocations` delimiting the start and end positions of that buggy code. So for each buggy code detected, you need to remember it's position into some kind of data structure.
3. After you have completely analyzed the source code file, you will have a complete set of locations of the buggy code in the file. So you then identify those locations in the source code text, and then you perform replacements and/or removals to fix the bugs.

The data structure that is used to store information about locations of buggy code detected is:

```
vector< pair<SourceLocation, SourceLocation> > SourcePairs;
```

A vector of pairs of `SourceLocations`. Each element in the vector has the starting position and the ending position of that problematic code construct so that later we can refactor it, knowing where it's at in the source code text, and replace it with new working code.

Recall from my description of AST Based Refactorings above, that you need a `Callback` class in order to run the refactorings and replace the problematic source code text with new code. Because that `Callback` class performs the replacements, it needs to know the starting and ending `SourceLocations` of the code to replace. Therefore, it makes more sense to have this vector of pairs of `SourceLocations` data structure to be a data member of this `Callback` class.

This class is called `RemoveAssignmentMatchCallback`, because the task that it performs is removing assignments of dead code, assignments that will never be referenced in the code again, making them redundant. However, this API code structure is generic enough that it can be implemented to take advantage of any Static Analysis based refactoring. I will explain you how the system works so you can create your own.

After creating the object, we call `RemoveAssignmentMatchCallback::getVector()`, which returns a (modifiable) reference to the `SourcePairs` vector of that class! A reference can be thought of as just another name for the original variable that it refers to, and it is frequently used to "extend" the scope of the original object, making it accessible in places where normally you would not. By virtue of passing in a reference to that vector into the constructor of the

`StaticAnalysisActionFactory` class, that factory class now has editable access to that same vector. Any changes to the reference affect the original object that it refers to!

```

//// Remove Assignment details
// This CallBack class gets the SourceLocations from the
// StaticAnalysisDiagnosticConsumer, and applies the replacements.
RemoveAssignmentMatchCallback
analysis_match_callback(&tool2.getReplacements());

...

// Create a custom ActionFactory.
StaticAnalysisActionFactory Factory(analysis_match_callback.getVector());

```

That reference to the vector is then saved as a private data member of the `StaticAnalysisActionFactory` class.

Recall that `StaticAnalysisActionFactory` is derived from `FrontendActionFactory`. It creates a new `StaticAnalysisAction` for each translation unit in order to process it. This reference to the vector is passed onto the `StaticAnalysisAction` objects when they are created.

```

inline FrontendAction *create() override
{ return new StaticAnalysisAction(SourcePairs); }

```

And of course, the `StaticAnalysisAction` class also has a data member that is a reference to this vector. Instances of the `StaticAnalysisAction` get created by the `StaticAnalysisActionFactory`. Each such instance is responsible for processing a single translation unit. That is important since the `StaticAnalysisAction` is responsible for configuring the Static Analyzer to run through that translation unit.

`StaticAnalysisAction` has several important methods. `getCheckersControlList()` is especially important since it is the main configuration of the Static Analyzer. This function returns a vector identifying by name which checker the Static Analyzer should run over the translation unit.

```

CheckersList StaticAnalysisAction::getCheckersControlList() {
    CheckersList List;
    List.push_back( make_pair("deadcode.DeadStores", true) );
    return List;
}

```

Different checkers check for different kinds of bugs. It is worth noting that it has to be one of the checkers that is currently installed into the code of Clang/LLVM. Since this particular refactoring is to remove “dead” unreferenced assignments, the name of the checker you want to use is

“deadcode.DeadStores”. It is one of the checkers that you get by default in Clang. This is the full list of default Clang checkers:

<https://clang.llvm.org/docs/analyzer/checkers.html>

Static Analysis is useful for finding path sensitive bugs. If you want to check for a specific kind of path sensitive bug, you can implement your own custom checker. Chapter 9 of Getting Started with LLVM Core Libraries tells you how to do that. So you can check for and refactor specific path sensitive bugs.

Anyway, `StaticAnalysisAction::getCheckersControlList()` returns a list of checkers that you want to run. Although you can put as many checkers as you want, I would recommend you to put only one or several related checkers in there because all of the problematic code constructs found by the checkers will eventually be refactored by the same `CallBack` class.

It is important to remember that `StaticAnalysisAction` class is itself a “factory” class. It creates `StaticAnalysisDiagnosticConsumer` classes which actually collect the `SourceLocations` of the problematic code constructs.

`StaticAnalysisAction::CreateASTConsumer()`, it creates and returns a `StaticAnalysisDiagnosticConsumer` to collect diagnostics from the Static Analyzer. It does not create and return that object directly though, there is some boilerplate code associated with that, which is beyond the scope of this book. The important things to remember is that this function creates a `StaticAnalysisDiagnosticConsumer`, and it sets the list of checkers to run by calling `getCheckersControlList()` within.

Because the class `StaticAnalysisAction` creates objects of type `StaticAnalysisDiagnosticConsumer`, it needs to have a reference to the vector `SourcePairs` as a private data member, in order to pass the vector onto that class. This reference to the vector is temporarily stored as a data member of the class, and when a `StaticAnalysisDiagnosticConsumer` is created, the reference to the vector is transferred to it.

The name `StaticAnalysisDiagnosticConsumer` is easy to understand. It is just a consumer of diagnostics emitted by the Static Analysis. A diagnostic in simple words is basically just a `SourceLocation` of the line with the buggy code, and a description of it. So `StaticAnalysisDiagnosticConsumer` ultimately gets the `SourceLocations` and saves them into the vector.

The way `StaticAnalysisDiagnosticConsumer` does this is it calls the function `FlushDiagnosticsImp()`. This is a function inherited from the base class `PathDiagnosticConsumer`. It is a call back method that gets called automatically when the Clang Static Analyzer has finished collecting all the Diagnostics. This method gets called at the end of running the Static Analysis. This method loops through all the diagnostics, and for each valid diagnostic, it saves the starting and ending `SourceLocations` of that diagnostic as a pair in

the vector reference `SourcePairs`. In fact the `SourceLocations` of the diagnosed codes are saved into the vector `SourcePairs` which is in the class `RemoveAssignmentMatchCallback`! In this way we can transfer information from the Static Analyzer into the `CallBack` class in order to perform the refactorings on these codes.

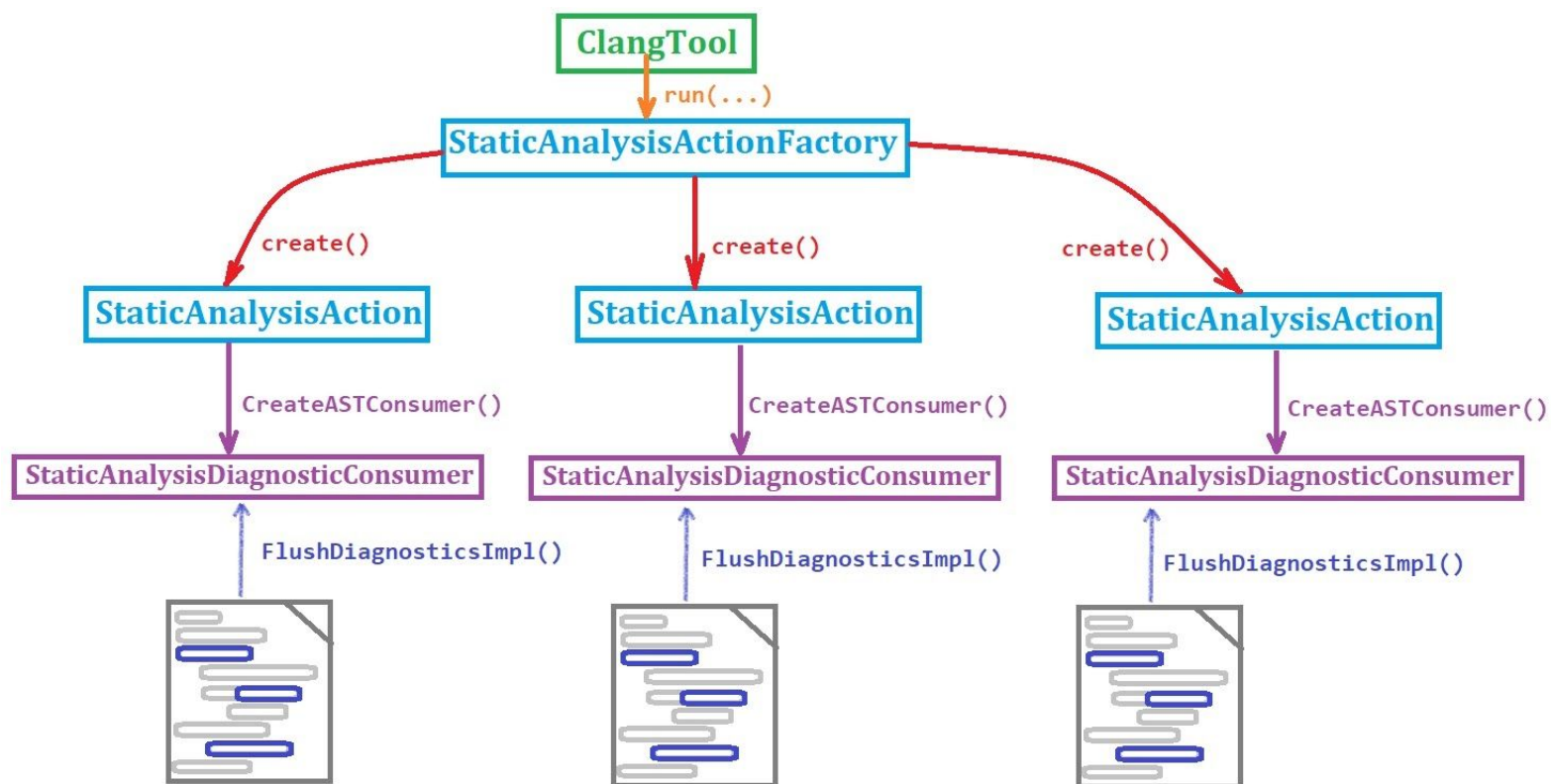
In summary, my `StaticAnalysisActionFactory` class is a parameter to `ClangTool::run()`, which runs that action over each translation unit.

```
ClangTool Tool(Compilations, SourcePaths);  
StaticAnalysisActionFactory Factory(...);  
Int result = Tool.run(&Factory);
```

This runs the Static Analyzer over all files specified in the command line (like `RefactoringTool`). The Static Analyzer internally keeps track of diagnosed code which violates the constraints of the checker(s) used.

The `StaticAnalysisActionFactory` creates instances of `StaticAnalysisAction`, one for each translation unit that there is in order to process it.

The `StaticAnalysisAction` in turn creates instances of class `StaticAnalysisDiagnosticConsumer` in order to facilitate collecting diagnostics from the Static Analyzer. After all the diagnostics have been collected by the Static Analyzer, then it internally calls the method `StaticAnalysisDiagnosticConsumer::FlushDiagnosticsImpl()`, which saves the `SourceLocations` of all the diagnostics into the vector `SourcePairs` in the `CallBack` class.



The Callback class, RemoveAssignmentMatchCallback, is not just for removing assignments only. It removes pretty much any source code text whose starting and ending SourceLocations have been saved as pairs into the SourcePairs vector. This Callback class could easily be extended to handle removals of other source code texts besides just assignments. The getASTmatchers() method for this class is interesting in that it matches the translationUnitDecl, which represents the entire translation unit. This means that its run() method gets called only once, upon which it loops through the SourcePairs vectors and deletes all the source code snippets.

Adding checkers for other path-sensitive buggy coding constructs, modify the method StaticAnalysisAction::getCheckersControlList() and push_back() more pairs to the list. The string is the name of the desired checker you want the Static Analyzer to run over the source code and true means you want to enable it.

All the other infrastructure is already in place so both my created Static Analysis API and the RemoveAssignmentMatchCallback should successfully delete the coding constructs matched by the new checkers, without any modification to their codes. If changes need to be made, they would not be very large ones.

Structure of the Application

In this chapter I will describe the structure of the application, how the code of the STAR tool is organized so that you understand what is where. I highly recommend that you read this chapter together with the doxygen documentation for the STAR tool. Then reading that will make a lot more sense.

The file with the `main()` function is called `refactoring_tool.cpp`. All the other files are supporting files of this project. They contain the API of the STAR tool that I created. This is an internal API, used by the `main()` function to offload it's low level details of tasks, so that we don't end up with enormous unreadable lines of code in the `main()` function.

My API of the STAR tool is structured in an object oriented format, just like Clang/LLVM's API. Each class has it's own *.h file with the interface, and it's *.cpp file with the implementation. If a class is a small one, then it can just be put into a *.h file, like a struct.

In order to understand the STAR tool we must understand both the `refactoring_tool.cpp` file containing the `main()` function, and the classes of the internal API that it uses. We can get an idea of what these classes are by going to the doxygen documentation for the STAR tool, and taking a look at the **Classes ▼** tab, either the Class List or the Class Hierarchy. There are several different kinds of classes:

- Callback classes, including the `BaseMatchCallback`, from which all the other Callback classes are inherited, and the derived classes themselves.
- Classes making up the Static Analysis API, described in the chapter “Static Analysis Based Refactorings”. They are:
 - `StaticAnalysisActionFactory`
 - `StaticAnalysisAction`
 - `StaticAnalysisDiagnosticConsumer`
 - `StaticAnalysisASTConsumer`
- Custom exception classes, which are only used in Callback classes that use exception handling, such as `RemoveMemcpyMatchCallback` and `RemoveMemsetMatchCallback`.

All these classes have descriptions about what they do in the doxygen documentation for the STAR tool. Every class needs to have it's own specific purpose, which is generally very well documented. Having God classes which are responsible for multiple independent functionalities is generally recognized as a bad software design practice in C++.

Moving on to the `refactoring_tool.cpp` file. I will first provide a quick overview before diving into the details. At the top of the file I have the header files and using statements. I like to put more project specific header files above the rest. Then header files for Clang/LLVM APIs and finally the C/C++ standard APIs. I also like to not using namespaces but instead use only the names that are actually in the code. This approach does not use everything that is included in the header files (to avoid possible naming collisions), and also you do not need to put the

namespace qualifier everywhere in your code. Please watch the following video and it will make more sense.

[Why I don't "using namespace std"](#)

Directly below that is code which specifies the valid command line arguments.

After that is the main() function of this application. It is often said that in order to understand how an application works, you start with the main() function. In professional software engineering, a main() function should specify the control flow of the application at a high level, using other functions and/or classes for the low level details of implementation.

The main() function of the STAR tool does three things:

1. It sets up the facilities to parse the command line arguments and actually parses them, determining which refactorings the user wants to run.
2. Then it actually does it's main task (pun intended) of the application, which is identifying code constructs in the input source code, and applying transformations on them. This procedure is separated into six rounds of refactorings, which I will go over later in this chapter.
3. Then at the end, information about the performed transformations is printed to the user, and the program exits.

In trying to understand the code we start from the top and make our way down.

One of the good things about Clang/LLVM API is that it has a relatively easy to use declarative style of programming specification and parsing of command line options. You can certainly read argc and argv in order to tell what command line arguments the user specified, but sometimes using an API to parse the command line arguments is more convenient. There are multiple different APIs for parsing command line arguments, such as getopt_long(), however Clang/LLVM has it's own way of parsing command line arguments which is straight forward. It is a separate API from the rest of Clang/LLVM APIs, so you can use it for both compiler tools and regular applications alike. To learn more about Clang/LLVM's command line arguments API, please read the document `llvm_CommandLine.pdf`

Directly below the using statements there are definitions of the command line options. The first line applies a custom category to the command-line options. It associates all the following command line options with the command to run this application from the terminal. The category tells the CommonOptionsParser how to parse the argc and argv.

```
OptionCategory StarToolCategory("refactoring_tool options");
```

The next several lines define the list of valid command line options that you can provide to the executable. They are specified as global variables of the opt template class. These global

variables act both as a specification of what valid command line arguments are accepted by the executable, as well as outputs to store the values of these command line arguments that the user typed.

After the command line options get parsed by the Clang/LLVM API, it automatically sets the global variables to the values entered by the user for the corresponding command line arguments. It scans the command line, and if it sees any arguments whose name matches, it gets the value of that argument from the argv, and stores it into the global variable according to its data type.

In the <> you specify the data type of that variable. And that is also the data type of the value that the user has to enter in the command line. For example, <bool> Says that this option has no extra value associated with it, and is to be treated as a bool value only. It is just a flag command line option. If this option is set, then its corresponding global variable becomes true, otherwise it becomes false.

```
wc --lines
```

Any other data types such as <int> or <string> will demand the user to specify a value for that argument, and they will set the global variable's value to whatever the user typed.

For more information the implementation of command line options in LLVM please read *Getting Started with LLVM Core Libraries* p. 273

I am using X Macros to define the list of tool specific command line options. In order to understand how this code works you must first read the chapter "Prerequisite Knowledge". This code creates a bunch of bool global variables that can be later referenced to find out if the user enabled that specific refactoring in the command line or not.

After specifying the typed command line options, there is an untyped command line option which is the extrahelp. Sometimes the user wants to just print the usage information.

The text that is passed into the constructor defines the message which is printed when the user provides the command `refactoring_tool --help` at the command prompt.

All these facilities must be above the main() function. Inside the main() function it calls a SetVersionPrinter() function, which sets the action to be taken when the user provides the command `refactoring_tool --version` at the command prompt. The difference between the extrahelp is that SetVersionPrinter() takes in a C++11 lambda function as the VersionPrinterTy func parameter. So you can have any code that does stuff in that lambda function, but for now it just prints the version information.

After that there a CommonOptionsParser is created, which takes the argc, argv, and the option category of the global variables specifying the command line arguments, and in its constructor it parses the command line and it sets the global variables to the values that the user provided.

CommonOptionsParser has to be constructed before any of the global variables can be used. All of the global variables have to be above the CommonOptionsParser constructor call, otherwise it will fail to parse the options provided by the user, and the refactoring_tool executable will fail to recognize that command line options.

Then the CompilationDatabase is created, which gets the compilations from the OptionsParser. This is described in details in the chapter “Important parts of Clang/LLVM API to know.” The SourcePaths vector is also initialized from the OptionsParser. That is how we know the name of the file(s) that are being processed by the application.

After that there is some code to enable all refactorings if RunAll is turned on. You don’t need to change this code, it automatically works even if you add new refactorings. Please see the chapter “Guidance for future development” about how to add a new refactoring to the application.

```
// If the user specified --all option,
// then all refactorings should be enabled.
if (RunAll) {
    #define X(VariableName, command_argument, description) \
    VariableName = true;
    OPTIONS_LIST
    #undef X
}
```

Directly below this initialization of settings and setting up boilerplate code, we have the actual meat of the application, which are the rounds of refactorings. We already know what a refactoring is, and how to do it. This was covered in the chapter “How to do Refactorings”. In that chapter we discussed how to perform an individual refactoring, either an AST based refactoring, or a Static Analysis based refactoring. Now in a compiler application we want to perform multiple refactorings onto a given source code file. The way that the STAR tool organizes multiple refactorings is by using what I call “rounds of refactorings”.

In a single round of refactorings, all the input source code files are parsed, and an AST is built. Then the installed matchers are run over the AST, and whenever a certain kind of matched code construct is matched, a special handler function is called to deal with that match and create the corresponding source code replacement for it. Then these replacements are applied to the file, and all changes are saved into the disk.

This process of creating an AST for the file, matching coding constructs against that AST, generating and applying appropriate refactorings, and saving the file to disk, is a process that is repeated every single round that we have. Indeed, every time you set up the RefactoringTool with the necessary information and call the runAndSave() method on it, this process of steps happens. This is precisely what distinguishes a round of refactoring

In the STAR tool, you can see that there are several layers of such refactorings rounds. Of course, because running the RefactoringTool and all tasks that it entails (parsing the code, creating an AST, matching coding constructs) is a computationally intense deed, ideally we want to have as few single distinct rounds as possible. And a new RefactoringTool is created each round of the refactorings to apply transformations in that round.

An important observation is that we can combine multiple individual refactorings which are handled by different CallBack classes, to be run by the same RefactoringTool. That way we can refactor multiple code constructs “in parallel” within the same round of refactorings. This feature for example is used extensively in the fourth round of refactorings, when multiple individual refactorings are run by the tool4. However, this technique can only be used for refactorings that do not have dependencies or interactions on each other. In order to identify such refactorings, you need to ask yourself the question, “do these two (or more) refactorings touch the same pieces of code?” If for example two individual refactorings touch different pieces of code, then there are no interactions between them, and hence they can run within that same round.

What happens if two individual refactorings, that are run in the same round, touch the same pieces of code? Then they have interactions between themselves. Actually, the order that individual refactorings touch the same pieces of code is well defined.

According to the official doxygen documentation for the MatchFinder class, it says:

The order of matches is guaranteed to be equivalent to doing a pre-order traversal on the AST, and applying the matchers in the order in which they were added to the [MatchFinder](#).

The order of matches is guaranteed to be equivalent to doing a pre-order traversal on the AST, meaning that the AST will be processed from top to bottom, and the parent nodes representing the more generic code constructs such as a function definition are matched before small statements are matched. Refactorings are applied in the order that their matchers were added to the MatchFinder.

So if you have two refactorings that are matched against the same pieces of code, the refactoring whose matchers are added to the MatchFinder first, will have its run() method be called first. After it's done, then the other refactoring's run() method will get called. Keep in mind that you cannot necessarily rely on this property of the MatchFinder to have refactorings that touch the same code run in sequence. Suppose the matchers of refactoring A were added to the MatchFinder first, then the matchers of refactoring B were added to the MatchFinder. Theoretically the refactoring A's run() method will be called first. It will see the AST elements of the matched code constructs, create satisfactory refactorings for them, and apply the refactoring, and then it will be done. After that refactoring B's run() method would get called, because its matchers were added to the MatchFinder second. Keep in mind that the AST had not changed since then. So refactoring B will see the AST as

it was at the time before refactoring A ran, but refactoring A has now already created source code replacements behind it's back. So there is a possibility that refactoring A's source code replacements would be applied before those of refactoring B. I just don't know for sure does the RefactoringTool immediately apply the Replacements when they are added, or does it aggregate all the source code Replacements by calling the run() methods for all the refactoring CallBack classes, and only apply the actual Replacements at the end?

According to the official doxygen documentation for RefactoringTool::runAndSave(), it says:

"Call `run()`, apply all generated replacements, and immediately save the results to disk."

This statement is vague and it could be interpreted differently. Depending on how the RefactoringTool is internally implemented, we could have refactorings which touch the same pieces of code be applied to that code in different orders. So this is an example of undesired implementation defined behavior, which we as application programmers just don't know about. So I would recommend a good rule of thumb, to only run refactorings in the same round that do not touch the same pieces of code in order to avoid potential race conditions! There are so many problems that could result when you try to run refactorings in the same round that touch the same pieces of code.

However, very often just one refactoring by itself can be not enough to accomplish a particular source code transformation. In that case we would need to have multiple refactorings executing sequentially in a known order, working together in order to perform a particular source code transformation.

Indeed, this is the main principle of the layered structure of the STAR tool, with it's multiple rounds of refactorings. The first round of refactorings passes over the input source code, performs it's replacements, and saves them out to the file. Then the next round of refactorings does it's work, and so on. This principle is similar to having long lines of Linux terminal commands, which are themselves made up of individual commands.

```
cat logfile.txt | awk -F "-" '{print $1 $3}' | wc -c
```

This is an example of a shell command. You don't need to know what it does. In fact it does nothing in particular. The main point is that the cat utility processes some input and it generates some output. That output is then piped into the input of the awk utility, which then does some processing on that input and generates it's own output, which is then piped into the input of the wc utility. Here we see multiple utilities that are not tied together by design, that merely work together as an assembly line. This is an analogy to how the rounds of refactorings work. One refactoring generates output (saving changes into a file), which is then processed by another refactoring in the next round.

Whenever you have multiple refactorings that have interactions or dependencies between themselves, especially if they touch the same code, or if the output of one refactoring is to be processed by the input of a future refactoring, you should always have a code organization structure

having multiple rounds of refactorings instead of putting them in the same round. Having multiple rounds of refactorings you no longer have to worry about the order in which the individual refactorings will apply their source code replacements, because the order is defined by you. Obviously a particular round of refactoring has to be completed and saved before the next round of refactoring starts. Callback classes in the future rounds of refactorings will be able to see the changes implemented in past refactorings directly in the AST, which means that they can implement other source code replacements that directly build on top of these transformations.

These rounds of refactorings are very modular. This means that additional refactorings can be added to existing rounds, or more rounds of refactorings can be added to the chain depending on future requirements and capabilities.

For example, the first round of refactorings is for removing an indirect recursion bug. It does this by stripping out the recursive function and creating an entirely new function that gets called instead. Because this refactoring generates entirely new code that might also have some possible coding constructs to be refactored by later modules, it has to be in the first round.

The second round runs the Static Analysis and the refactorings that are dependent on it. Because these refactorings will get rid of any redundant code, or code that is not executed during the running time of the program and is to be removed, it is advantageous to have these refactorings be executed before the general AST-based refactorings. Static Analysis based refactorings will remove some code, leaving less work for future rounds of refactorings. The Static Analysis based refactorings also have to be before the first round, and any refactorings that create new code, because the generated code also should be analyzed and refactored.

The third round is for refactorings which remove significant large blocks of code, which might have problematic code constructs that the future refactorings need to remove. So run these refactorings first, and save the changes back out to the file, and only then after that run the other refactorings in a different round so that these removed blocks of code do not show up in matching the AST. Among these third round refactorings are removal of the initialize function and separating step functions.

The fourth round is for simple refactorings that modify only the line on which they reside. These are localized refactorings which do not need to take into account the source code file as a whole. Because these refactorings are rather simple, modify only local parts of the source code, and do not have any interactions among themselves, they can all be run in the same round.

The fifth and sixth rounds of refactorings identify and delete any variables in the input source code that were left unused by the above refactorings. The fifth round finds these variables and the sixth round removes them. They have to be split up into two rounds necessarily because FindVariables keeps a list of all variable declarations and a list of all variable uses. At the end of running the round, both lists are filled and they are compared to identify the unused variables. That's why we cannot just remove a variable from the source code as soon as we see it. No, we have to look through the entire source code to identify possible uses of that variable. Only after

doing that can we say if that variable is unused. And of course, another round is required to actually run over the AST again and remove the unused variable declarations.

After running all the rounds of refactorings, the STAR application should print details of it's execution to the user if the debug flag is set. It displays information for each individual refactoring that was run, as well as a total sum the number of refactorings that were run by it upon a particular model or input source file. While being run over the AST, the Callback classes keep internal counters to accumulate the number of individual transformations that they did. They have getter methods to get these numbers and print them to the screen in this final part of the application.

Guidance for future development

This chapter describes how to add and integrate a new Refactoring Module into the STAR Application.

First point is: all CallbackClasses have to be inherited from BaseMatchCallback because then it will have access to utility functions and also for consistency with the other Callback classes. You don't have to include the "Callback" at the end of the class's name. I think it is concise without that. In the past I used to include that suffix, for example RemoveHypotMatchCallback, but later I stopped doing that. For example my newest Callback class, CreateMinorStepFunction, is without that suffix. You can edit all the Callback classes and remove the "Callback" suffix from them. Be sure to modify the declarations, definitions, Doxygen comments, instantiations of the classes in the main() function, the Makefile, and also this book to reflect the changed names.

Each Callback class should have this line in the *.cpp file to know if the user enabled diagnostic output to be printed.

```
extern bool print_debug_output; // defined in refactoring_tool.cpp
```

These are the changes that you need to make in refactoring_tool.cpp in order to integrate a new Callback class into the project.

First you need to create a new command line option for running that module. More than one Callback class can be functionally in a single module.

You need to add a new line. The first parameter is the name of the bool variable that will be used in the code to keep track of whether the user enabled that refactoring or not. The second parameter is the actual command line option that the user types into the terminal for enabling that refactoring. The third parameter is a string description of what the refactoring does.

```
// Define the list of tool specific command line options.
#define OPTIONS_LIST \
    X(RunRemoveMemcpy, "remove-memcpy", "This option turns on replacement of memcpy().") \
    X(RunRemoveMemset, "remove-memset", "This option turns on replacement of memset().") \
    X(RunMakeStatic, "make-static", "This option turns all dynamic memory allocations " \
        "into stack ones, gets rid of calloc() and free().") \
    X(RunRemovePointer, "remove-pointer", "This option turns on removal of the global pointer." \
    X(RunRemoveHypot, "remove-hypot", "This option turns on replacement of hypot().") \
    X(RunRemoveVariables, "remove-variables", "This option removes unreferenced variables.") \
    X(RunRemoveAssignment, "remove-assignment", "This option removes unreferenced assignments." \
    X(RunRemoveInitialize, "remove-initialize", "This option removes the initialize function.") \
    X(RunRemoveIndirectRecursion, "remove-indirect-recursion", "This option removes any indirec
        "recursion in the step function by splitting it into major and minor step functions.")
```

Then identify the round of refactoring, and add the callback class to it by getting the Replacements from the RefactoringTool and adding the matchers to the MatchFinder.

```
// Create a fourth RefactoringTool to run the fourth round of refactorings.
RefactoringTool tool4(Compilations, SourcePaths);
// Create a fourth MatchFinder to run the new RefactoringTool through the source code again,
// to apply refactorings in the fourth round.
MatchFinder mf4;

//// Remove memcpy details
RemoveMemcpyMatchCallback remove_memcpy_match_callback(&tool4.getReplacements());
if (RunRemoveMemcpy) {
    remove_memcpy_match_callback.getASTmatchers(mf4);
}

//// Make static details
MakeStaticMatchCallback make_static_match_callback(&tool4.getReplacements());
if (RunMakeStatic) {
    make_static_match_callback.getASTmatchers(mf4);
}

//// Remove pointer details
RemovePointerMatchCallback remove_pointer_match_callback(&tool4.getReplacements());
if (RunRemovePointer) {
    remove_pointer_match_callback.getASTmatchers(mf4);
}

//// Remove hypot details
RemoveHypotMatchCallback remove_hypot_match_callback(&tool4.getReplacements());
if (RunRemoveHypot) {
    remove_hypot_match_callback.getASTmatchers(mf4);
}
```

Lastly, after running the rounds of refactorings, the application prints diagnostic output. Please add an if () statement with the bool variable enabling that refactoring. The number of each kind of refactoring should be printed separately as well as added into the num_refactorings also, for recording the total number of refactorings.

Now your refactoring has been integrated into the STAR Tool, and you can run it by enabling the corresponding command line option.

Useful Resources

Presentations

- ../Konstantin's Presentation/
- ../Nari Visits/

Books

- Create Your Own Refactoring Tool with Clang
- Getting Started with LLVM Core Libraries
- LLVM Essentials
- LLVM Programmer's Manual
- llvm_CommandLine
- diagnostics_in_clang

Lectures

- What is LLVM? What Makes Swift Possible?
<https://www.youtube.com/watch?v=KA8hFBh2eiw>
- What goes behind LLVM Compiler Infrastructure
https://www.youtube.com/watch?v=IR_L1xf4PrU
- Chandler Carruth: Refactoring C++ with Clang
<https://www.youtube.com/watch?v=yulOGfcOH0k>
- Create your own Refactoring Tool in Clang
<https://www.youtube.com/watch?v=8PndHo7jjHk>
- Automatic C++ source code generation with clang - Sergei Sadovnikov [ACCU 2017]
<https://www.youtube.com/watch?v=aPTyatTI42k>
- The Clang AST - a Tutorial
<https://www.youtube.com/watch?v=VqCkCDFLSsc>
- CppCon 2018: James Bennett "Refactoring Legacy Codebases with LibTooling"
<https://www.youtube.com/watch?v=tUBUqJSGr54>
- Pacific++ 2017: Chandler Carruth "LLVM: A Modern, Open C++ Toolchain"
https://www.youtube.com/watch?v=uZI_Qla4pNA
- P. Goldsborough "clang-useful: Building useful tools with LLVM and clang for fun and profit"
<https://www.youtube.com/watch?v=E6i8jmiy8MY>
- emBO++ 2019 - Arvid Gerstmann: Refactoring C++ Using LibTooling

<https://www.youtube.com/watch?v=rbbCgVQrjWs>

Online Tutorials

- <https://learngitbranching.js.org/>

Programming Tools Applications

- Atom IDE.
<https://atom.io/>
- Sourcetrail - The open-source cross-platform source explorer
<https://www.sourcetrail.com/>
- Meld
<https://meldmerge.org/>

Index

- | | |
|------------------------------|--------|
| • Clang/LLVM | page 1 |
| • command line options | page 4 |
| • compiler tool | page 1 |
| • definition list (X Macros) | page 3 |
| • dynamic_cast | page 5 |
| • implementation (X Macros) | page 3 |
| • isa | page 7 |
| • refactoring | page 8 |
| • X Macros | page 2 |