ПРАКТИКА ПРОГРАММИРОВАНИЯ

ОБРАТНЫЕ ВЫЗОВЫ В C++

ПРОЕКТИРОВАНИЕ И АНАЛИЗ

12+

Виталий Ткаченко

Виталий Ткаченко

Обратные вызовы в C++

«ЛитРес: Самиздат»

2020

Ткаченко В. Е.

Обратные вызовы в C++ / В. Е. Ткаченко — «ЛитРес: Самиздат», 2020

В практике разработки ПО зачастую встает задача динамической модификации программного кода в зависимости от текущих или настраиваемых значений параметров. Для решения этой задачи широко используются обратные вызовы. В языке C++ обратные вызовы реализуются различными способами, и далеко не всегда очевидно, какой из них лучший для конкретной ситуации. В книге рассмотрены теоретические и практические аспекты организации обратных вызовов, проанализированы достоинства и недостатки различных реализаций, выработаны рекомендации по выбору в зависимости от требований к проектируемому ПО. В первую очередь книга предназначена для программистов среднего (middle) уровня, т.е. тех, кто уже достаточно хорошо знает язык C++, но хотел бы расширить и углубить свои знания в области проектирования и дизайна. В определенной степени она также будет интересна опытным разработчикам, с одной стороны, как систематизация знаний, с другой стороны, как источник идей и методов для решения практических задач.

Содержание

Виталий Ткаченко
Обратные вызовы в C++

Введение

Однажды со мной консультировался начинающий разработчик. Не помню точно, о чем шла речь (да это и не важно), но вопрос был в стиле «есть проблема – как ее решить?». Первой моей мыслью, которую я и озвучил, было – «сделай обратный вызов». Следующий, вполне ожидаемый, вопрос был «а как его реализовать?». Почти не думая, я ответил первое, что пришло в голову – «используй указатель на функцию». «Хорошо», сказал разработчик, «я почитаю про эти указатели». Через какое-то время он снова пришел с вопросом – «ну, что такое указатель на функцию, я понял, но как внутри функции узнать, какому классу предназначается вызов?» Так, слово за слово, вопрос за вопросом, и я вдруг начинаю осознавать, что вопросы совсем не такие уж простые, как вначале могло показаться, и что одно понятие тянет за собой другое, что есть множество альтернатив при выборе способа реализации, и что так сразу и не скажешь, какой из них лучше подходит именно для вот этого случая… Так и родилась идея книги, которую вы сейчас держите перед глазами.

Формат представления информации в виде книги имеет одно неоспоримое преимущество: здесь отсутствуют ограничения по объему. Появляется возможность изложить весь материал обстоятельно, подробно, в деталях, охватывая множество аспектов и нюансов. Это выгодно отличает книгу от других форматов, таких, как статьи, лекции, презентации и т. п. В них всегда приходится идти на компромиссы, выделяя главное и отбрасывая детали, которые, на первый взгляд, кажутся несущественными, но их наличие существенно облегчает освоение материала и избавляет читателя от необходимости самостоятельно искать ответы на вопросы, которые неизбежно возникают при изучении незнакомых предметов.

Сами по себе обратные вызовы является узкоспециализированной темой, однако при этом они охватывают ряд смежных концепций как в сфере использования языка программирования, так и в сфере архитектурно-проектных решений. В связи с этим, изучение обратных вызовов значительно повышает компетенции специалиста и обогащает его арсенал приемов и способов решения нетривиальных задач.

В первую очередь книга предназначена для разработчиков среднего (middle) уровня, т. е. тех, кто уже достаточно хорошо знает язык C++, но хотел бы расширить и углубить свои знания в области проектирования и дизайна. Безусловно, не лишней она будет и для начинающих, но нужно быть готовым к тому, что для изучения материала придется приложить значительные усилия: рассматриваемые концепции являются достаточно сложными и предполагают хорошее знание синтаксиса C++, а также некоторый опыт в программировании. Надеюсь, опытные разработчики также найдут книгу полезной как в плане систематизации знаний, так и в плане новых идей и методов, которые можно использовать в практике разработки.

Структурно книга состоит из разделов, глав и параграфов. В первом разделе излагаются теоретические основы, даются определения и термины. Во втором разделе рассматриваются способы реализации обратных вызовов в языке C++. В третьем разделе проводится сравнительный анализ реализаций, вырабатываются рекомендации для выбора в конкретных случаях. В четвертом разделе рассматривается использование шаблонов – пожалуй, наиболее интересной концепции C++, активно развивающейся в новых стандартах. И в заключение, чтобы изложенный материал не показался совсем уж абстрактным и оторванным от жизни, в пятом разделе демонстрируется практическое использование обратных вызовов на примере проектирования программного компонента.

В книге иллюстрируется, как используются те или иные конструкции C++, но не раскрывается их сущность – предполагается, что читатель об этом осведомлен. Поэтому для успешного понимания материала необходимо ориентироваться в следующих темах:

- базовый синтаксис C++;
- классы и наследование, перегрузка операторов;
- лямбда-выражения и захват переменных;
- контейнеры стандартной библиотеки;
- семантика шаблонов C++;
- шаблоны с переменным числом параметров, частичная специализация шаблонов.

Теоретические положения проиллюстрированы многочисленными примерами, оформленными в виде листингов. После каждого листинга (за исключением совсем уж тривиальных случаев) идет пояснение, которое облегчает понимание кода. Примеры создавались, ориентируясь на стандарт C++ 17; некоторые из них используют специфические особенности указанного стандарта и не будут компилироваться в более ранних версиях. Исходные тексты всех примеров можно найти в **https://github.com/tkachenko-vitaliy/Callbacks**, там же указан адрес электронной почты для связи с автором.

Во втором издании исправлены некоторые опечатки, а также переработана глава 5.5, в которой представлены улучшенные технические решения, основываясь на новых возможностях стандарта C++ 17.

На этом вступительную часть можно считать оконченной, приступим теперь непосредственно к изучению обратных вызовов.

1. Теоретические основы обратных вызовов

1.1. Концепция обратных вызовов

1.1.1. Интуитивное определение

Представьте следующую ситуацию. Вам нужно совершить платеж в банке. Вы идете в банк, берете талон, дожидаетесь, пока вас пригласят, и совершаете платеж. Но ведь столько времени придется потратить, в банке всегда такие очереди… Есть вариант получше: попросить свою маму (или бабушку) зайти в банк и занять очередь. Когда очередь подойдет, мама (или бабушка) позвонит, и вам остается только прийти и сделать платеж. Если же вы в этот день сильно заняты, тогда можно оставить телефон друга, и он сделает платеж вместо вас.

Итак, результат один и тот же, но последовательность действий различная. В первом случае вы сами идете в банк, отстаиваете очередь и совершаете платеж, т. е. выполняете все необходимые операции. Во втором случае вы сидите и ожидаете, когда вам позвонят, т. е. сделают вызов, и делаете только одно действие, а именно – совершаете платеж. Либо это делает ваш друг, если маме (или бабушке) дали его, а не ваши контакты. Можно утверждать, что ваша мама (или бабушка) инициировала, а вы выполнили обратный вызов.

1.1.2. Обратный вызов как паттерн

Перейдем теперь на язык программирования и дадим формальное определение.

> **Обратный вызов** – это паттерн, в котором какой-либо исполняемый код как аргумент передается в другой код, при этом ожидается, что через сохраненный аргумент исполняемый код будет запущен в требуемый момент времени.

Возвращаясь к неформальному примеру: здесь выполнение платежа можно считать исполняемым кодом, номер телефона – аргументом, телефонный звонок – запуском кода на выполнение.

Графически описанную концепцию можно проиллюстрировать следующим образом (Рис. 1). В программе существует код, выполняющий какие-либо операции, или исполняемый код. Когда программа запускается, исполняемый код как аргумент передается в другой код, или вызывающий код. Вызывающий код сохраняет переданный аргумент и начинает работу. В нужный момент времени, используя сохраненный аргумент, вызывающий код запускает исполняемый код, т. е. осуществляет обратный вызов.

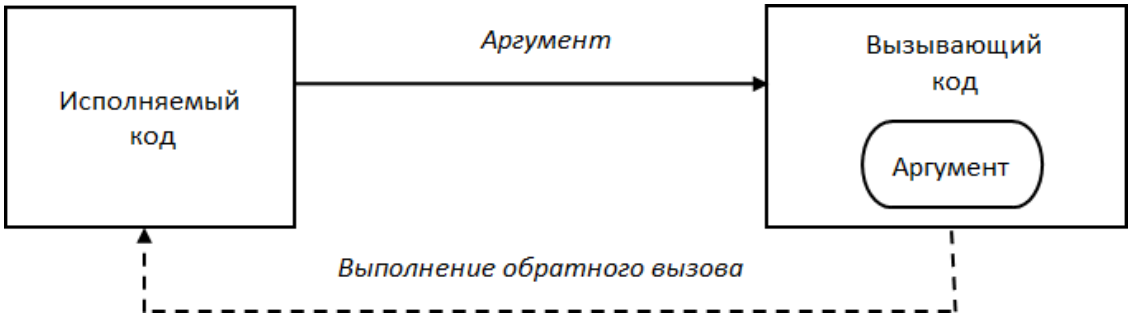

Рис. 1. Концепция обратных вызовов

1.1.3. Прямые и обратные вызовы

Различие между прямым и обратным вызовом проиллюстрировано на Рис. 2. В первом случае поток управления запускает вызывающий код, из которого вызывается исполняемый код, и далее управление возвращается в точку вызова. Во втором случае поток управления идет мимо исполняемого кода и настраивает аргумент в вызывающем коде, а вызов исполняемого кода осуществляет уже вызывающий код, т. е. поток управления идет в обратном направлении. Таким образом, мы имеем обратный вызов.

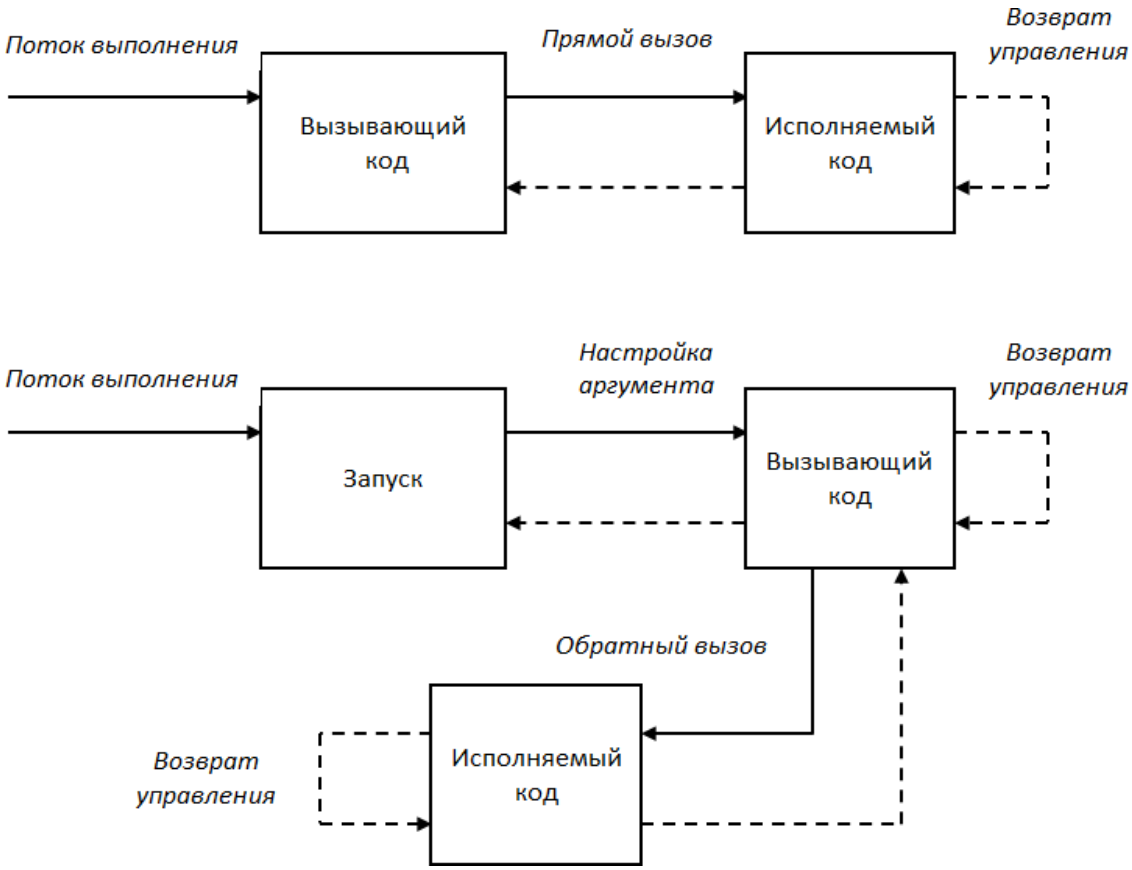

Рис. 2. Прямой и обратный вызов

1.2. Задачи, решаемые с помощью обратных вызовов

Все многообразие задач, решаемых с помощью обратных вызовов, можно разделить на следующие группы.

1.2.1. Запрос данных

Представим, что мы разрабатываем программное обеспечение для микроконтроллера управления технологическими процессами. Контроллеру требуется периодически получать показания датчиков, таких как температура, влажность, давление и т. д. Как это реализовать?

Самое простое решение – код для опроса датчиков непосредственно реализовать в ПО контроллера. Но здесь возникает множество вопросов. А если в системе понадобится использовать другую модель датчика, код опроса которого должен быть другим? А если нам нужно использовать различные датчики для различных режимов? А как быть, когда мы вообще не знаем, какие датчики будут использоваться?

Эффективный способ решения указанных проблем – разработка драйвера, т. е. модуля, поддерживающего единый интерфейс вызовов для различных реализаций. Однако одно дело подать идею, а вот реализовать – тут все гораздо сложнее: интерфейс должен быть универсальным и покрывать все возможные требования; необходимо разработать механизм для загрузки нужной реализации интерфейса; требуется каким-то образом связывать интерфейс и реализацию – в итоге нам понадобится сервис поддержки драйверов. Для операционной системы это вполне оправдано, однако для микроконтроллера с его очень ограниченными ресурсами внедрение такого сервиса чревато потерей производительности как из-за большого объема кода, так и из-за дополнительного расхода памяти.

Можно предложить не такое универсальное, зато более простое и менее ресурсоемкое решение с помощью обратных вызовов(Рис. 3). Код опроса упаковывается в отдельный компонент. Перед началом работы происходит настройка, т. е. указанный код как аргумент сохраняется в рабочем коде контроллера. В нужный момент рабочий код делает обратный вызов, выполняет соответствующую функцию и получает требуемое значение. Если необходимо, в процессе работы можно изменять хранимый аргумент, изменяя, таким образом, код опроса датчиков.

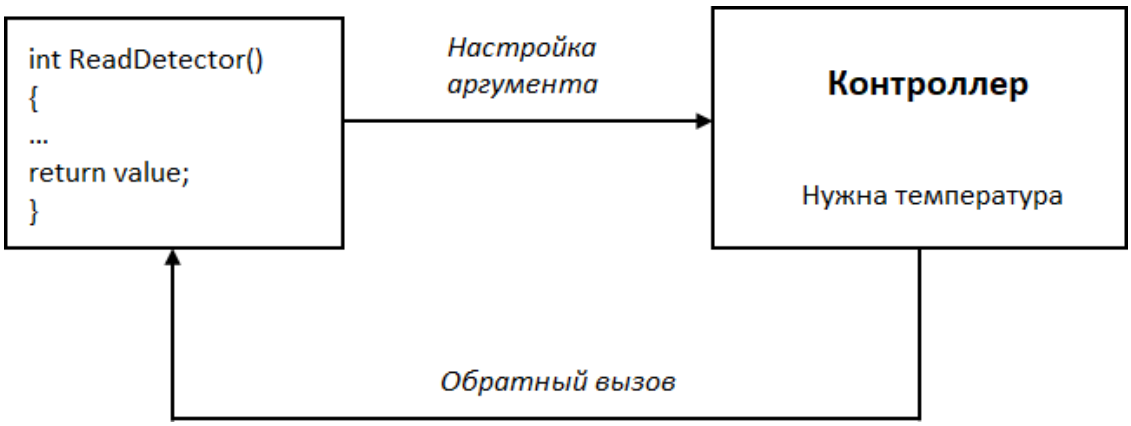

Рис. 3. Опрос датчиков с помощью обратного вызова

1.2.2. Вычисления по запросу

Представим, что мы разрабатываем супербыстрый алгоритм сортировки, оптимизированный для работы на нашем многопроцессорном суперкомпьютере. Было потрачено массу усилий, реализовано много кода, и, наконец, алгоритм почти готов. Но вот незадача: мы не знаем заранее, что именно нам нужно сортировать. Сортировка чисел – это самый простой случай, а что делать, если понадобится сортировать, допустим, структуры, содержащие записи из базы данных? Пусть в структуре содержатся сведения о сотрудниках – фамилия, имя, отчество. Как реализовать сортировки по отдельным полям, по совокупности полей? Неужели придется дублировать код для каждого случая?

Простое и эффективное решение указанной проблемы представлено на Рис. 4. Код для сравнения полей упаковывается в отдельный компонент. Когда запускается алгоритм, этот компонент передается как аргумент. В требуемый момент времени алгоритм через указанный аргумент вызовет код сравнения, передавая элементы данных как параметры. Таким образом, можно реализовать различные правила сравнения и передавать их алгоритму без изменения рабочего кода.

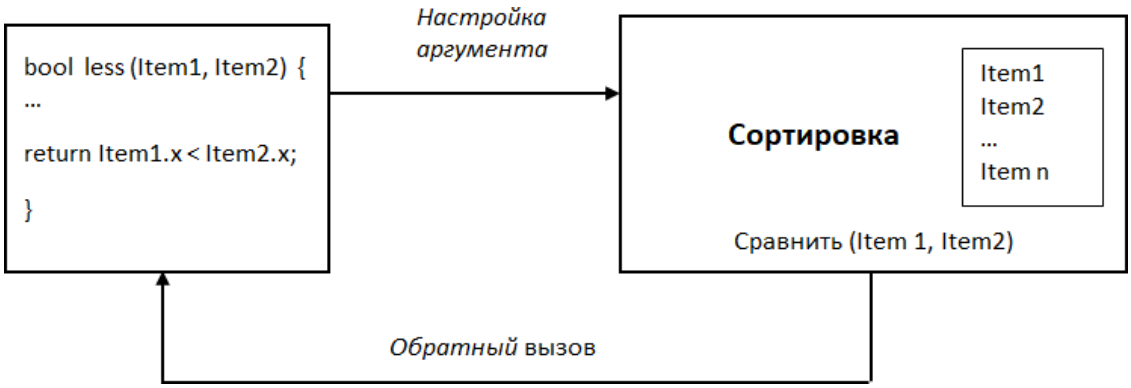

Рис. 4. Результат вычисления с помощью обратного вызова

1.2.3. Перебор элементов

Представим, что мы разрабатываем модуль сетевого обмена. Как пользователю узнать, какие протоколы поддерживаются?

Самое простое решение – получить количество поддерживаемых протоколов, а затем запрашивать их имена по порядковому номеру. Данный способ легко реализуем, если внутри модуля имена протоколов хранятся в массиве. А если имена нужно хранить в списке? Тогда задача усложняется: нужно сделать перебор элементов списка, чтобы получить нужное значение по порядковому номеру. А если имена должны храниться в виде двоичного дерева?

Возможное решение: разработать итератор – специальный класс, который будет осуществлять навигацию по контейнеру. Такой подход реализован, к примеру, в стандартной библиотеке STL, где для каждого контейнера имеется соответствующий итератор. Недостаток этого решения проявляется в том, что мы ограничиваем сферу применения модуля, построенного таким образом: его использовать могут только те компоненты, которые способны интерпретировать вызовы методов C++. Кроме того, итератор привязан к типу используемого контейнера, и при его изменении приходится перекомпилировать все связанные компоненты.

А что, если реализовать итератор с помощью набора функций, без использования классов? Интерфейс получается довольно сложным: необходимы отдельные функции для создания

итератора, запроса значений, уничтожения итератора; необходимо объявить тип данных для хранения итератора; необходимо предусмотреть уничтожение итератора в случае возникновения исключений.

Простое и эффективное решение указанных проблем представлено на Рис. 5. Код, обрабатывающий имена поддерживаемых протоколов (например, отображение в пользовательском интерфейсе), упаковывается в отдельный компонент. Для получения протоколов вызывается функция, в которую указанный компонент передается как аргумент. Функция перебирает хранимые значения, для каждого значения через сохраненный аргумент вызывается код обработки, имя протокола передается как параметр.

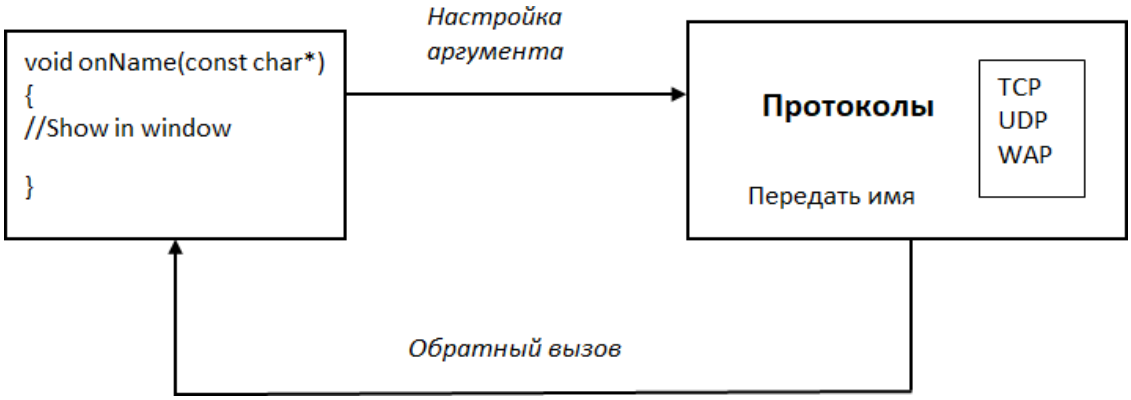

Рис. 5. Просмотр элементов с помощью обратных вызовов

1.2.4. Уведомление о событиях

Представим, что мы в системе запустили таймер, и нам нужно получить уведомление о срабатывании таймера. Самое простое решение – в процессе выполнения опрашивать таймер и анализировать, не истекло ли время. Как часто нужно делать опрос? Слишком часто – теряется производительность, слишком редко – теряется точность. Кроме того, приходится постоянно в определенных участках кода вставлять вызов опроса. Учитывая, что в программе могут работать несколько потоков, опрашивать таймер они будут с разной частотой, и каждый поток обнаружит срабатывание таймера в разное время.

Простое и эффективное решение указанных проблем представлено на Рис. 6. Код, обрабатывающий срабатывание таймера, упаковывается в отдельный компонент. Когда запускается таймер, этот компонент как аргумент передается таймеру, и когда таймер сработает, через сохраненный аргумент будет вызван код обработки. По такому же принципу можно организовать асинхронный ввод-вывод, обработку прерываний и т. п.

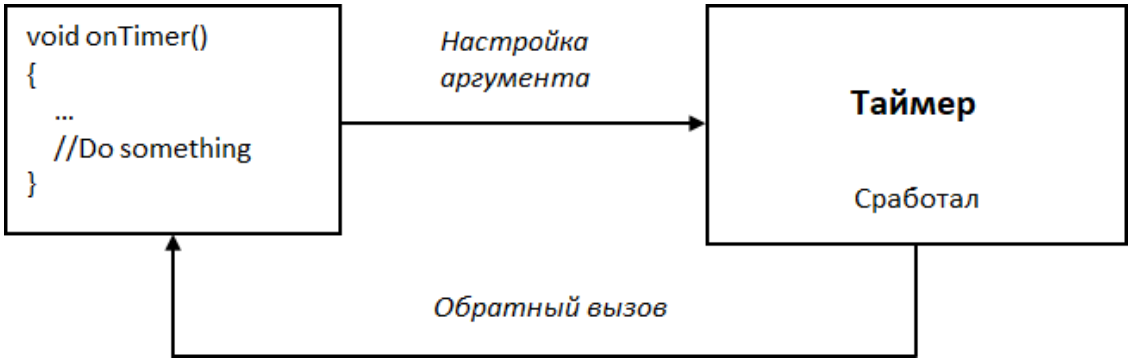

Рис. 6. Уведомление о срабатывании таймера с помощью обратного вызова

Итак, мы рассмотрели типовые задачи, в которых используются обратные вызовы. Как видим, подставляя соответствующие аргументы, можно запускать на выполнение различные участки программного кода. Отсюда можно сделать вывод, что обратные вызовы целесообразно использовать в случаях, когда требуется **динамическая модификация поведения программы во время выполнения.**

1.3. Модель обратных вызовов

1.3.1. Определения и термины

Модель обратных вызовов изображена на Рис. 7. Структурно она состоит из двух частей: исполнитель и инициатор.

Исполнитель – это компонент, в который упаковывается код обратного вызова (исполняемый код). Исполнитель также содержит контекст, который представляет собой совокупность данных, влияющих на поведение исполняемого кода.

Инициатор – это компонент, который осуществляет обратный вызов. Перед началом работы выполняется настройка, при которой исполнитель как аргумент вместе с контекстом сохраняются в инициаторе. Затем инициатор запускается, и в нужный момент, используя хранимый аргумент, он делает вызов исполняемого кода. В качестве входных параметров в этот код передается сохраненный контекст и информация вызова, которая представляет собой значения, формируемые инициатором.

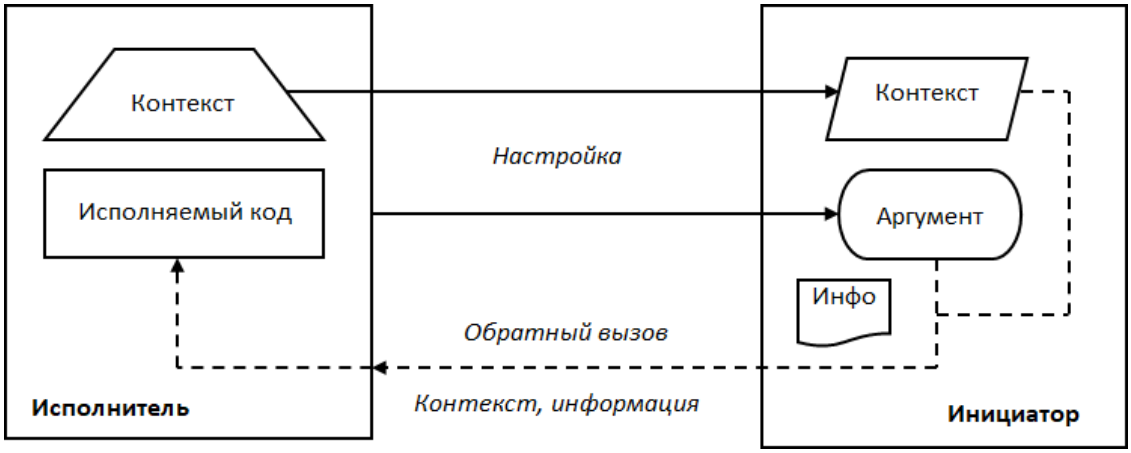

Рис. 7. Модель обратных вызовов

Дадим формальные определения используемых терминов.

Исполнитель: компонент, который реализует исполняемый код обратного вызова.
Инициатор: компонент, который осуществляет обратный вызов.
Аргумент: хранимая точка входа в код обратного вызова.
Настройка: процедура сохранения аргумента.
Информация вызова: значения, которые формируются инициатором и передаются в исполнитель.
Контекст: множество переменных и состояний, которые влияют на поведение исполняемого кода.

В процессе реализации обратного вызова нам нужно ответить на следующие вопросы.
1. Как оформить исполняемый код, чтобы он мог быть вызван инициатором?
2. Как хранить аргумент?
3. Как передавать контекст?

Различные способы реализации дают свои ответы на поставленные вопросы. Но прежде, чем приступить к их изучению, необходимо осветить еще несколько моментов.

1.3.2. Контекст

Вне зависимости от того, каким способом реализован исполнитель, исполняемый код всегда находится внутри тела некоторой функции. Если результат выполнения функции зависит только от входных параметров, то контекст оказывается ненужным. В качестве примера можно привести случай, когда обратный вызов возвращает результат сравнения переданных аргументов.

Однако такая ситуация встречается далеко не всегда, в большинстве случаев требуется знать значения переменных, внешних по отношению к функции исполнителя. Другими словами, необходимо получить контекст вызова.

Важность контекста можно проиллюстрировать на следующем примере. Пусть мы реализуем подсистему сетевого обмена, которая осуществляет передачу данных по каналам связи. Для управления каналом создается отдельный класс, задачей которого является формирование и отправка пакетов через вызовы соответствующих функций операционной системы. Операционная система, в свою очередь, подтверждает о доставке пакета через обратный вызов (Рис. 8). Как нам узнать в коде обработчика вызова, для какого класса предназначено подтверждение? Здесь-то и необходим контекст вызова, в качестве которого выступает указатель на класс, управляющий нужным каналом. Этот указатель не хранится внутри кода обработчика, он должен каким-то образом ему передаваться. Другими словами, обработчик вызова должен получить контекст. Различные реализации обратных вызовов предлагают свои собственные способы передачи и интерпретации контекста, которые будут подробно рассматриваться в соответствующих главах.

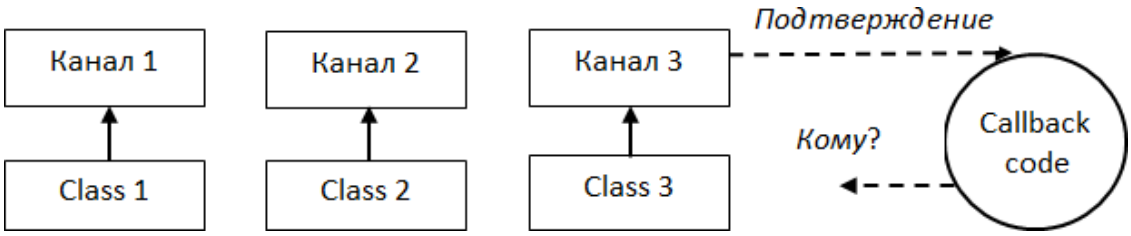

Рис. 8. Сетевой обмен и контекст вызова

1.4. Архитектурный дизайн вызовов

1.4.1. Синхронные и асинхронные вызовы

С точки зрения архитектурного дизайна обратные вызовы можно разделить на синхронные и асинхронные. Если при вызове какой-либо функции инициатора обратный вызов происходит внутри тела этой функции, которая затем возвращает управление, то вызов является синхронным (другое название – блокирующий). Если обратный вызов может произойти в любое время, то этот вызов является асинхронным (другое название – отложенный).

> **Синхронный вызов** – архитектурный дизайн, в котором при вызове функции инициатора обратный вызов происходит до выхода из тела этой функции.
>
> **Асинхронный вызов** – архитектурный дизайн, в котором обратный вызов может быть выполнен в любое время.

Различие между синхронными и асинхронными вызовами проиллюстрировано на Рис. 9. В первом случае поток управления входит в функцию *Run*, из которой вызывается функция обратного вызова, и затем управление возвращается в точку вызова. Во втором случае функция *Run* вначале производит сохранение аргумента, а затем выполняет некоторое действия (*Action*), внутри которого делает обратный вызов. В качестве действия может выступать циклический опрос, обработка очереди сообщений, создание отдельного потока и т. п.

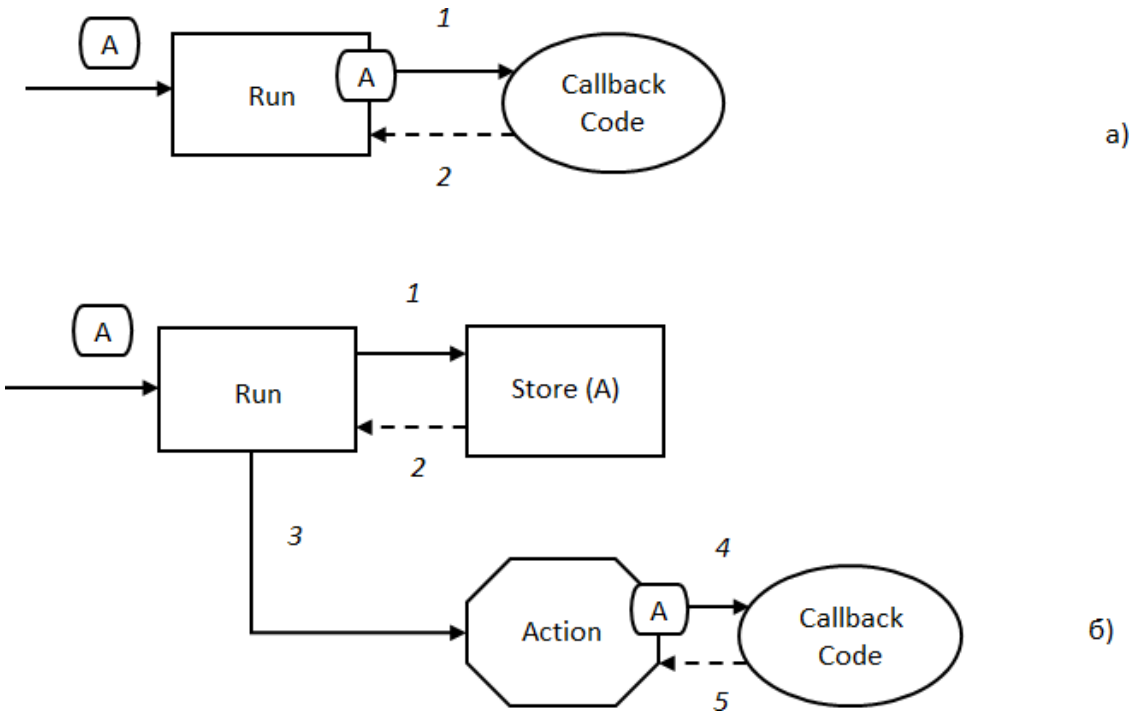

Рис. 9. Синхронные и асинхронные вызовы: а) синхронный; б) асинхронный

Особенностью реализации синхронных вызовов является то, что здесь не нужно хранить аргумент: он передается как параметр в функцию инициатора и используется только внутри этой функции. В случае асинхронных вызовов необходима предварительная настройка аргумента, который должен быть сохранен в какой-либо нелокальной переменной.

1.4.2. Использование вызовов в API

API (Application Programming interface, интерфейс прикладных программ) – это программный код, реализующий некоторую функциональность, а также объявления, через которые некоторая программа может вызывать этот код. Указанные объявления реализуют интерфейс API.

Интерфейс API – набор объявлений для вызова кода API.

При проектировании API должны соблюдаться следующие требования.

1. *Интерфейс должен следовать определённым соглашениям*. Следуя указанным соглашениям, стороннее приложение может осуществлять вызовы кода API.
2. *Интерфейс должен быть изолирован от реализации.* Должна существовать возможность изменения кода реализации без изменения интерфейса.
3. *Код должен быть подготовлен к выполнению*. Для C++ это означает, что код должен быть предварительно откомпилирован.

С точки зрения C++ интерфейсы API могут быть разделены на два больших класса.

Системный API: интерфейс объявляется в виде набора функций, поддерживающих стандартный протокол вызова. Любая программа, независимо от того, на каком языке она написана, может обратиться к указанному API путем вызова функций интерфейса. Как правило, системные API реализуются в виде динамически разделяемых библиотек. В качестве примера можно назвать всем известный Windows API, реализация которого находится в системной библиотеке User32.dll. Любое приложение может загрузить эту библиотеку и вызывать требуемые функции для выполнения системных вызовов.

C++ API: интерфейс объявляется в виде набора классов C++. Как и системные, C++ API чаще всего реализуются в виде динамических библиотек, но могут поставляться также в виде статических. Использовать такие API могут только те программные компоненты, которые могут интерпретировать вызовы C++. Так, например, среда выполнения для языка Python может вызывать методы классов C++, а вот у Visual Basic такая возможность отсутствует.

Интерфейсы системных API должны объявляться в стиле языка C, т. е. в них должны использоваться функции с фиксированным числом параметров и простые структуры данных, такие, как числа, символы, указатели и структуры. Это связано с тем, что такие объявления следуют стандартным соглашениям операционной системы, в силу чего любая программа, независимо от используемого языка программирования (даже написанная на ассемблере), может использовать указанный API. Однако из-за требования описания интерфейсов в стиле C на реализацию обратных вызовов накладываются ограничения, которые будут рассматриваться в соответствующих главах.

1.5. Итоги

Обратный вызов – это паттерн, в котором какой-либо исполняемый код как аргумент передается в другой код, при этом ожидается, что через сохраненный аргумент исполняемый код будет запущен в нужный момент времени. Основные классы задач, решаемые с помощью обратных вызовов, следующие: запрос данных; вычисления по запросу; перебор элементов; уведомления о событиях.

Модель обратных вызовов включает в себя следующие понятия: исполнитель, инициатор, аргумент, настройка, контекст.

В синхронных вызовах при вызове функции инициатора обратный вызов осуществляется до выхода из тела функции. В асинхронных вызовах вызов может быть выполнен в любое время.

Обратные вызовы часто используются в системных и C++ API. При использовании в системных API на реализацию обратных вызовов накладываются ограничения.

Рассмотрев общую концепцию, приступим к обзору способов реализации обратных вызовов.

2. Реализация обратных вызовов

2.1. Указатель на функцию

2.1.1. Концепция

Графическое изображение реализации обратного вызова с помощью указателя на функцию представлено на Рис. 10. Исполнитель реализован в виде глобальной функции, в качестве контекста могут выступать любые данные. При настройке указатель на функцию как аргумент и указатель на данные как контекст сохраняются в инициаторе. Инициатор осуществляет обратный вызов посредством вызова функции через сохраненный указатель, передавая ей требуемые значения и контекст – указатель на данные. Поскольку инициатор не интерпретирует контекст и не выполняет с ним никаких операций, для хранения контекста используется нетипизированный указатель.

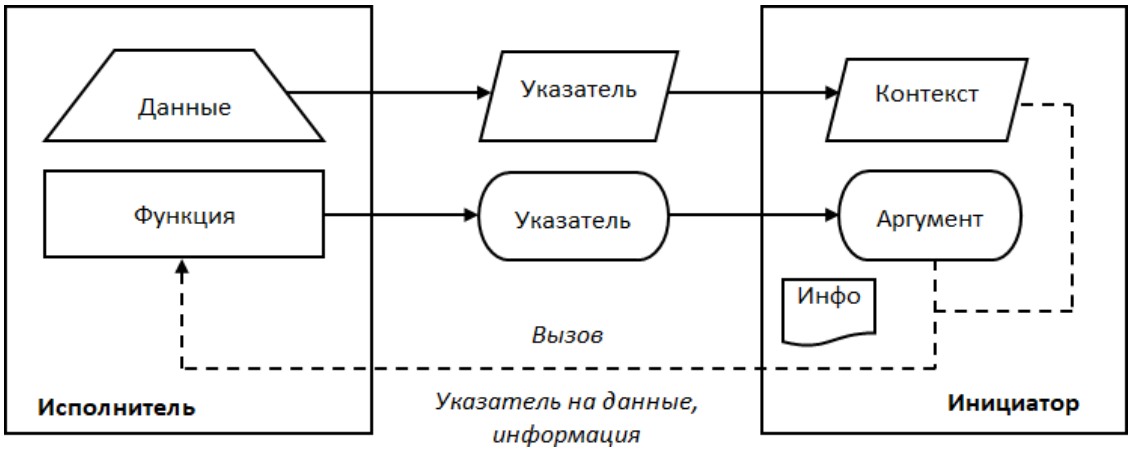

Рис. 10. Обратный вызов с указателем на функцию

2.1.2. Инициатор

Реализация инициатора представлена в Листинг 1[2].

Листинг 1.Иинициатор с указателем на функцию

```
typedef void(*ptr_callback) (int eventID, void* pContextData);  //

ptr_callback ptrCallback = NULL;  // (2)
void* contextData = NULL;         // (3)

void setup(ptr_callback pPtrCallback, void* pContextData)  // (4)
```

[2] Мы здесь (и в дальнейших листингах тоже) не будем разделять заголовочные файлы и файлы реализации: это всего лишь пример, а разделение загромождает описание и усложняет понимание.

```
    {
       ptrCallback = pPtrCallback;
       contextData = pContextData;
    }

    void run()                                  // (5)
    {
      int eventID = 0;
      //Some actions
      ptrCallback(eventID, contextData);  // (6)
    }
```

В строке 1 объявлен тип – указатель на функцию, в строке 2 объявлена переменная этого типа, в строке 3 объявлен указатель на данные контекста. В строке 4 объявлена функция для настройки указателей, в которой инициализируются соответствующие переменные. В строке 5 объявлена функция запуска, внутри этой функции инициатор в строке 6 производит вызов функции по сохраненному указателю. Сигнатура функции, объявленная в строке 1, в качестве первого параметра принимает значение, которое передается инициатором, т. е. информацию вызова, а второй параметр – это контекст. Указанная сигнатура здесь только для примера; конечно же, в зависимости от поставленных задач количество параметров и их порядок может быть произвольным. Мы также опустили моменты, связанные с созданием потока, ожиданием окончания работы сервера и т. п. – для понимания принципов организации вызова это несущественно.

Итак, мы реализовали инициатор в процедурно-ориентированном дизайне. Приведенная реализация имеет серьезный недостаток: указатель на функцию и указатель на контекст хранятся в глобальных переменных. Это создает множество проблем: изменения настроек указателей в разных частях программы не изолированы, т. е. влияют друг на друга; инициатор может работать только с одним-единственным исполнителем; невозможна одновременная работа нескольких потоков. Выходом из сложившейся ситуации будет реализация инициатора в объектно-ориентированном дизайне[3] (Листинг 2).

Листинг 2. Инициатор с указателем на функцию в объектно-ориентированном дизайне

```
    class Initiator  //(1)
    {
    public:
      using ptr_callback  =  void(*) (int, void*);
  //(2)

      void setup(ptr_callback pPtrCallback, void* pContextData)    // (
      {
          ptrCallback = pPtrCallback; contextData = pContextData;  // (
      }
```

[3] Конечно же, описанные проблемы могут быть решены и в процедурном дизайне, но код при этом значительно усложняется. В общем-то, объектно-ориентированная парадигма и разрабатывалась как средство борьбы с возрастающей сложностью программного кода.

```
  void run()                                    // (5)
  {
      int eventID = 0;
      //Some actions
      ptrCallback (eventID, contextData);  // (6)
}
private:
  ptr_callback ptrCallback = nullptr;       // (7)
  void* contextData = nullptr;              // (8)
};
```

В строке 1 мы объявляем класс – инициатор, в строке 2 мы объявляем тип указателя на функцию. В строке 3 объявляем функцию настройки указателей, соответствующие переменные – (указатель на функцию и указатель на контекст) объявлены соответственно в строках 7 и 8. В строке 5 объявлена функция запуска, внутри этой функции в строке 6 производится вызов функции по соответствующему указателю. Как видим, объектная реализация практически полностью повторяет процедурную, только все объявления сделаны внутри класса. Другими словами, мы провели инкапсуляцию данных и процедур внутри некоторой сущности, в качестве которой выступает класс.

Конечно, поскольку мы программируем на C++, мы должны следовать объектно-ориентированному дизайну, и любые реализации делать в его рамках. Для чего тогда мы привели реализацию инициатора в процедурном дизайне, в стиле языка C? Дело в том, что процедурный дизайн является единственно возможным для проектирования системных API, поскольку в объявлениях интерфейсов таких API допускается использование только глобальных функций и простых структур данных (см. п. 1.4.2).

2.1.3. Исполнитель

Реализация исполнителя для случая, когда инициатор разработан в процедурном дизайне, представлена в Листинг 3.

Листинг 3. Исполнитель для инициатора в процедурном дизайне

```
struct ContextData  // (1)
{
    //some context data
};

void callbackHandler(int eventID, void* somePointer)     // (2)
{
  //It will be called by initiator
  ContextData* pContextData = (ContextData*)somePointer;  // (3)
  //Do something here
}

int main()                                    // (4)
{
```

```
  ContextData clientContext;                // (5)
  setup(callbackHandler, &clientContext);  // (6)
  run();                                    // (7)
  //Wait finish
}
```

В строке 1 объявляется тип данных для контекста. Структура здесь показана для примера, в качестве контекста могут выступать любые типы: числа, указатели, смеси и т. п. В строке 2 объявляется функция – обработчик обратного вызова, ее сигнатура должна совпадать с сигнатурой, с которой работает инициатор. Указанная функция будет вызвана инициатором, в нее будут переданы два параметра: первый передается инициатором (информация вызова, в нашем случае это **eventID**), а второй – это контекст. Клиент должен интерпретировать контекст; нет другого способа это сделать, кроме как приведением типов (строка 3).

Далее, в строке 4 объявлена основная функция, в которой осуществляются все необходимые операции. В строке 5 объявляются данные контекста; в строке 6 производится настройка обратного вызова, в функцию настройки передаются указатель на функцию-обработчик и указатель на контекст; в строке 7 инициатор запускается.

Реализация исполнителя для случая, когда инициатор реализован в объектно-ориентированном дизайне, представлена в Листинг 4. Как видим, она очень похожа на предыдущую реализацию с той разницей, что мы объявляем экземпляр класса-инициатора (строка 5), и все вызовы осуществляем через вызов соответствующих методов класса.

Листинг 4. Исполнитель для инициатора в объектно-ориентированном дизайне

```
struct ContextData // (1)
{
  //some context data
};

void callbackHandler(int eventID, void* somePointer) // (2)
{
  //It will be called by initiator
  ContextData* pContextData = static_cast<ContextData*>(somePointer
  //Do something here
}

int main() // (4)
{
  Initiator  initiator;                                  // (5)
  ContextData clientContext;                             // (6)
  initiator.setup(callbackHandler, &clientContext); // (7) callback
   initiator.run();
// (8) initiator has been run
  //Wait finish
}
```

2.1.4. Синхронный вызов

Реализация инициатора для синхронного вызова приведена в Листинг 5. Как видим, для синхронных вызовов код значительно упрощается: нет необходимости хранить переменные, информация вызова и контекст передаются непосредственно в функцию.

Листинг 5. Инициатор для синхронного обратного вызова с указателем на функцию

```
using ptr_callback  =  void(*) (int, void*);

void run(ptr_callback ptrCallback, void* contextData = nullptr)
{
  int eventID = 0;
  //Some actions
  ptrCallback (eventID, contextData);
}
```

2.1.5. Преимущества и недостатки

Достоинства и недостатки реализации обратных вызовов с помощью указателя на функцию представлены в Табл. 1.

Табл. 1. Преимущества и недостатки обратных вызовов с указателем на функцию

Преимущества	Недостатки
Простая реализация	Инициатор хранит контекст исполнителя
Независимость инициатора и исполнителя	Небезопасный способ трансляции контекста
Совместим с кодом на языке C	
Подходит для реализации любых API	

Простая реализация. Как мы видели, инициатор реализуется достаточно просто: две переменных, синтаксис вызова функции через указатель очень похож на вызов обычной функции.

Независимость инициатора и исполнителя. Любое изменение кода исполнителя никак не влияет на код инициатора, который при этом остается неизменным

Совместим с кодом на языке C. В некоторых случаях приходится разрабатывать смешанный код, т. е. часть кода пишется C, а часть – на C++. Если код исполнителя написан на C++, и этот код должен быть вызван инициатором, написанным на C, то использование указателей на функцию является единственно доступным механизмом. [4]

[4] В качестве примера можно привести практику моделирования embedded-систем. В самом общем виде Embedded-системы представляют собой микроконтроллер, который встраивается в какое-либо устройство и выполняет функции управления, мониторинга и контроля. В силу определенных причин так сложилось, что ПО для управляющих контроллеров (такое ПО называют firmware) пишется на языке C. В процессе разработки подобных устройств часто используется моделирование, когда firmware запускается на обычном компьютере в имитационном окружении, а реальные аппаратные устройства заменяются их программными моделями. Модели и имитаторы обычно пишутся на языке C++, а firmware, как правило, написано на C – получается смешанный код.

Подходит для реализации любых API. Можно реализовать как C++, так и системные API. Для C++ API инициатор разрабатывается в виде набора классов, для системных API – в виде набора функций.

Инициатор хранит контекст исполнителя. Как мы видели, инициатор вынужден сохранять контекст исполнителя. Это усложняет реализацию и способствует увеличению расхода памяти.

Небезопасный способ трансляции контекста. Контекст передается клиенту в виде нетипизированного указателя, интерпретация указателя возлагается на клиента. В большой программной системе это чревато ошибками, поскольку нет никакой возможности проверить корректность полученного указателя.

2.2. Указатель на статический метод класса

2.2.1. Концепция

Графическое изображение обратного вызова с помощью указателя на статический метод класса представлено на Рис. 11. Исполнитель реализуется в виде класса, код упаковывается в статический метод класса, в качестве контекста выступает указатель на экземпляр класса. При настройке указатель на статический метод как аргумент и указатель на класс как контекст сохраняются в инициаторе. Инициатор осуществляет обратный вызов посредством вызова метода, передавая ему требуемую информацию и контекст – указатель на класс.

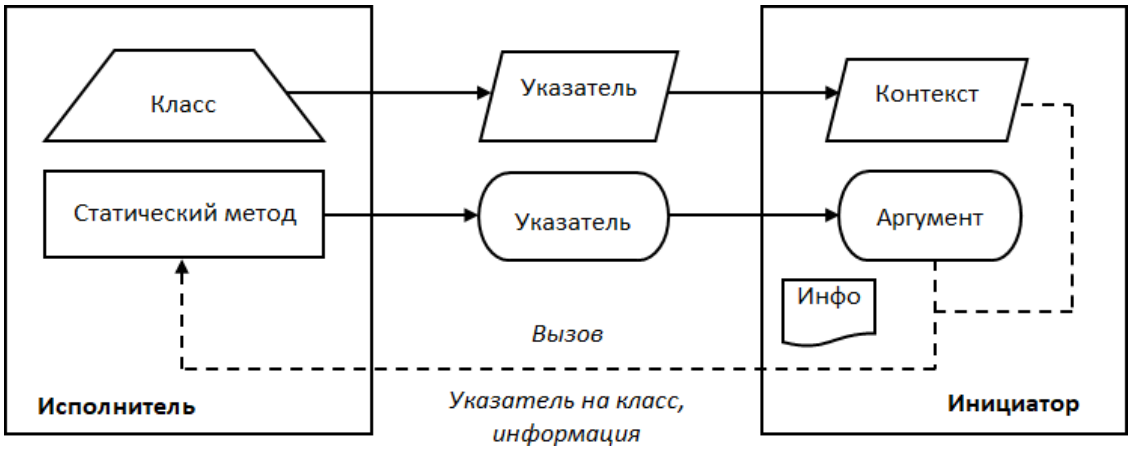

Рис. 11. Обратный вызов с указателем на статический метод класса

2.2.2. Инициатор

По своей сути статический метод класса – это обычная функция, ограниченная областью видимости класса. Поэтому реализация инициатора, представленная в Листинг 6, практически полностью повторяет реализацию для указателей на функцию, только в качестве контекста выступает указатель на экземпляр класса.

Листинг 6. Инициатор с указателем на статический метод класса

```
class Executor;  //(1)

class Initiator  // (2)
{
public:
 using ptr_callback_static = void(*) (int, Executor*);
// (3)

  void setup(ptr_callback_static pPtrCallback, Executor* pContextDa
  {
  ptrCallback = pPtrCallback; contextData = pContextData;
// (5)
```

```
    }

    void run()                              //  (6)
    {
      int eventID = 0;
      //Some actions
      ptrCallback(eventID, contextData);  // (7)
    }

  private:
    ptr_callback_static ptrCallback = nullptr;  // (8)
    Executor* contextData = nullptr;            // (9)
  };
```

В строке 1 делается предварительное объявление типа класса исполнителя. В строке 2 объявляется класс – инициатор, в строке 3 объявляется тип указателя на функцию с контекстом – экземпляром класса. В строке 4 объявлена функция для настройки указателей, соответствующие переменные (указатель на статический метод и указатель на контекст – экземпляр класса) объявлены в строках 8 и 9. В строке 6 объявлена функция запуска, внутри этой функции в строке 7 производится вызов функции по соответствующему указателю с передачей информации вызова и контекста.

2.2.3. Исполнитель

Реализация исполнителя приведена в Листинг 7.

Листинг 7. Исполнитель с указателем на статический метод класса

```
  class Executor                    // (1)
  {
  public:
    Executor(Initiator* initiator)  // (2)
    {
      initiator->setup(callbackHandler, this);
    }

    static void callbackHandler(int eventID, Executor* executor)  //
    {
      //It will be called by initiator
      executor->onCallbackHandler(eventID);
// (4)
    }

  private:
    void onCallbackHandler(int eventID)  // (5)
    {
      //Do what is necessary
    }
  };
```

```
int main() // (6)
{
  Initiator initiator;             // (7)
  Executor executor(&initiator);   // (8)
  initiator.run();                 // (9)
  //Wait finish
}
```

В строке 1 объявляется класс – исполнитель. В строке 2 объявляется конструктор с входным параметром – указателем на инициатор, здесь происходит настройка обратного вызова.[5]

В строке 3 объявлен статический метод как обработчик обратного вызова. Входными параметрами здесь являются информация вызова (в нашем случае это **eventID**) и указатель на контекст, в качестве которого выступает указатель на экземпляр класса. Внутри метода можно обращаться к содержимому класса, используя полученный указатель как квалификатор. Таким образом, прямо здесь можно реализовать код обработчика, а можно вызвать обычный (нестатический) метод класса (строка 4).

Далее, в строке 6 объявлена основная функция, в которой осуществляются все необходимые операции. В строке 7 объявлен класс-инициатор; в строке 8 объявлен класс- исполнитель, в конструктор передается указатель на инициатор; в строке 9 происходит запуск инициатора.

Особенностью реализации исполнителя с помощью указателя на статический метод является возможность работы с инициатором, предназначенным для указателей на функцию. В этом случае метод класса в качестве контекста должен принимать нетипизированный указатель с последующим приведением типов. Пример использования показан в Листинг 8, инициатор здесь используется из Листинг 1 п. 2.1.2.

Листинг 8. Исполнитель с указателем на статический метод класса для инициатора с нетипизированным контекстом

```
class Executor  // (1)
{
public:
  Executor()    // (2)
  {
    setup(callbackHandler, this);
  }

  static void callbackHandler(int eventID, void* somePointer)  // (
  {
    //It will be called by initiator
    Executor* executor = static_cast<Executor*>(somePointer);
// (4)
    executor->onCallbackHandler(eventID);
  }
```

[5] Это необязательно делать в конструкторе, соответствующие операции можно выполнить после объявлений экземпляров инициатора и исполнителя в функции main. Однако инициализация в конструкторе представляется более удобной, потому что настройка вызова будет сделана сразу при объявлении экземпляра класса – исполнителя без дополнительных операций.

```
private:
  void onCallbackHandler(int eventID)  // (5)
  {
    //Do what is necessary
  }
};

int main()             // (6)
{
  Executor executor;   // (7)
  run();               // (8)
  //Wait finish
}
```

Настройка обратного вызова осуществляется в конструкторе (строка 2). В обработчике обратного вызова (строка 3) мы делаем приведение типов (строка 4), чтобы получить указатель на экземпляр класса. В главной функции (строка 6) происходит запуск инициатора.

2.2.4. Синхронный вызов

Реализация инициатора для синхронного вызова приведена в Листинг 9. Как видим, она практически полностью повторяет реализацию, рассмотренную в предыдущей главе, только в качестве указателя на контекст используется указатель на экземпляр класса.

Листинг 9. Инициатор для синхронного обратного вызова с указателем на статический метод класса

```
class Executor;
using ptr_callback_static = void(*) (int, Executor*);

void run(ptr_callback_static ptrCallback, Executor * contextData =
{
  int eventID = 0;
  //Some actions
  ptrCallback (eventID, contextData);
}
```

2.2.5. Преимущества и недостатки

Преимущества и недостатки реализации обратных вызовов с помощью указателя на статический метод класса приведены в Табл. 2.

Табл. 2. Преимущества и недостатки обратных вызовов с указателем на статический метод класса

Преимущества	Недостатки
Простая реализация	Инициатор хранит контекст исполнителя
Совместим с инициатором в процедурном дизайне	

Простая реализация. Не сложнее, чем для указателей на функцию.

Совместим с инициатором в процедурном дизайне. Можно использовать для работы с системными API.

Инициатор хранит контекст исполнителя. Так же, как и в случае указателей на функцию, усложняет реализацию и способствует увеличению расхода памяти.

2.3. Указатель на метод-член класса

2.3.1. Концепция

В предыдущей главе мы рассматривали использование указателя на статический метод класса, в который в качестве контекста передавали указатель на экземпляр класса. А почему бы нам напрямую не вызвать метод-член класса, минуя прослойку в виде статического метода, из которого вызывается метод-член класса? Для этого нам понадобятся указатель на класс и указатель на метод.

Графическое изображение обратного вызова с помощью указателя на метод-член класса (далее – метод класса) представлено на Рис. 12. Исполнитель реализуется в виде класса, код упаковывается в метод класса, в качестве контекста выступает экземпляр класса. При настройке указатель на метод и указатель на класс как как аргументы сохраняются в инициаторе. Инициатор осуществляет обратный вызов посредством вызова метода, передавая ему требуемую информацию. Контекст здесь передавать не нужно, поскольку внутри метода доступно все содержимое класса.

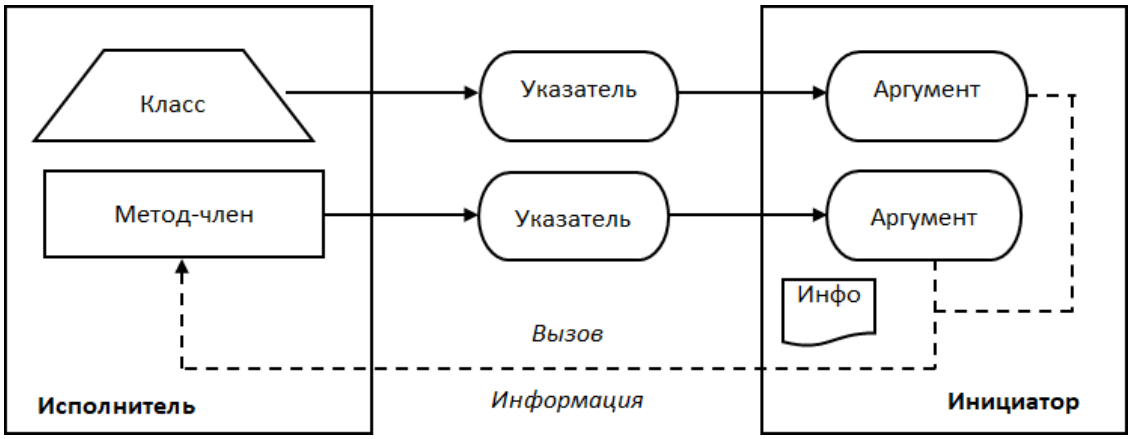

Рис. 12. Реализация обратного вызова с помощью указателя на метод-член класса

2.3.2. Инициатор

Реализация инициатора приведена в Листинг 10.

Листинг 10. Инициатор с указателем на метод-член класса

```
class Executor;  // (1)

class Initiator  // (2)
{
public:
  using ptr_callback_method = void(Executor::*)(int);  // (3)

void setup(Executor* pArgCallbackClass, ptr_callback_method pArgCallbackMethod)
  // (4)
```

```
  {
    ptrCallbackClass = argCallbackClass; ptrCallbackMethod = argCal
  }

  void run()  // (6)
  {
    int eventID = 0;
    //Some actions
    (ptrCallbackClass->*ptrCallbackMethod)(eventID);  // (7)
  }

private:
  Executor* ptrCallbackClass = nullptr;              // (8)
  ptr_callback_method ptrCallbackMethod = nullptr;  // (9)
};
```

В строке 1 делается предварительное объявление типа класса исполнителя. В строке 2 объявляется класс-инициатор, в строке 3 объявляется тип указателя для класса-исполнителя. В строке 4 объявляется функция для настройки указателей, соответствующие переменные (указатель на метод класса и указатель на экземпляр класса) объявлены в строках 8 и 9. В строке 6 объявлена функция запуска, внутри этой функции в строке 7 через соответствующий указатель производится вызов метода класса.

2.3.3. Исполнитель

Реализация исполнителя приведена в Листинг 11.

Листинг 11. Исполнитель с указателем на метод-член класса

```
class Executor                          // (1)
{
public:
  void callbackHandler(int eventID)  // (2)
  {
    //It will be called by initiator
  }
};

int main()                                                    // (3)
{
  Initiator initiator;                                        // (4)
  Executor executor;                                          // (5)
  initiator.setup(&executor, &Executor::callbackHandler);  // (6)
  initiator.run();                                            // (7)
}
```

В строке 1 объявляется класс-исполнитель. В строке 2 объявлен метод класса, который будет выполнять функцию обработчика обратного вызова. В указанный метод передается информация вызова (в нашем случае это **eventID**). В строке 3 объявлена основная функция, в

которой осуществляются все необходимые операции. В строке 4 объявлен класс-инициатор, в строке 5 объявлен класс-исполнитель. В строке 6 осуществляется настройка обратного вызова, в строке 7 производится запуск инициатора.

2.3.4. Управление контекстом

Рассматриваемая реализация позволяет осуществлять управление контекстом тремя способами: настройка экземпляра класса-исполнителя, настройка указателя на метод, переопределение виртуальных функций. Это приводит к интересным эффектам.

Пусть у нас будут объявления классов-исполнителей с наследованием, как показано в Листинг 12. Графически иерархия наследования изображена на Рис. 13.

Листинг 12. Классы-исполнители с наследованием

```
class Executor
{
public:
  virtual void callbackHandler1(int eventID);
  virtual void callbackHandler2(int eventID);
};

class Executor1: public Executor
{
public:
  void callbackHandler1(int eventID) override;
};

class Executor2: public Executor
{
public:
  void callbackHandler2(int eventID) override;
};

class Executor3: public Executor1, public Executor2
{
};
```

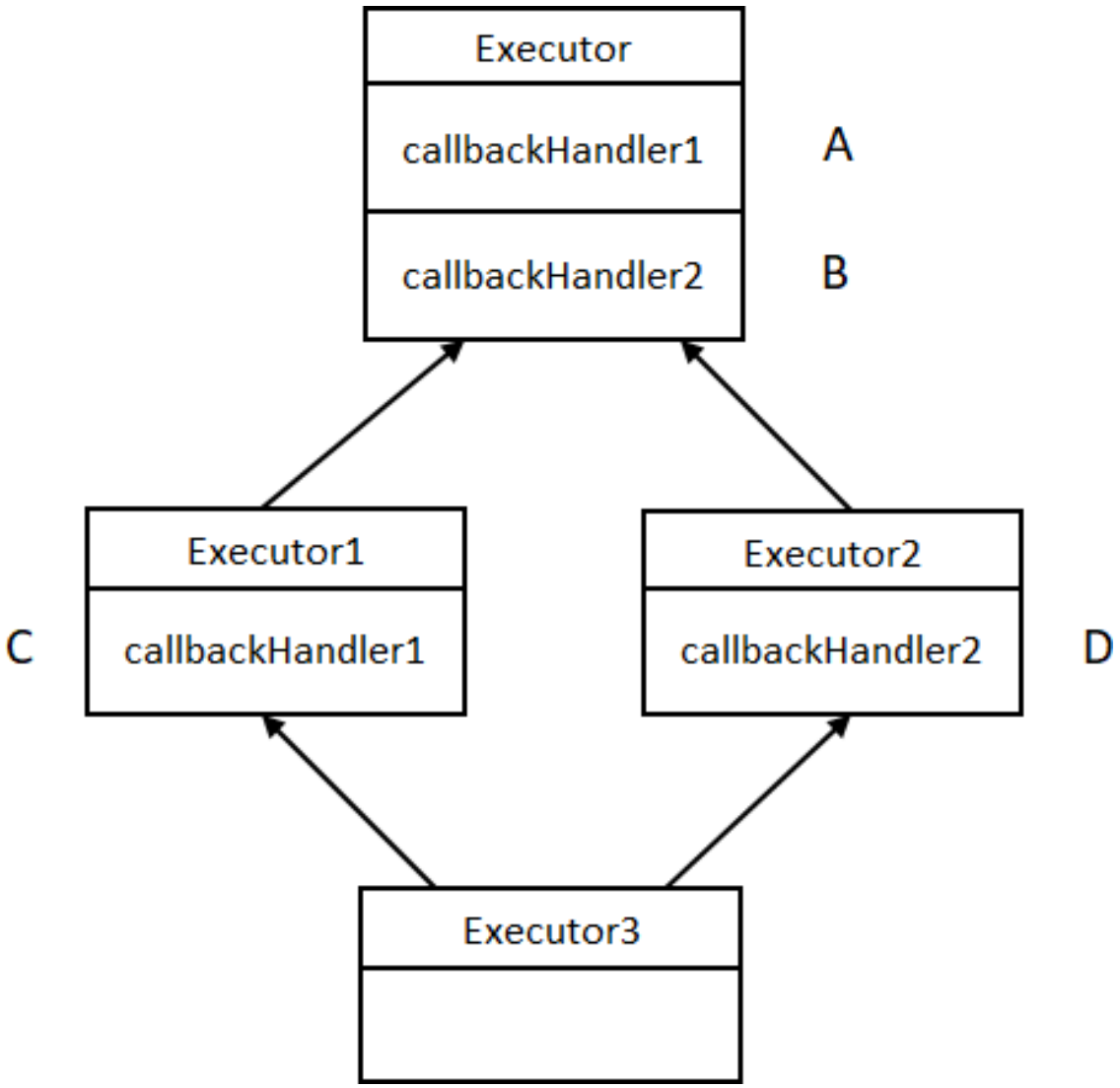

Рис. 13. Иерархия наследования классов-исполнителей

Итак, будем назначать различные указатели на экземпляры классов и методы-члены, как показано в Листинг 13.

Листинг 13. Настройка указателей на классы и методы

```
int main()
{
  Initiator initiator;
  Executor  executor;
  Executor1 executor1;
  Executor2 executor2;
  Executor3 executor3;

  initiator.setup(&executor, &Executor::callbackHandler1);   // (1)
  initiator.setup(&executor, &Executor::callbackHandler2);   // (2)
  initiator.setup(&executor1, &Executor::callbackHandler1);  // (3)
  initiator.setup(&executor1, &Executor::callbackHandler2);  // (4)
  initiator.setup(&executor2, &Executor::callbackHandler1);  // (5)
```

```
      initiator.setup(&executor2, &Executor::callbackHandler2);  // (6)

                                                                 //
initiator.setup(&executor3,    &Executor::callbackHandler1);    //
Incorrect, base class is ambiguous  // (7)
                                                                 //
initiator.setup(&executor3,    &Executor::callbackHandler2);    //
Incorrect, base class is ambiguous  // (8)

      initiator.setup((Executor1*)&executor3, &Executor::callbackHandle
      initiator.setup((Executor1*)&executor3, &Executor::callbackHandle
      initiator.setup((Executor2*)&executor3, &Executor::callbackHandle
      initiator.setup((Executor2*)&executor3, &Executor::callbackHandle
    }
```

В строках 1 и 2 все прозрачно: какой метод назначен, такой и будет вызван.

В строке 3 мы назначаем указатель на метод **Executor::callbackHandler1**, но поскольку в классе **Executor1** он переопределен, будет вызван метод **Executor1::callbackHandler1**.

В строке 4 мы назначаем указатель на **Executor::callbackHandler2**; в классе **Executor1** такого метода нет (т.е. он не переопределен), поэтому будет вызван метод базового класса **Executor::callbackHandler2**.

В строке 5 мы назначаем указатель на **Executor::callbackHandler1**; в классе **Executor2** метод не переопределен, поэтому будет вызван метод базового класса **Executor::callbackHandler2**.

В строке 6 мы назначаем указатель на **Executor::callbackHandler2**; в классе **Executor2** он переопределен, поэтому будет вызван метод **Executor2:: callbackHandler2**.

С классом **Executor3** ситуация еще интереснее, поскольку он использует множественное наследование[6]. Мы не можем напрямую назначать указатели на методы базового класса, как это приведено в строках 7 и 8, потому что если взглянуть на иерархию наследования, то можно увидеть, что к базовому классу можно добраться двумя путями – через **Executor1** либо через **Executor2**. Таким образом, компилятор не знает, по какому пути выполнять поиск методов, и выдает ошибку. По указанной причине мы должны явно указать в цепочке наследования класс-предшественник. Если в пути наследования какая-нибудь функция окажется переопределена, то она будет вызвана, в противном случае будет вызвана функция базового класса.

В строке 9 мы в качестве предшественника указываем класс **Executor1** и назначаем указатель на метод **callbackHandler1**. В **Executor1** этот метод переопределен, и он будет вызван. В строке 10 мы назначаем указатель на метод **callbackHandler2**; в **Executor1** этот метод не переопределен, поэтому будет вызван метод базового класса **Executor::callbackHandler2**. Если мы в качестве предшественника будем указывать **Executor2**, как это показано в строках 11 и 12, то получится все наоборот: в строке 11 будет вызван метод базового класса **Executor:: callbackHandler1**, а в строке 12 будет вызван соответствующий переопределенный метод **Executor2::callbackHandler2**.

Для наглядности сведем результаты в Табл. 3.

[6] Вообще, множественное наследование – неоднозначный механизм, который часто подвергается критике. В большинстве современных языков (например, Java, C#, Ruby и др.) множественное наследование не поддерживается. Тем не менее, в C++ множественное наследование существует, поэтому необходимо рассмотреть и такой случай.

Табл. 3. Вызовы методов по цепочке наследования

Настойка	Вызов	На рисунке
&executor, &Executor::callbackHandler1	Executor::callbackHandler1	A
&executor, &Executor::callbackHandler2	Executor::callbackHandler2	B
&executor1, &Executor::callbackHandler1	Executor1::callbackHandler1	C
&executor1, &Executor::callbackHandler2	Executor::callbackHandler2	B
&executor2, &Executor::callbackHandler1	Executor::callbackHandler1	C
&executor2, &Executor::callbackHandler2	Executor2::callbackHandler2	D
(Executor1*)&executor3, &Executor::callbackHandler1	Executor1::callbackHandler1	C
(Executor1*)&executor3, Executor::callbackHandler2	Executor::callbackHandler2	B
(Executor2*)&executor3, &Executor::callbackHandler1	Executor::callbackHandler1	A
(Executor2*)&executor3, Executor::callbackHandler2	Executor2::callbackHandler2	D

Используя рассмотренные способы управления контекстом, можно реализовать довольно изощренную логику обработки и динамически ее изменять в процессе выполнения программы.

2.3.5. Синхронный вызов

Реализация инициатора для синхронного вызова представлена в Листинг 14. В отличие от асинхронного вызова, здесь аргументы не хранятся, а передаются как входные параметры функции.

Листинг 14. Инициатор для синхронного обратного вызова с указателем на метод-член класса

```
class Executor;
using ptr_method_callback_t = void(Executor::*)(int);

void run(Executor* ptrClientCallbackClass,
ptr_method_callback_t ptrClientCallbackMethod)
{
int eventID = 0;
//Some actions
(ptrClientCallbackClass->*ptrClientCallbackMethod)(eventID);
}
```

2.3.6. Преимущества и недостатки

Преимущества и недостатки реализации обратных вызовов с помощью указателя на метод – член класса приведены в Табл. 4.

Табл. 4. Преимущества и недостатки реализации обратных вызовов с помощью указателя на метод-член класса

Преимущества	Недостатки
Гибкость	Сложность
Отсутствие трансляции контекста	Тип класса должен объявляться в инициаторе
	Инициатор должен хранить указатель на метод и указатель на класс

Гибкость. Управлять контекстом можно тремя способами, подобные возможности отсутствуют в других реализациях.

Отсутствие трансляции контекста. Контекст транслировать не нужно, метод-член имеет полный доступ к содержимому класса.

Сложность. Код получается довольно громоздким и запутанным.

Тип класса должен объявляться в инициаторе. Здесь достаточно только предварительного объявления класса. Полное объявление класса в инициаторе делать необязательно и даже нежелательно, потому что логически это обработчик обратного вызова, то есть он относится к исполнителю и должен быть в нем реализован. Тем не менее, требование предварительного объявления класса ограничивает независимость исполнителя: он может использовать только те типы классов, которые были предварительно объявлены в инициаторе.

Инициатор должен хранить указатель на метод и указатель на класс. Увеличивается расход памяти.

2.4. Функциональный объект

2.4.1. Концепция

С точки зрения C++ функциональный объект – это класс, который имеет перегруженный оператор вызова функции[7].

Графическое изображение обратного вызова с помощью функционального объекта представлено на Рис. 14. Исполнитель реализуется в виде класса, код упаковывается в перегруженный оператор вызовы функции, в качестве контекста выступает экземпляр класса. При настройке экземпляр класса как аргумент сохраняется в инициаторе[8]. Инициатор осуществляет обратный вызов посредством вызова перегруженного оператора, передавая ему требуемую информацию. Контекст здесь передавать не нужно, поскольку внутри оператора доступно все содержимое класса.

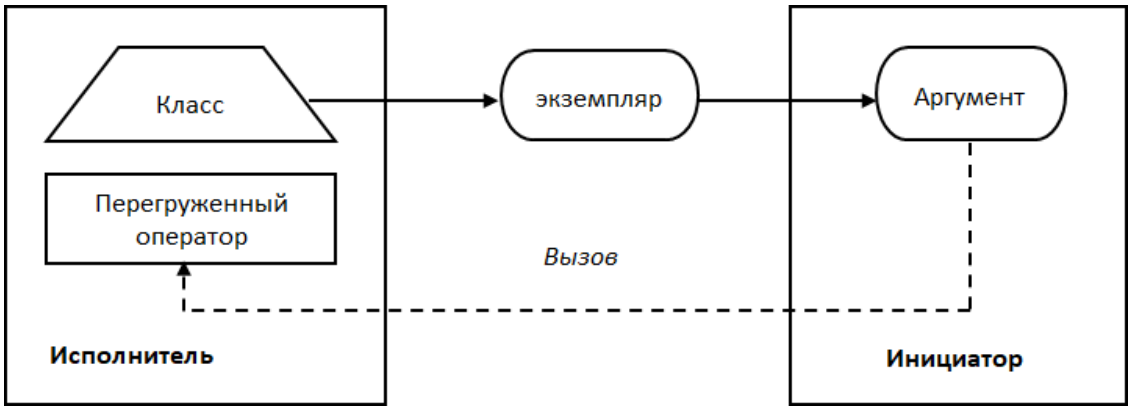

Рис. 14. Реализация обратного вызова с помощью функционального объекта.

2.4.2. Инициатор

Предварительно необходимо объявить функциональный объект (см. Листинг 15), потому что его объявление должен видеть как инициатор, так и исполнитель.

Листинг 15.Объявление функционального объекта

```
class CallbackHandler
{
public:
  void operator() (int eventID) //This is an overloaded operator
   {
     //It will be called by server
   };
};
```

[7] Другое название, которое встречается в литературе, – функтор.

[8] В инициаторе хранится копия экземпляра класса. Не ссылка, не указатель, а именно копия. Из этого вытекает несколько важных следствий, которые будут рассмотрены далее.

Реализация инициатора приведена в Листинг 16.

Листинг 16. Инициатор с функциональным объектом

```
class Initiator  // (1)
{
public:
  void setup(const CallbackHandler& callback)  // (2)
  {
    callbackObject = callback;
  }

  void run()  // (3)
  {
    int eventID = 0;
    //Some actions
    callbackObject(eventID);  // (4)
  }

private:
  CallbackHandler callbackObject;  // (5)
};
```

В строке 1 мы объявляется класс-инициатор. В строке 2 объявляется функция для настройки вызова, в которую передается ссылка на функциональный объект. Данный объект присваивается переменной-аргументу, объявленному в строке 5. В строке 3 объявлена функция запуска, внутри этой функции в строке 4 производится вызов перегруженного оператора. Как видим, синтаксис вызова перегруженного оператора совпадает с синтаксисом вызова обычной функции.

2.4.3. Исполнитель

Реализация исполнителя приведена в Листинг 17.

Листинг 17. Исполнитель с функциональным объектом

```
int main()
{
  Initiator initiator;         // (1)
  CallbackHandler executor;    // (2)
  initiator.setup(executor);   // (3)
  initiator.run();             // (4)
}
```

В строке 1 объявляется переменная класса-инициатора, в строке 2 объявляется функциональный объект, в строке 3 производится настройка, в строке 4 – запуск.

2.4.4. Синхронный вызов

Реализация инициатора для синхронного вызова представлена в Листинг 18. В отличие от асинхронного вызова, здесь функциональный объект не сохраняется как аргумент, он передается через входные параметры функции.

Листинг 18. Инициатор для синхронного вызова с функциональным объектом

```
void run(CallbackHandler& callbackObject)
{
  int eventID = 0;
  //Some actions
  callbackObject(eventID);
}
```

2.4.5. Преимущества и недостатки

Преимущества и недостатки реализации обратных вызовов с помощью функционального объекта приведены в Табл. 5.

Табл. 5. Преимущества и недостатки обратных вызовов с помощью функционального объекта

Преимущества	Недостатки
Простая реализация	Общий функциональный объект
Безопасность	Невозможность реализации API
Отсутствие трансляции контекста	
Высокое быстродействие	

Простая реализация. Самая простая из всех рассмотренных. Необходима только одна переменная – экземпляр класса, весь контекст хранится внутри этого класса. Прозрачный и понятный синтаксис.

Безопасность. При настройке в инициаторе создается копия переданного функционального объекта. Исходный экземпляр становится ненужным, его можно безопасно удалить.

Отсутствие трансляции контекста. Код вызова хранится внутри перегруженного оператора, контекст инкапсулирован внутри класса вместе с кодом.

Общий функциональный объект. Инициатор и исполнитель связаны через единый функциональный объект, они оба должны видеть его объявление. Вся логика обработки реализуется внутри объекта. Это приводит к монолитной архитектуре, что сильно затрудняет модификацию поведения обработчика. По сути дела, исполнитель встраивается в инициатор и становится его составной частью[9].

Невозможность реализации API. Следствие монолитной архитектуры: использование API предполагает возможность модификации поведения исполнителя без изменения кода инициатора. Поскольку они оба связаны через единый объект, выполнение указанного требования является нереализуемым.

Высокое быстродействие. А вот здесь недостатки монолитной архитектуры превращаются в достоинства. Дело в том, что поскольку инициатор сохраняет у себя объект, он имеет

[9] Частично этот недостаток устраняется с помощью шаблонов, что будет рассматриваться в соответствующем разделе.

доступ к коду перегруженного оператора, т. е. к коду обработчика вызова. Как следствие, оптимизирующий компилятор получает возможность встроить код обработчика непосредственно в точку вызова, опуская вызов функции (перегруженный оператор тоже является функцией), что значительно ускоряет выполнение вызова. Рассмотрим этот момент подробнее.

2.4.6. Производительность

С точки зрения машинных команд, вызов функции – не слишком быстрая операция. Необходимо несколько команд для сохранения стека[10]; команда перехода к коду функции; команда возврата управления; несколько команд для восстановления стека. А если код тела функции небольшой, к примеру, всего лишь сравнение двух величин, то время, затраченное на вызов функции, может значительно превысить время выполнения кода функции.

Поясним сказанное на примере. Напишем маленькую простую программу, которая считывает из консоли два числа, складывает их и результат выводит на экран (Листинг 19).

Листинг 19. Маленькая простая программа

```
#include <iostream>

int Calculate(int a, int b)
{
  return a + b;
}

int main()
{
  int a, b;
  std::cin >> a >> b;
  int result = Calculate(a, b);
  std::cout << result;
}
```

Откомпилируем код с выключенной оптимизацией и запустим на выполнение. Посмотрим дизассемблерный участок кода [11], в котором производится вызов функции (Листинг 20):

Листинг 20. Дизассемблерный код с выключенной оптимизацией:

```
int Calculate(int a, int b)
{
00007FF6DA741005  and         al,8                // 1
return a + b;
00007FF6DA741008  mov         eax,dword ptr [b]   // 2
00007FF6DA74100C  mov         ecx,dword ptr [a]   // 3
00007FF6DA741010  add          ecx,eax            // 4
00007FF6DA741012  mov         eax,ecx             // 5
```

[10] Количество таких команд зависит от количества входных параметров функции.

[11] Этот код получен с помощью компилятора Microsoft Visual studio версии 19.23.28106.4. Другие компиляторы могут генерировать отличающийся код, но принцип останется прежним.

```
}
00007FF6DA741014  ret                                    // 6

int main()
{
…….
int result = Calculate(a, b);
00007FF6DA741053  mov        edx,dword ptr [b]            // 7
00007FF6DA741057  mov        ecx,dword ptr [a]            // 8
00007FF6DA74105B  call      Calculate (07FF6DA741000h)    // 9
00007FF6DA741060  mov       dword ptr [result],eax       // 10
…….
```

В строках 7 и 8 введенные значения a и b сохраняются в регистрах. В строке 9 выполняется вызов функции. В строке 1 выполняется обнуление результата, в строках 2 и 3 переданные значения копируются в регистры, в строке 4 выполняется сложение, в строке 5 результат копируется обратно в регистр, в строке 6 выполняется выход из функции, в строке 10 результат вычисления функции копируется в переменную результата.

Теперь включим оптимизацию, откомпилируем и посмотрим на код (Листинг 21):

Листинг 21. Дизассемблерный код с включенной оптимизацией

```
int main()
{
…….
int result = Calculate(a, b);
00007FF7D5B11033  mov         edx,dword ptr [b]
00007FF7D5B11037  add          edx,dword ptr [a]
```

Как видим, для вычислений у нас всего две операции: запись в регистр значения b и добавление к нему значения a. Код встроен в поток выполнения, вызов функции не производится. Ощутимая разница, не правда ли?

2.5. Лямбда-выражение

2.5.1. Концепция

Лямбда-выражение[12] – это локальная неименованная функция, которая, подобно обычной функции, может принимать входные параметры и возвращать результат. Особенностью лямбда-выражений, отличающих их от обычных функций, является возможность захвата переменных.

Графическое изображение обратного вызова с помощью лямбда-выражения представлено на Рис. 15. Исполнитель реализуется в виде какой-либо исполняемой функции, в качестве которой могут выступать глобальная функция, статический метод класса, метод-член класса, перегруженный оператор. Код обратного вызова упаковывается в лямбда-выражение, в качестве контекста выступают захваченные переменные. При настройке лямбда-выражение как аргумент сохраняется в инициаторе. Инициатор осуществляет обратный вызов посредством вызова хранимого выражения, передавая ему требуемую информацию. Контекст здесь передавать не нужно, поскольку внутри тела лямбда-выражения доступны все захваченные переменные.

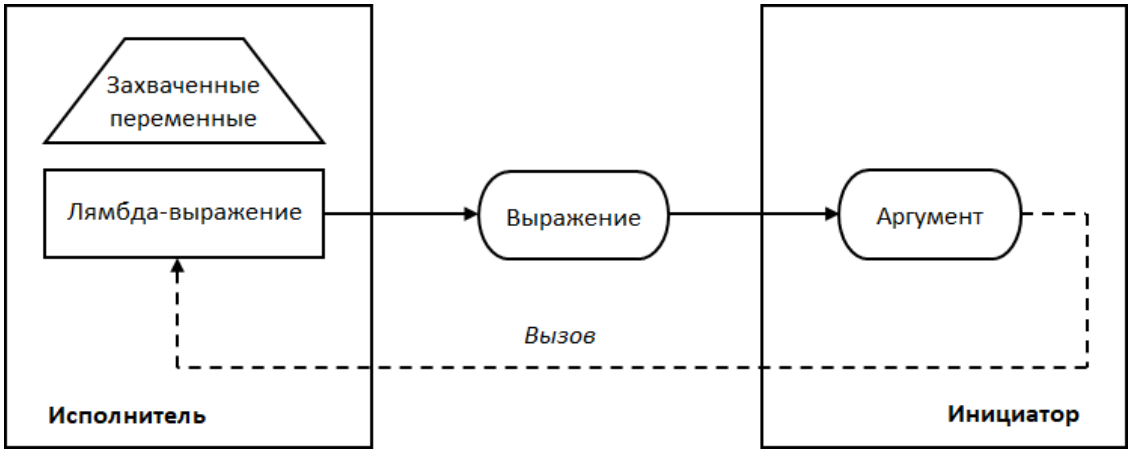

Рис. 15. Реализация обратного вызова с помощью лямбда-выражения

2.5.2. Инициатор

Как хранить и передавать лямбда-выражение как аргумент? Если оно не захватывает переменные, то стандарт допускает неявное преобразование лямбда-выражения к указателю на функцию. В этом случае реализация инициатора полностью совпадает с рассмотренной в 2.1. Однако использование лямбда-выражений без захвата переменных не дает никакого преимущества по сравнению с обычной функцией, использовать их в таком виде не имеет смысла.

Другое дело, когда лямбда-выражение осуществляет захват переменных, в этом случае мы получаем мощный и гибкий инструмент управления контекстом. Однако использование таких выражений в качестве аргумента вызывает определенные сложности. Связано это с тем, что тип лямбда-выражения является анонимным. Как следствие, имя типа нам неизвестно, и

[12] В литературе можно встретить термин «лямбда-функция», но в стандарте C++ он именуется как “lambda-expression”, что в переводе означает «лямбда-выражение».

мы не можем просто объявить переменную нужного типа и присвоить ей лямбда-выражение, как это происходит, например, с указателями или классами. Решается указанная проблема с помощью шаблонов, что будет рассмотрено позже в соответствующих главах. Забегая вперед, отметим, что для хранения лямбда-выражений можно объявлять шаблон с параметром – типом лямбда-выражения (п. 4.4.2) либо использовать специальные классы библиотеки STL (п. 4.6.1).

2.5.3. Исполнитель

Исполнитель реализовывается в виде лямбда-выражения, а передача его как аргумента инициатору зависит от способа реализации последнего. Если исполнитель реализован в виде шаблона класса (п. 4.4.2), лямбда-выражение должно присваиваться в конструкторе класса. В случае использования классов STL (п. 4.5.1) лямбда-выражение передается подобно любому другому аргументу. Подробно эти вопросы рассматриваются в разделе 4, посвященном использованию шаблонов.

2.5.4. Синхронный вызов

Инициатор для синхронного вызова с лямбда-выражением реализуется в виде шаблонной функции, параметром шаблона выступает тип аргумента. Подробно этот вопрос рассмотрен в п. 4.2.1.

2.5.5. Преимущества и недостатки

Преимущества и недостатки реализации обратных вызовов с помощью лямбда-выражения приведены в Табл. 6.

Табл. 6. Преимущества и недостатки обратных вызовов с помощью лямбда-выражения

Преимущества	Недостатки
Гибкое управление контекстом	Требует использования шаблонов

Гибкое управление контекстом. Возможность захвата переменных предоставляет простые и удобные средства изменения контекста. Изменяя состав захваченных переменных, мы легко можем добавлять значения, необходимые для контекста, при этом нет необходимости изменять код инициатора. Захватив указатель this, мы получаем доступ к содержимому класса, т. е. фактически лямбда-выражение превращается в «метод внутри метода» (см. пример в Листинг 22). Элегантно, не правда ли?

Требует использования шаблонов. Использование шаблонов накладывает архитектурные ограничения на реализацию программных модулей. Это связанно с тем, что шаблоны не предполагают присутствие предварительно откомпилированного кода. Подробнее об этом мы будем говорить в соответствующей главе (4.7), посвященной ограничениям при использовании шаблонов.

Листинг 22. Лямбда-выражение с захватом указателя this

```
class EventCounter
{
public:
  void AddEvent(unsigned int event)
  {
```

```
    callCounter_++;
    lastEvent_ = event;
  }
private:
  unsigned int callCounter_ = 0;
  int lastEvent_ = 0;
};

class Executor
{
public:
  Executor(EventCounter* counter): counter_(counter)
  {
    auto lambda = [this](int eventID)
    {
      //It will be called by initiator
      counter_->AddEvent(eventID);
      processEvent(eventID);
    };
    //Setup lambda in initiator
  }
private:
  EventCounter* counter_;
  void processEvent(int eventID) {/*Do something*/}
};
```

2.6. Итоги

В C++ обратные вызовы могут быть реализованы с помощью следующих конструкций:

- указатель на функцию;
- указатель на статический метод класса;
- указатель на метод-член класса;
- функциональный объект;
- лямбда-выражение.

Каждая реализация имеет свои достоинства и недостатки. Так какую все-таки выбрать? Чтобы ответить на этот вопрос, необходимо выполнить сравнительный анализ.

3. Сравнительный анализ реализаций

3.1. Методологические подходы

3.1.1. Обобщенный алгоритм

Итак, мы рассмотрели различные способы реализации обратных вызовов. Какая из них наилучшим образом подходит для использования в конкретной ситуации? Чтобы ответить на этот вопрос, необходимо сравнить реализации, т. е. требуется сравнительный анализ.

Обобщенный алгоритм сравнительного анализа включает следующие шаги.

1. Выбрать объекты анализа.
2. Определить критерии сравнения.
3. Построить матрицу соответствия, в которой отобразить, насколько объекты анализа соответствуют выбранным критериям.
4. Проанализировать полученные результаты и выбрать объект, наилучшим образом удовлетворяющий совокупности критериев.

Рассмотрим указанные шаги подробнее.

1. **Объект анализа** – это некая сущность, которая будет подвергаться анализу. В нашем случае такими сущностями выступают реализации обратных вызовов.

2. **Выбор критериев** – пожалуй, самый сложный и в то же время наиболее важный этап сравнительного анализа. Критерии должны отражать значимость показателя, который они определяют; неверный выбор критериев приводит к неправильным результатам. Так, например, в качестве критерия можно выбрать количество строк кода, но насколько этот показатель значим при разработке? В нашем случае совершенно не значим: не имеет значения, займет реализация 10 или 50 строк, важно то, насколько она обеспечивает качество выполняемых функций. Качество, в свою очередь, определяется степенью выполнения требований, предъявляемых к проектируемой системе. По этой причине именно требования наилучшим образом подходят для использования в качестве критериев.

3. **Матрица соответствия** строится в виде таблицы. В заголовки строк таблицы вписываются критерии, в заголовки столбцов – объекты анализа. В ячейках таблицы для каждой пары «объект-критерий» выставляется степень соответствия объекта заданному критерию. Степень выполнения может быть качественной (выполняется/не выполняется) или количественной (выставляется оценка по заданной шкале).

4. Полученные **результаты** суммируются. Объект, набравший наибольшее количество положительных утверждений (качественная оценка), или наибольшее количество баллов (количественная оценка), будет оптимальным.

Итак, мы описали обобщенный алгоритм сравнительного анализа. Далее рассмотрим, как выполняются шаги алгоритма применительно к поставленной задаче – выбору оптимальной реализации для конкретного случая. Первый шаг – выбор объектов анализа – здесь тривиальный, объектами анализа являются реализации обратных вызовов. Перейдем ко второму шагу –определим критерии, в качестве которых выступают требования.

3.1.2. Требования как критерии

Обозначим требования, предъявляемые при разработке программного кода обратных вызовов. Состав требований не претендует на полноту, читатель может добавить свои, если посчитает их значимыми или актуальными для конкретного случая.

Простота. Показывает, насколько просто и быстро можно написать, отладить и сопровождать код.

Независимость компонентов. Показывает, нужно ли изменять код одного компонента при изменении другого. Чем меньше зависимости между компонентами (в нашем случае это инициатор и исполнитель), тем проще разработка и отладка программной системы. Кроме того, упрощается ее сопровождение и повышается надежность.

Отсутствие трансляции контекста. Отсутствие необходимости трансляции контекста упрощает разработку, улучшает прозрачность кода и повышает независимость компонентов. И наоборот, трансляция контекста усложняет код и заставляет инициатор выполнять дополнительные операции для хранения и передачи контекста

Безопасность. Показывает устойчивость системы к потенциальным ошибкам.

Гибкость. Показывает, насколько просто модифицировать код при появлении новых требований.

Полиморфизм. Показывает, поддерживается ли полиморфизм в реализации исполнителя. Поддержка полиморфизма упрощает разработку и повышает гибкость в рамках объектно-ориентированной парадигмы.

Быстродействие. Показывает, насколько быстро осуществляется вызов кода исполнителя.

Системный API. Показывает возможность реализации системных API.

C++ API. Показывает возможность реализации C++ API.

Итак, объекты анализа выбраны, критерии определены. Теперь нужно построить матрицу соответствия. Для начала мы будем использовать качественный анализ, поскольку он более простой в реализации.

3.2. Качественный анализ

3.2.1. Матрица соответствия

Матрица соответствия строится в виде таблицы. В строках выписываются требования, в столбцах – способы реализации, в ячейках – признаки, указывающие, насколько реализация поддерживает соответствующий критерий (Табл. 7.)

Табл. 7. Качественный анализ реализаций обратных вызовов

Легенда: ▪ полностью поддерживается; ▫ поддерживается частично; пустое поле – не поддерживается

Требования	Указатель на функцию	Указатель на статический метод	Указатель на метод-член	Функциональный объект	Лямбда-выражение
Простота	▫	▫		▪	
Независимость компонентов	▪	▫	▫		▫
Отсутствие трансляции контекста			▪	▪	▪
Безопасность		▫	▫	▪	▪
Гибкость	▫	▫	▪		▫
Полиморфизм			▪		▪
Быстродействие	▫	▫		▪	▪
Системный API	▪				
C++ API	▪	▪	▪		

По каким соображениям мы назначили оценки?

Простота. Самой сложной реализацией будет, пожалуй, указатель на метод-член класса: запутанный и не слишком наглядный синтаксис. Довольно сложной выглядит реализация лямбда-выражений, поскольку приходится использовать шаблоны. Несколько проще выглядит реализация с помощью указателей на функцию, но там немного запутывает необходимость приведения типов. На этом фоне остальные реализации выглядят достаточно простыми.

Независимость компонентов. Полностью независимыми будет реализация с помощью указателей на функцию: как бы мы не модифицировали код исполнителя, как бы не меняли используемый контекст, код инициатора остается неизменным, даже не требуется его переком-

пиляция. Это одна из причин, почему указанная реализация подходит для построения системных API. Лямбда-выражения являются относительно независимыми: при любом изменении состава и типов захваченных переменных код инициатора остается неизменным, но он будет требовать перекомпиляции, поскольку реализован с использованием шаблонов. Указатели на методы классов являются частично независимыми, поскольку требуют предварительного объявления класса в инициаторе. Использование функциональных объектов порождает монолитную архитектуру, где инициатор и исполнитель зависят друг от друга.

Отсутствие трансляции контекста. Указатели на функции и статические методы требуют трансляции контекста, остальные реализации этого не требуют.

Безопасность. Самыми безопасными являются функциональные объекты и лямбда-выражения, потому что в инициаторе хранятся их копии, никак не зависящие от исполнителя. Указатели на методы класса поддерживают безопасность лишь частично: управление временем жизни экземпляра класса возлагается на исполнителя, и потенциально возможны ситуации, когда последний уничтожает экземпляр класса, указатель на который остается в инициаторе и может быть вызван. Указатель на функцию не является безопасным, поскольку исполнитель интерпретирует контекст приведением типов, и нет никакой возможности проверить полученный указатель.

Гибкость. Самым гибким является указатель на метод класса, поскольку здесь имеются несколько способов модификации поведения обработчика. Другие реализации не предлагают таких возможностей, а функциональные объекты в силу монолитной структуры гибкими не являются.

Полиморфизм. Указатель на метод-член класса поддерживает полиморфизм подтипов (наследование и виртуализация), лямбда-выражения поддерживают специализированный полиморфизм (различный код в зависимости от состава и типов захваченных переменных). Остальные реализации полиморфизм не поддерживают.

Конец ознакомительного фрагмента.