Аннотации как средство управления сложностью проектов

В программировании довольно часто возникает ситуация, когда требуется привязать к коду некоторые утверждения «о коде», которые сами по себе выполняемым кодом не являются. Высокоуровневые языки, C# или Java, содержат концепцию т.н. метаданных, атрибутов или аннотаций, которые и являются встроенным средством, для записи подобного рода информации внутри файлов исходных текстов.
Широко используемые сегодня в системном программировании языки C/C++ имеют ограниченные возможности для описания аннотаций или метаданных, хотя нуждаются в них не менее, а может быть даже и более, чем вышеупомянутые высокоуровневые языки, главным образом потому, что при программировании следует учитывать особенности нижележащей аппаратуры. Например, иногда возникает необходимость в явном указании выравнивания структур, особенностей их расположения в памяти, ограничения при использовании функций и многое другое. В настоящее время, большинство из этих проблем решаются нестандартными методами (с помощью нестандартных расширений компилятора) либо хранятся отдельно от кода в каких-то внешних файлах настроек или файлах сборки (например, явное задание каких-то параметров в виде ключей компилятора, которые записываются для определенных файлов внутри makefile). В некоторых случаях, подобная метаинформация, которая логически неразрывно связана с содержимым файла, и вовсе в явном виде нигде не присутствует, а просто записывается в readme.txt (примеры – требуемые внешние константы, требования к расположению определенных функций по определенному адресу и т.п.).
Обсуждаемый сейчас новый стандарт C++14 предполагает введение новых синтаксических конструкций для атрибутов, однако это не сможет решить вышеописанные проблемы, поскольку набор самих атрибутов остается фиксированным и определенным в самом языке (то есть пользователь не может определять свои собственные атрибуты, как это можно делать в Java/C#).

В настоящей статье предлагаются методы, основанные на открытых форматах и стандартах, позволяющие ввести пользовательские метаданные или атрибуты в текст программ на C/C++, также рассмотрен тесно связанный с этим вопрос модульности на уровне исходных текстов в языке C. Созданные таким образом модули могут быть повторно использованы без внесения в них каких-либо изменений. При этом используется стандартный инструментарий для компиляции.

Сложность и модульность

Одной из основных проблем разработки ПО является управление сложностью. Можно рассмотреть проблему расширяемости с формальной стороны. Обычно, под «расширяемой системой», подразумевают такую, в которой добавление функциональности условным объемом n, должно происходить так, чтобы «трудозатраты» при этом составляли n*C, где C – некоторая константа, которая должна быть минимальной. Так или иначе, все паттерны проектирования и различные методологии направлены на достижение этого соотношения для разных сценариев этой самой производимой работы. Идеально расширяемой архитектурой является такая, для которой соотношение сохраняется, для добавления любой функциональности. В реальном мире о «любой» функциональности речи не идет, подразумевается обычно какая-то конкретика. Например, ОС работающая на одной платформе, должна в принципе оказаться способной работать и на любой другой платформе, это есть причина существования слоя абстракции HAL. Добавление N новых платформ, потребует создания N новых модулей HAL, если принять трудозатраты на создание одного HAL за константу X, то общий объем работ становится N*X+N*C = N*(X+C), где C – объем работ, необходимый для интеграции нового HAL в проект и возможности его использования (в идеале C = 0). Это при условии правильно спроектированных интерфейсов. Если же ОС написана без четких интерфейсов платформы, легко увидеть, что это потребует вероятных изменений во всех или в большинстве модулей самой ОС, то есть трудозатраты на поддержку N новых платформа составят N*X+N*m*Y=N(X+m*Y), где m – количество модулей в которые нужно внести изменения помимо HAL, а Y –

объем изменений в каждом из этих модулей. Легко увидеть, что это хуже, чем первоначальный вариант, а начиная с некоторого m и N (в случае сильносвязанного «спагетти-кода») система становится практически нерасширяемой, потому что трудозатраты на изменение текущей системы могут превысить затраты на разработку аналогичной системы с нуля. Любопытно, что косвенно, все ООП которое лежит на идее абстракций и интерфейсов можно считать техническими средствами достижения описанных соотношений функциональности и трудозатрат на их реализацию.
Таким образом, модульность является средством достижения «хорошей архитектуры», неотъемлемым свойством которой является расширяемость. Фактически, модульная система позволяет изменять свое поведение в тех предалах, в которых позволяют ее интерфейсы. При этом, правда, возникают инфраструктурые проблемы: замена или добавление модуля – непростая задача, даже если новый модуль идеально реализует интерфейс старого, если старый должен сохраниться в дереве проекта (а обычно это необходимое условие), то внесения изменений требуют конфигурационные файлы (которые могут быть общие для обоих модулей, а теперь должны узнать об их различиях), файлы сборки (которые должны в зависимости от конфигурации включать в сборку тот или иной вариант модуля) и т.д. То есть объем работы при добавлении N модулей заключается не просто в написании этих модулей, но предполагает еще и внесение изменений в некоторое количество инфраструктурных сущностей. Кроме того, по мере разрастания проекта, возрастает сложность связей между компонентами, проект не только сложно поддерживать и развивать, но и затрудняется расширение команды, так как новым людям требуется много времени для того чтобы «войти в курс» и разобраться в хитросплетении зависимостей в кодовой базе. Поэтому среди основных требований к реализации модульности – отсутствие «дополнительной информации» которую надо писать вручную в каких-либо внешних файлах, без которой модульность «не работает».
С модульностью также тесно связан вопрос повторного использования кода, поскольку основным методом управления сложностью является использование компонентов. Обычно

разбиение программы на компоненты происходит «бинарным» образом, то есть повторно используемый код выносится в библиотеки, которые могут быть использованы в других проектах. Такой подход обладает как достоинствами, так и недостатками, среди достоинств можно отметить простоту использования: библиотеки уже скомпилированы, поэтому нет нужды возиться с настройкой окружения для их компиляции и возможность использования из разных языков, а среди недостатков – тот факт, что скомпилированный код лишается любых возможностей конфигурирования времени компиляции. Последний пункт критически важен для встраиваемых систем по соображениям производительности и переносимости: при большом разнообразии процессоров бывает более целесообразно распространять исходных код библиотеки, нежели N вариантов ее самой, скомпилированной всеми возможными компиляторами под все возможные процессоры, при этом пользователи исходных кодов библиотеки сталкиваются с ее зависимостями и необходимостью поддержки инфраструктуры для ее компиляции.
Как было показано выше, модульность и ее поддержка в языке является важным вопросом при реализации больших и сложных проектов. Поскольку аннотации – утверждения «о коде», и им требуется некоторая сущность, с которой они могут быть ассоциированы, хотелось бы, чтобы они могли быть ассоциированы с понятием «модуля», а не только с элементами языка. Например, в C# атрибуты могут относиться как к элементам языка, таким как переменные, методы, классы, так и к «сборке» в целом, которая является программным «компонентом».
В C отсутствует концепция каких-либо компонентов (хотя логически они всегда присутствуют в архитектуре) поэтому при реализации аннотаций, приходится реализовывать модульность с помощью самих же аннотаций. В некоторых случаях было бы удобно выделить в коде на C «компоненты» и ассоциировать аннотации с «модулем» целиком, по аналогии с тем, как это делается в C#/Java. Данный подход хорош еще и тем, что ассоциация метаданных с элементами языка подразумевает «знание» грамматики языка, в случае же работы с метаданными на уровне модулей, это знание зачастую не требуется и реализация аннотаций получается более простой.

Аннотации в C

Ключевой идеей, которая может помочь в решении всех этих проблем – снабдить модули необходимой информацией, об их внешних зависимостях, требуемых внешних константах и прочим. Хотя, из-за невозможности внесения изменений в язык и компилятор, исследовать аннотации можно только некоторым внешним инструментом, который может и не знать грамматики C/C++ и поэтому в настоящей редакции использование аннотаций распространяется на весь модуль, хотя синтаксис теоретически позволяет аннотировать и отдельные функции/переменные. Существует несколько взглядов на то, как должны выглядеть аннотации в проекте на C, Microsoft использует макро-подобные определения (SAL), в основном же метаданные записываются в виде комментариев (doxygen, Keil configuration wizard). Предлагаемый подход предполагает использование специального макроса FX_METADATA. Этот макрос должен отображаться в пустой макрос во время компиляции (внешним определением). Хотя пустой макрос требует внешних определений по сравнению с записью метаданных в комментариях, в целом такой подход более наглядный, т.к. метаданные можно закомментировать по аналогии с кодом, а также на метаданные не будет распространяться применяемый в IDE стиль для комментариев.

Обычно (в частности, в C#) содержащаяся в аннотациях декларативная информация является набором пар ключ-значение, которое обладает свойством замыкания, то есть значение, в свою очередь, может быть другой такой парой ключа и значения и т.д. Поскольку писать аннотации предполагается вручную, необходим язык разметки, который был бы достаточно универсальным, совместимым с синтаксисом C и при этом достаточно удобным для чтения и написания человеком. Языком, специально создававшимся для этих целей является JSON (или его более расширенный аналог – YAML), он и был выбран в качестве универсального языка аннотаций, записываемых внутри макроса FX_METADATA. Также JSON также является достаточно интуитивно понятным и дружественным к изменениям даже для программистов незнакомых с ним. Единственное отступление от синтаксиса JSON – отсутствие требования заключения всех строк в

кавычки, это сильно улучшает читаемость текста. Отсутствие кавычек препятствует использованию стандартного парсера JSON, поэтому используется парсер YAML, но полностью YAML не поддерживается из-за неполной совместимости его синтаксиса с C. Подобный «JSON без кавычек» называется здесь и далее YAML/JSON.
Аннотации в тексте программы выглядят следующим образом:
FX_METADATA(({…}))
Многострочные директивы пишутся также в стандартной форме:
FX_METADATA(({...
...
}))
Поскольку аннотации всегда являются набором пар «ключ-значение», они всегда представляют собой ассоциативный массив, заключаемый в фигурные скобки.

Модули и зависимости

Как говорилось выше, модульность сама по себе также реализуется с помощью аннотаций. Файлы содержат внутри себя аннотации, связывающие данный файл с именем модуля, все файлы (с некоторыми ограничениями, о которых ниже) в которых имя модуля в аннотации совпадает представляют собой модуль. То есть появляются «надфайловые» сущности и отношения между ними, что упрощает анализ исходных текстов и визуализацию их структуры.
Вышеупомянутые высокоуровневые языки изначально проектировались с учетом существования в них модулей (и с возможностью импорта модуля в другой модуль), в C ситуация прямо противоположная, даже #include сделан на уровне препроцессора, а компилятор всегда обрабатывает один «файл» и не подозревает ни о каких зависимостях. Было бы желательно, чтобы #include каким-то образом работал с именами модулей, по аналогии с тем, как это сделано в языках с поддержкой модульности.
Прежде чем начинать обсуждение, следует ввести несколько определений.
Совокупность типов данных, прототипов функций, определений и прочей информации, которая может быть расположена в

заголовочном файле называется его **интерфейсом**. Нужно отметить, что под это определение подпадают только те сущности, которые могут быть использованы пользователем данного заголовочного файла (документированные). Внутренние определения, и функции внутри заголовочного файла интерфейсом не являются.
Связанные с заголовочными файлами исходные тексты, содержащие реализацию функций описанных в соответствующих заголовочных файлах называются **реализацией интерфейса** или просто **реализацией**.
Зависимостью модуля А от модуля Б называется включение заголовочного файла модуля Б в заголовочный файл или файл исходного текста модуля А, так как это предполагает использование интерфейса модуля Б внутри интерфейса или реализации модуля А. Использование функций, макросов и прочих элементов их модуля Б внутри модуля А формирует, таким образом, набор точек соединения или срез (pointcut в терминах АОП).
Для включения интерфейса по его имени, а не по имени файла, используется оговоренная в стандарте возможность писать макросы в качестве аргумента директивы #include. Создается специальный макрос FX_INTERFACE, который принимает имя интерфейса/модуля в качестве аргумента.
Таким образом, заголовочный файл соответствующий модулю MY_MODULE должен содержать внутри себя следующую строку:
FX_METADATA(({ interface: [MY_MODULE] }))
По аналогии с Java, рекомендуется давать файлу такое же имя, как имя экспортируемого им интерфейса, хотя и не обязательно.
Если интерфейс не реализован полностью в заголовочном файле, тогда каждый файл исходного текста, соответствующий модулю MY_MODULE должен содержать директиву:
FX_METADATA(({ implementation: [MY_MODULE] }))
Импорт модуля осуществляется в виде:
#include FX_INTERFACE(MY_MODULE)
Данный макрос должен браться извне (также как и FX_METADATA - из ключей компилятора или из включенного принудительно файла), он должен тем или иным образом возвращать имя файла, которое может быть обработано препроцессором (как имя включаемого файла). Это может быть сделано несколькими

способами, которые применимы в различных ситуациях, они будут рассмотрены далее в разделе «Сборка».
Дополнительным преимуществом такой системы сборки является то, что сборка перестает зависеть от структуры директорий и зависит только от внутренних отношений между файлами. Проводя аналогии далее, можно сказать, что получившаяся система напоминает Managed Extensibility Framework (MEF), имеющийся в платформе Microsoft.NET, только «работает» она во время компиляции. Теперь, имея только исходные тексты, можно сгенерировать список файлов для сборки и собрать систему, причем все зависимости между модулями будут извлечены из самих модулей.

Извлечение зависимостей

Для построения графа зависимостей модулей (и определения файлов, которые должны войти в сборку) можно воспользоваться несколькими методами.
Наиболее универсальным является метод, основанный на использовании препроцессора. В этом случае, после анализа исходных текстов внешним инструментом, генерируется файл, содержащий отображения всех найденных модулей, на имена соответствующих им файлов, в этом файле также определяется макрос FX_INTERFACE/FX_METADATA. Данный файл должен быть неявно включен (с помощью директивы компилятора) в файл, зависимости которого следует извлечь, после обработки последнего препроцессором, и найдя в выводе все метки, соответствующие интерфейсам (которые можно получить, определив FX_METADATA как непустой макрос), можно узнать обо всех директивах #include которые содержатся в файле, то есть появляется возможность узнать все зависимости. Данный метод хорошо работает, если пути к файлам указаны так, что существует только один модуль, реализующий данный интерфейс и не возникает никакой неоднозначности, требующей внешней информации.
Второй способ более трудоёмок, но зато может использоваться и в случае, когда есть несколько модулей, реализующих данный интерфейс и возникают неоднозначности. Поскольку включаемые модули неизвестны, компилировать и обрабатывать файлы

препроцессором невозможно, поэтому единственным способом извлечения зависимостей остается «ручной» разбор содержащихся в файле директив #include. Хотя и этот процесс может быть автоматизирован, компилятор GCC содержит встроенную функциональность, с помощью которой можно узнать зависимости файла, даже если файлов, которые он включает, не существует, после этого, зная, что модули импортируются с помощью макроса FX_INTERFACE, можно выделить в выводе имена модулей с помощью регулярных выражений.
Хотя существует важное отличие между этими двумя методами: первый метод позволяет использовать в последующих директивах #include макроопределения, определенные в предыдущих включаемых файлах, а разбор #include-ов без использования препроцессора естественно не позволяет этого, можно пренебречь этой возможностью, т.к. это является дурным стилем программирования и не поощряется даже и без использования основанной на модулях сборке.

Сборка

Конфигурирование, и определение файлов, которые должны войти в сборку должно осуществляться без привязки к конкретной системе сборки вроде make и представляет собой отдельную фазу сборки проекта, которая выполняется до компиляции.
Сборку можно производить тремя основными способами.
Если система поставляется в виде исходных текстов, то с точки зрения пользователя удобно когда предварительно сконфигурированная система поставляется в виде плоского набора файлов, это требует минимума настроек для его проекта, упрощает отладку и сценарии сборки. Для относительно небольших проектов поставка может осуществляться в плоском виде: поскольку в системе может быть только один интерфейс с заданным именем, а в каждую конфигурацию может входить только одна реализация каждого интерфейса, при именовании файлов учитывающем экспортированные ими интерфейсы можно гарантировать, что имена файлов не совпадут, даже если все файлы входящие в сборку скопировать в одну папку.
Макрос FX_INTERFACE при этом должен быть определен как:

#define FX_INTERFACE(i) <i.h>
Хотя «относительно небольшой» проект это условность, ничто не мешает сделать подобный трюк для системы любого объема, но когда в одной папке лежат несколько сотен а то и тысяч файлов, уже с трудом можно говорить об упрощении работы с ними.
Второй способ состоит в использовании неявно включаемого файла, который содержит отображения имен модулей на имена файлов, а также макрос FX_INTERFACE. При этом не нужно указывать компилятору папки содержащие файлы для включения (default include directories), т.к. неявно включаемый файл уже содержит все нужные отображения в виде абсолютных путей. Файлы исходных текстов могут быть выведены в виде плоского списка или в каком-либо формате вроде make для последующей сборки.
Наконец, третий способ, если включаемые неявно файлы нежелательны по тем или иным причинам, для каждого могут быть из его зависимостей определены include directories, которые также могут быть выведены в каком-либо формате, который может быть проинтерпретирован используемой системой сборки. Важно отметить, что вся информация о зависимостях и компилируемых файлах извлекается непосредственно из исходных текстов, поэтому для того, чтобы модуль можно было использовать, надо только чтобы он содержал нужные аннотации и присутствовал в дереве проекта.

Примеры использования аннотаций

Рассмотренный выше пример реализации модульности на основе аннотаций, хотя и полезен, но не очень впечатляющий: того же эффекта можно добиться с помощью системы сборки, соглашений об именовании файлов и т.д., без привлечения целой инфраструктуры анализа и интерпретации аннотаций внутри исходных текстов. В данном разделе будут рассмотрены некоторые примеры, которые являются естественным расширением предложенной выше модульной системы, но функционал которых уже не так просто реализовать другими методами.

Внедрение зависимостей во время компиляции

Помимо просто иерархии модулей, крупные проекты могут содержать также несколько вариантов для некоторых из этих модулей. Если в кодовой базе может одновременно присутствовать несколько модулей с одинаковым именем, нужно их как-то различать, поэтому в список YAML/JSON содержащий имя интерфейса добавляется еще один элемент, который называется «имя реализации указанного интерфейса, которую содержит данный модуль».

FX_METADATA(({ interface: [MY_MODULE, MY_IMPLEMENTATION] }))

То же самое делается и с файлами исходных текстов:

FX_METADATA(({ implementation: [MY_MODULE, MY_IMPLEMENTATION] }))

Модулем теперь считаются файлы, которые содержат одинаковые метки как названия интерфейса так и реализации. Рассмотренная выше схема, где указывается только имя модуля продолжает работать без изменений т.к. анализирует только первый элемент списка, содержащего имя модуля.

Директива #include по-прежнему содержит включение абстрактного интерфейса (поскольку включение конкретной реализации обычно является нарушением принципа взаимозаменяемости модулей, реализующих один интерфейс. Возникает недостаток информации, который должен быть устранен из какого-то внешнего источника. Вводится файл, называемый «отображением» который указывает, какие реализации каких интерфейсов следует использовать. Нетрудно заметить, что это абсолютно аналогично внешней конфигурационной информации для внедрения зависимостей, используемой в различных прикладных библиотеках для высокоуровневых языков. Подробно внедрение зависимостей будет рассмотрено в отдельной статье.

Опции

Отдельной темой является конфигурирование опций. Опциями называются некоторые константы, которые нужны для работы модулей (и обычно задаваемые в виде директив #define). Часто для опций, а также и для включения-исключения самих модулей, делается файл типа config.h, содержащий нужные определения.

Как уже говорилось выше, данный подход плох тем, что при дабовлении какого-то нового модуля, существует несколько мест, куда надо внести изменения, для того чтоб встроить этот модуль в систему. Также это является камнем преткновения для конфигурирования, опции и #define распространяются только на исходные тексты и не могут влиять на уже скомпилированные библиотеки, поэтому, если какой-либо модуль предполагает широкий диапазон конфигурирования с помощью внешних констант, становится невозможным его использования в виде библиотеки. Например, если в каком-то модуле используется статические глобальные данные зависящие от максимального количества потоков в системе, то для «продвинутых» систем вроде настольных, эта константа может быть порядка тысячи или десятка тысяч, напротив, для маленьких встраиваемых систем количество потоков редко превышает 10, поэтому в скомпилированном виде такой модуль использовать нельзя, и необходимо прибегать либо к динамическим структурам данных (которые тоже не всегда могут быть применимы), либо перекладывать заботу о внешних константах на плечи того, кто будет этот модуль использовать (грубо говоря, писать в readme.txt, что данные исходные тексты требуют таких-то внешних определений, которые надо добавить в ваш аналог config.h).
Теперь, после того, как определены принципы работы модульности, можно вернуться к вопросам поставленным в начале статьи и сказать, что пользовательские метаданные ассоциированные с модулем содержат также описания и названия констант, которые этот модуль использует, возможно с указанием типа и граничных значений для упрощения анализа допустимых значений.
Каждый модуль содержит в качестве метаданных описание используемых им внешних констант, например:

```
FX_METADATA(({ options: [
  MY_MODULE_OPTION_1: {
   type: int, range: [0, 0xffffffff], default: 0x1000,
   description: "Friendly description."},
  MY_MODULE_OPTION_2: {
   type: enum, values: [VARIANT_1: 0x200, VARIANT_2: 0xF],
default: 0,
```

description: "Friendly description."}]})

Для системы без внедрения зависимостей поддерживать список констант конфигурации в актуальном состоянии еще как-то можно вручную, в случае же использования внедрения зависимостей и большого разнообразия модулей и их комбинаций, задача указать все нужные опции становится нетривиальной и чревата ошибками, т.к. полностью ложится на плечи программиста. При использовании аннотированных опций, все опции для текущей конфигурации могут быть собраны автоматически. Также, при указании границ и описаний, становится возможным написать графическое приложение для конфигурирования опций, аналогичное Keil cfg wizard, только работающее для модулей и проекта в целом, а не только для отдельных файлов, то есть, в отличие от Keil, конфигурирование производится уже на более высоком уровне – в терминах модулей, а не переменных в конкретных файлах исходных текстов.

Верификация

Аннотации также могут быть с успехом использованы в целях формальной верификации ПО. Например, такие требования как «работа в реальном времени» могут быть преобразованы в требования «все вызываемые функции должны запрещать прерывания либо вытеснение на фиксированное время не зависящее от условий времени выполнения». После чего вводится атрибут функции «соответствует требованию реального времени». Проверка выполнения этого условия для любой функции является алгоритмически неразрешимой, поэтому установка атрибутов должна проиводиться вручную, как результат code review, но проверка того, что все используемые функции обладают этим атрибутом, может быть произведена автоматически, что экономит ресурсы, т.к. review отдельной функции несравним по сложности с рассмотрением всей системы. Помимо этого, возможно аннотирование ограничений по цикломатической сложности функций, квоты стека и многие другие параметры, которые могут быть проверены полностью автоматически, что упрощает поиск ошибок, а также упрощает документацию – некоторые сведения и допущения, которыми пользовался программист содержатся непосредственно в коде и

описание этих ограничений можно опустить при документировании функции.

Аспекты

При разработке слабосвязанных модульных систем часто возникает необходимость в знании модулей о конфигурации, которая может быть получена только после определения набора модулей, входящих в ее состав. Например, необходимость сопоставить каждому модулю уникальный идентификатор от 1 до N, где N - количество модулей в системе. Это может быть необходимо для получения уникальных идентификаторов для кодов ошибок каждого модуля. Модули могут реализовывать совместно некоторую сквозную функциональность, например, добавлять поля в структуры данных других модулей и т.д. Все эти задачи сводятся к тому, что после определения конфигурации, нужно собрать некоторую информацию из всех модулей в системе, после чего собрать всю эту информацию, или аспекты, в некотором одном месте, откуда все заинтересованные модули могли бы получить к ней доступ.
Эта задача также может быть эффективно решена с помощью аннотаций внутри исходных текстов. В простейшем случае, каждый модуль может содержать набор пар <"ключ"-"набор значений">, после обработки всех модулей, входящих в систему, все наборы значений с данным ключом объединяются в единый набор, после чего, каждый модуль, знающий о ключе, получает доступ к полному набору значений, а не только к своему.

Доступ к метаданным во время выполнения

В качестве экзотики, можно рассмотреть, каким образом можно проанализировать аннотации в бинарном виде во время выполнения (реализовав, таким образом некоторое подобие Reflection). У JSON есть соответствующий ему бинарный формат BSON, в который могут быть сериализованы содержащиеся в модулях метаданные. После того, как файл скомпилирован, в соответствующем ему бинарном файле может быть создана секция метаданных (для ELF может быть использована стандартная утилита objcopy), затем, эти секции должны быть

объединены и добавлен символ, соответствующий началу секции. Затем, во время выполнения, секция с бинарными данными может быть доступна через имя символа.

```
void test_func(void)
{
extern void metadata;
…
}
```

Таким образом, метаданные всего проекта могут быть соединены компоновщиком в одну секцию и проанализированы во время выполнения.

Заключение

Аннотации являются важным средством описания относящихся к коду декларативных метаданных и существуют в том или ином виде в большинстве современных языков. Язык C хотя и не содержит встроенной поддержки метаданных, содержит все необходимые средства, чтобы эти возможности были в него встроены совместимыми со стандартом методами. Использование аннотаций помогает в создании программных модулей уровня исходных текстов, упрощает конфигурирование, документирование, верификацию/анализ и сборку проекта. Любой существующий проект, при условии наличия в нем четко определенных межмодульных интерфейсов, может быть быстро портирован на данную платформу. При этом не потеряется совместимость ни со стандартом, ни с одним из инстументов. Формат самих аннотаций основан на широко используемых языках JSON/YAML, которые обладают неограниченными возможностями для описания декларативной информации, являются дружественными как для чтения так и для написания вручную. Кроме того, доступно бинарное представление информации, записанной на этих языках, и удобная работа с ней программным способом во время выполнения.

Также рассмотрена тема модульности в C и связь модулей с аннотациями, методы внедрения зависимости и замены модулей. Помимо рассмотренных случаев, аннотации могут быть также использованы для записи любой пользовательской информации, которая впоследствии может быть проанализирована внешними

инструментами, либо самим кодом во время выполнения.