СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 1](#_Toc197185546)

[ВВЕДЕНИЕ 2](#_Toc197185547)

[ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 3](#_Toc197185548)

[1.1. Обоснование актуальности работы 3](#_Toc197185549)

[1.2. Обзор существующих решений 3](#_Toc197185550)

[1.3. Анализ существующих решений 4](#_Toc197185551)

[QSettings (Qt Core) 5](#_Toc197185552)

[Qt.labs.settings (QML) 5](#_Toc197185553)

[KConfig (KDE Framework) 5](#_Toc197185554)

[Заключение 6](#_Toc197185555)

[ГЛАВА 2. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ К РАЗРАБАТЫВАЕМОЙ СИСТЕМЕ 8](#_Toc197185556)

[2.1. Функциональные требования 8](#_Toc197185557)

[2.2. Нефункциональные требования 8](#_Toc197185558)

[ГЛАВА 3. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ 8](#_Toc197185559)

[ЗАКЛЮЧЕНИЕ 8](#_Toc197185560)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 8](#_Toc197185561)

ВВЕДЕНИЕ

Почти в каждом десктопном приложении в том или ином виде присутствует окно настроек. При разработке таких приложений на C++ и Qml создание таких настроек с нуля является очень трудоёмкой задачей. Если не создать отдельный модуль для этой задачи – то может появиться огромное количество boilerplate кода, который очень сильно затруднит поддержку и развитие в любой области разработки. Поэтому нашей компанией было принято решение создать такую библиотеку, сделать её максимально обобщенной и гибкой в использовании.

ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

Перед тем как на самом деле приступать к задаче, необходимо обратиться к открытым источникам и выяснить, не написал ли кто-нибудь уже такой же или подобный проект чтобы использовать его или хотя бы взять за основу для создания нового.

1.1. Обоснование актуальности работы

Как уже было упомянуто во введении – подобная работа может очень сильно упростить и ускорить разработку приложений и избавить от лишнего написания кода. Описывать QML-интерфейс для каждого параметра и каждый раз писать логику загрузки/сохранения настроек может очень сильно усложнить процесс разработки. Благодаря библиотеке достаточно будет только предоставить json с описанием настроек, не будет необходимости в перекомпиляции и написании своих компонентов. При обнаружении ошибок в коде библиотеки достаточно будет лишь в одном месте их поправить, и исправления применятся во всех проектах которые используют данный модуль.

1.2. Обзор существующих решений

Для начала рассмотрим класс **QSettings**. Класс является частью большого популярного фреймворка Qt. QSettings позволяет сохранять и загружать настройки из ini файлов, реестра Windows и других платформенно зависимых хранилищ. Тем самым мы имеем кроссплатформенный способ манипулирования настроек. Однако есть очень важные замечания, связанные с этим решением. Во-первых, это всего лишь backend часть для настроек. Во-вторых, мы ограничиваемся возможностями, реализованными в QSettings. То есть мы не сможем сериализовать свои типы данных, поскольку QVariant содержит предопределенные типы.

Далее рассмотрим **Qt.labs.settings (QML)**, Qml модуль который предоставляет доступ к QSettings внутри qml. С этим модулем проблема ровно та же самая, что и с просто классом QSettings, то есть надо писать интерфейс самому.

Также существует фреймворк **KConfig (KDE Framework)**. Наверное, из названия сразу же становится понятно почему это может стать очень плохим решением для проекта. Используя KConfig мы создаём прямую зависимость от KDE, что означает полную скованность и невозможность запустить проект например на операционной системе **Windows**, на которой поддержки KDE, конечно же, не будет.

Ну и в завершение обзора существующих решений упомянем альтернативный вариант, когда мы просто не создаем библиотеку и каждый раз в проекте пишем своё решение для настроек. В таком случае мы получаем полное отсутствие стандартизации и, как говорится, каждый раз «изобретаем велосипед». Такой подход к программированию очень быстро погубит архитектуру существующей системы и безусловно приведет к неподдерживаемым проектам.

1.3. Анализ существующих решений

Рассмотрев все имеющиеся решения, их недостатки и достоинства, можно сделать вывод, что предложенная библиотека действительно имеет место быть. Но для эффективного анализа необходимо ввести критерии чтобы полноценно провести анализ.

Критерии оценивания:

1. Гибкость – возможность работы с разными типами данных (строки, числа, списки, вложенные структуры).
2. Интеграция с QML – насколько просто подключить решение в QML-проект без переписывания кода.
3. Автоматизация UI – требуется ли ручное создание интерфейса или он генерируется автоматически.
4. Формат хранения – поддержка JSON, INI, бинарных форматов и т. д.
5. Расширяемость – возможность добавления новых типов настроек и кастомизации.
6. Зависимости – наличие сторонних библиотек, усложняющих развёртывание.

QSettings (Qt Core)

**Плюсы:**

* Встроен в Qt, не требует дополнительных зависимостей.
* Поддерживает кроссплатформенное хранение (INI, реестр Windows, plist macOS).
* Простой API для сохранения/загрузки примитивных типов.

**Минусы:**

* Нет встроенного механизма генерации UI – интерфейс нужно писать вручную.
* Ограниченная поддержка сложных структур (например, вложенных объектов).
* Нет валидации и описаний параметров.

Таким образом, данное решение подходит для простых случаев, но не решает задачу автоматизации интерфейса настроек.

Qt.labs.settings (QML)

**Плюсы:**

* Позволяет работать с настройками напрямую из QML.
* Интегрируется с QSettings, сохраняя кроссплатформенность.

**Минусы:**

* Только базовые типы данных (нет массивов, сложных объектов).
* Интерфейс всё равно создаётся вручную.
* Не поддерживает динамическое обновление схемы настроек.

Так, данное решение удобно для минималистичных проектов, но не подходит для сложных конфигураций.

KConfig (KDE Framework)

**Плюсы:**

* Поддержка сложных структур, групп настроек и зависимостей.
* Возможность удалённого управления конфигурацией.

**Минусы:**

* Зависит от KDE Frameworks, что неприемлемо для многих Qt-проектов.
* Ориентирован на C++/QtWidgets, интеграция с QML затруднена.
* Избыточность для большинства приложений.

Мощный инструмент, но слишком тяжёлый и специализированный для обычных задач.

1.4 Заключение

Анализ показал, что существующие решения либо слишком примитивны (QSettings, Qt.labs.settings), либо требуют значительных доработок (самописные системы), либо перегружены ненужной функциональностью (KConfig).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Решение | Гибкость | QML интеграция | Авто-UI | Формат хранения | Расширяемость | Зависимости |
| QSettings | Низкая | Частичная | Нет | INI/реестр | Нет | Qt Core |
| Qt.labs.settings | Низкая | Да | Нет | INI/реестр | Нет | Qt Quick |
| Самописные | Высокая | Да | Частично | JSON/другие | Да | Нет |
| KConfig | Очень высокая | Нет | Нет | Свой | Да | KDE |
| Новое решение | Высокая | Да | Да | JSON/другие | Да | Только Qt |

Таблица 1 - Сравнение доступных решения

Преимущества предлагаемой библиотеки:

1. Автоматизация – генерация UI из JSON без ручного описания QML.
2. Гибкость – поддержка вложенных структур, динамических списков и кастомных типов.
3. Простота интеграции – минимальный код для подключения в QML-проект.
4. Отсутствие лишних зависимостей – только стандартные компоненты Qt.

Таким образом, разработка данной библиотеки оправдана отсутствием готовых решений, сочетающих удобство, гибкость и лёгкость интеграции в QML-приложения.

ГЛАВА 3. ВЫБОР СПОСОБА ПОСТАВКИ БИБЛИОТЕКИ

Разработчикам доступно много способов поставки библиотеки. Однако очень важно сразу выбрать правильный, чтобы было удобно пользоваться итоговым продуктом и не было никаких неприятностей. Основные проблемы, которые призван решать этот раздел – каким способом поставить конечный продукт и каким способом поставить зависимости конечного продукта. Предварительно можно назвать как минимум три зависимости конечного продукта.

* Jsoncpp – библиотека для эффективной работы с json файлами
* Fmtlib – библиотека для удобного форматирования строк
* Spdlog – библиотека для быстрого логирования

Три самых популярных способа поставки библиотек в области С++:

* Установить зависимость системно
* Подтянуть в качестве git submodule проекта
* Использовать пакетный менеджер conan

Необходимо проанализировать каждый, узнать какие у него плюсы, минусы и критические проблемы.

3.1. Установка системной зависимости

Начать стоит с системных зависимостей. На Debian-based дистрибутивах можно легко поставить, например, fmtlib с помощью пакетного менеджера apt командой apt install libfmt-dev и дальше спокойно им пользоваться. Но одним из требований к проекту является кросс платформенность.

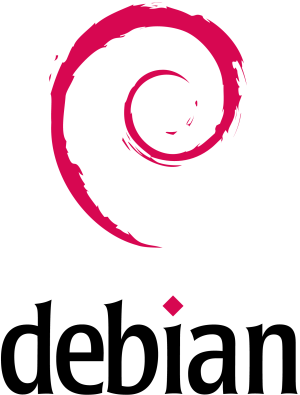


Рисунок 1 - логотип Debian

Итак, рассмотрим случай. Допустим мы разрабатываем библиотеку на Ubuntu 22.04 LTS. Поставили libfmt-dev пакет, он оказался версии 10.1.1, мы пишем #include <fmt/format.h> и спокойно разрабатываем нашу библиотеку.

Далее мы взяли свежий Debian 12 образ, хотим на нём использовать нашу библиотеку, ставим пакет libfmt-dev 10.1.1, всё работает, всё отлично. Мы можем спокойно пользоваться нашим проектом.

Потом коллега на дистрибутиве линукса Manjaro внезапно тоже захотел поработать с нашим проектом, читает README и видит, что необходимо поставить библиотеку libfmt. Пишет команду sudo pacman -S fmt и получает версию библиотеки 11.1.4. Оказалось, что в этой версии библиотеки заголовочный файл fmt/format.h заменили на fmt/core.h, который был в версии libfmt 10.1.1.

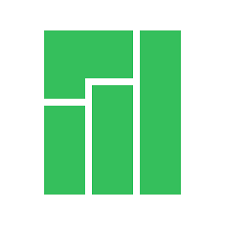


Рисунок 2 - логотип Manjaro

Чтобы обойти эту проблему можно использовать, например, следующий код.

#if FMT\_VERSION >= 110000 // 11.0.0+

#include <fmt/core.h>

#else

#include <fmt/format.h>

#endif

Не нужно быть большим экспертом чтобы понять, что такой подход приведёт в никуда. Неужели так придется делать с каждой библиотекой? А что если API библиотеки поменяется? А как тестировать такой проект? Можно было бы полностью инкапсулировать версию fmt в отдельный модуль, но тогда программисты вместо того чтобы писать код для проекта – большую часть времени занимались бы решением проблем совместимости.

Но это еще не конец истории. Далее пришло задание собрать проект, использующий нашу библиотеку, на платформе Windows. Там и вовсе никакого пакетного менеджера нету, так что придется собирать и ставить libfmt полностью вручную, что очень сильно усложняет поставку библиотеки.

Таким образом, вместо «Нам нужно поддерживать две разные версии fmt» необходимо задать вопрос: «Почему мы вообще допускаем такое неконтролируемое разнообразие версий?».

Теперь проблема с системными зависимостями видна и становится понятно, что нельзя использовать этот способ для поставки библиотек.

3.2. Использовать git submodule



Рисунок 3 - логотип Git Submodule

Альтернативой системным библиотекам является git submodule. Этот инструмент позволяет вкладывать одни репозитории в другие. Таким образом разработчики смогут полностью поместить весь исходный код libfmt и зафиксировать нужную версию прямо у себя в проекте. Делается это с помощью следующий команд:

Добавить библиотеку (spdlog, googletest, fmt)

* git submodule add

Обновить до свежей версии

* git pull
* git submodule update —init —recursive

Выбрать версию

* git checkout v0.1.0

Такой подход имеет явное преимущество перед системными библиотеками. Но это всё еще не идеально. Система Git Submodules содержит несколько очень разрушительных проблем, которые всё еще не позволят благополучно использовать нашу библиотеку.

Представим ситуацию: в проекте есть две библиотеки, LibA и LibB, и обе зависят от одной и той же LibX, но требуют разных её версий. Если подключить их через Git Submodules, то CMake выберет только одну версию LibX — ту, которая встретилась первой. Вторая версия просто проигнорируется, и при этом не будет никакой ошибки компиляции.

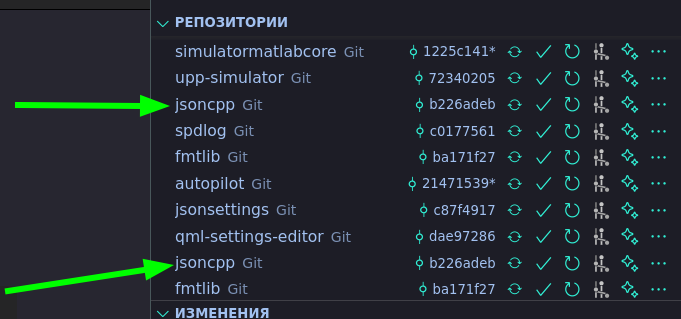


Рисунок 4 - проблема разных версий библиотек

Проблема в том, что система сборки не видит конфликта. Она находит LibX в одном из подмодулей и использует её, даже если другая часть проекта ожидает совершенно другую версию. В результате возможны трудноотлавливаемые ошибки:

* Несовместимые изменения API в разных версиях LibX могут приводить к падениям в рантайме.
* Ошибки линковки, если символы из одной версии LibX конфликтуют с другой.
* Неочевидные баги, когда код работает в одних условиях и ломается в других.

Если зависимости сами используют подмодули (а их зависимости — свои подмодули), структура проекта быстро превращается в "матрешку". Это усложняет:

* Клонирование проекта — нужно рекурсивно подтягивать все подмодули, и если какой-то из них недоступен (например, приватный репозиторий), процесс ломается.
* Обновление зависимостей — если LibX обновилась в одном месте, её нужно вручную синхронизировать во всех подмодулях, где она используется.
* Размер репозитория — история подмодулей может занимать много места, даже если сам проект небольшой.

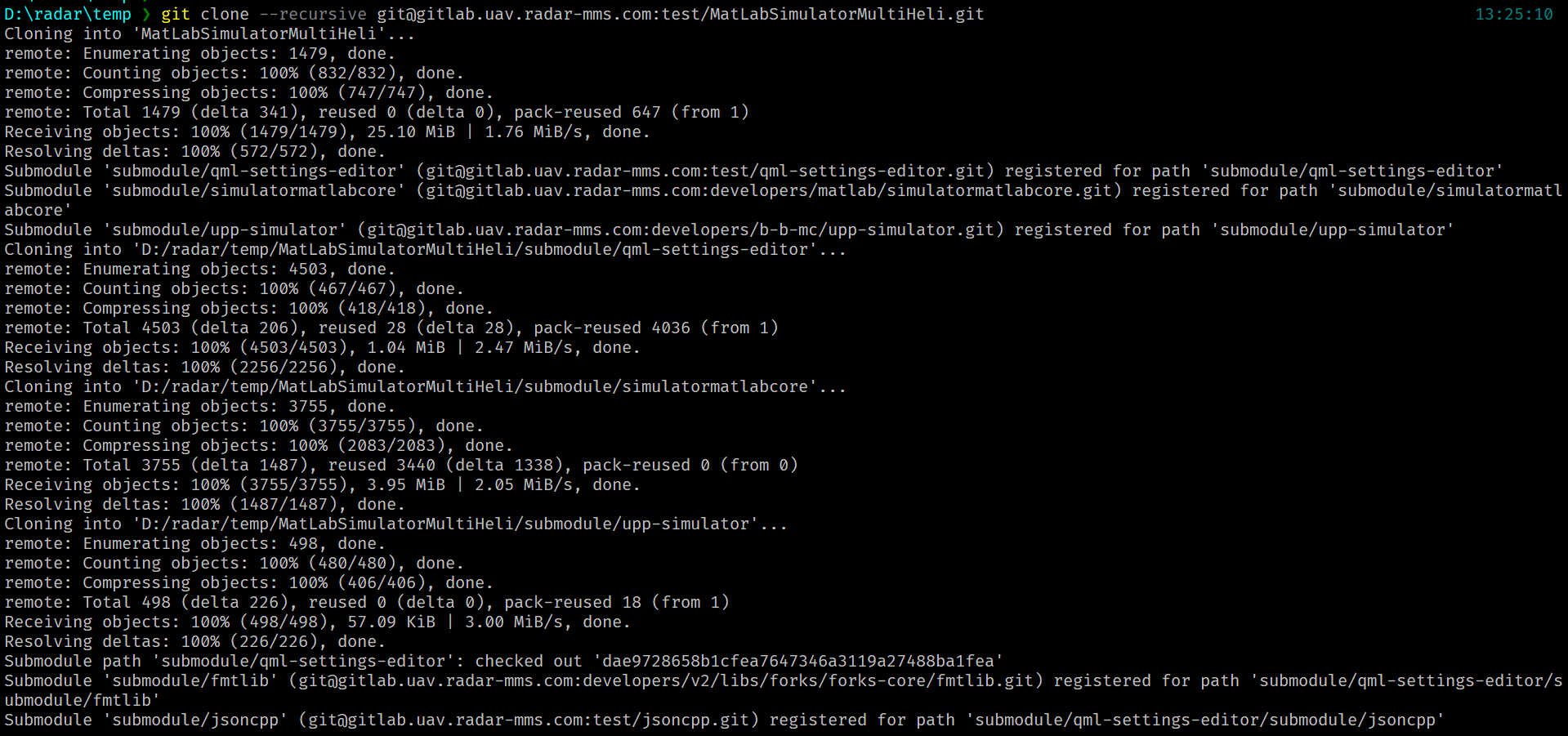


Рисунок 5 - пример клонирования проекта с сабмодулями

В отличие от пакетных менеджеров, где зависимости чётко указываются в конфигурационном файле, в подмодулях версии зафиксированы лишь хешами коммитов. Это значит:

* Невозможно быстро проверить, какие версии библиотек используются — нужно вручную заглядывать в каждый подмодуль.
* Сложно автоматизировать обновления — нет механизма "обнови все зависимости до последних стабильных версий".
* Риск "поломать" проект, если обновление подмодуля задевает другие части системы.

При использовании submodules каждая зависимость включается в проект в виде исходного кода и компилируется непосредственно в его рабочей директории. Это создаёт несколько проблем:

Если несколько подмодулей зависят от одной библиотеки (например, zlib или fmt), она может компилироваться несколько раз — по одному для каждого подмодуля. Это не только увеличивает время сборки, но и расходует дисковое пространство.

В отличие от систем управления пакетами, где скомпилированные библиотеки сохраняются в кэше, submodules требуют полной пересборки всех зависимостей при каждом обновлении или изменении конфигурации.

Поскольку системы непрерывной интеграции обычно начинают сборку с чистого состояния, они вынуждены каждый раз заново загружать и компилировать все подмодули, даже если изменения затронули лишь небольшую часть проекта.

Таким образом, использование Git Submodules для управления зависимостями существенно усложняет процесс сборки, увеличивает время разработки и создаёт дополнительные риски при обслуживании проекта. В отличие от современных пакетных менеджеров, submodules не обеспечивают кэширование, контроль версий и надёжное восстановление зависимостей, что делает их менее предпочтительным выбором для серьёзных проектов.

3.3. Использовать пакетный менеджер conan

В контексте данной работы использование системы управления пакетами Conan было выбрано в качестве основного инструмента для решения этих задач.



Рисунок 6 - логотип пакетного менеджера conan

Во-первых, Conan предоставляет механизм точного контроля версий зависимостей. В отличие от Git Submodules, где версии библиотек фиксируются лишь хешем коммита и могут быть непрозрачными для разработчика, Conan позволяет явно указывать требуемые версии пакетов в файле conanfile.py. Это позволяет легко следить за версиями зависимостей и избегать проблем, связанных с использованием различной версии одной и той же библиотеки, тем самым ускоряя разработку и делая ее гораздо проще.

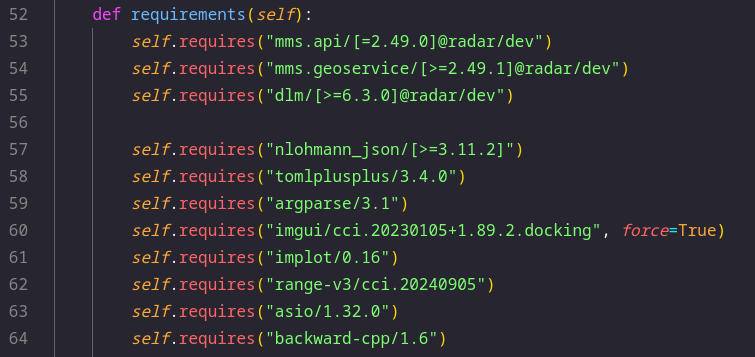


Рисунок 7 - пример conanfile.py

Во-вторых, Conan реализует эффективную систему кэширования собранных библиотек. При работе с подмодулями каждая зависимость компилируется непосредственно в дереве проекта, что при удалении директории сборки или изменении конфигурации приводит к необходимости повторной полной сборки всех зависимостей. Conan сохраняет скомпилированные пакеты в ~/.conan2/projects, что значительно ускоряет повторные сборки и делает процесс разработки более предсказуемым.

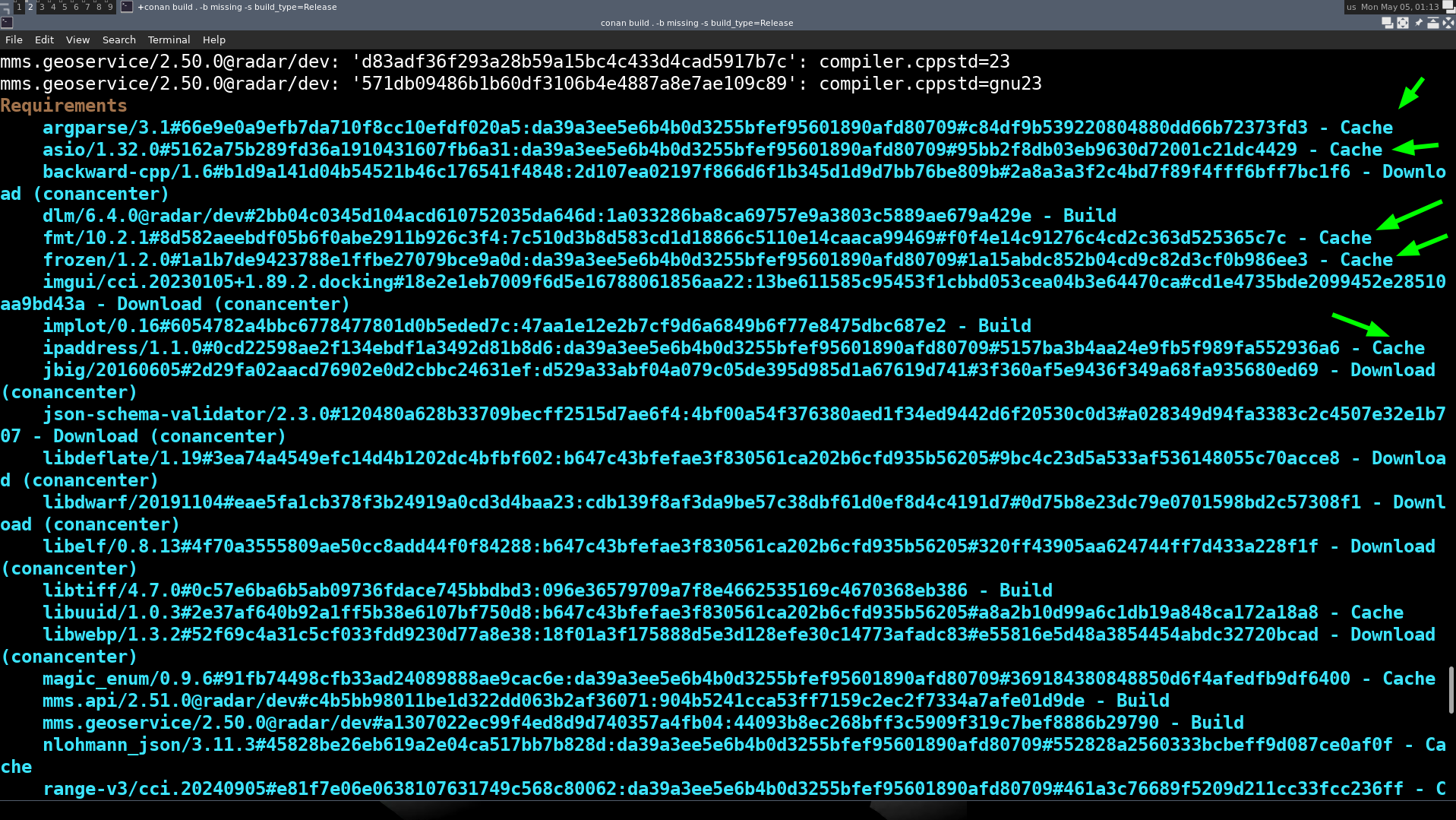


Рисунок 8 - прпимер кеширования библиотек

Важным преимуществом является также поддержка кроссплатформенности. Conan обеспечивает корректную работу зависимостей на разных операционных системах и с различными инструментами сборки, автоматически разрешая специфичные для платформы особенности. Это особенно ценно при разработке библиотеки, которая должна работать в разнородных окружениях.

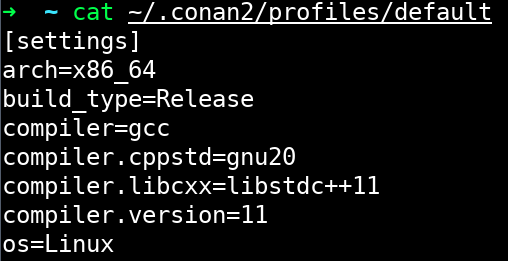


Рисунок 9 - пример профиля conan для обычной сборки проекта

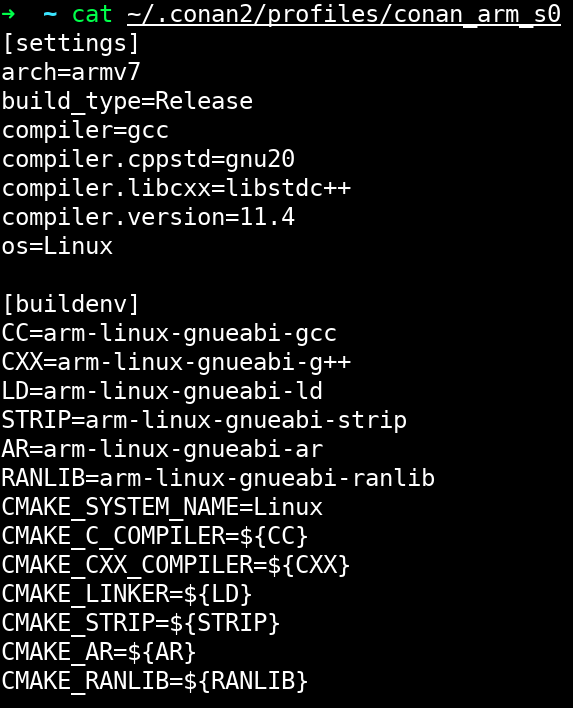


Рисунок 10 - пример профиля conan для кросс компиляции всех библиотек под arm

Кроме того, Conan предоставляет удобные механизмы для создания и распространения собственных пакетов. Это позволяет не только использовать внешние зависимости, но и легко упаковывать компоненты проекта для их повторного использования в других решениях. В контексте данной работы это означает возможность простого распространения разрабатываемой библиотеки среди потенциальных пользователей.

Наконец, интеграция Conan с современными системами сборки, такими как CMake, выполняется практически прозрачно для разработчика, сохраняя привычный рабочий процесс. При этом обеспечиваются все перечисленные преимущества в сравнении с альтернативными подходами к управлению зависимостями.

Таким образом, выбор Conan в качестве основы для управления зависимостями в проекте обусловлен необходимостью обеспечения надежности, воспроизводимости сборок и удобства сопровождения кода. Этот подход соответствует современным практикам разработки программного обеспечения и позволяет избежать многочисленных проблем, характерных для более примитивных методов работы с внешними библиотеками.

ГЛАВА 2. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ К РАЗРАБАТЫВАЕМОЙ СИСТЕМЕ

2.1. Функциональные требования

2.2. Нефункциональные требования

ГЛАВА 3. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ