Upgrading For Loops: A Tutorial in Clang Tooling

This document is intended to show how to build a useful source-to-source translation tool based on Clang's LibTooling. It is explicitly aimed at people who are new to Clang, so all you should need is a working knowledge of C++ and the command line.

Ready?

Step 0: Obtaining Clang

As Clang is part of the LLVM project, you'll need to download LLVM's source code first. Both Clang and LLVM are maintained as Subversion repositories, but we'll be accessing them through the git mirror.

First, choose a place for LLVM to live. I created mine in ~/clang-llvm.

```
$ mkdir ~/clang-llvm && cd ~/clang-llvm
$ git clone http://llvm.org/git/llvm
$ cd llvm/tools
$ git clone http://llvm.org/git/clang.git clang
```

All done? Great! Next you need to obtain the CMake build system and Ninja build tool. You may already have CMake installed, but current binary versions of CMake aren't built with Ninja support.

```
$ mkdir ~/cmake && cd ~/cmake
$ git clone https://github.com/martine/ninja.git
$ cd ninja
$ git checkout release
$ ./bootstrap.py
$ sudo cp ninja /usr/bin/
$ git clone git://cmake.org/stage/cmake.git
$ cd cmake
$ git checkout next
$ ./bootstrap
$ make
$ sudo make install
```

Okay. Now we'll build Clang!

```
$ cd ~/clang-llvm
$ mkdir llvm_build
$ cd llvm_build
$ cmake -G Ninja ../ -DLLVM_BUILD_TESTS=ON  # Enable tests; default is off.
$ ninja
$ ninja check       # Test LLVM only.
$ ninja clang-test  # Test Clang only.
$ ninja install
```

And we're live.

All of the tests should pass, though there is a (very) small chance that you can catch LLVM and Clang out of sync. Running `git svn rebase` in both the llvm and clang directories should fix any problems.

Finally, we want to set Clang as its own compiler. The second command will bring up a GUI for configuring Clang; you need to set the entry for CMAKE_CXX_COMPILER.

```
$ cd ~/clang-llvm/llvm/llvm_build
$ ccmake ../
```

        Scroll down to CMAKE_BUILD_TYPE, and set it to Debug.
        Press 't' to turn on advanced mode.
        Scroll down to CMAKE_CXX_COMPILER, and set it to /usr/bin/clang++, or wherever you installed it.
        Press 'c' to configure, then 'g' to generate CMake's files.

Finally, run ninja one last time, and you're done.

Intermezzo: About Clang's Abstract Syntax Tree (AST)

In order to work on the compiler, you need some basic knowledge of the abstract syntax tree. There is some documentation available on Clang's website, but I will mention some of the relevant parts here as well.

In C++, code is comprised of statements and declarations[1]. Clang uses a systematic hierarchy of classes to represent all the components of the AST, which is extensively documented on the Clang doxygen. It's worth some time to explore the basic kinds of statement and expression, since you will need to deal with them directly.

Just looking at the Stmt documentation, you will immediately notice some familiar C++ constructs. For example, it doesn't require much effort to understand the roles of ForStmt, WhileStmt, ReturnStmt, and CompoundStmt, while others such as MSDependentExistsStmt and ObjCForCollectionStmt will probably not turn up in this exercise. Because Clang considers expressions - pieces of code which evaluate to values - to be statements, the Expr class is a subclass of Stmt. You can also see some of the types of expression, such as BinaryOperator,

[1] Actually, there are others, such as preprocessor directives, but we're not concerned with them right now.

CallExpr, and ArraySubscriptExpr from the Stmt documentation, but the Expr class has too many subclasses for the Expr documentation to display a pretty inheritance graph.

Here, it's important to note that Clang's AST contains a surprising amount of information not directly specified by the programmer, such as implicit casts and dealings with temporary expressions. To see what I mean, try asking Clang to show you the AST for this simple program.

```
$ mkdir ~/test-files/
$ gvim ~/test-files/simple.cc
struct T { int x; };
void f() {
  T t1  = {0};
  int i = t1.x + 2;
}
```

This command instructs Clang to dump a text-friendly version of the AST:

```
$ clang -cc1 -ast-dump ~/test-files/simple.cc
```
[2]

Which outputs a bunch of text, ending with

```
(CompoundStmt 0x3697f98
  (DeclStmt 0x3697e48
    0x3697d40 "T t1 =
      (InitListExpr 0x3697e00 'struct T'
        (IntegerLiteral 0x3697d98 'int' 0))")
  (DeclStmt 0x3697f80
    0x3697e70 "int i =
      (BinaryOperator 0x3697f58 'int' '+'
        (ImplicitCastExpr 0x3697f40 'int' <LValueToRValue>
          (MemberExpr 0x3697ef0 'int' lvalue .x 0x3664a80
            (DeclRefExpr 0x3697ec8 'struct T' lvalue Var 0x3697d40 't1'
'struct T')))
        (IntegerLiteral 0x3697f20 'int' 2))"))
```

As you can see, the function we defined has a body which consists of a compound statement. This compound statement contains two declarations - one of which has an LValue-to-RValue implicit conversion listed.

To really get the feel of the AST, it's a good idea to try compiling a few simple programs with the `-cc1 -ast-dump` arguments, though you won't be able to include any standard headers[3]. I

[2] A similar `-ast-dump-xml` option is available too, though it will occasionally drop back to Lisp-style output.
[3] This is because the well-hidden `-cc1` option tells Clang to talk directly to the compiler, rather than automatically including system header files. Even `clang -help` doesn't mention it!

happen to be a fan of the syntax highlighting offered by Vim’s Lisp mode, which is easily accessed by piping the output of the AST dump into my text editor, then setting the filetype to Lisp:

```
$ clang -cc1 -ast-dump ~/test-files/simple.cc | gvim -
```

The output is even prettier when all of the extraneous quotes are deleted.

Finally, Clang's system for handling types is split between two places: QualType and Type. We will usually work with the former, though the latter expresses Clang’s type hierarchy, which will come into play later. Just remember that you might need to check both classes for a given method.

And that’s enough about the AST for now.

Step 1: Creating a ClangTool

Now that we have enough background knowledge, it’s time to create the simplest productive ClangTool in existence: a syntax checker. While this already exists as clang-check, it’s important to understand what’s going on.

First, we’ll need to create a new directory for our tool and tell CMake that it exists.

```
$ cd ~/clang-llvm/llvm/tools/clang
$ mkdir tools/loop-convert
$ echo 'add_subdirectory(loop-convert)' >> tools/CMakeLists.txt
$ vim tools/loop-convert/CMakeLists.txt
```

CMakeLists.txt should have the following contents:

```
set(LLVM_LINK_COMPONENTS support)
set(LLVM_USED_LIBS clangTooling clangBasic clangAST)

add_clang_executable(loop-convert
 loop-convert.cpp
 )
target_link_libraries(loop-convert
 clangTooling
 clangBasic
 clangASTMatchers
 )
```

With that done, Ninja will be able to compile our tool. Let’s give it something to compile!

```
$ vim tools/loop-convert/loop-convert.cpp
```

Even though this is a short code segment, quite a bit is going on. Here’s a breakdown:

First, the includes needed to get a ClangTool up and running:

```
#include "clang/Basic/FileManager.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Tooling/Tooling.h"
#include "clang/Tooling/Refactoring.h"
```

Next: shortcuts for qualified names

```
using std::vector;
using std::string;
namespace cl = llvm::cl;
using namespace clang::tooling;
```

Here, we create two command-line arguments: the build path, and a list of source files to hand to the tool. There aren't many surprises as to what they specify: `BuildPath` is an optional string parameter, and `SourcePaths` is a (potentially empty) `vector<string>`. This relies on the LLVM CommandLine library.

```
static cl::opt<string> BuildPath(
   cl::Positional,
   cl::desc("<build-path>"));

static cl::list<string> SourcePaths(
   cl::Positional,
   cl::desc("<source0> [... <sourceN>]"),
   cl::OneOrMore);
```

Now we get to the main function. We first need to tell Clang how it should treat the source files we wish to examine, which can either be accomplished by referencing a compilation database or by specifying the compilation arguments on the command line.
Note that we first try to decide the compiler options from the command line before falling back on the compilation database.

```
int main(int argc, const char **argv) {
 // OwningPtr is one of LLVM's RAII smart pointers.
 llvm::OwningPtr<CompilationDatabase> Compilations(
   FixedCompilationDatabase::loadFromCommandLine(argc, argv));
 cl::ParseCommandLineOptions(argc, argv);
 if (!Compilations) {
   string ErrorMessage;
   Compilations.reset(CompilationDatabase::loadFromDirectory(BuildPath,
                                                             ErrorMessage));
   if (!Compilations)
     llvm::report_fatal_error(ErrorMessage);
 }
```
Clang uses a compilation database to store all the commands issued to the compiler specifically so that tools don't have to repeat the build system's work.

Now, we create the tool and run it over some source code.
```
 ClangTool SyntaxTool(*Compilations, SourcePaths);
 // First, let's check to make sure there were no errors.
 if (int result = SyntaxTool.run(
     newFrontendActionFactory<clang::SyntaxOnlyAction>())) {
   llvm::errs() << "Error compiling files.\n";
   return result;
 }
 return 0;
}
```

And that's it! You can compile our new tool by running ninja from the llvm_build directory.
```
$ cd ~/clang-llvm/llvm/llvm_build
$ ninja
```

You should now be able to run the syntax checker, which is located in llvm_build/bin, on any source file. Try it!
```
$ bin/loop-convert . \
  ../tools/clang/tools/loop-convert/test-files/simple.cpp --
```
Note the two dashes after we specify the source file. The additional options for the compiler are passed after the dashes rather than loading them from a compilation database - there just aren't any options needed right now.

The full source is available at array-step-1.

Intermezzo: Learning ASTMatchers

Clang recently introduced the ASTMatcher library to provide a simple, powerful, and concise way to specify the shape of an AST. Implemented as a DSL powered by macros and templates (see include/clang/ASTMatchers/ASTMatchers.h if you're curious), matchers offer the feel of algebraic data types common to functional programming languages.

For example, suppose you wanted to examine only binary operators. There is a matcher to do exactly that, conveniently named `binaryOperator`. I'll give you one guess what this matcher does:
```
binaryOperator(hasOperatorName("+"), hasLHS(integerLiteral(equals(0))))
```
Shockingly, it will match against addition expressions whose left hand side is exactly the literal `0`. It will *not* match against other forms of `0`, such as `'\0'` or `NULL`, but it will match against

macros that expand to `0`. The matcher will also *not* match against calls to the overloaded operator '+', as there is a separate `operatorCallExpr` matcher to handle overloaded operators.

The AST matchers are divided into three types: `StatementMatchers`, `DeclarationMatchers`, and `TypeMatchers`, which respectively match against statements, declarations, and types. In addition, I mentally divide machers into a different set of four categories, based on the roles they play.

The first category represents a particular object in the AST. In particular, each kind of node in the AST has a similarly-named matcher - `BinaryOperator` expressions are matched by the `binaryOperator` matcher, `ForStmt` statements are matched by the `forStmt` matcher, and so on.

There are also a good number of “qualifier” matchers which let you deconstruct parts of a particular AST node, such as `hasLHS` for binary operators, `hasLoopInit` for for statements, and `argumentCountIs` for function calls. Clang’s developers have tried to name each of these “qualifier” matchers consistently - they usually begin with a prefix or suffix of “has” or “is”[4].

“Connective” matchers, which combine other matchers, form the third kind. The well-named `anyOf`, `allOf`, `unless`, and `anything` matchers from this category. I find that I rarely use the `allOf` matcher directly, as it is integrated into many of the AST node matchers[5]. One important note for debugging matchers is that C++’s type deduction is unfortunately not as awesome as certain other languages’. Some matchers, and the “connective” matchers in particular, will occasionally need to be wrapped in the `expression` or `statement` matchers in order to avoid ambiguous template deduction errors.

Finally, we arrive at a special matcher and member function, `id` and `bind`, which bind a matched AST node to a string identifier. We need some way to refer to matched nodes in order to do anything interesting with them, after all! `id` and `bind` behave identically[6], with the distinction that `id`’s string identifier is passed *before* the matcher it identifies while bind’s identifier is specified *after* its respective matcher. For example, these two matchers are functionally equivalent:

```
variable(hasType(isInteger())).bind("intvar")
id("intvar", variable(hasType(isInteger())))
```

They both define DeclarationMatchers which match variable declarations of integer type, identified by the string `"intvar"`. I tend to prefer using `bind` for short matchers and `id` for long ones.

[4] There are some exceptions, such as the `callee` matcher, which I would have named `hasCallee`.
[5] If you look at include/ASTMatchers/ASTMatchers.h, you will notice that most of the AST node matchers are defined as `VariadicDynCast`**`AllOf`**`Matchers`, which contain an implicit `allOf`.
[6] Technically, `bind` is a method implemented by bindable matchers, but it serves the same role as `id`.

One last note about the matcher library before we move on to our first real example: Some matchers are only permitted within the contexts of other matchers, such as hasArgument, which will cause a compilation error if it's not used within the context of a function or constructor call. The compiler will give you a somewhat helpful error message in case something is incorrect, usually of the form:

```
../tools/clang/include/clang/ASTMatchers/ASTMatchers.h:1244:3: error:
'instantiated_with_wrong_types' declared as an array with a negative size
  TOOLING_COMPILE_ASSERT((llvm::is_base_of::value ||
  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
../tools/clang/include/clang/ASTMatchers/ASTMatchersInternal.h:53:43: note:
expanded from macro 'TOOLING_COMPILE_ASSERT'
  typedef CompileAssert<(bool(Expr))> Msg[bool(Expr) ? 1 : -1]
                                          ^~~~~~~~~~~~~~~~~~~
../tools/clang/include/clang/ASTMatchers/ASTMatchersInternal.h:518:27: note:
in instantiation of member function
'clang::ast_matchers::internal::matcher_hasArgumentMatcher >::matches'
requested here
    return Matcher(new MatcherT(Param1, Param2));
                          ^
../tools/clang/tools/loop-convert/LoopActions.cpp:15:14: note: in
instantiation of function template specialization
'clang::ast_matchers::internal::PolymorphicMatcherWithParam2 >::operator
Matcher' requested here
     forStmt(hasArgument(0, expression())
```

This is a pre-C++11 compile-time assertion failure of llvm's inheritance check. What deserves your attention is the name of the array declared with a negative size, "`instantiated_with_wrong_types`" in this case. See the LLVM programmer's manual for an explanation of LLVM's type system.

Step 2: Using AST Matchers

Okay, on to using matchers for real. Let's start by defining a matcher which will capture all for statements that begin by defining a new variable initialized to zero. The loop shell will need to attach an identifier, so a good beginning is

```
id("forLoop", forStmt())
```

Next, we want to specify that a single variable is declared in the first portion of the loop, so we can extend the matcher to

```
id("forLoop",
  forStmt(hasLoopInit(declarationStatement(hasSingleDecl(variable())))))
```

Finally, we can add the condition that the variable is initialized to zero.

```
id("forLoop", forStmt(hasLoopInit(declarationStatemnt(hasSingleDecl(variable(
```

```
   hasInitializer(integerLiteral(equals(0)))))))))
```

It is fairly easy to read and understand the matcher definition (“match loops whose init portion declares a single variable which is initialized to the integer literal `0`”), but deciding that every piece is necessary is more difficult. Note that this matcher will *not* match loops whose variables are initialized to `‘\0’`, `0.0`, `NULL`, or any form of zero besides the integer `0`.
The last step is giving the matcher a name:

```
StatementMatcher LoopMatcher =
  id("forLoop",
     forStmt(hasLoopInit(declarationStatement(hasSingleDecl(variable(
         hasInitializer(integerLiteral(equals(0))))))))));
```

Once you have defined your matchers, you will need to add a little more scaffolding in order to run them. Matchers are paired with a MatchCallback and registered with a MatchFinder object, then run from a ClangTool. More code!

```
In ForActions.h:
StatementMatcher LoopMatcher = forStmt().bind("forLoop");

class LoopPrinter : public MatchFinder::MatchCallback {
public :
 virtual void run (const MatchFinder::MatchResult &Result) {
   if (const ForStmt *FS = Result.nodes.getStmtAs<ForStmt>("forLoop"))
     FS->dump();
}

In loop-convert.cpp
// Starting from the definition of SyntaxTool in main()
ClangTool SyntaxTool(*Compilations, SourcePaths);
if (int status = Tool.run(newFrontendActionFactory(&Finder))) {
 llvm::errs() << "Error compiling files.";
 return status;
}
ClangTool LoopTool(*Compilations, SourcePaths);
MatchFinder Finder;
LoopPrinter Printer;
Finder.addMatcher(LoopMatcher, &Printer);
if (int status = Tool.run(newFrontendActionFactory(&Finder)) {
 llvm::errs() << "Error encountered during translation.\n";
 return status;
}
```

Now, you should be able to recompile and run the code to discover for loops. Create a new file with a few examples, and test out our new handiwork:

```
$ cd ~/clang-llvm/llvm/llvm_build/
$ ninja loop-convert
$ vim ~/test-files/simple-loops.cc
$ bin/loop-convert ~/test-files/simple-loops.cc
```

The complete source code with examples is available at array-step-2a

Step 2.5: More Complicated Matchers

Our simple matcher is capable of discovering for loops, but we would still need to filter out many more ourselves. We can do a good portion of the remaining work with some cleverly chosen matchers, but first we need to decide exactly which properties we want to allow.

How can we characterize for loops over arrays which would be eligible for translation to range-based syntax? Range based loops over arrays of size `N`

1. Start at index `0`
2. Iterate consecutively
3. end at index `N - 1`

We already check for (1), so all we need to add is a check to the loop's condition to ensure that the loop's index variable is compared against `N` and another check to ensure that the increment step just increments this same variable. The matcher for (2) is straightforward: require a pre- or post-increment of the same variable declared in the init portion.

Unfortunately, such a matcher is impossible to write. Matchers contain no logic for comparing two arbitrary AST nodes and determining whether or not they are equal[7], so the best we can do is matching *more* than we would like to allow, and punting extra comparisons to the callback.

In any case, we can start building this sub-matcher. We can require that the increment step be a unary increment like this:

```
hasIncrement(unaryOperator(hasOperatorName("++")))
```

Specifying *what* is incremented introduces another quirk of Clang's AST: usages of variables are represented as `DeclRefExpr`'s, or *declaration reference expressions*, because they are expressions which refer to variable declarations.

```
hasIncrement(unaryOperator(
 hasOperatorName("++"),
 hasUnaryOperand(declarationReference())))
```

Further requiring the incremented variable to be an integer type changes our matcher again

[7] There are several possible notions of "equal" here. And backreferences, which would be necessary to do this in a matcher, make regular expression engines significantly more complicated.

```
hasIncrement(unaryOperator(
 hasOperatorName("++"),
 hasUnaryOperand(declarationReference(to(variable(hasType(isInteger())))))))
```

And the last step will be to attach an identifier to this variable, so that we can retrieve it in the callback:

```
hasIncrement(unaryOperator(
 hasOperatorName("++"),
 hasUnaryOperand(declarationReference(to(
   variable(hasType(isInteger())).bind("incrementVariable"))))))
```

We can add this code to the definition of LoopMatcher and make sure that our program, outfitted with the new matcher, only prints out only loops that declare a single variable initialized to zero and have an increment step consisting of a unary increment of some variable.

The adjusted code is available at array-step-2b. Compile and run with the new matchers - does loop-convert behave as you would expect?

Now, we just need to add a matcher to check if the condition part of the for loop compares a variable against the size of the array. There is only one problem - we don't know which array we're iterating over without looking at the body of the loop! We are again restricted to approximating the result we want with matchers, filling in the details in the callback.

```
hasCondition(binaryOperator(hasOperatorName("<"))
```

It makes sense to ensure that the left-hand side is a reference to a variable, and that the right-hand side has integer type.

```
hasCondition(binaryOperator(
 hasOperatorName("<"),
 hasLHS(expression(hasType(isInteger()))),
 hasRHS(declarationReference(to(variable(hasType(isInteger())))))))
```

This code, with a bind added to both the LHS and RHS of the less-than operator, is available at array-step-2c. It is especially important to build and test this version on a standard for loop.

Why? Because it doesn't work. Of the three loops provided in test-files/simple.cpp, zero of them have a matching condition. A quick look at the AST dump of the first for loop, produced by the previous iteration of loop-convert, shows us the answer:

```
(ForStmt 0x173b240
  (DeclStmt 0x173afc8
    0x173af50 "int i =
      (IntegerLiteral 0x173afa8 'int' 0)")
  <<<NULL>>>
  (BinaryOperator 0x173b060 '_Bool' '<'
    (ImplicitCastExpr 0x173b030 'int' <LValueToRValue>
      (DeclRefExpr 0x173afe0 'int' lvalue Var 0x173af50 'i' 'int'))
    (ImplicitCastExpr 0x173b048 'int' <LValueToRValue>
      (DeclRefExpr 0x173b008 'const int' lvalue Var 0x170fa80 'N' 'const
int')))
  (UnaryOperator 0x173b0b0 'int' lvalue prefix '++'
    (DeclRefExpr 0x173b088 'int' lvalue Var 0x173af50 'i' 'int'))
 (CompoundStatement …
```

We already know that the declaration and increments both match, or this loop wouldn't have been dumped. The culprit lies in the implicit cast applied to the first operand (i.e. the LHS) of the less-than operator, an L-value to R-value conversion applied to the expression referencing `i`. Thankfully, the matcher library offers a solution to this problem in the form of `ignoringParenImpCasts`, which instructs the matcher to ignore implicit casts and parentheses before continuing to match. Adjusting the condition operator will restore the desired match.

```
hasCondition(binaryOperator(
 hasOperatorName("<"),
 hasLHS(expression(hasType(isInteger()))),
 hasRHS(ignoringImpCasts(declarationReference(
   to(variable(hasType(isInteger()))))))))
```

After adding `bind`s to the expressions we wished to capture and extracting the identifier strings into variables, we have array-step-2 completed.

Step 3: Retrieving Matched Nodes

So far, the matcher callback isn't very interesting: it just dumps the loop's AST. At some point, we will need to make changes to the input source code. Next, we'll work on using the nodes we bound in the previous step.

The `MatchFinder::run()` callback takes a `MatchFinder::MatchResult&` as its parameter. We're most interested in its Context and Nodes members. Clang uses the `ASTContext` class to represent contextual information about the AST, as the name implies, though the most functionally important detail is that several operations require an `ASTContext*` parameter. More immediately useful is the set of matched nodes, and how we retrieve them.

Since we bound three variables (identified by `ConditionVarName`, `InitVarName`, and `IncrementVarName`), we can obtain the matched nodes by using the `getDeclAs()` member function.

In LoopActions.cpp

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

void LoopPrinter::run(const MatchFinder::MatchResult &Result) {
  ASTContext *Context = Result.Context;
  const ForStmt *FS = Result.Nodes.getStmtAs<ForStmt>(LoopName);
  // We do not want to convert header files!
  if (!FS || !Context->getSourceManager().isFromMainFile(FS->getForLoc()))
    return;
  const VarDecl *IncVar = Result.Nodes.getDeclAs<VarDecl>(IncrementVarName);
  const VarDecl *CondVar = Result.Nodes.getDeclAs<VarDecl>(ConditionVarName);
  const VarDecl *InitVar = Result.Nodes.getDeclAs<VarDecl>(InitVarName);
```

Now that we have the three variables, represented by their respective declarations, let's make sure that they're all the same, using a helper function I call `areSameVariable()`.

```
  if (!areSameVariable(IncVar, CondVar) || !areSameVariable(IncVar, InitVar))
    return;
  llvm::outs() << "Potential array-based loop discovered.\n";
}
```

If execution reaches the end of `LoopPrinter::run()`, we know that the loop shell that looks like

```
for (int i= 0; i < expr(); ++i) { ... }
```

For now, we will just print a message explaining that we found a loop. The next section will deal with recursively traversing the AST to discover all changes needed.

As a side note, here is the implementation of `areSameVariable`. Clang associates a `VarDecl`[8] with each variable to represent the variable's declaration. Since the "canonical" form of each declaration is unique by address, all we need to do is make sure neither `ValueDecl` is `NULL` and compare the canonical `Decls`.

```
static bool areSameVariable(const ValueDecl *First, const ValueDecl *Second)
{
  return First && Second &&
         First->getCanonicalDecl() == Second->getCanonicalDecl();
}
```

It's not as trivial to test if two expressions are the same, though Clang has already done the hard work for us by providing a way to canonicalize expressions:

```
static bool areSameExpr(ASTContext* Context, const Expr *First,
                        const Expr *Second) {
  if (!First || !Second)
```

[8] VarDecl is a subclass of ValueDecl, which is itself a subclass of Decl. See the VarDecl documentation for details.

```
    return false;
  llvm::FoldingSetNodeID FirstID, SecondID;
  First->Profile(FirstID, *Context, true);
  Second->Profile(SecondID, *Context, true);
  return FirstID == SecondID;
}
```

This code relies on the comparison between two `llvm::FoldingSetNodeID`s. As the documentation for Stmt::Profile() indicates, the `Profile()` member function builds a description of a node in the AST, based on its properties, along with those of its children. FoldingSetNodeID then serves as a hash we can use to compare expressions.

We will need `areSameExpr` later. For now, the next checkpoint is aray-step-3. Before you run the new code on the additional loops added to `test-files/simple.cpp`, try to figure out which ones will be considered potentially convertible.

Step 4: Finding Usages

We now arrive at the most complicated part of the journey: checking the body of the for loop to discover all usages of the loop variable we identified in the previous step. My first instinct would be to write a matcher that checks for `ArrayIndexExprs` to find permitted usages, and a matcher that checks for other references to the loop variable.

```
StatementMatcher IndexUsageMatcher =
  declarationReference(to(variable(fromAST(Var)).bind(IndexName)));
StatementMatcher ArrayUsageMatcher =
  ArrayIndexExpression(hasBase(expression().bind(ArrayName)),
                       hasIndex(ignoringImplicitCasts(IndexUsageMatcher)));
```

Unfortunately, the ASTMatchers library doesn't support running a MatchFinder over an arbitrary Stmt, nor does it permit us to create matchers that compare themselves to AST nodes. For now, we will fall back upon the previously available tool for exploring the AST: the `RecursiveASTVisitor`[9], which implements a flexible and powerful way to crawl an AST. Don't worry, as this use case doesn't rely on the class's more complicated aspects.

First, we need to subclass RecursiveASTVisitor.
In LoopActions.h,

```
#include "clang/AST/RecursiveASTVisitor.h"
class ForLoopASTVisitor : public RecursiveASTVisitor<ForLoopASTVisitor> {
 public :
```

First, some type aliases. We'll need some way to return a collection of permitted expressions

[9] If you have never seen the curiously recurring template pattern, this would be a good time to learn.

that we intend to change, along with a collection of the arrays that were indexed.

```
  typedef const Expr* Usage;
  typedef llvm::SmallVector<Usage, 8> UsageResult;
  typedef llvm::SmallPtrSet<const Expr *, 1> ContainerResult;
```

Next, this class will need to remember the variable we are checking, along with the expressions that need converting.

```
 private:
  bool OnlyUsedAsIndex;
  ASTContext *Context;
  UsageResult Usages;
  ContainerResult ContainersIndexed;
  const VarDecl *TargetVar;
```

And finally, we will need a way to traverse the body of a for loop.

```
 public:
  bool findUsages(const Stmt *Body) {
    TraverseStmt(const_cast<Stmt *>(Body));
    return OnlyUsedAsIndex;
  }

  // Methods for AST traversal
  bool TraverseArraySubscriptExpr(ArraySubscriptExpr *ASE);
  bool VisitDeclRefExpr(DeclRefExpr *DRE);
  bool VisitDeclStmt(DeclStmt *DS);
};
```

Add a few accessors and a constructor, and you will have the code in LoopActions.h.

Next, we need to implement the interesting part: deciding if our index variable `TargetVar` is actually only used as an array index. The first step will be disallowing all `DeclRefExprs` referring to the `TargetVar` by setting `OnlyUsedAsIndex` to false should we encounter it.

```
bool ForLoopASTVisitor::VisitDeclRefExpr(DeclRefExpr *DRE) {
  const ValueDecl *TheDecl = DRE->getDecl();
  if (areSameVariable(TargetVar, TheDecl))
    OnlyUsedAsIndex = false;
  return true;
}
```

Of course, we explicitly allow references to `TargetVar` if they are as the index of an `ArraySubscriptExpr`. We therefore intercept the traversals of `ArraySubscriptExprs`, and make sure that the index is *not* visited if we consider it to be a valid use of `TargetVar`.

```
bool ForLoopASTVisitor::TraverseArraySubscriptExpr(ArraySubscriptExpr *ASE) {
  const Expr *Arr = ASE->getBase();
  if (isValidSubscriptExpr(ASE->getIdx(), TargetVar, Arr)) {
    const Expr *ArrReduced = Arr->IgnoreParenCasts();
```

We keep track of the arrays TargetVar is used to index.

```
    if (!containsExpr(Context, &ContainersIndexed, ArrReduced))
      ContainersIndexed.insert(ArrReduced);
```

Since we consider this a valid (convertible) use of `TargetVar`, we prune the AST traversal by *not* calling `VisitorBase::TraverseStmt()`.

```
    Usages.push_back(ASE);
    return true;
  }
```

If we didn't decide that the ArraySubscriptExpr was exactly what we were looking for, continue the recursive traversal.

```
  return TraverseStmt(Arr) && TraverseStmt(ASE->getIdx());
}
```

This implementation relies on two helper functions: `isValidSubscriptExpr(), which` checks to see if the index of an ArrayIndexExpr was TargetVar, and `containsExpr()`, which checks to see if our collecion of indexed arrays contains an expression equivalent to the discovered array.

With our RecursiveASTVisitor built, all we need to do is use it. Some simple adjustments to the end of LoopPrinter::run() can exercise the new functionality.

```
// in LoopPrinter::run() {
  ForLoopASTVisitor Finder(Context, LoopVar);
  if (!Finder.findUsages(FS->getBody()))
    return;
```

If we make it here, then LoopVar was only used as an array index, if at all. Did it index into exactly one array?

```
  const ForLoopASTVisitor::ContainerResult &ContainersIndexed =
      Finder.getContainersIndexed();
  if (ContainersIndexed.size() != 1)
    return;
```

At this point, we know that there was exactly one array. We now need to make sure that it has the correct length.

```
  const Expr *ContainerExpr = *(ContainersIndexed.begin());
  if (!arrayMatchesConditionExpr(Context, ContainerExpr->getType(),
                                 BoundExpr))
    return;
  llvm::outs() << "Discovered translatable loop: index variable is "
               << LoopVar->getNameAsString() << ".\nArray expression is: ";
  ContainerExpr->dump();
  llvm::outs() << "\n";
}
```

The helper function `arrayMatchesConditionExpr()` checks to see if the integer bound BoundExpr is a compile-time constant equal to the array's length. It fiddles with the [`llvm::APInt`] and [`llvm::APSInt`] classes, comparing the two compile-time values by converting BoundExpr to an unsigned integer value, if possible[10].

Whew! We're finally done with this step, ready to try out our new code, available in array-step-4 on some more loops!

Step 5: Editing The Source Code

The next touch is to modify the existing source code through the `tooling::Replacements` class. We can use the Replacements& reference provided by the RefactoringTool created in the main function (in LoopConvert.cpp), simply by adding a Replacements& member to LoopPrinter. Come to think of it, LoopPrinter isn't a wonderful name for the class any more, since it will actually modify the source code of array-based loops when we're done with this step; I renamed it to LoopFixer.

In LoopActions.h

```
class LoopFixer : public MatchFinder::MatchCallback {
 private:
  tooling::Replacements &Replace;

 public:
  explicit LoopFixer(tooling::Replacements &Replace) : Replace(Replace) { }
  virtual void run(const MatchFinder::MatchResult &Result);
};
```

Note that we mark the constructor `explicit` to avoid accidental conversions; because we've added a parameter, the usage in LoopConvert.cpp also needs to be adjusted

```
Replacements &Replace = LoopTool.getReplacements();
LoopFixer Fixer(Replace);
```

[10] This is because comparing a signed integer and an unsigned integer is always a bug at the "low" level of LLVM, as is comparing two integers of different bit widths. The length of an array is nonnegative, so we need to convert BoundExpr's value into an unsigned value - which can be done without losing precision by extending each to be one bit larger than the bigger of the two.

Most of the interesting work in this step is in creating a function to replace the text-dumping. Rather than printing some textual information about the loops, we will add a call to a new function:

```
doConversion(Context, Replace, LoopVar, ContainerExpr, Finder.getUsages(),
             FS);
```

And now we need to define this `doConversion`.

```
static void doConversion(ASTContext *Context, Replacements &Replace,
                         const VarDecl *IndexVar, const Expr *ContainerExpr,
                         const UsageResult &Usages, const ForStmt *TheLoop) {
```

We need some method of choosing a variable name and ensuring that it does not conflict with existing declarations. For now, we'll just default to "elem".

```
  std::string VarName = "elem";
```

First, replace all usages of the array subscript expression with our new variable. Clang represents the location of a piece of the AST with a SourceRange.

```
 for (UsageResult::const_iterator I = Usages.begin(), E = Usages.end();
       I != E; ++I) {
    SourceRange ReplaceRange = (*I)->getSourceRange();
    std::string ReplaceText = VarName;
    Replace.insert(Replacement(Context->getSourceManager(),
                               CharSourceRange::getTokenRange(ReplaceRange),
                               ReplaceText));
  }
```

Now, we need to construct a new range expression.

```
  SourceRange ParenRange(TheLoop->getLParenLoc(), TheLoop->getRParenLoc());
  StringRef ContainerString =
      getStringFromRange(Context->getSourceManager(), Context->getLangOpts(),
                         ContainerExpr->getSourceRange());
```

Finally, we designate the type of the variable as "`const auto &`"

```
  QualType AutoRefType =
      Context->getLValueReferenceType(Context->getAutoDeductType());
  std::string TypeString = AutoRefType.getAsString();

  std::string Range = ("(" + TypeString + " " + VarName + " : "
                           + ContainerString + ")").str();
  Replace.insert(Replacement(Context->getSourceManager(),
                             CharSourceRange::getTokenRange(ParenRange),
                             Range));
}
```

And we're (almost) done. There's one last helper function to explain: getStringFromRange, which turns a SourceRange into its associated text in the source code, if the SourceRange consists of a start and end SourceLocation in the same file[11].

```
static StringRef getStringFromRange(SourceRange Range,
                                    SourceManager &SourceMgr,
                                    const LangOptions &LangOpts) {
```
Source Files are identified by a FileID, and are managed by a SourceManager.
```
  if (SourceMgr.getFileID(Range.getBegin()) !=
      SourceMgr.getFileID(Range.getEnd()))
    return NULL;
  CharSourceRange SourceChars(Range, true);
  return Lexer::getSourceText(SourceChars, SourceMgr, LangOpts);
}
```
With that, our simple array-based loop converter is complete, and found in array-step-5.

Step 6: Adding FileCheck Tests

The Clang test suite includes just over 5000 tests, and Clang developers are unlikely to accept any new code unless it comes with tests. Thankfully, there is an almost-painless way to write tests as annotated C++ files using the LLVM test infrastructure. If we drop a test in the test/ directory within Clang's repository, it will automatically be included in the test suite.
```
$ vim clang/test/Tooling/loop-convert-array.cpp
```
First, we need to instruct Clang how to run the test. The lines beginning with RUN: are almost-shell scripts, the main difference being that a few substitutions are made. In this example,

- `%t` is replaced with the name of a temporary file
- `%s` is replaced by the name of the current file (i.e. loop-convert-array.cpp)

We run the test's source code through grep to remove any of the commented annotations.
```
// RUN: rm -rf %t.cpp
// RUN: grep -Ev "//\s*[A-Z-]+:" %s > %t.cpp
// RUN: loop-convert . %t.cpp -- && FileCheck -input-file=%t.cpp %s
// RUN: rm -rf %t.cpp
```
This test will focus on positive changes - loops that should be converted. There are five tests included in the example, intended to cover particular cases.

- The first test uses the index as an r-value
- The second test uses the index as an l-value
- The third test uses a more complicated expression as the array bound
- The fourth test uses a more complicated expression as the array
- The fifth test calls a member function on an array of structs

[11] Using `#include`, we can create expressions where this is not the case.

We will walk through instrumenting a few of the loops for testing.
Consider this snippet:

```
  const int N = 6;
  int arr[N] = {1, 2, 3, 4, 5, 6};
  for (int i = 0; i < N; ++i) {
    sum += arr[i];
  }
```

We expect the loop to be converted to something along the lines of

```
  for (auto & VARNAME : arr) {
    sum += VARNAME;
  }
```

Clang uses FileCheck to run these tests, so we need to add a line to describe our expectations. FileCheck allows us to specify a particular sequence of strings that should (or should not) occur in the output of our command. The checks for this loop are

```
 // CHECK: for (auto & [[VAR:[a-z_]+]] : arr)
 // CHECK-NEXT: sum += [[VAR]];
 // CHECK-NEXT: }
```

This example illustrates several useful FileCheck properties. First, FileCheck allows us to capture a regular expression[12] and use it later. The text `[[VAR:[a-z_]+]]` matches against the regular expression `[a-z_]+`, saving the text to a variable named VAR. The next line references VAR (by surrounding the variable name with double brackets); regexes that do not need to be saved can be placed in double braces {{like this}}.
Finally, the CHECK directive instructs FileCheck to find the string after the colon and issue an error if it's not found. FileCheck also requires that the order of your CHECKs be followed: each CHECK is attempted from the end of the last successful CHECK. Additionally, it's possible to require that the matching line be the *next* line, using CHECK-NEXT.

It is also important to write *negative tests*, which capture instances that should *not* be modified. Some examples include computations done with the indices, loops with empty bodies, and loops whose array bounds don't match correctly. The RUN: header for negative tests is identical to the RUN lines for the positive tests, though they are implemented with a single CHECK line:

```
  // CHECK-NOT: for ({{.*[^:]:[^:].*}})
```

This line instructs FileCheck to verify that the supplied regex is not matched, namely that there are no for loops that contain a single colon on the inside of the loop parentheses[13]. Any converted loops would be written with range-based syntax, which includes a single colon. You should copy one of the loops from the positive tests over to the negative test to make sure that it fails.
The entire Clang test suite can be run from your `llvm_build` directory with the command

[12] I'm not sure which kind of regular expressions (perl? grep? grep -E? vim? sed?), so it can take a few tries.
[13] Remember, `::` is a perfectly valid construct.

```
$ ninja clang-test
```
[14]

One last touch would be modifying test/CMakeLists.txt to include loop-convert as a dependency. Look for the line which begins with "`add_dependencies(clang-test`".
Tests are included in array-step-6.

Epilogue: Finalizing the Tool

That's it!
Wait a minute, you ask, are these conversions *safe*? The answer is "yes, *usually*." One can imagine the (somewhat evil example) of intentionally calling a function that swaps which arrays a variable refers to in the middle of the loop, or on another thread. C++11 range-based for loops will continue iterating over the original array referred to at the beginning of the loop, though the original code would index into the new array after the swap. While it is conceivable that we could detect such problems, it would require a def-use analysis of each iterated array, which is not always possible to do perfectly[15]. The general advice will be to migrate potentially problematic code manually.

Actually, this tool is less correct than it should be.

- It can potentially try to write conflicting replacements to source code, which is not handled correctly by Replacements.
- It does not verify that the variables it introduces do not conflict, particularly in nested loops where it uses the same name
- The resulting code is not automatically recompiled with the -std=c++11 flag added.
- The array can be manipulated as stated above

These changes add far more complexity than I felt should be added at this point, though all four are addressed in the more complete version of the loop converter[16].

I also deliberately left a simple bug in the array converter - a loop such as this one will not be converted:

```
const int N = 2;
  int arr[N];
  int res = 0;
 for (unsigned i = 0; i < N; ++i) {
    res += arr[i] + 1;
  }
```

We therefore end this tutorial with an exercise: look at the AST for the above loop, and figure out why it is not converted (Is it the matcher? the AST visitor? the logic executed between the two?). Then add a positive unit test to to confirm that the original code did not migrate this loop, but the new code does.

[14] Yes, I understand that you're paying attention and that this was stated at the beginning of the tutorial.
[15] Consider a templated function that takes two pointers to arrays of size N...
[16] To the extent that the array changing out from under the loop directly in its own body.

Hopefully this tutorial has provided enough basic knowledge about developing refactoring tools with Clang. Now go make C++ development even more awesome with what you've learned!