

# Введение в теорию трансляторов

## Практика

### “Проверка стиля языка СИ”

Anatoly Talamanov  
AI Frameworks Engineer

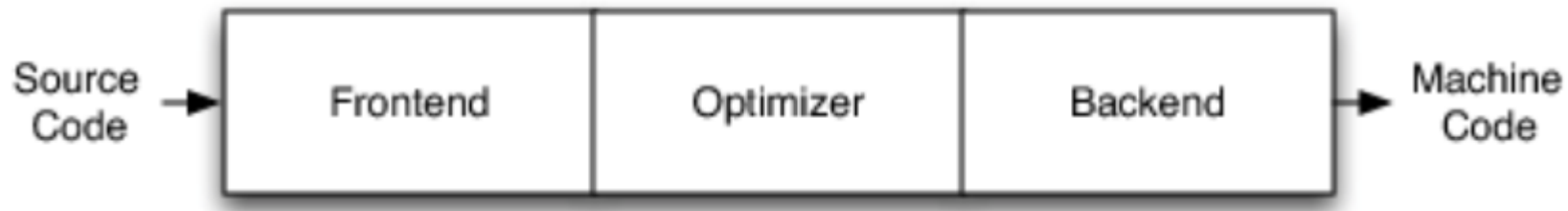
Elizaveta Shulankina  
Development Tools Engineer

Clang и его интерфейсы

# Clang

Clang - это **C language family front end** для LLVM.

В архитектуре компилятора front end берет на себя часть анализа, отвечающего за разбиение исходного кода на части в соответствии с грамматической структурой. Результатом является промежуточное представление, которое back end преобразует к целевой программе.



# Clang AST (Abstract Syntax Tree)

Front end отвечает за синтаксический анализ исходного кода, проверку его на наличие ошибок и преобразование данного кода в **AST (Abstract Syntax Tree – абстрактное синтаксическое дерево)**.

AST – это структурированное представление, которое можно использовать для различных целей, таких как создание таблицы символов, выполнение проверки типов и, наконец, генерация кода.

Для нашей задачи мы будем использовать AST, чтобы находить в исходном коде выражения, подлежащие замене.

# Задача

Требуется реализовать tool с помощью интерфейсов Clang, который будет находить в исходном коде приложения все приведения типов в стиле СИ и заменять их на аналог из C++.

Пример:

```
double d = 4.5;  
int i = (int)d; // int i = static_cast<int>(d);
```

# Какие интерфейсы Clang потребуются? (1)

Потребуется познакомиться с базовыми структурами AST:

## 1. `ASTContext`

Вся информация об AST для единицы перевода собрана в классе [`ASTContext`](#). Он позволяет обойти всю единицу трансляции, начиная с [`getTranslationUnitDecl`](#), получить доступ к таблице идентификаторов Clang для проанализированной единицы трансляции.

## 2. `AST Classes/Nodes`

AST построен с использованием трех групп основных классов/узлов: [объявлений](#) (`Decl`), [операторов](#) (`Stmt`) и [типов](#) (`Type`). Эти три класса составляют основу целого ряда специализаций.

# Какие интерфейсы Clang потребуются? (1)

Пример [AST](#) для оператора If:

Рассмотрим оператор if, который представлен классом/узлом `IfStmt` в AST.

Он состоит из условного Expr (`BinaryOperator` класс)

и двух `CompoundStmt`,

один для `then-case` и

один для `else-case` соответственно.

```
$ cat example.cpp
```

```
int f(int i) {  
    if (i > 0) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

```
$ clang -Xclang -ast-dump -fsyntax-only example.cpp
```

```
[part of the AST left out for conciseness]
```

```
`-IfStmt 0x2ac4638 <line:2:5, line:6:5>
```

```
| -<<<NULL>>>
```

```
| -BinaryOperator 0x2ac4540 <line:2:9, col:13> '_Bool' '>'
```

```
| | -ImplicitCastExpr 0x2ac4528 <col:9> 'int' <LValueToRValue>
```

```
| | | -DeclRefExpr 0x2ac44e0 <col:9> 'int' lvalue ParmVar 0x2ac4328 'i' 'int'
```

```
| | -IntegerLiteral 0x2ac4508 <col:13> 'int' 0
```

```
| -CompoundStmt 0x2ac45b0 <col:16, line:4:5>
```

```
| | -ReturnStmt 0x2ac4598 <line:3:9, col:16>
```

```
| | | -ImplicitCastExpr 0x2ac4580 <col:16> 'int' <IntegralCast>
```

```
| | | -CXXBoolLiteralExpr 0x2ac4568 <col:16> ' Bool' true
```

```
`-CompoundStmt 0x2ac4618 <line:4:12, line:6:5>
```

```
`-ReturnStmt 0x2ac4600 <line:5:9, col:16>
```

```
`-ImplicitCastExpr 0x2ac45e8 <col:16> 'int' <IntegralCast>
```

```
`-CXXBoolLiteralExpr 0x2ac45d0 <col:16> ' Bool' false
```

## Какие интерфейсы Clang потребуются? (2)

Для нахождения приведений в стиле СИ в исходном коде мы будем использовать [AST Matchers](#) (сопоставители).

[AST Matchers](#) предоставляют предметно-ориентированный язык (DSL) для сопоставления предикатов в AST Clang. Например, соответствие выражению вызова с именем [doSomething](#) будет выглядеть так:

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

Справочник по [AST Matches](#) объясняет, как использовать DSL для создания сопоставителей для интересующих узлов в дереве AST.



# Типы AST Matchers: Node Matchers

Node Matchers (сопоставители узлов) лежат в основе выражений сопоставителей - они определяют тип ожидаемого узла. Каждое выражение сопоставления начинается с **node matcher**, который затем может быть уточнен с помощью сопоставителя сужения (**narrowing matcher**) или обхода (**traversal matcher**). Traversal matchers принимают сопоставители узлов в качестве аргументов.

**Node Matchers** единственные средства сопоставления, которые поддерживают вызов **bind("id")** для привязки сопоставленного узла к определенной строке, которая позже может быть извлечена из **MatchCallback**.

Matcher<[Decl](#)>

decl

Matcher<[Decl](#)>...

Matcher<[Decl](#)>

declaratorDecl

Matcher<[DeclaratorDecl](#)>...

Matcher<[Decl](#)>

decompositionDecl

Matcher<[DecompositionDecl](#)>...

Matcher<[Decl](#)>

enumConstantDecl

Matcher<[EnumConstantDecl](#)>...

# Типы AST Matchers: Narrowing Matchers

Narrowing Matchers (сужающие сопоставители) соответствуют определенным атрибутам на текущем узле, тем самым сужая набор узлов текущего типа для сопоставления. Существуют специальные логические сужающие сопоставления (allOf, anyOf, something и except), которые позволяют пользователям создавать более мощные выражения сопоставления.

Matcher<\*>

Matcher<[BinaryOperator](#)>

Matcher<[BinaryOperator](#)>

Matcher<[BinaryOperator](#)>

Matcher<[BinaryOperator](#)>

Matcher<[CXXBaseSpecifier](#)>

Matcher<[CXXBaseSpecifier](#)>

Matcher<[CXXBaseSpecifier](#)>

unless

hasAnyOperatorName

hasOperatorName

isAssignmentOperator

isComparisonOperator

isPrivate

isProtected

isPublic

Matcher<\*>

StringRef, ..., StringRef

std::string Name

# Типы AST Matchers: Traversal Matchers

Traversal Matchers (сопоставители обхода) определяют отношение к другим узлам, доступным из текущего узла. Обратите внимание, что существуют специальные средства сопоставления обхода (`has`, `hasDescendant`, `forEach` и `forEachDescendant`), которые работают на всех узлах и позволяют пользователям писать более общие выражения сопоставления.

<code>Matcher&lt;*&gt;</code>	<code>hasParent</code>	<code>Matcher&lt;*&gt;</code>
<code>Matcher&lt;*&gt;</code>	<code>invocation</code>	<code>Matcher&lt;*&gt;...Matcher&lt;*&gt;</code>
<code>Matcher&lt;*&gt;</code>	<code>optionally</code>	<code>Matcher&lt;*&gt;</code>
<code>Matcher&lt;*&gt;</code>	<code>traverse</code>	<code>TraversalKind TK, Matcher&lt;*&gt; InnerMatcher</code>
<code>Matcher&lt;<a href="#">AbstractConditionalOperator</a>&gt;</code>	<code>hasCondition</code>	<code>Matcher&lt;<a href="#">Expr</a>&gt; InnerMatcher</code>
<code>Matcher&lt;<a href="#">AbstractConditionalOperator</a>&gt;</code>	<code>hasFalseExpression</code>	<code>Matcher&lt;<a href="#">Expr</a>&gt; InnerMatcher</code>
<code>Matcher&lt;<a href="#">AbstractConditionalOperator</a>&gt;</code>	<code>hasTrueExpression</code>	<code>Matcher&lt;<a href="#">Expr</a>&gt; InnerMatcher</code>
<code>Matcher&lt;<a href="#">AddrLabelExpr</a>&gt;</code>	<code>hasDeclaration</code>	<code>Matcher&lt;<a href="#">Decl</a>&gt; InnerMatcher</code>

# Пример использования AST Matchers

Давайте рассмотрим пример использования AST Matchers для нахождения в исходном коде вектора, переданного по значению.

Входный тестовый файл:

```
#include <vector>

void foo(std::vector<int> is);
void bar(const std::vector<int> is);
void foobar(const std::vector<int>& cis);
void fooboo(std::vector<int>& cis);
```

# Пример использования AST Matchers: AST Nodes

`Decl` и `Stmt`, это узлы AST, с которыми мы столкнемся при обработке исходного файла.

С различными `Decl`, `Stmt` и их производными классами, такими как `NamedDecl` и `CallExpr`, мы можем получить много информации, такой как тип переменной, имя, информация об определении и т. д.

# Пример использования AST Matchers

Чтобы проверить все объявления функций с любым параметром типа `std::vector`, нам нужно будет сопоставить конкретный узел AST, который:

- Является объявлением или определением функции
- Имеет хотя бы один параметр типа `std::vector`

# Пример использования AST Matchers

Чтобы использовать сопоставители AST, нам нужно вызвать группу функций создания сопоставления, связать их вместе, чтобы получить нужный сопоставитель, и/или связать целевой узел с именем, чтобы мы могли извлечь его позже.

# Пример использования AST Matchers

```
// Соответствие объявлению функции:
DeclarationMatcher Matcher = functionDecl();
// Сопоставление параметров:
// Чтобы сопоставить параметр, мы можем использовать hasParameter(N, ParamMatcher),
// который будет соответствовать N-му параметру с заданным сопоставителем параметров.
// Но, поскольку нам нужно будет сопоставить любой параметр типа std::vector,
// мы будем использовать hasAnyParameter.
DeclarationMatcher Matcher = functionDecl(hasAnyParameter(...));
// Сопоставление типа параметра:
DeclarationMatcher Matcher = functionDecl(
    hasAnyParameter(hasType(recordDecl(matchesName("std::vector")))));
// ! Все сопоставители, являющиеся существительными, описывают сущности в AST
// и могут быть связаны,
// чтобы на них можно было ссылаться всякий раз, когда будет найдено совпадение.
// Для этого требуется вызвать метод bind для нужных сопоставителей.
// В данном примере нам требуется сослаться на объявление функции:
DeclarationMatcher Matcher = functionDecl(
    decl().bind("funcDeclId"),
    hasAnyParameter(hasType(recordDecl(matchesName("std::vector")))));
```



# Пример использования AST Matchers: MatchCallback

Когда полученный AST matcher найдет правильный узел, будет вызван соответствующий `clang::ast_matchers::MatchFinder::MatchCallback` с результатом сопоставления.

Предоставляя `MatchCallback`, мы можем распечатать объявления/определения функций, принимающие любой параметр типа `std::vector`, который передается по значению.

# Пример использования AST Matchers: MatchCallback

```
class VecCallback : public clang::ast_matchers::MatchFinder::MatchCallback {
public:
    virtual void
    run(const clang::ast_matchers::MatchFinder::MatchResult &Result) final {
        llvm::outs() << ".";
        if (const auto *F =
            Result.Nodes.getNodeAs<clang::FunctionDecl>("funcDeclId")) {
            const auto& SM = *Result.SourceManager;
            const auto& Loc = F->getLocation();
            llvm::outs() << SM.getFilename(Loc) << ":"
                << SM.getSpellingLineNumber(Loc) << ":"
                << SM.getSpellingColumnNumber(Loc) << "\n";
        }
    }
};
```

# Какие интерфейсы Clang потребуются? (3)

Для замены исходного кода программы будем пользоваться классом [Rewriter](#).

[Rewriter](#) - ключевой компонент в задачах “source-to-source transformation”.

Вместо того, чтобы обрабатывать каждый возможный узел AST для вывода кода из AST, Rewriter предлагает подход, заключающийся в хирургическом изменении исходного кода в ключевых местах для выполнения преобразования.

Rewriter - это диспетчер буферов, который использует структуру данных [rope](#), чтобы обеспечить эффективное нарезание и разрезание исходного кода. В сочетании с сохранением исходных местоположений для всех узлов AST в Clang, Rewriter позволяет очень точно удалять и вставлять код.

# Какие интерфейсы Clang потребуются? (3)

## clang::Rewriter Class Reference

**Rewriter** - This is the main interface to the rewrite buffers. More...

```
#include "clang/Rewrite/Core/Rewriter.h"
```

### Classes

```
struct RewriteOptions
```

### Public Types

```
using buffer_iterator = std::map< FileID, RewriteBuffer >::iterator
```

```
using const_buffer_iterator = std::map< FileID, RewriteBuffer >::const_iterator
```

### Public Member Functions

```
Rewriter ()=default
```

```
Rewriter (SourceManager &SM, const LangOptions &LO)
```

```
void setSourceMgr (SourceManager &SM, const LangOptions &LO)
```

```
SourceManager & getSourceMgr () const
```

```
const LangOptions & getLangOpts () const
```

```
int getRangeSize (SourceRange Range, RewriteOptions opts=RewriteOptions()) const  
getRangeSize - Return the size in bytes of the specified range if they are
```

```
int getRangeSize (const CharSourceRange &Range, RewriteOptions opts=RewriteOptions()) const  
getRangeSize - Return the size in bytes of the specified range if they are
```

```
bool InsertText (SourceLocation Loc, StringRef Str, bool InsertAfter=true, bool indentNewLines=false)  
InsertText - Insert the specified string at the specified location in the original buffer. More...
```

```
bool InsertTextAfter (SourceLocation Loc, StringRef Str)  
InsertTextAfter - Insert the specified string at the specified location in the original buffer. More...
```

```
bool InsertTextAfterToken (SourceLocation Loc, StringRef Str)  
InsertTextAfterToken - Insert the specified string after the token in the specified location. More...
```

```
bool InsertTextBefore (SourceLocation Loc, StringRef Str)  
InsertTextBefore - Insert the specified string at the specified location in the original buffer. More...
```

```
bool RemoveText (SourceLocation Start, unsigned Length, RewriteOptions opts=RewriteOptions())  
RemoveText - Remove the specified text region. More...
```

```
bool RemoveText (CharSourceRange range, RewriteOptions opts=RewriteOptions())  
RemoveText - Remove the specified text region. More...
```

```
bool RemoveText (SourceRange range, RewriteOptions opts=RewriteOptions())  
RemoveText - Remove the specified text region. More...
```

```
bool ReplaceText (SourceLocation Start, unsigned OrigLength, StringRef NewStr)  
ReplaceText - This method replaces a range of characters in the input buffer with a new string. More...
```

```
bool ReplaceText (CharSourceRange range, StringRef NewStr)  
ReplaceText - This method replaces a range of characters in the input buffer with a new string. More...
```

```
bool ReplaceText (SourceRange range, StringRef NewStr)  
ReplaceText - This method replaces a range of characters in the input buffer with a new string. More...
```

```
bool ReplaceText (SourceRange range, SourceRange replacementRange)  
ReplaceText - This method replaces a range of characters in the input buffer with a new string. More...
```

# Какие интерфейсы Clang потребуются? (4)

Один из основных классов AST, который потребуется в реализации задачи:

CStyleCastExpr, представляющий явное приведение в C (C99 6.5.4) или приведение в стиле C в C ++ (C ++ [expr.cast]), в котором используется синтаксис `(type) expr`.

# Предусловия для выполнения задачи

# Предусловия для выполнения задачи

1. Установка окружения
2. Сборка LLVM & Clang

# Установка окружения

## Windows 10

Требуется установить WSL (Windows Subsystem for Linux) согласно [инструкции](#) и выбрать дистрибутив Ubuntu20 или Ubuntu18.

Далее можно переходить к следующему шагу.

## Ubuntu 20/18

Можно переходить к следующему шагу.



# Сборка LLVM & Clang

Требуется выполнить шаги согласно [инструкции](#), чтобы собрать LLVM & Clang.

*Либо*

Можно следовать [инструкции](#), чтобы использовать уже собранный LLVM & Clang.

Шаги для выполнения задачи

# Шаги для выполнения задачи

Требуется следовать следующей инструкции для реализации задачи:

[Полная инструкция для реализации задачи](#)

Кратко:

1. Сделать Fork [репозитория](#) с подготовленным [скелетом](#) для реализации задачи.
2. Создать отдельную ветку для работы с репозиторием.
3. Собрать проект с заданием.
4. Реализовать tool, который будет находить в исходном коде приложения все приведения типов в стиле СИ и заменять их на аналог из C++.
5. Отправить решение, создав pull request из вашей ветки в master главного репозитория.

# Дополнительно

1. Дописать код для сохранения результата модификации исходного кода. Залить изменения в свою ветку.
2. Вывести AST для версий входного файла до и после модификации. Сохранить версии в отдельный текстовый файл, указать на измененный участок. Залить файл в свою ветку.
3. Написать вариант реализации задачи с использованием [RefactoringTool](#) класса для замены исходного кода. Залить изменения в свою ветку.
4. Описать преимущества использования [RefactoringTool](#) над [Rewriter](#) (можно комментарием к коду, либо в отдельном файле).

Детали реализации задачи

# Задача

Требуется реализовать tool с помощью интерфейсов Clang, который будет находить в исходном коде приложения все приведения типов в стиле СИ и заменять их на аналог из C++.

Пример:

```
double d = 4.5;  
int i = (int)d; // int i = static_cast<int>(d);
```

# Детали реализации

Потребуется дописать класс [CastCallback](#), отвечающий за действие, которое нужно совершить при нахождении узла [cStyleCastExpr](#) в [AST](#).

В данном случае мы хотим понять тип преобразования и заменить исходный код с помощью [Rewriter](#).

```
class CastCallback : public MatchFinder::MatchCallback {
public:
    CastCallback(Rewriter& rewriter) {
        // Your code goes here
    };

    void run(const MatchFinder::MatchResult &Result) override {
        // Your code goes here
    }
};
```

# Детали реализации

После реализации `CastCallback` требуется пересобирать проект и запустить:

```
$ make -j8
```

```
$ ./c-style-checker ../test/test.cpp --extra-arg=-I/home/<your-root-name>/compiler-course/llvm-project/llvm/build/lib/clang/<version>/include/
```



# Проверка результата

Если реализация сделана правильно, то на экране вы должны увидеть следующее:

```
#include <iostream>
int main() {
    float f;
    int i = static_cast<int>(f);
    return 0;
}
```

Полезные ссылки

# Полезные ссылки

1. [Понимание Clang AST](#)
2. [Руководство по созданию инструментов с использованием LibTooling и LibASTMatchers](#)
3. [Усложненная реализация данной задачи](#)
4. [Матчеры которые есть в clang](#)
5. [Сборка LLVM](#)