

Санкт-Петербургский государственный политехнический
университет
Институт информационных технологий и управления
Кафедра компьютерных систем и программных технологий



ВЫПУСКНАЯ РАБОТА БАКАЛАВРА

Тема: **Разработка модуля расширения статических
проверок для компилятора Clang**

Студент гр. 43501/3 Н.А. Салин

Санкт-Петербургский государственный политехнический
университет
Институт информационных технологий и управления
Кафедра компьютерных систем и программных технологий

Диссертация допущена к защите
зав. кафедрой

_____ В.Ф. Мелехин

«____» _____ 2015 г.

ВЫПУСКНАЯ РАБОТА БАКАЛАВРА

Тема: **Разработка модуля расширения статических
проверок для компилятора Clang**

Направление: 230100 – Информатика и вычислительная техника

Выполнил студент гр. 43501/3

_____ Н.А. Салин

Научный руководитель,
м.т.т., аспирант

_____ М.Х. Ахин

Рецензент,
к. т. н., доц.

_____ Р.Е. Цензент

Консультант по нормоконтролю,
к. т. н., доц.

_____ Р.Е. Вьюер

Эта страница специально оставлена пустой.

РЕФЕРАТ

Отчет, 47 стр., 3 рис., 1 прил.

CLANG, AST, СТАТИЧЕСКИЙ АНАЛИЗ, PLUGIN

Что тут должно быть?

ABSTRACT

Report, 47 pages, 3 figures, 1 appendicies

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	7
1. СТАТИЧЕСКИЕ АНАЛИЗАТОРЫ	9
1.1. Введение	9
1.2. Решаемые задачи	9
1.2.1. Выявление ошибок в программах	9
1.2.2. Рекомендации по оформлению кода	10
1.2.3. Подсчет метрик	10
1.2.4. Поиск уязвимостей	10
1.2.5. Другие	11
1.3. Преимущества и недостатки	11
1.4. Существующие статические анализаторы	13
1.4.1. PVS-Studio	13
1.4.2. Coverity	14
1.4.3. Flint	15
1.4.4. Clang Static Analyzer	16
2. КОМПИЛЯТОР CLANG И МОДУЛИ ДЛЯ НЕГО	17
2.1. О Clang	17
2.2. Абстрактное синтаксическое дерево	18
2.3. Структура абстрактного синтаксического дерева Clang	19
2.4. Пример абстрактного синтаксического дерева	20
2.5. Способы использования Clang	25
2.6. RecursiveASTVisitor и ASTMatcher	27
3. РАЗРАБОТКА МОДУЛЯ ДЛЯ CLANG	33
3.1. Список доступных проверок	33
3.1.1. Одиночные условия	33
3.1.2. Одинаковые ветки if-else	33
3.1.3. Одинаковые условия в if-else-if	33
3.1.4. Ошибки использования функции memset	34
3.1.5. Неверный размер для выделения памяти под строку	35
3.1.6. Опечатка при использовании strlen	35
3.1.7. Потенциальная ошибка при использовании strcmp в условии	35

3.1.8.	Одинаковые аргументы функций strcmp, strncmp, memmove, memcpy, strstr	36
3.1.9.	Выражение внутри sizeof	36
3.1.10.	Перемножение операторов sizeof	36
3.1.11.	Выделение памяти для одного простого типа с инициализацией	37
3.1.12.	Сравнение указателя и 0	37
3.2.	Диаграмма классов	39
3.3.	Динамическое подключение проверок	39
3.4.	Добавление новых проверок	41
4.	ТЕСТИРОВАНИЕ И АНАЛИЗ РАЗРАБОТАННОГО МОДУЛЯ	43
4.1.	Тестирование	43
4.2.	Анализ	43
	ЗАКЛЮЧЕНИЕ	45
	ПРИЛОЖЕНИЕ А. ЛИСТИНГИ	47

ВВЕДЕНИЕ

Описать что такое ошибки в коде, какие существуют средства нахождения ошибок.

Рассказать про Therac-25, Agiane 5 и падение вертолета 1994

Рассказать что в данной работе будет использоваться статический анализ.

1. СТАТИЧЕСКИЕ АНАЛИЗАТОРЫ

1.1. Введение

Статический анализ кода - процесс выявления ошибок и недочетов в исходном коде без реального выполнения исследуемой программы. В большинстве случаев анализ производится над какой-либо версией исходного кода, однако анализу могут подвергаться объектный код. В основном данный термин применяют к анализу, который производится автоматизированно, с использованием специального программного обеспечения. В настоящее время синтаксические анализаторы являются частью всех современных компиляторов. Найденные подозрительные участки кода компилятор выдает в виде предупреждений (warnings). Однако для того чтобы провести более детальный анализа необходимо больше процессорного времени и памяти. Поскольку компиляция должна происходить быстро, в компиляторах используются проверки только на простые ошибки без полноценного анализа. Поэтому для выявления более сложных ошибок в исходном коде, необходимо использовать специализированные инструменты для статического анализа кода.

В настоящее время статический анализ широко используется в областях:

- ПО для медицинских устройств
- ПО для ядерных станций и систем защиты реактора
- ПО для авиации

1.2. Решаемые задачи

1.2.1. Выявление ошибок в программах

Статический анализ в большинстве случаев используется для выявления ошибок в исходном коде.

test code

1.2.2. Рекомендации по оформлению кода

Стив МакКоннелл (Steve McConnell) сказал в своём выступлении на SD West '04: "код должен удобно читаться, а не удобно писаться". Граматное оформление кода крайне важное условие для людей, работающих в одной команде. Следует помнить, что стиль и соглашения о форматировании у каждой организации свои. Обычно эти соглашения определяются на ранней стадии создания проекта. В результате соблюдения принятых соглашений, код будет легче читаться и станет более единообразным и, как следствие, уменьшется стоимость поддержки и отладки существующего кода. Статический анализ для решения данной задачи применяют как для контроля уже давно работающих сотрудников, так и для обучения новых, которые недостаточно хорошо знакомы с принятыми соглашениями.

1.2.3. Подсчет метрик

Метрики кода представляют собой набор оценок программного обеспечения, которые дают разработчикам более глубокое представление о разрабатываемом коде. Так к примеру с помощью метрик кода разработчики могут оценить сложность разработанного ПО, понять какие места необходимо переработать или более тщательно протестировать. Важно помнить, что метрики кода не являются основным показателем оценки кода. К примеру если программист написал малое количество строк кода, это не значит, что он менее эффективен, чем программист, написавший большее количество кода.

1.2.4. Поиск уязвимостей

Статический анализ можно успешно применять для поиска уязвимостей в ПО (Static Application Security Testing, SAST). Однако не только разработчики программы могут использовать статический анализ. В случае, если исходный код находится в открытом доступе или был украден, злоумышленники могут использовать статический анализ для нахождения потенциальных уязвимостей. Вместо того чтобы искать слабые места программы вслепую или просматривать огромное количество исходного кода, куда проще провести статический анализ. Затем после нахождения слабых мест, злоумышленник может начать исследование слабых мест для атаки. Хотя в большинстве случаев хакеру не доступен исходный код, все равно имеется воз-

возможность проведения статического анализа с помощью специальных анализаторов, которые работают с двоичным кодом.

1.2.5. Другие

Перечисленные выше задачи, не единственное применение для статического анализа. К примеру при написании кроссплатформенного кода довольно не просто учесть все особенности каждой ОС или аппаратной платформы. На помощь в данном случае могут прийти специализированные анализаторы, которые смогут найти большинство небезопасных участков кода. Так же статический анализ можно использовать в обучении. Чтобы проверить работу ученика на ошибки не нужно просматривать весь вручную, взамен можно автоматизировать проверку всех работ сразу. Это очень удобно когда у преподавателя много учеников и на ручную проверку всех работ уйдет много времени.

1.3. Преимущества и недостатки

Как и у любого инструмента, статический анализ не является универсальным инструментом и имеет как преимущества, так и недостатки. К преимуществам можно отнести:

- Полное покрытие кода. В отличие от динамического анализа, статические анализаторы проверяют даже те участки кода, которые получают управление крайне редко.
- Независимость от компилятора и среды выполнения. В виду того, что статический анализ производится над исходным кодом без реального выполнения, появляется возможность находить скрытые ошибки. Данные ошибки могут появляться в определенных ситуациях и зависеть от реализации компилятора или заданных ключей компилятора. Примером скрытых ошибок могут служить ошибки "неопределенного поведения"(англ. undefined behaviour).
- Использование на ранних этапах жизненного цикла ПО. Основным преимуществом является возможность нахождения программных ошибок в программе на раннем этапе. Вследствии чего происходит существенное уменьшение стоимости исправления

ошибки. Так же данное преимущество особенно удобно для проектов больших встраиваемых систем, в случае если невозможно использовать средства динамического анализа до тех пор, пока ПО не будет готово к запуску на целевой системе.

- Низкие стоимостные затраты. Для того, чтобы использовать статический анализ, не нужно тратить дополнительные средства на создание тестовых програм или фиктивных модулей (stubs).

Недостатки:

- Статический анализ слаб в выявлении утечек памяти и параллельных ошибок. Для диагностирования данных ошибок необходимо очень много памяти и процессорного времени. Помимо ресурсных затрат, такой алгоритм крайне сложно реализовать, поэтому в большинстве случаев диагностируются только простейшие случаи.
- Ложные срабатывания. При поиске ошибок статический анализатор пытается предсказать поведение программы используя ее исходный код. Это приводит к тому, что происходит множество ложных срабатываний. В большинстве случаев подозрительный участок кода будет компилироваться и работать верно. Определить является ли подозрительный участок кода ошибкой или нет может только человек. В больших проектах соотношение ложных срабатываний и действительных ошибок может быть 100:1. Так ложные срабатыванию могут отвлекать от участков кода, где есть реальная ошибка.
- Скорость. Поскольку для нахождения потенциальных ошибок необходимо проводить детальный анализ каждого участка кода, время анализа может занимать значительное время. Для больших проектов скорость работы исчисляется в часах, поэтому обычно статический анализ используется при "ночных сборках".
- Статические анализаторы полностью не заменяют ручной аудит кода. Такие категории дефектов, как логические, архитектурные уязвимости и проблемы с производительностью, могут быть обнаружены только экспертом.

1.4. Существующие статические анализаторы

Существует множество статических анализаторов кода, у каждого есть свои плюсы и минусы. Ниже будут рассмотрены некоторые из существующих анализаторов. С большим списком синтаксических анализаторов можно ознакомиться на сайте Wikipedia.

1.4.1. PVS-Studio

PVS-Studio это статический анализатор для языка C/C++ отечественной разработки. В нем можно выделить 4 набора правил для диагностики ошибок:

1. Диагностика общего назначения
2. Диагностика возможных оптимизаций
3. Диагностика 64-битных ошибок (Viva64)
4. Диагностика параллельных ошибок (VivaMP)

Подробный список осуществляемых проверок можно найти на сайте PVS-Studio.

Одним из главных преимуществ является интеграция с Microsoft Visual Studio начиная с версии 2005. При этом анализатор поставляется в виде плагина и для начала работы с анализатором не нужно проводить настройку. Для удобства использования предоставляется пользовательский интерфейс для навигации по коду, анализа файлов и получения справочной информации. Также разработчиками реализованна работа анализатора на всех ядрах и процессорах, что существенно ускоряет процесс нахождения ошибок.

Однако из плюсов PVS-Studio вытекают и недостатки. Так как анализатору для работы необходима Visual Studio, то недостатками является:

- Анализ можно производить только используя операционную систему Windows
- Для возможности использования плагинов не подходит бесплатная версия Visual Studio (Express)

Хотя на сайте разработчиков есть описание того, как можно произвести запуск PVS-Studio на Linux, но данный запуск является только как эксперимент, а не полноценная рабочая версия.

Для демонстрации возможностей PVS-Studio разработчики постоянно проводят проверки open source проектов. Со списком уже проверенных проектов и найденными ошибками можно ознакомиться на официальном сайте.

1.4.2. Coverity

Это серьезный набор программ, который используется для выявления и исправления дефектов безопасности и качества в программах критического назначения. Одной из программ выявления дефектов является статический анализатор Coverity Code Advisor умеющий находить дефекты и уязвимости в исходном коде, написанном на языке C, C++, Java и C#.

Отличительной особенностью Coverity является минимальное количество ложных срабатываний и большинство из выдаваемых предупреждений действительно соответствует дефектным участкам кода, которые в последствии могут привести к серьезным ошибкам.

Coverity Code Advisor может выявлять следующие дефекты:

1. Ошибки использования API
2. Переполнение буфера
3. Неправильная обработка ошибок
4. Переполнения целлых типов данных
5. Проблемы с производительностью
6. Неинициализированные переменные
7. SQL инъекции
8. Разименование нулевого указателя
9. Другие

Недостатком данного анализатора является высокая цена, однако есть бесплатный онлайн сервис Coverity Scan доступный для зарегистрированных open source проектов. Данный сервис начал сотрудничать со Стенфордским университетом с 2006 года и в течении первого года было обнаружено и устранено более 6000 программных ошибок в 50 C/C++ open source проектах.

Статический анализатор Coverity использовали в ЦЕРНе для анализа програм, использующихся в большом адронном коллайдере. В результате было найдено и устранено более 40000 программных ошибок, которые могли повлиять на точность физических исследований частиц. Так же статический анализ широко использовался во время разработки программного обеспечения для полета марсохода Curiosity.

1.4.3. Flint

Flint - это статический анализатор от компании Facebook с открытым исходным кодом предназначенный для анализа програм написанных на C++. Изначально Flint был написан на языке C++, но затем был переписан на язык D, в следствии чего ускорилось время анализа исходного кода.

Так как Flint разрабатывался для внутренних нужд компании, в качестве доступных проверок используются только часто встречающиеся ошибки:

1. Запрет на идентификаторы из черного списка
2. Инициализация переменной сама собой
3. Все исключения должны передаваться по ссылке
4. Ошибки при определении конструктора
5. Передача небольших типов по значению
6. Другие

Хотя Flint не сможет найти большинство ошибок как PVS-Studio или Coverity, большим достоинством является открытость кода. Если необходимо добавить собственную проверку, это легко сделать.

1.4.4. Clang Static Analyzer

Clang Static Analyzer является инструментом для статического анализа, который находит ошибки для программ на языке C, C++ и Objective-C. В данный момент имеется возможность использовать статический анализатор как отдельную программу или в интегрированной среде разработки XCode. Отдельная программа запускается из командной строки и разработана для запуска вместе с компиляцией исходного кода. Весь исходный код анализатора открыт и является частью проекта Clang. Анализатор реализован как C++ библиотека, которая может быть использована другими инструментами и приложениями.

В настоящее время ведется активная разработка и производятся постоянные улучшения как для повышения точности, так и для увеличения возможностей алгоритмов анализа.

Доступные проверки разделены на 6 категорий:

1. Базовые проверки. Производятся проверки общего назначения, такие как деление на 0, разыменование NULL указателя, использование неинициализированных переменных и так далее
2. C++ проверки. Сюда входят проверки, специфичные для языка C++
3. Проверки для нахождения неиспользуемого кода.
4. OS X проверки. Данные проверки необходимы для нахождения ошибок, специфичных для языка Objective-C и ошибок неправильного использования Apple SDK (OS X и iOS)
5. Проверки безопасности. Используются для выявления небезопасного использования API. Производятся на основе CERT Secure Coding Standards
6. Unix проверки. Необходимы для проверки корректного использования Unix и POSIX API

С полным списком доступных проверок можно ознакомиться на сайте анализатора.

2. КОМПИЛЯТОР CLANG И МОДУЛИ ДЛЯ НЕГО

2.1. О Clang

Clang является компилятором для языков программирования C, C++, Objective-C, Objective-C++ и OpenCL. Для оптимизации исходного кода и затем кодогенерации используется фреймворк LLVM. Хотя Clang разрабатывается как фронтенд для LLVM, теоретически возможно использовать и другие бэкенды. К примеру в качестве бэкенда можно использовать GCC. Однако комбинация Clang и LLVM предоставляет набор инструментов, позволяющий полностью заменить GCC.

Разработка Clang началась ввиду того, что был необходим компилятор, который предоставлял бы детальные диагностики ошибок, удобную интеграцию с интегрированной средой разработки и имел бы лицензию, позволяющую использовать компилятор в коммерческих целях. В результате в 2007 году был представлен компилятор Clang под лицензий University of Illinois/NCSA Open Source License. Для интегрированной среды разработки XCode, Clang становится основным компилятором начиная с версии 3.2. С ноября 2012 года в качестве основного компилятора для FreeBSD используется Clang. Вероятно в ближайшие годы Clang сменит GCC и будет основным компилятором для языков C и C++ во многих дистрибутивах и других Unix-подобных системах.

Одной из главных целей Clang является поддержка инкрементной компиляции, которая позволяет более тесно интегрировать компилятор и графический интерфейс среды разработки. Для GCC это непросто сделать так как он разрабатывался для использования в классическом цикле "компиляция-линковка-отладка". Поэтому Clang стремится предоставить фреймворк, позволяющий производить парсинг, индексацию, статический анализ и компиляцию языков семейства Си.

Ввиду того, что изначально Clang был спроектирован для максимального сохранения информации во время процесса компиляции, появляется возможность предоставлять контекстно-ориентированные сообщения об ошибках. Такие сообщения будут понятны программистам и удобны для сред разработки. Так же благодаря модульности

дизайна компилятора, можно использовать необходимые модули в среде разработки для индексирования кода, подсветки синтаксиса и рефакторинга.

Подводя итог выше сказанного, можно выделить основные цели использования Clang:

Особенности для конечного пользователя:

- Быстрая компиляция и небольшое использование памяти
- Понятные диагностики
- Совместимость с GCC

Преимущества для приложений:

- Модульная архитектура
- Поддержка разнообразных инструментов (рефакторинг, статический анализ, генерация кода и так далее)
- Тесная интеграция с средой разработки
- Использование BSD-подобной лицензии

Внутренний дизайн и реализация:

- Простой и легко изменяемый исходный код
- Унифицированный парсер для языков C, Objective-C, C++ и Objective-C++
- Поддержка последних версий C/C++/Objective-C

С полным и подробным списком всех особенностей можно ознакомиться на сайте Clang.

2.2. Абстрактное синтаксическое дерево

Абстрактным синтаксическим деревом (Abstract Syntax Tree) называется представление исходного кода в виде дерева, в котором вершинами являются операторы языка программирования, а листья - операндами. Листья могут представлять только переменные и константы.

Так как Clang не производит упрощения исходного кода во время компиляции (как это делает GCC), то гарантируется точное воспроизведение исходного кода в абстрактном синтаксическом дереве, что необходимо для полноценного статического анализа. К примеру скобочные выражения и константы времени компиляции доступны в исходной форме в абстрактном синтаксическом дереве.

2.3. Структура абстрактного синтаксического дерева Clang

Узлы в абстрактном синтаксическом дереве Clang организованы так, что у них нет общего предка. Вместо этого, есть несколько больших иерархий классов для простых типов, таких как Decl и Stmt.

- Decl представляет объявления (declarations). От данного класса наследуются разнообразные объявления типов. К примеру класс FunctionDecl отвечает за объявление функции, а класс ParmVarDecl за объявление параметра функции.
- Stmt представляет операторы (statements). Все классы, описывающие операторы, наследуются от данного класса. К примеру для оператора "if" существует класс IfStmt, оператору "return" соответствует класс ReturnStmt.

Многие важные AST узлы наследуются от классов Type, Decl, DeclContext или Stmt, а некоторые классы наследуются от Decl и DeclContext. В Clang AST выражения (expressions) представляются классом Expr, наследуются от класса Stmt и являются так же операторами. Однако существует множество классов, которые не являются частью большой иерархии классов и доступны только из определенных узлов, к примеру класс CXXBaseSpecifier. Так же важно помнить, что комментарии не входят в состав абстрактного синтаксического дерева.

Все информация о абстрактном синтаксическом дереве для единицы трансляции (translation unit) собрана в классе ASTContext. Он предоставляет обход всей единицы трансляции или доступ к таблице идентификаторов Clang для разобранный единицы трансляции.

2.4. Пример абстрактного синтаксического дерева

Рассмотрим пример генерации абстрактного синтаксического дерева для следующего кода:

```
int gVar=0;

void func(int arg)
{
    if (arg == 0)
        arg++;

    for (int i=0; i<arg; i++)
        gVar++;
}

int main(int argc, char *argv[])
{
    int arg = 2;
    func(arg);
    return 0;
}
```

Для получения абстрактного синтаксического дерева в текстовом виде у Clang есть встроенный режим AST-dump, который может быть включен с помощью флага ast-dump". Так же отключем кодогенерацию флагом fsyntax-only". В результате для получения AST необходимо ввести в терминал следующую команду:

```
clang -Xclang -ast-dump -fsyntax-only ast.c
```

где ast.c - файл с исходным кодом.

В коноль выведется текстовое представление абстрактного синтаксического дерева для приведенного выше исходного кода:

```
TranslationUnitDecl 0x7f8383830ec0 <<invalid sloc>> <invalid sloc>
... внутренние объявления clang ...
|-VarDecl 0x7fc5f98333d0 <ast.c:1:1, col:10> col:5 used gVar 'int' cinit
| '-IntegerLiteral 0x7fc5f9833428 <col:10> 'int' 0
|-FunctionDecl 0x7fc5f9833520 <line:3:1, line:10:1> line:3:6 used func 'void (int)'
| |-ParmVarDecl 0x7fc5f9833460 <col:11, col:15> col:15 used arg 'int'
| |-CompoundStmt 0x7fc5f986fef8 <line:4:1, line:10:1>
| | |-IfStmt 0x7fc5f9833698 <line:5:2, line:6:6>
| | | |-<<NULL>>
| | | |-BinaryOperator 0x7fc5f9833628 <line:5:6, col:13> 'int' '=='
| | | | |-ImplicitCastExpr 0x7fc5f9833610 <col:6> 'int' <LValueToRValue>
| | | | | '-DeclRefExpr 0x7fc5f98335c8 <col:6> 'int' lvalue ParmVar 0x7fc5f9833460
'arg', 'int'
```

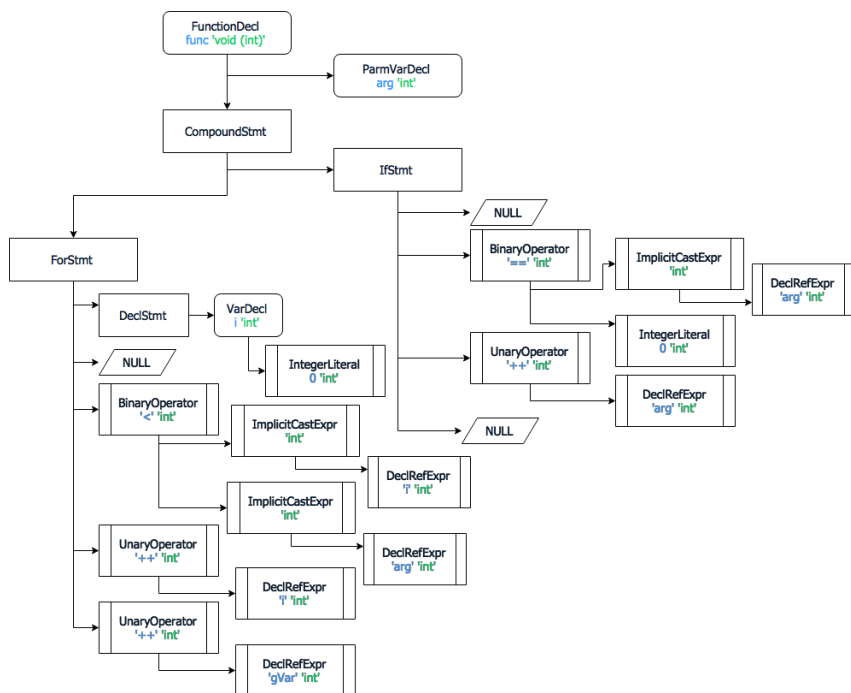



Рисунок 2.1. Clang AST для объявления функции func

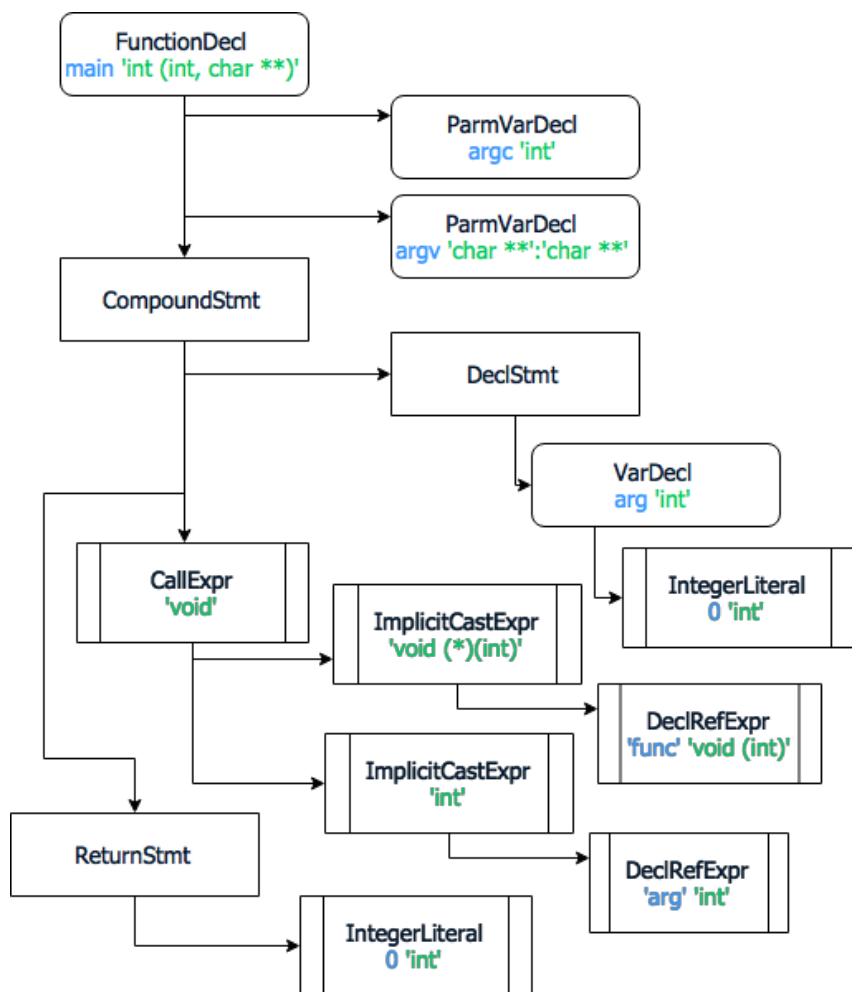


Рисунок 2.2. Clang AST для объявления функции main

Decl

Рассмотрим примеры использования типа Decl. Так, для объявления функции, корнем в абстрактном синтаксическом дереве должна быть вершина с типом Decl. Из рисунков 2.1 и 2.2 видно, что корне-

вая вершина имеет тип `FunctionDecl`, который является подклассом `Decl`. Данный класс отвечает за объявление функции. Если у функции есть параметры, то класс `FunctionDecl` содержит вершины с типом `ParmVarDecl`, которые отвечают за объявление этих параметров. На рисунке 2.1 функции "func" всего один параметр (arg) и поэтому одна вершина `ParmVarDecl`, в то время как у функции "main" (рисунок 2.2) два параметра и соответственно две вершины `ParmVarDecl`. Для объявления локальной или глобальной переменной используется `VarDecl`. На рисунке 2.1 видно, что для цикла `for` объявляется переменная "i" типа `int`, а на рисунке 2.2 `VarDecl` используется для объявления локальной переменной "arg". Так же из текстового представления абстрактного синтаксического дерева, приведенного выше, видно, что для объявления глобальной переменной "gVar" аналогично используется `VarDecl`. В случае, если у объявляемой переменной есть начальное значение, то `VarDecl` имеет дочернюю вершину с этим значением. В приведенных примерах дочерней вершиной является вершина с типом `IntegerLiteral`, которая содержит начальное значение.

Stmt

Как уже было рассмотрено ранее, `Stmt` представляет операторы. На рисунках 2.1 и 2.2 подклассами `Stmt` являются:

- `CompoundStmt` представляет группу операторов заключенных в фигурные скобки
- `DeclStmt` необходим для определения локальной переменной
- `ReturnStmt` соответствует оператору "return"
- `IfStmt` соответствует оператору "if"
- `ForStmt` соответствует оператору "for"

`Expr` представляет выражения и тоже является подклассом `Stmt`. На рисунках 2.1 и 2.2 подклассами `Expr` являются:

- `CallExpr` соответствует вызову функции
- `ImplicitCastExpr` необходим для неявного приведения типов
- `DeclRefExpr` используется для ссылки на объявленные переменные и функции

- `IntegerLiteral` для целых литерал
- `UnaryOperator` и `BinaryOperator` обозначает унарный и бинарный оператор соответственно

Из абстрактного синтаксического дерева видно, что у оператора может быть несколько дочерних вершин, которые содержат дополнительную информацию. К примеру у выражения `CallExpr` может быть несколько вершин: первая соответствует ссылке на вызываемую функцию, а остальные являются параметрами этой функции. У `BinaryOperator` всегда два операнда: левая и правая часть выражения. В отличие от операторов (`Stmt`) у выражений (`Expr`) есть возвращаемый тип.

2.5. Способы использования Clang

Clang предоставляет необходимую инфраструктуру для создания инструментов, которым необходима синтаксическая и семантическая информация из исходного кода. Для того, чтобы использовать все возможности Clang есть три способа создания инструментов: использование библиотеки `LibClang`, использование библиотеки `LibTooling` или создание плагина для Clang. Рассмотрим каждый из методов.

LibClang

`LibClang` представляет стабильный высокоуровневый интерфейс к Clang, написанный на языке C. Является хорошим выбором если необходим стабильный API, так как Clang периодически меняется. В случае если использовать `LibTooling` или создавать плагин, то нужно обновлять написанный код, чтобы он соответствовал изменениям в Clang. Так же `LibClang` позволяет использовать другие языки программирования для доступа к Clang API, а не только C++. Однако `LibClang` не предоставляет полного доступа к абстрактному синтаксическому дереву, только высокоуровневый доступ.

LibTooling

`LibTooling` это C++ интерфейс, используемый для написания отдельного приложения. Типовые примеры использования `LibTooling`:

- Простой контролер синтаксиса
- Инструменты рефакторинга

В отличие от LibClang, предоставляется полный доступ к абстрактному синтаксическому дереву. Анализировать можно как один файл, так и заданный набор файлов, независимо от системы сборки.

Примерами использования LibTooling являются Clang Tools. Clang Tools - это коллекция специальных инструментов для разработчика, используемых для автоматизации и улучшения разработки на языках C и C++. Примеры созданных или планируемых инструментов как часть проекта Clang:

- Проверка синтаксиса (clang-check)
- Автоматическое исправление ошибок компиляции (clang-fixit)
- Автоматическое форматирование кода (clang-format)
- Инструменты для миграции кода под новый стандарт языка (clang-modernize)
- Инструменты для рефакторинга

Плагин для Clang

Использование плагина для Clang позволяет производить дополнительные действия при обходе абстрактного синтаксического дерева как часть процесса компиляции. Плагины представляют из себя динамические библиотеки, которые подключаются во время выполнения компилятором. Благодаря динамическому подключению к компилятору, плагин легко интегрировать в среду сборки проекта. Так же плагин имеет возможность останавливать процесс сборки. Как и при использовании LibTooling, плагину предоставляется полный контроль над абстрактным синтаксическим деревом.

Однако стоит помнить, что нецелесообразно создавать плагин, если:

- Необходимо использовать плагин вне среды сборки
- Необходимо использовать плагин для определенного набора файлов в проекте, которые не обязательно вызывают пересборку проекта

2.6. RecursiveASTVisitor и ASTMatcher

Для обхода и нахождения интересных мест в абстрактном синтаксическом дереве Clang есть два подхода: используя RecursiveASTVisitor и с помощью AST Matchers.

Рассмотрим как с помощью каждого подхода найти интересный фрагмент кода. В качестве примера будет рассмотрено нахождение фрагмента кода, содержащий оператор if и в качестве условия происходит сравнение указателя и 0:

```
int main()
{
    int var = 5;
    int *ptr = &var;
    if (ptr < 0) // <==
        var++;
    return 0;
}
```

Абстрактное синтаксическое дерево для интересующего фрагмента кода:

```
-IfStmt 0x7fdb75024790 <line:18:2, line:19:6>
|-<<<NULL>>>
|-BinaryOperator 0x7fdb75024720 <line:18:6, col:12> 'int' '<'
| |-ImplicitCastExpr 0x7fdb750246f0 <col:6> 'int *' <LValueToRValue>
| | |-DeclRefExpr 0x7fdb750246a8 <col:6> 'int *' lvalue Var 0x7fdb750245f0 'ptr' 'int *'
| | |-ImplicitCastExpr 0x7fdb75024708 <col:12> 'int *' <NullToPointer>
| | |-IntegerLiteral 0x7fdb750246d0 <col:12> 'int' '0'
| |-UnaryOperator 0x7fdb75024770 <line:19:3, col:6> 'int *' postfix '++'
| |-DeclRefExpr 0x7fdb75024748 <col:3> 'int *' lvalue Var 0x7fdb750245f0 'ptr' 'int *'
|-<<<NULL>>>
```

RecursiveASTVisitor

Класс RecursiveASTVisitor позволяет обойти каждую вершину абстрактного синтаксического дерева. Для этого необходимо создать класс, который будет отвечать за обход абстрактного синтаксического дерева. Данный класс должен наследоваться от класса RecursiveASTVisitor. Чтобы обойти интересные вершины, достаточно переопределить функцию "bool VisitNodeType(NodeType *)". Так для обхода всех Stmt необходимо переопределить функцию "bool VisitStmt(Stmt *)". Так же можно переопределить функцию для поиска вершин, которые являются подклассами Stmt. К примеру для нахождения оператора "if" нужно переопределить функцию "bool

VisitIfStmt(IfStmt *)". В этом случае для оператора "if" будет вызываться два обработчика: VisitStmt и VisitIfStmt.

Для Visit* функций необходимо возвращать значение "true чтобы продолжить обход остальных вершин абстрактного синтаксического дерева. Если же вернуть значение "false обход дерева полностью прекращается и Clang завершает работу.

Рассмотрим как стоит использовать RecursiveASTVisitor для нахождения участка кода из вышеописанного примера.

Проанализировав полученное для исходного кода абстрактное синтаксическое дерево видно, что корнем для искомого фрагмента является IfStmt. Поэтому в классе, созданном для обхода AST следует переопределить функцию с именем "VisitIfStmt". Так же нетрудно заметить, что условием оператора "if" должен быть оператор сравнения (BinaryOperator), в котором в левой части находится указатель, а в правой константа, равная 0. Между левой и правой частью оператора сравнения должен быть символ «". Основываясь на вышесказанном, можно написать код, для выявления заданного шаблона.

```
class Example:
    public RecursiveASTVisitor<Example>
{
public:
    bool VisitIfStmt(IfStmt *s) {
        if (const BinaryOperator *binOP =
            llvm::dyn_cast<BinaryOperator>(s->getCond()))
        {
            if (binOP->getOpcode() == BO_LT)
            {
                const Expr *LHS = binOP->getLHS();
                if (const ImplicitCastExpr *Cast =
                    llvm::dyn_cast<ImplicitCastExpr>(LHS))
                {
                    LHS = Cast->getSubExpr();
                }

                const Expr *RHS = binOP->getRHS();
                if (const ImplicitCastExpr *Cast =
                    llvm::dyn_cast<ImplicitCastExpr>(RHS))
                {
```

```

        RHS = Cast->getSubExpr();
    }

    if (
const DeclRefExpr *dRef=llvm::dyn_cast<DeclRefExpr>(LHS) &&
const IntegerLiteral *intVal =
        llvm::dyn_cast<IntegerLiteral>(RHS))
    {
        if (const VarDecl *Var =
            llvm::dyn_cast<VarDecl>(dRef->getDecl()))
        {
            if (Var->getType()->isPointerType() &&
                intVal->getValue().getZExtValue() == 0)
            {
                s->dump(); // found
            }
        }
    }
}
}
return true;
}
};

```

Как видно для нахождения простого шаблона нужно написать довольно много кода. Главной проблемой является то, что данный код трудно читать. Чтобы понять, какой шаблон ищет приведенный код, необходимо вчитываться и смотреть, что делает каждая строчка кода. Намного лучше для выявления шаблонов использовать `ASTMatcher`.

ASTMatcher

С недавнего времени в Clang появилась библиотека `ASTMatcher`, которая предоставляет простой, мощный и краткий способ определенных шаблонов в абстрактном синтаксическом дереве. Реализованные как предметно ориентированный язык на основе макрасов и шаблонов, `matchers` похожи на алгебраические типы данных из функциональных языков программирования.

К примеру для исследования только бинарных операторов есть `matcher` который делает именно это и называется `"binaryOperator"`.

Для того, чтобы исследовать только бинарный оператор, левой части у которого используется литерал 0, необходимо написать следующий matcher:

```
binaryOperator (hasLHS (integerLiteral (equals (0))))
```

Данный matcher не будет находить фрагменты кода, где используются другие формы 0, такие как `'\0'` или `NULL`, но matcher будет работать если используется макрос, который раскрывается в 0. Matcher так же не будет срабатывать если используется перегруженный бинарный оператор, так как для этого есть специальный matcher, который называется `"operatorCallExpr"` и используется для обработки перегруженных операторов.

Есть три простых категории для matchers:

1. *Node Matchers*. Используются для поиска в AST по определенному типу. Любой шаблон определяющий выражение должен начинаться с Node Matcher, который затем можно уточнить с помощью Narrowing Matcher или Traversal Matcher. Все Traversal Matcher используют Node Matcher в качестве аргументов. Так же только Node Matcher имеют метод `"bind"` для связывания заданного узла и имени, для того чтобы в дальнейшем можно было получить найденный узел по имени.
2. *Narrowing Matchers*. Используются для поиска по атрибутам вершины абстрактного синтаксического дерева. Есть специальные логические Narrowing Matchers которые позволяют пользователям более мощные выражения.
3. *Traversal Matchers*. Используются для нахождения узла по связям между узлами абстрактного синтаксического дерева. Есть специальные Traversal Matchers которые работают для всех узлов и позволяют пользователям писать более универсальные выражения.

Для полного списка всех доступных AST Matchers смотрите документацию.

Если Matcher описывает сущность в абстрактном синтаксическом дереве Clang и может быть связан с именем, то на него можно получить ссылку когда шаблон найден. Для этого нужно вызвать метод `"bind"` на нужном шаблоне. К примеру следующий шаблон находит все переменные типа `int` и связывает с этой переменной имя `"intVar"`:


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

Для обработки фрагментов кода, удовлетворяющих заданному шаблону необходимо создать класс обработчик. Данный класс должен наследоваться от класса `MatchCallback`. Класс `MatchCallback` является вложенным классом для класса `MatchFinder`. У класса `MatchCallback` есть три функции, которые можно переопределить:

1. **void run(const MatchResult &Result).** Является обязательной для переопределения. Вызов данной функции происходит каждый раз, когда был найден участок кода, соответствующий заданному шаблону
2. **void onStartOfTranslationUnit().** В отличие от функции "void run(const MatchResult &Result)" данная функция не является обязательной для переопределения. Вызывается каждый раз в начале разбора для каждой единицы трансляции
3. **void onEndOfTranslationUnit().** Функция "void onEndOfTranslationUnit()" аналогична ранее рассмотренной функции "void onStartOfTranslationUnit()" с единственным отличием, что вызывается каждый раз в конце разбора для каждой единицы трансляции.

Как видно из заголовка функции "run в качестве параметра данной функции передается ссылка на результат класс `MatchResult`. Данный класс необходим для получения всей информации при совпадении абстрактного синтаксического дерева с заданным шаблоном.

За работу по нахождению участков кода, соответствующих заданному шаблону отвечает класс `MatchFinder`. Для регистрации обработчика и шаблона, который данный обработчик будет анализировать необходимо использовать функцию `addMatcher`. Первым параметром данной функции является шаблон который необходимо выявить в абстрактном синтаксическом дереве. Вторым параметром необходимо передать указатель на экземпляр класса обработчика.

Рассмотрим насколько просто нахождение участка кода из вышеописанного примера нахождения сравнения указателя и 0 с использованием `ASTMatcher`.

Как уже было рассмотрено ранее, корнем для искомого фрагмента кода должен быть `IfStmt` обозначающий оператор "if". Затем условием оператора "if" должен быть оператор сравнения

(BinaryOperator), в левой части которого находится указатель, а в правой литерала 0. В результате AST matcher будет выглядеть следующим образом:

```
ifStmt( hasCondition( binaryOperator(
    hasOperatorName("<"),
    hasLHS(ignoringParenImpCasts(declRefExpr(
        to(varDecl(hasType(pointsTo(AnyType))).bind("lhs")))),
    hasRHS(ignoringParenImpCasts(integerLiteral(equals(0))))
    )).bind("if")
```

Обработчик, вызываемый при нахождении данного совпадения выглядит так:

```
class Example:
public MatchFinder::MatchCallback
{
public:
virtual void run(const MatchFinder::MatchResult &Result)
{
const IfStmt *s = Result.Nodes.getNodeAs<clang::IfStmt>("if");
    if (s)
        s->dump();
}
};
```

Как видно необходимо писать намного меньше кода, чем при использовании RecursiveASTVisitor, в следствии чего труднее допустить ошибки. Так же важным преимуществом использования AST Matcher является наглядность. Увидев такой шаблон, можно сразу понять какому исходному коду он соответствует.

3. РАЗРАБОТКА МОДУЛЯ ДЛЯ CLANG

3.1. Список доступных проверок

3.1.1. Одиноквые условия

Часто во время написания кода программист может допустить опечатку и не заметить следующей логической ошибкой: в исходном коде имеется оператор сравнения, логический или побитовый в левой и правой части которого находятся одинаковые выражения.

Рассмотрим пример:

```
if (pos.x==0 && pos.x==0)
```

В данном случае слева и справа от оператора && расположены одинаковые выражения `pos.x==0`, что скорее всего свидетельствует о наличии ошибки, допущенной из-за невнимательности. Корректный код, не вызывающий ошибок должен выглядеть так:

```
if (pos.x==0 && pos.y==0)
```

3.1.2. Одинаковые ветки if-else

Данное предупреждение выдается на фрагмент кода, в котором полностью совпадают истинная и ложная ветка оператора "if". Такую опечатку нетрудно допустить при копировании кода.

Пример:

```
if (isCorrect())  
    flags |= CORRECT;  
else  
    flags |= CORRECT;
```

Скорее всего это неверный код и его стоит пересмотреть.

3.1.3. Одинаковые условия в if-else-if

Данная ошибка часто появляется в результате множественного копирования исходного кода после чего некоторые условия не были исправлены и остались прежними.

Рассмотрим пример:

```

if (x==1)
    Func1 ();
else if (x==2)
    Func2 ();
else if (x==1)
    Func3 ();

```

В приведенном выше коде функция "Func3()" никогда не будет вызвана так как условия "x==1" уже проверялось. Корректный код должен быть:

```

if (x==1)
    Func1 ();
else if (x==2)
    Func2 ();
else if (x==3)
    Func3 ();

```

3.1.4. Ошибки использования функции `memset`

Для обнуления области памяти часто используется функция `memset`. Но из-за невнимательности программист может перепутать местами второй и третий аргумент в результате чего не происходит обнуления:

```
memset(buf, sizeof(buf), 0)
```

Помимо приведенной выше ошибки, часто неопытные специалисты не знают, что третьим аргументом функции `memset` принимается значение в байтах.

Рассмотрим пример:

```

int arr[10];
memset(arr, 0, 10);

```

В данном примере только первые 10 байт массива `arr` будут заполнены 0. Анализатор выдает предупреждения, только в том случае, когда третий аргумент не является кратным размеру типа первого аргумента. Если первым аргументом передается указатель, имеющий тип не `char`, `short`, `int` или `long`, а третьим константа, то выдается предупреждение, что не стоит использовать константу.

3.1.5. Неверный размер для выделения памяти под строку

Часто многие начинающие и даже опытные программист забывают что в C/C++ строки должны заканчиваться терминальным символом '0'.

В результате появляется ошибка недостаточного выделения памяти под строку.

Пример:

```
char * str = (char) malloc(strlen(name));  
strcpy(str, name);
```

В результате выполнения приведенного кода будет выход за предел выделенной памяти. Поэтому необходимо использовать следующий код:

```
char * str = (char) malloc(strlen(name)+1);  
strcpy(str, name);
```

Ошибка так же находится в случае, если для выделения памяти используется оператор **new**

3.1.6. Опечатка при использовании strlen

Как уже описывалось ранее, для хранения строк в C/C++ необходимо не забыть про наличие терминального символа '0'.

Однако иногда из-за невнимательности можно поставить неправильно закрывающуюся скобку:

```
char * str = (char) malloc(strlen(name+1));  
strcpy(str, name);
```

В результате выделяется на двай байта меньше памяти, чем необходимо.

Предупреждение о потенциальной ошибке выдастся только в случае, если в качестве аргумента strlen является выражение сложения, правой частью которого является литерал 1.

3.1.7. Потенциальная ошибка при использовании strcmp в условии

Часто по незнанию начинающие специалисты не знают как работают функции "strcmp" и "strncmp". Данные функции используются

для сравнения двух строк и возвращают 0, если строки совпадают. Рассмотрим пример:

```
if (strcmp(str1 , str2))
```

Условие в операторе if выполняется только в том случае, когда строки не совпадают. Однако человек, редко использующий данные функции может подумать, что условие истинно когда строки совпадают. Хотя приведенный код не содержит ошибку, лучше переписать его следующим образом:

```
if (strcmp(str1 , str2) != 0)
```

3.1.8. Одинаковые аргументы функций strcmp,strncmp,memmove,memcpy,strstr

Часто в программе происходит вызов функции, в которую передается два одинаковых аргумента и для многих функций такая ситуация является нормальной. Однако для таких функций как strcmp,strncmp,memmove,memcpy,strstr такая ситуация является подозрительной.

3.1.9. Выражение внутри sizeof

Выражения написанные внутри оператора sizeof не выполняются, поэтому такой фрагмент кода является подозрительным.

Рассмотрим пример:

```
int i=0;
size_t normalSize = sizeof(i++);
size_t doubleSize = sizeof(i *2);
```

В результате выполнения переменная i не увеличится и останется равной 0, а переменная doubleSize будет равна переменной normalSize, хотя скорее всего в данном случае программист надеялся получить значение в два раза больше.

3.1.10. Перемножение операторов sizeof

Исходный код, в котором происходит перемножение двух операторов sizeof(), практически всегда свидетельствует о наличии ошибки.

Рассмотрим пример:

```
TCHAR buf[256];  
DWORD count = sizeof(buf)*sizeof(TCHAR);
```

В данном коде тип TCHAR определяется во время компиляции и в зависимости от настроек может иметь размер один и более байт. В результате при размере TNCAR больше одного байта, количество элементов буфера будет неверным.

Для корректного определения количество элементов необходимо использовать следующий код:

```
TCHAR buf[256];  
DWORD count = sizeof(buf)/sizeof(TCHAR);
```

3.1.11. Выделение памяти для одного простого типа с инициализацией

Если в коде происходит динамическое создание одного единственного целочисленного объекта с инициализацией, то скорее всего произошла опечатка и программист хотел использовать квадратные скобки вместо круглых.

Пример:

```
int *arr = new int(N);
```

Предупреждение появляется только при попытке создания объекта с типом char, short, int или long.

3.1.12. Сравнение указателя и 0

Анализатор выдает предупреждение при нахождении сравнения указателя и литеры 0 с использованием оператора <.

Пример кода:

```
class ResourceManager  
{  
public:  
    SomeObj* Find(const char * filename);  
    void LoadResource(const char * filename);  
    ...  
} resources;  
...
```

```
if (resources.Find("foo") < 0)
    resources.LoadResource("foo");
```

Приведенный выше код мог легко появиться в процессе рефакторинга если раньше был следующий код:

```
class ResourceManager
{
public:
    int Find(const char * filename);
    SomeObj* Get(int index);
    void LoadResource(const char * filename);
    ...
} resources;
...
if (resources.Find("foo") < 0)
    resources.LoadResource("foo");
```

Метод "Find" изначально возвращала индекс, по которому можно было получить ресурс, если же ресурс не был найден, возвращалось -1. Но после рефакторинга метод "Find" стал возвращать указатель на ресурс или 0, в результате метод LoadResource никогда не будет вызван.

3.2. Диаграмма классов

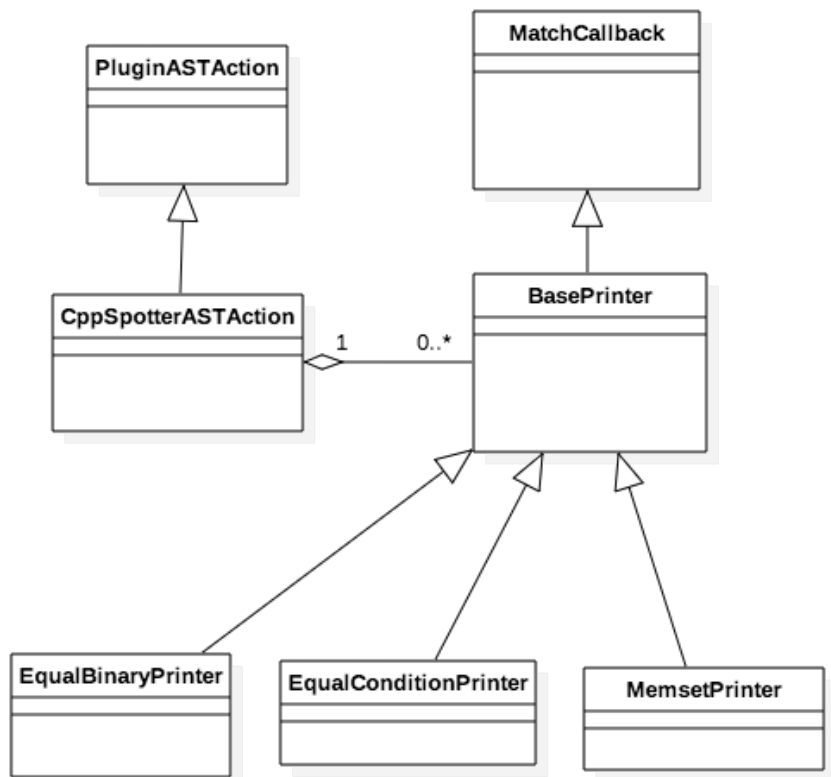


Рисунок 3.1. UML диаграмма классов

Тут стоит описать как относятся классы друг к другу?

3.3. Динамическое подключение проверок

Невсегда удобно использовать все доступные проверки. В случае если какая-то проверка на ошибки выдает слишком много ложных предупреждений, ее лучше отключить и сосредоточиться на более важных ошибках.

Для включения нужных проверок необходимо использовать следующие флаги:

- **-eqBin** Одинаковые условия, см. 3.1.1
- **-eqStmt** Одинаковые ветки if-else, см. 3.1.2
- **-eqCond** Одинаковые условия в if-else-if, см. 3.1.3
- **-memset** Неверное использование memset, см. 3.1.4
- **-allocStr** Неверный размер для выделения памяти под строку, см. 3.1.5
- **-strlen** Опечатка при использовании strlen, см. 3.1.6
- **-strcmp** Потенциальная ошибка strcmp, см. 3.1.7
- **-eqArgs** Одинаковые аргументы функций, см. 3.1.8
- **-sizeof** Выражение внутри sizeof, см. 3.1.9
- **-sizeofMul** Перемножение операторов sizeof, см. 3.1.10
- **-new** Выделение памяти для одного простого типа, см. 3.1.11
- **-ptrCmp** Сравнение указателя и 0, см. 3.1.12
- **-all** Включение всех вышеперечисленных проверок

Для передачи каждого флага в плагин необходимо использовать следующий формат:

```
—plugin—arg—<имя плагина> <флаг>
```

Где <имя плагина> соответствует имени плагина, которое задается в исходном коде при регистрации плагина, а <флаг> соответствует флагу, который получит плагин. Так для того, чтобы включить проверки одинаковых условий (3.1.1) и сравнения указателя с 0 (3.1.12), нужно передать Clang такие ключи:

```
—plugin—arg—CppSpotter —eqBin \  
—plugin—arg—CppSpotter —ptrCmp
```

3.4. Добавление новых проверок

Добавление новой проверки происходит в три этапа:

1. Создание класса обработчика. Данный класс необходим для обработки подозрительных мест в исходном коде и в случае нахождения ошибки отвечает за отображение предупреждения. Созданный класс должен наследоваться от класса `BasePrinter` и переопределять функцию

```
void addToFinder(ast_matchers::MatchFinder * finder)
```

Эта функция отвечает за регистрацию созданного обработчика.

2. Создание `AST Matcher` для выявления узлов абстрактного синтаксического дерева, потенциально содержащих ошибки. Созданный `AST Matcher` используется при регистрации обработчика в функции **`addToFinder`**.
3. Последним этапом является добавление нового флага для созданной проверки в функции **`ParseArgs`**. Если пользователь передал флаг на включение новой проверки, необходимо добавить данную проверку в список используемых проверок с помощью функции **`addPrinter`**.

4. ТЕСТИРОВАНИЕ И АНАЛИЗ РАЗРАБОТАННОГО МОДУЛЯ

4.1. Тестирование

Показать нахождение ошибок.

Список ложных срабатываний:

- `*++scan == *++match && *++scan == *++match`
- Диагностическое сообщение не выдается, если сравниваются два идентичных выражения типа `float` или `double`. Такое сравнение позволяет определить, является ли значение `NaN`. Пример кода, реализующего подобную проверку:

```
bool isnan(double X) { return X != X; }
```

- Наличие двух пустых веток считается корректной и безопасной ситуацией. Подробные конструкции можно часто встретить при использовании макросов. Пример безопасного кода:

```
if (exp) {  
} else {  
}
```

- Функция `foo` каждый вызов меняет свое состояние или глобальные переменные

```
if (foo() || foo())
```

- `strlen(text+1)` так и задумывалось
- `memset`

Список найденных ошибок:

4.2. Анализ

Тут будет сравнение скорости компиляции с плагином, без него и с частично отключенными проверками.

ЗАКЛЮЧЕНИЕ

Заключение

ПРИЛОЖЕНИЕ А

ЛИСТИНГИ