**Вступление:**

Добрый вечер уважаемые коллеги!

Я являюсь студентом курса OTUS «Программист С» группы OTUS-C-2023-07.

Меня хорошо видно и слышно?

Меня зовут Богов Сергей. Я работаю инженером-программистом 1 категории группы аппаратных средств отдела СКЗИ и сертификации.

Хочу представить вашему вниманию защиту проектной работы по теме «Построение графа вызовов функций для файла с исходным кодом на языке С».

**Предисловие:**

Наша компания занимается производством и продажей программных и программно-аппаратных средств СКЗИ. В текущий момент одним из новых проектов является встраивание нашего программного продукта в аппаратные средства различных производителей, их сертификация как ПАК с СКЗИ и выведение данной сертифицированной продукции на Российский рынок.

**Описание целей проекта:**

В рамках данного проекта производится оценка встраивания нашего программного продукта в прикладное ПО производителей (1). В ходе проведения оценки одной из задач является анализ встраивания и применения библиотечных функций нашего продукта в общем архитектурном решении прикладного ПО (2). Для этого производится определение структуры вызовов библиотечных функций на фоне общего архитектурного решения (3) прикладного ПО. Полученные результаты анализируются, по ним выполняется описание встраивания функционала нашего продукта в процессы прикладного ПО, в дальнейшем оценивается корректность или некорректность применения это функционала при выполнении тех или иных задач при работе устройства в целом, а также безопасность использования данного функционала.

В ходе проведения данных работ возникла потребность отображать результаты, полученные при определении структуры вызовов библиотечных функций, в удобном для восприятия графическом виде (4). Изначально данные результаты получались при ручном анализе кода и было принято решение отражать их в виде некой блок-схемы. Однако их получение процесс не всегда простой и порой весьма трудоемкий. Поэтом было принято решение искать средства автоматизации для упрощения проведения и получения визуализированных результатов анализа, в целях сокращения времени и трудозатрат (5).

**Этапы планирования проекта:**

Сначала планировалось использовать какое-то готовое решение, которое решало бы имеющуюся задачу. Одним из таких знакомых нам решений был Doxigen (1). Однако попытки его использования не принесли желаемого результата. Далее были попытки найти в открытом доступе проекты решающие похожие задачи и приспособить их под свои нужды (2). Но пришли к такому мнению, что поиск, анализ и переделывание чужого решения не лучший вариант и может оказаться весьма непростым, да и безопасность применения готовых или чужих решений может оказаться под вопросом. Поэтому пришли к мнению, что необходимо попробовать создать собственное решение для выполнения имеющейся задачи (3). Для этого необходимо было подобрать мощный, доступный, универсальный инструмент для парсинга исходного кода (4), а также гибкий и доступный инструмент, применяемый для визуализации результатов парсинга (5). Затраты на реализацию собственного решения должны быть сопоставимы с его применением в рабочих целях и не занимать много времени (6).

**Применяемые инструменты:**

При создании проекта применялись следующие инструменты:

1. Редактор исходного кода Visual Studio Code
2. Компилятор clang
3. Библиотека libclang для генерации и анализа абстрактного синтаксического дерева (AST).
4. llvm - для подключения библиотеки libclang
5. Graphviz — пакет утилит для автоматической визуализации графов, заданных в виде описания на языке DOT.
6. CMake — кроссплатформенное программное средство автоматизации сборки программного обеспечения из исходного кода.
7. Make — компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

**Описание структуры проекта:**

1. build - директория сборки проекта;
2. graph\_image - директория содержащая конечный граф вызовов;
3. output\_graphviz - директория содержащая выходной файл .dot формата Graphviz;
4. src - директория с исходным кодом программы;
5. tests - директория с тестовыми анализируемыми файлами;
6. CMakeLists.txt - файл сборки;

Ссылка на репозиторий с исходными кодами:

<https://github.com/Sbogov/call_graph_with_graphviz.git>

**Компиляция проекта:**

Сборка и все команды выполняются из директории /build

Предусмотрена отладочная и релизная версии. В отладочной версии дополнительно в консоль будет выводиться лог обхода абстрактного синтаксического дерева (AST).

Флаги компиляции:

1. cmake -DCMAKE\_BUILD\_TYPE=Debug ..
2. cmake -DCMAKE\_BUILD\_TYPE=Release ..

**Сборка и запуск проекта:**

1. Сборка проекта осуществляется командой make.
2. Все управление осуществляется из консоли
3. При запуске программа на вход принимает аргументы командной строки – путь к файлу, анализ которого необходимо провести.
4. После выполнения анализа указанного файла в директории /output\_graphviz создается файл с описанием на языке DOT, необходимый для построения графа.

**Построение графа:**

Построение графа выполняется командой make graph. В результате в директории /graph\_image появится отрисованный с помощью Graphviz граф вызовов функций, в консоли отобразится содержимое файла .dot, на экране появится построенный граф (при наличии используемого в проекте средства для просмотра изображений eog).

**Тестирование и отладка:**

Возможен запуск проекта на имеющихся в структуре проекта тестовых файлах (make test1, make test2). При запуске проекта на тестовых файлах предусмотрено создание ast-dump (абстрактного синтаксического дерева AST), получающегося в результате парсинга исследуемого файла с исходным кодом с помощью библиотеки clang.

Данное дерево так же использовалось для понимания того, как происходит обход AST узлов и какие типы курсоров встречаются в анализируемом файле (чтобы выделить необходимые).

**Описание основных конструкций исходного кода проекта:**

Создаем индекс clang, проверяем корректность создания индекса. Если индекс не создался, выводим сообщение об ошибке, очищаем память, выделенную для создания индекса.

Выполняем парсинг файла с исходным кодом с помощью clang\_parseTranslationUnit. Проверяем успешность парсинга. Если парсинг файла выполнить не удалось выводим сообщение об ошибке, очищаем ресурсы, выделенные для создания единицы парсинга и индекса clang.

Далее создаем и открываем для записи DOT файл. Проверяем создание и открытие файла. Если файл открыть не удалось, выводим сообщение об ошибке и также очищаем ресурсы, выделенные для создания единицы парсинга и индекса clang.

Если файл создался успешно, записываем в него заголовок, необходимый для Graphviz и построения графа.

Далее получаем корневой курсор для созданной единицы парсинга, а также имя нулевого курсора.

Инициализируем структуру для промежуточного хранения данных, очищаем память, выделенную для получения имени нулевого курсора, мы ее передали в структуру, она нам больше не понадобится.

Далее вызываем функцию clang\_visitChildren для обхода AST, в нее передаем полученный корневой курсор созданной единицы парсинга, функцию посетителя для обхода узлов AST, а также нашу структуру для промежуточного хранения данных (о функции посетителя visitorCallback расскажу немного позже).

После завершения обхода дерева AST, закрываем DOT файл, а также очищаем ресурсы, связанные с созданными ранее единицей парсинга и контекстом clang.

Теперь посмотрим на функцию посетителя visitorCallback. Это коллбэк функция которая вызывается при обходе каждого узла AST и имеет такие же аргументы как clang\_visitChildren. В данном случае аргумент visitor –функция посетитель нам не нужен. clientData - указатель, представляющий данные клиента, которые будут передаваться обратным вызовам и посетителям. Приводим этот указатель к типу нашей структуры для временного хранения данных.

Проверяем что наш курсор находится в главном файле, соответствующем единице перевода (парсинга).

При обходе узлов дерева AST, если мы встретим вызов функции, для того чтобы сделать запись в DOT файл для построения графа нам нужен будет не только текущий курсор, который является вызовом функции, но и его родитель. Родительским узлом в данном случае будет являться определение функции, внутри которой встречается вызов других функций, если таковые имеются.

Для этого начав обход узлов дерева AST, если тип текущего курсора будет являться определением функции (CXCursor\_FunctionDecl), то мы будем считать данный курсор родительским узлом для всех встретившихся далее вызовов функций (CXCursor\_CallExpr), пока не встретим новое определение функции для смены родительского узла. Родительский узел мы будем хранить в нашей структуре для промежуточного хранения данных между рекурсивными вызовами visitorCallback в процессе обхода узлов дерева AST.

Далее при обходе узлов AST, если наш текущий курсор — это вызов функции (CXCursor\_CallExpr), то мы вызываем функцию для занесения найденного вызова функции в DOT файл для Graphviz и продолжаем рекурсивный обход узлов дерева AST.

Функция печати printFunctionCall использует макросы для дополнительной печати проходимых узлов дерева AST, показа зафиксированных в процессе обхода родительских улов, и значения родительского и дочернего узла во время вызова функции печати. Также функция заносит все найденные вызовы функции вместе с их родителем в DOT файл в формате Graphviz для дальнейшей отрисовки графа.

В файле CMakeLists.txt задано несколько команд для упрощения работы с проектом. Все команды выполняются из директории сборки **/build**

Для запуска парсинга любого файла необходимо выбрать исполняемый файл проекта и указать путь к анализируемому файлу.

Основной командой является make graph – это построение графа вызовов для проанализированного файла.

Так же возможен запуск парсинга тестовых примеров, для того чтобы ознакомиться с тем, как работает проект.

Возможна так же полная очистка всех результатов сборки и работы проекта командой make clean\_all.

**Выводы и планы по развитию:**

1. Libclang – мощный инструмент для анализа исходного кода, требующий более детального изучения для эффективной работы с ним.
2. AST-dump содержит гораздо больше информации чем RTL (gcc -fdump-rtl-expand) для парсинга кода (здесь можно сделать небольшое отступление про Egypt, если будет время и коллегам будет интересно и сказать пару слов про этот инструмент).
3. Доработка программы для анализа проектов, состоящих из множества файлов с исходным кодом.
4. Поиск и построения графа вызовов только для определенных функций, использующихся в проекте (в рамках рабочих задач).