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.

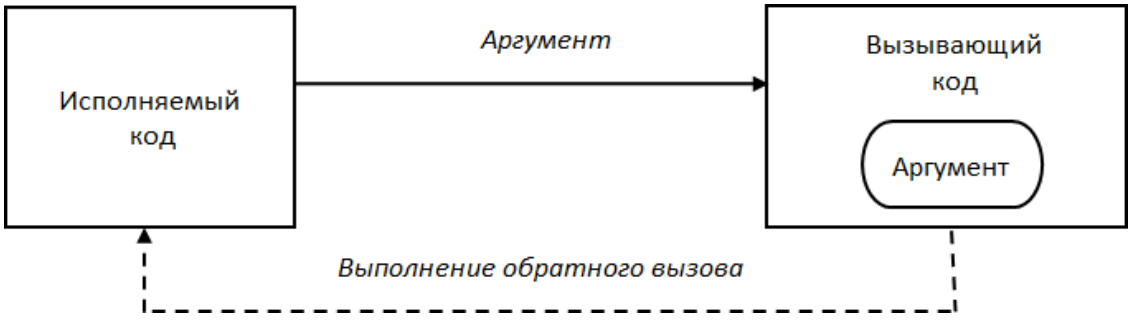

Рис. 1. Концепция обратных вызовов

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.

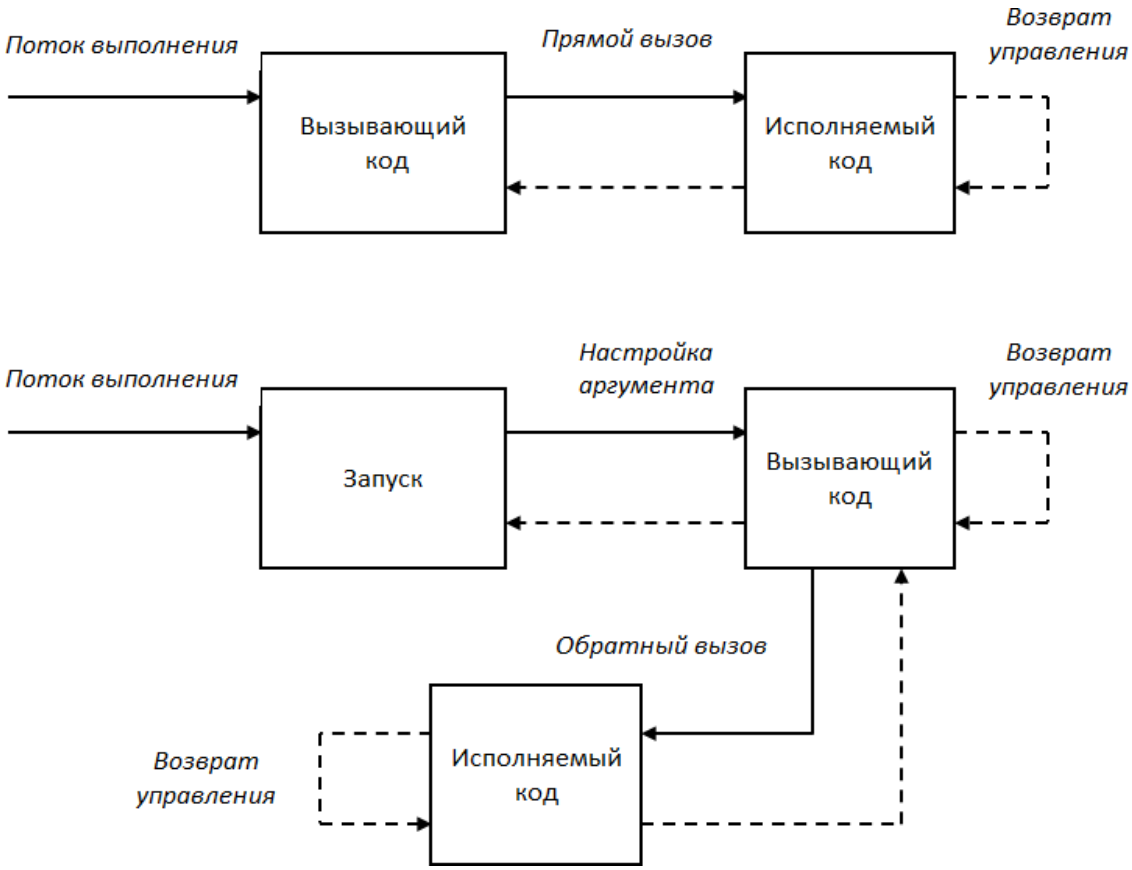

Рис. 2. Прямой и обратный вызов

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.

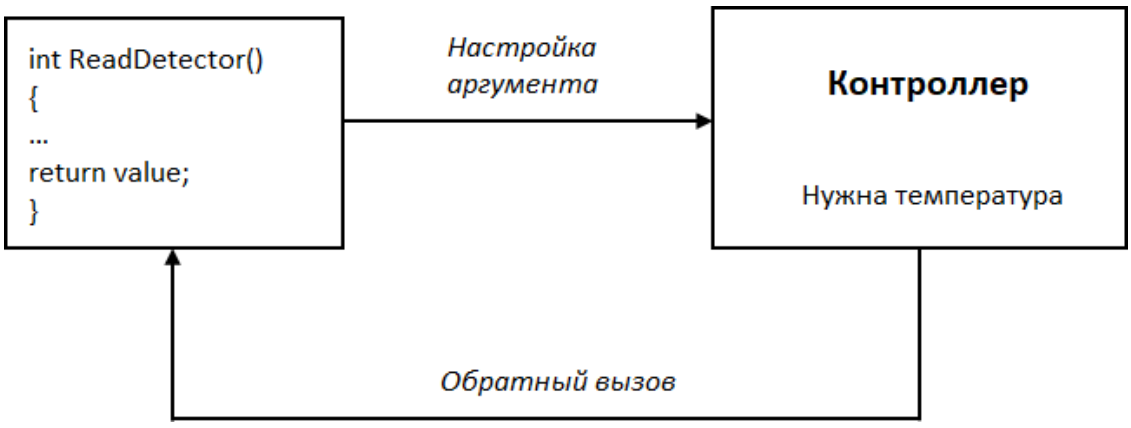

Рис. 3. Опрос датчиков с помощью обратного вызова

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.

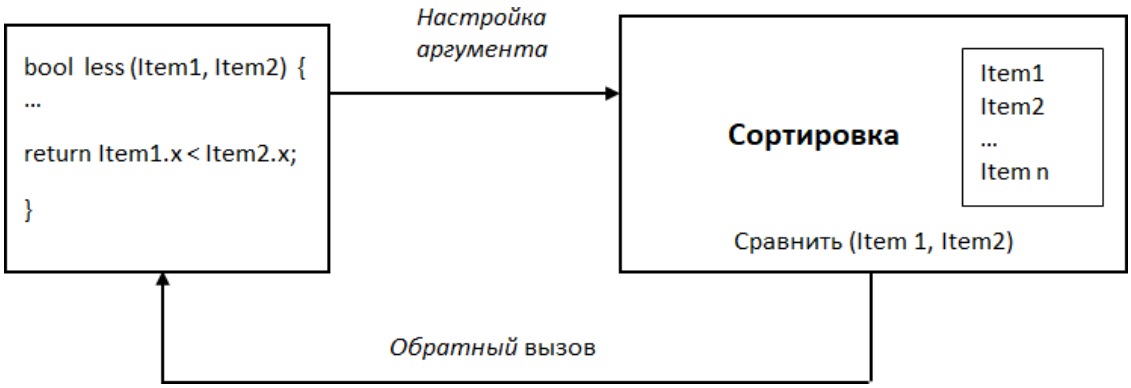

Рис. 4. Результат вычисления с помощью обратного вызова

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.

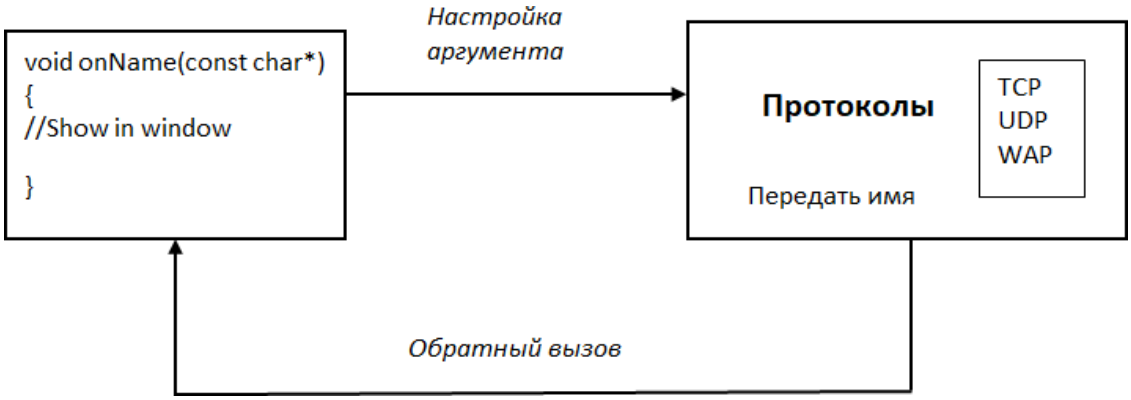

Рис. 5. Просмотр элементов с помощью обратных вызовов

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.

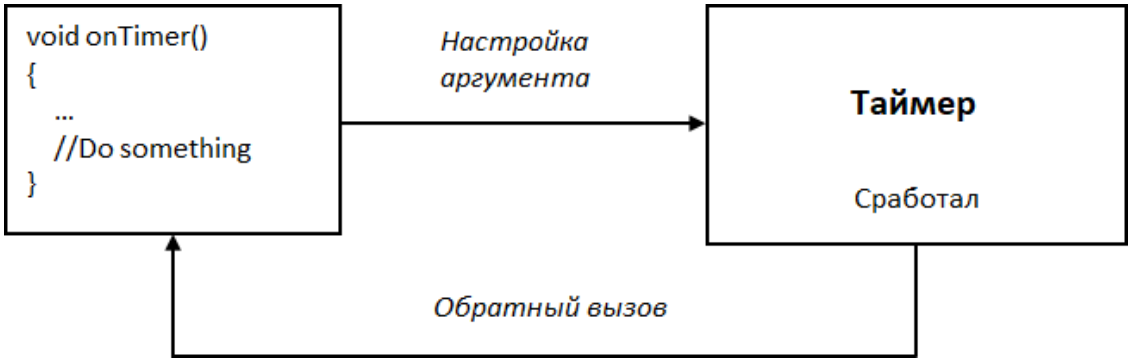

Рис. 6. Уведомление о срабатывании таймера с помощью обратного вызова

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.

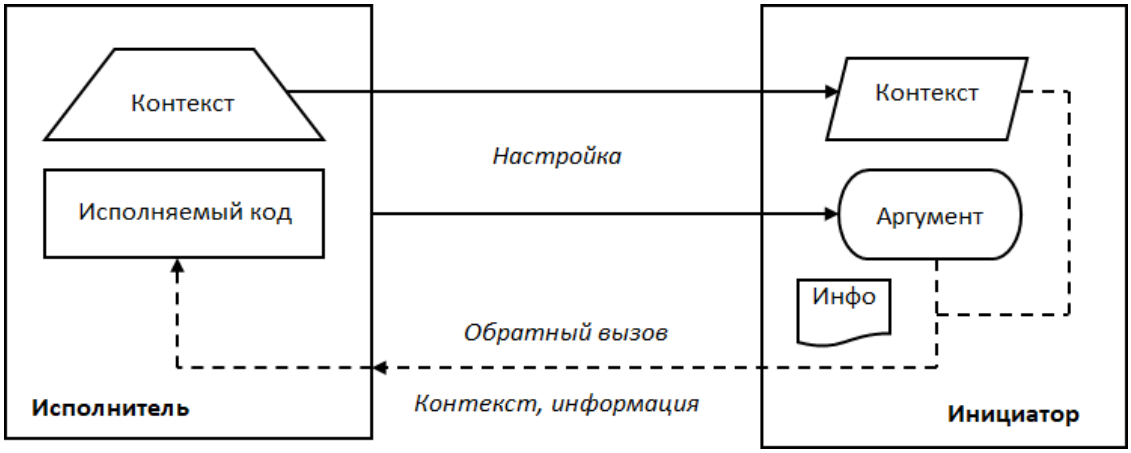

Рис. 7. Модель обратных вызовов

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.

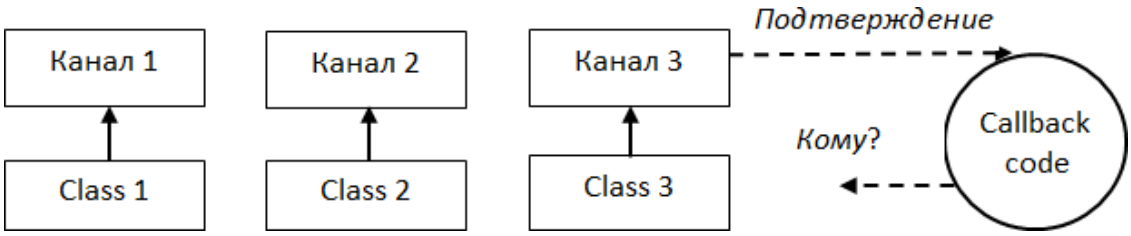

Рис. 8. Сетевой обмен и контекст вызова

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.

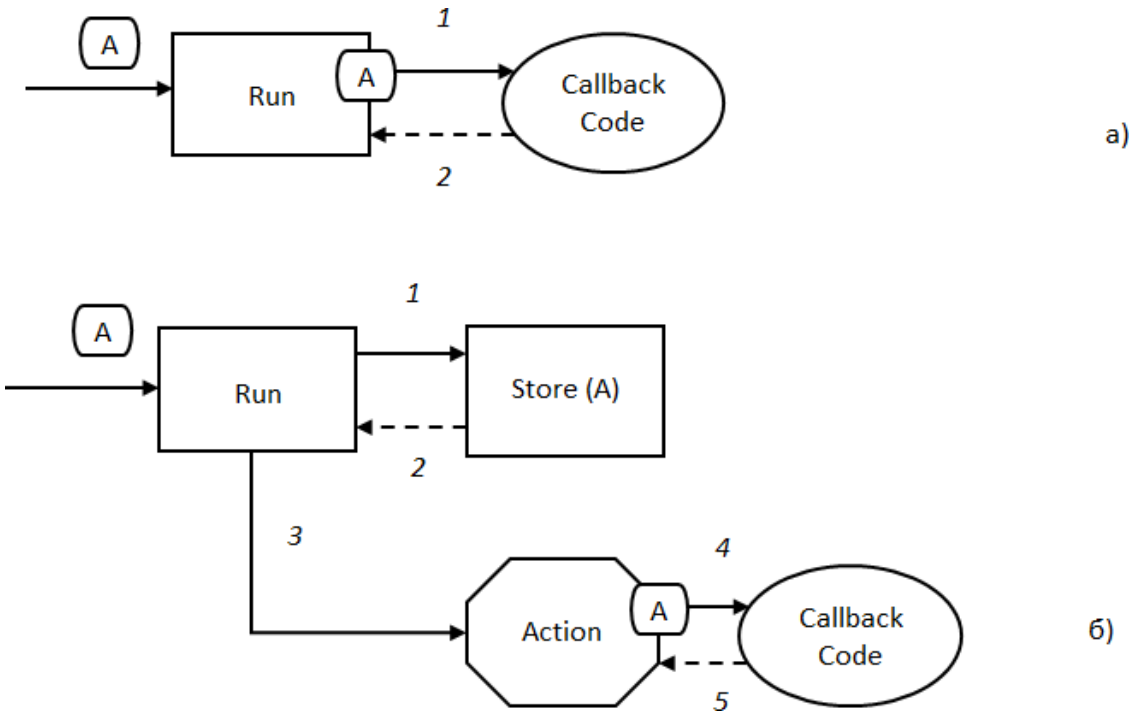

Рис. 9. Синхронные и асинхронные вызовы: а) синхронный; б) асинхронный

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.

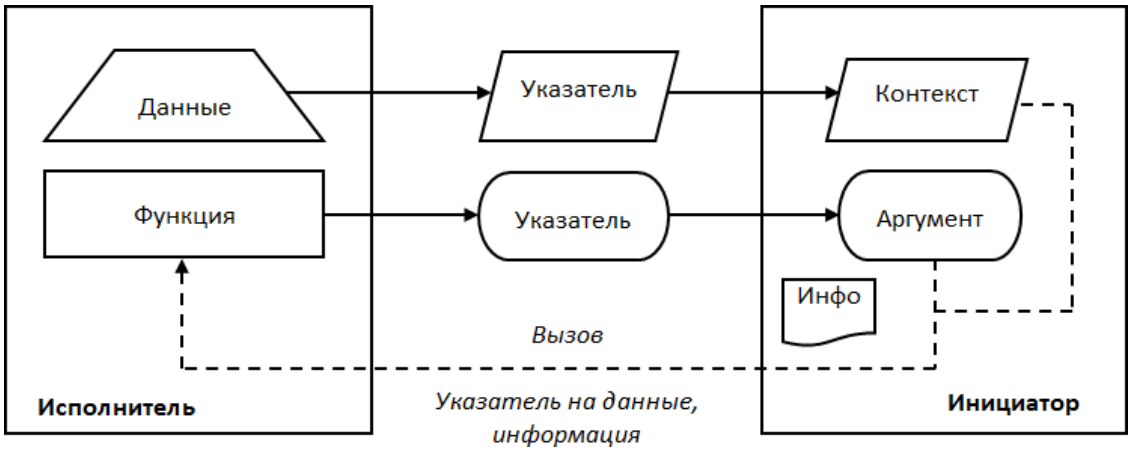

Рис. 10. Обратный вызов с указателем на функцию

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.

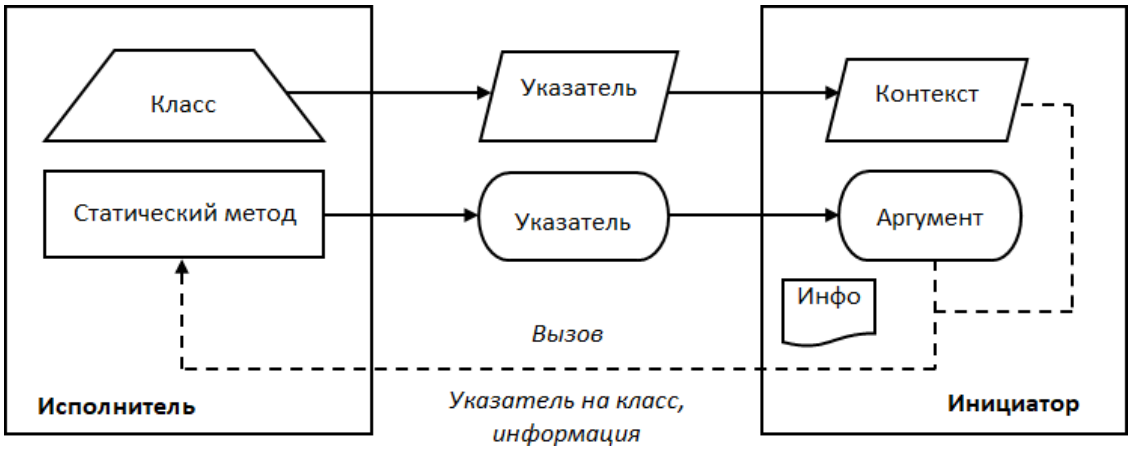

Рис. 11. Обратный вызов с указателем на статический метод класса

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.

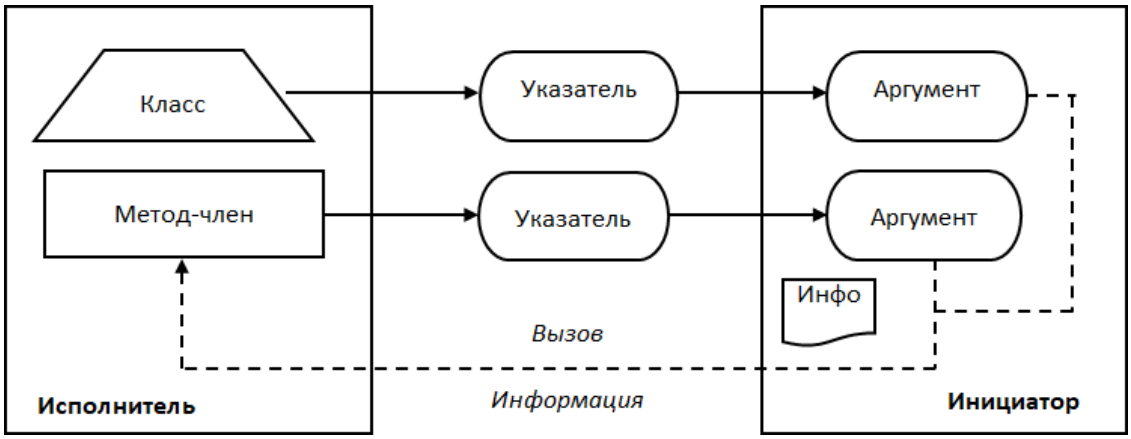

Рис. 12. Реализация обратного вызова с помощью указателя на метод-член класса

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
{
};
```

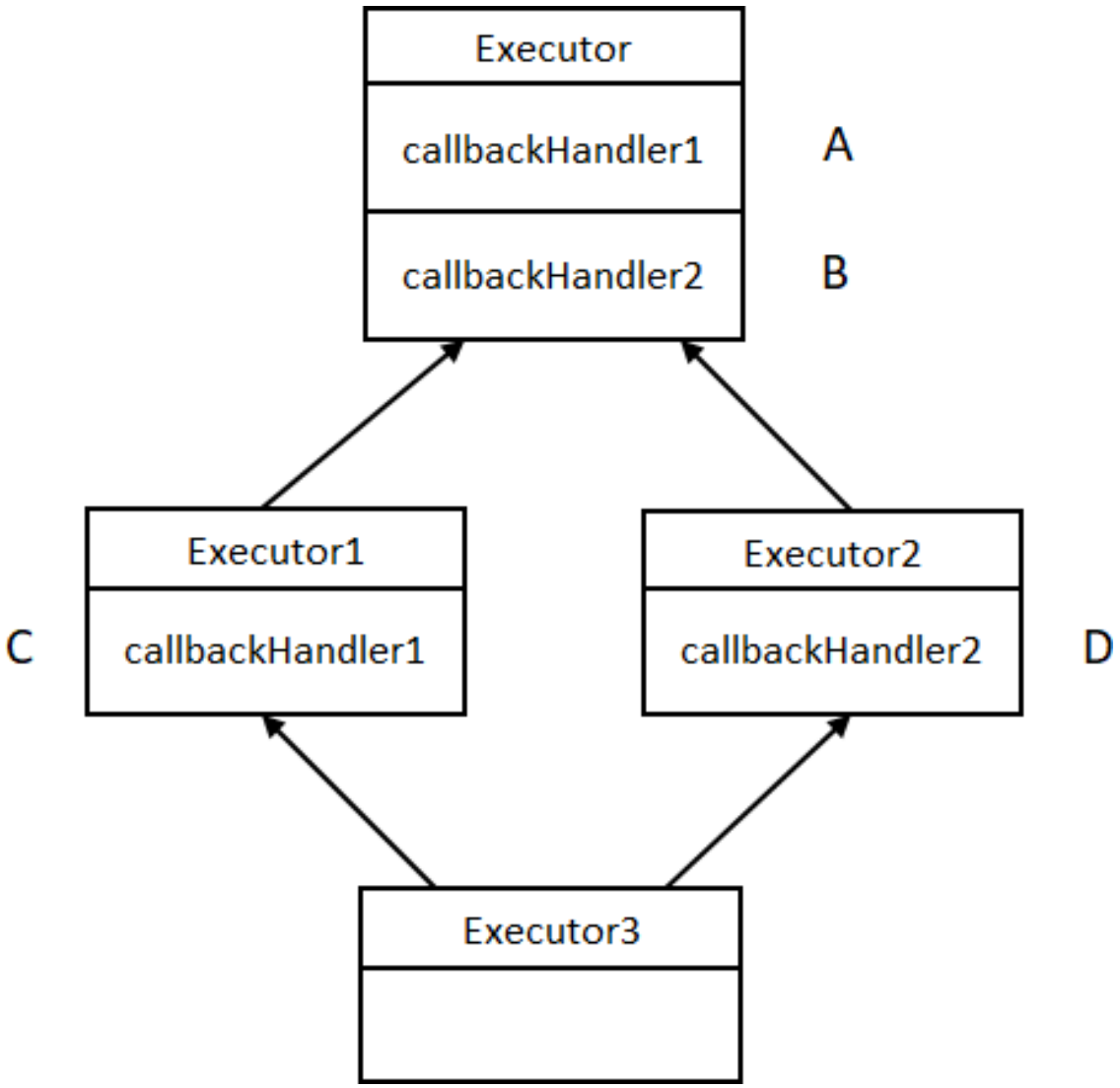

Рис. 13. Иерархия наследования классов-исполнителей

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.

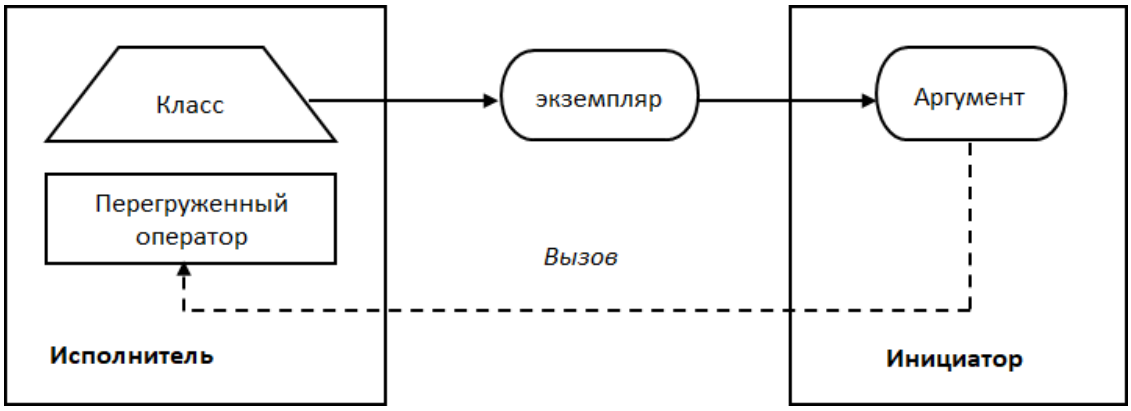

Рис. 14. Реализация обратного вызова с помощью функционального объекта.

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.

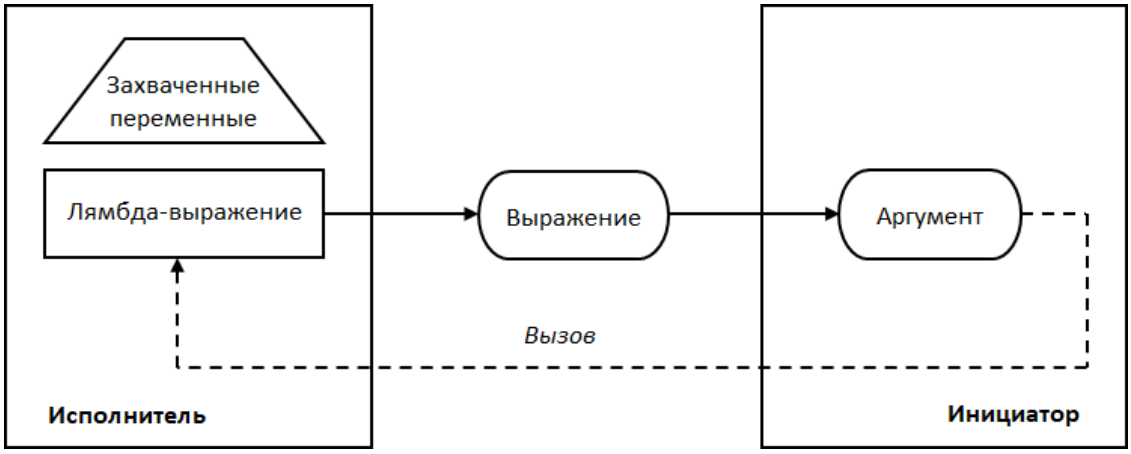

Рис. 15. Реализация обратного вызова с помощью лямбда-выражения

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.

Быстродействие. Самым быстродействующим является функциональный объект, практически не отстает от него и лямбда-выражение. Это связано с тем, что их код может встраиваться в точку вызова. Несколько медленнее работают указатели на функцию и на статический метод, поскольку их код выполняется через вызов функции[13]. Медленнее всего работает указатель на метод-член класса, поскольку ему необходимо обращение к таблице виртуальных функций.

Системный API. Указатель на функцию – единственный способ, с помощью которого можно использовать обратные вызовы при проектировании системных API.

C++ API. Лямбда-выражения не подходят для использования в C++ API: хотя инициатор не требует изменений при модификации исполнителя, но ему требуется перекомпиляция. Не подходят для C++ API также функциональные объекты, поскольку изменение функционального объекта затрагивает как инициатор, так и исполнитель.

3.2.2. Выбор реализации

Итак, мы построили матрицу соответствия, проанализировали, насколько реализации соответствуют выбранным критериям. Что же выбрать для конкретного случая? Для решения этого вопроса необходимо определить, какой критерий сейчас является наиболее важным, и выбрать реализацию по этому критерию. Так, например, если самым важным является возможность проектирования системного API, то следует выбрать указатели на функцию. Если

[13] При использовании указателей на функцию их код встроить невозможно, потому что заранее неизвестно, какая функция будет использоваться.

самым важным является быстродействие, то следует выбрать функциональные объекты. Если самым важным является гибкость, то следует выбрать указатели на член класса.

А как сделать выбор, если нам одновременно важны несколько критериев, причем некоторые из них противоречат друг другу (а чаще всего именно так и происходит)? У нас появляется проблема многокритериального выбора, решить которую позволяет метод интегральных оценок.

3.3. Метод интегральных оценок

3.3.1. Количественные оценки

По своей сути метод интегральных оценок повторяет качественный анализ, но с одним отличием – в матрице соответствия вместо качественных вводятся количественные оценки. В ячейках матрицы проставляются числовые значения, отражающие, насколько объект анализа поддерживает (другими словами, в какой степени реализует) соответствующее требование. Диапазон возможных значений задается шкалой оценок, которая зависит от точности, которую мы хотим получить. Примеры различных шкал оценок изображены на Рис. 16.

0	Не реализуемо
1	Реализуемо

0	Не реализуемо
1	Частичная реализация
2	Полная реализация

0	Не реализуемо
1	Ограниченная реализация
2	Реализация с оговорками
3	Полная реализация

Рис. 16. Шкалы оценки реализуемости требований

Итак, строим матрицу соответствия, в ячейках выставляем числовые оценки, суммируем оценки по столбцам. Реализация, набравшая наибольшее количество баллов, является оптимальной.

Пример интегральных оценок по трем критериям с использованием трехбалльной шкалы приведен в Табл. 8. Здесь наибольшее количество балов набирает реализация с использованием функционального объекта, которая для конкретного случая является оптимальной.

Табл. 8. Интегральные оценки по трехбалльной шкале

Требования	**Указатель на функцию**	**Указатель на статический метод**	**Указатель на метод-член**	**Функциональный объект**	**Лямбда-выражение**
Простота	1	1	0	2	0
Независимость компонентов	2	1	1	0	1
Безопасность	0	1	1	2	2
Итого	**3**	**3**	**2**	**4**	**3**

3.3.2. Коэффициенты важности

Зачастую оказывается, что некоторые требования являются более важными, чем остальные. Например, быстродействие важно, но в то же время гибкость еще важнее; в свою очередь, безопасность является приоритетным фактором. Чтобы учесть такие ситуации, вводятся коэффициенты важности.

Каждому требованию присваивается коэффициент, который отражает, насколько данное требование является важным для обеспечения качества функционирования системы в конкретном случае. При расчете числовых оценок каждое значение в ячейке таблицы умножается на этот коэффициент; таким образом вносятся поправки в итоговые значения. Целесообразно

предварительно ранжировать требования по важности: наименее важному присвоить коэффициент 1, и для каждого требования, более важного, чем предыдущее, увеличивать значение на единицу.

Введем коэффициенты важности для предыдущего примера. Ранжируем требования: считаем, что наименее важным для нас является простота, наиболее важным – безопасность. Результаты приведены в Табл. 9.

Табл. 9. Ранжирование требований

1	Простота
2	Безопасность
3	Независимость компонентов

Пересчитаем показатели с учетом коэффициентов важности. Для коэффициентов важности вводим отдельный столбец, где проставляем соответствующие значения. В ячейках в скобках отображаются значения оценки без учета коэффициента, без скобок отображаются новые значения с учетом поправок (Табл. 10).

Табл. 10. Интегральные оценки с учетом коэффициентов важности.

Требования	КВ	Указатель на функцию	Указатель на статический метод	Указатель на метод-член	Функциональный объект	Лямбда-выражение
Простота	1	(1) 1	(1) 1	(0) 0	(2) 2	(0) 0
Независимость компонентов	3	(2) 6	(1) 3	(1) 3	(0) 0	(1) 3
Безопасность	2	(0) 0	(1) 2	(1) 2	(2) 4	(2) 4
Итого		**7**	**6**	**5**	**6**	**7**

Как видим, после введения коэффициентов важности результаты изменились: теперь максимальное количество балов набирают две реализации – указатель на функцию и лямбда-выражение.

3.3.3. Учет прогнозных показателей

Как мы видели в предыдущем примере, может оказаться, что по результатам расчетов несколько реализаций имеют одинаковое количество балов. В этом случае целесообразно заглянуть в будущее.

Из списка требований выбираем те, которые не актуальны сейчас, но которые, возможно, станут актуальны в последствии. Сводим эти требования в таблицу, аналогично предыдущему примеру, но для числовых значений используем инверсную шкалу: если реализация полностью поддерживает соответствующее требование, выставляем 0, если не поддерживает, то выставляем минимальное отрицательное значение[14]. Так, например, если используется трехбалльная

[14] Минимальное отрицательное, по модулю оно будет максимальным.

шкала, то 0 превращается в -2, 1 превращается в -1, а 2 превращается в 0. Инверсная шкала показывает, насколько сильно новые требования ухудшают текущую интегральную оценку: чем меньше значение[15], тем в большей степени уменьшается текущая оценка.

Далее, полученные оценки суммируются, получившаяся отрицательная интегральная оценка для каждого столбца суммируется с соответствующей текущей оценкой, внося, таким образом, поправки. Из получившихся итоговых значений выбирается реализация, у которой количество балов после коррекции получается наибольшим.

Вернемся к примеру из предыдущего параграфа. Представим, что мы поразмыслили и решили, что в будущем для нас может стать актуальным быстродействие и необходимость реализации C++ API. Сводим эти критерии в таблицу с инверсной шкалой, считаем, что важность этих критериев одинакова. Подсчитываем сумму (Табл. 11).

Табл. 11. Интегральные оценки с инверсной шкалой

Требования	**КВ**	**Указатель на функцию**	**Указатель на статический метод**	**Указатель на метод-член**	**Функциональный объект**	**Лямбда-выражение**
Быстродействие	1	-1	-1	-2	0	0
C++ API	1	0	0	0	-2	-2
Итого		**-1**	**-1**	**-2**	**-2**	**-2**

Получившиеся результаты суммируем с результатами, полученными с использованием обычной шкалы (Табл. 12).

Табл. 12. Поправки с учетом инверсной шкалы

Шкала	**Указатель на функцию**	**Указатель на статический метод**	**Указатель на метод-член**	**Функциональный объект**	**Лямбда-выражение**
Прямая	9	8	5	6	9
Инверсная	-1	-1	-2	-2	-2
Итого	**8**	**7**	**3**	**4**	**7**

Итак, после внесенных поправок для прогнозных показателей максимальное количество балов набирает указатель на функцию, который рекомендуется к применению.

Может оказаться, что даже после учета прогнозных показателей остаются реализации с одинаковым количеством баллов. В этом случае выбор остается на усмотрение разработчика. Он может, к примеру, взять критерий, который лично для него является более предпочтительным (например, простота), и выбрать реализацию по этому критерию. Или просто выбрать, что называется, первую попавшуюся.

[15] Мы говорим «меньше», поскольку числа здесь отрицательные. По модулю это значение будет «больше».

3.4. Итоги

Сравнительный анализ реализаций обратных вызовов необходим для выбора наилучшей в конкретной ситуации. Методика анализа включает в себя выбор объектов, определение критериев сравнения, построение матрицы соответствия, выбор оптимального решения.

Качественный анализ используется, если необходимо выбрать реализацию, оптимальную по какому-нибудь единственному критерию. Если у нас имеется несколько критериев, то необходим количественный анализ, в качестве которого применяется метод интегральных оценок.

Рассмотренные методики подходят не только для исследования обратных вызовов, их можно применять в любых других случаях, когда необходим выбор оптимального архитектурного решения из множества возможных.

4. Обратные вызовы и шаблоны

4.1. Общие понятия шаблонов

Шаблоны в C++ являются инструментом, реализующим параметрический полиморфизм, что означает возможность построения единого (обобщенного) кода для различных типов данных[16]. В таком коде не задаются конкретные типы, а вводятся параметры, в которые затем подставляется нужный тип данных. Чтобы код работал корректно, типы должны удовлетворять некоторым соглашениям, или, другими словами, поддерживать определенный интерфейс.

Обобщенный код – это код, реализующий заданную функциональность без привязки к типам данных.

Шаблоны объявляются ключевым словом **template**, после которого в угловых скобках перечисляются параметры. Параметрами шаблона могут быть как типы данных, так и значения.

Пример объявления шаблона:

```
template SomeTemplate<typename type, int value>
```

Здесь объявлен шаблон с одним параметром-типом **type** и параметром-значением **value**.

Параметрам шаблона, как типам, так и значениям, могут быть назначены значения по умолчанию:

```
template SomeTemplate<typename type = SomeStruct(), int value
= 0>
```

После объявления шаблона следует код шаблона, в качестве которого выступает функция либо класс. В этом коде вместо имен типов и числовых значений можно подставлять имена параметров. Конкретные типы и значения, подставляемые в эти параметры, станут известны после инстанциирования шаблона, под которым понимается объявление экземпляра шаблона с заданными типами.

Инстанциирование шаблона – это объявление экземпляра шаблона с заданными типами.

Инстанциирование шаблона может быть явным и неявным. При явном инстанциировании типы параметров шаблона объявляются, а при неявном – выводятся, исходя из типов входных аргументов. Пример объявления шаблонов и их инстанциирование представлены в Листинг 23.

Листинг 23. Объявление шаблона и его инстанциирование

```
template<typename type, int size = 1>  // (1)
```

[16] В противоположность полиморфизму подтипов, который подразумевает исполнение потенциально разного кода для каждого типа или подтипа. В C++ полиморфизм подтипов реализуется с помощью наследования и виртуальных функций.Термины «параметрический полиморфизм» и «полиморфизм подтипов» больше характерны для академической литературы, в C++ обычно используются их эквиваленты «статический полиморфизм» и «динамический полиморфизм». С точки зрения теории, такая терминология не совсем корректна, потому что она скорее отражает не сущность полиморфизма, а способ его реализации в конкретном языке программирования. Тем не менее, в C++ эти термины прижились.

```
class StaticArray
{
public:
  type array[size];
};

template <typename TYPE>          // (2)
TYPE Sum(TYPE s2, TYPE s3)
{
  return s2 + s3;
}

int main()
{
  StaticArray<int, 1> someArray;  // (3)

  int a = 0; double x = 8;
  Sum(a, a);                      // (4)
  Sum<double> (a, x);             // (5)
}
```

В строке 1 объявлен шаблон класса, в строке 2 объявлен шаблон функции. В строке 3 производится явное инстанциирование шаблона класса, типами параметров выступают int и числовое значение. В строке 4 производится неявное инстанциирование шаблона функции, тип параметра шаблона здесь будет int, который выводится из типа входного аргумента. В строке 5 производится явное инстанциирование; оно здесь необходимо, потому что из типов входных аргументов нельзя однозначно определить, какой тип параметра должен использоваться в шаблоне.

Вообще, шаблоны в C++ – это обширная тема, заслуживающая отдельной книги, поэтому изложить ее полностью не представляется возможным. Для лучшего понимания дальнейшего материала, кроме уже изложенных базовых понятий, рекомендуется ознакомиться со следующими темами: шаблоны с переменным числом параметров; частичная специализация шаблонов; автоматический вывод типов[17].

Программирование с использованием шаблонов, или, как его еще называют, метапрограммирование, достаточно сложное, поскольку предполагает высокий уровень абстракции в сочетании с неявным генерированием кода на этапе компиляции. Здесь используется другая парадигма, которая очень отличается от привычного объектно-ориентированного подхода; по своей природе шаблоны ближе к функциональному программированию. Однако именно благодаря указанным особенностям они позволяют легко и естественно решать многие задачи, в которых использование классических средств C++ порождает немало проблем.

Применительно к нашей теме, т. е. проектированию обратных вызовов, с помощью шаблонов можно реализовать множество интересных вещей, как это будет показано в следующих главах. Начнем с синхронных вызовов, как наиболее простых.

[17] Для изучения можно порекомендовать книгу «Вандевурд, Джосаттис, Грегор. Шаблоны C++: справочник разработчика», где подробно рассматриваются соответствующие темы.

4.2. Синхронные вызовы

4.2.1. Инициатор

Проанализируем различные реализации инициатора синхронных вызовов (Листинг 24):

Листинг 24. Реализации инициатора для синхронных вызовов

```
  class Executor
  {
  public:
    void callbackHandler(int eventID);
    void operator() (int eventID);
  };

  using ptr_callback = void(*) (int, void*);
  using ptr_callback_static = void(*) (int, Executor*);
  using ptr_callback_method = void(Executor::*)(int);

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

  void run(ptr_callback_static ptrCallback, Executor* contextData = nullptr)
// (2)
  {
    int eventID = 0;
    ptrCallback(eventID, contextData);
  }

  void run(Executor* ptrClientCallbackClass, ptr_callback_method ptrClientCallbackMethod)
// (3)
  {
    int eventID = 0;
    (ptrClientCallbackClass->*ptrClientCallbackMethod)
(eventID);
  }

  void run(Executor callbackHandler)  // (4)
  {
    int eventID = 0;
    callbackHandler(eventID);
  }
```

Можно заметить, что все реализации, по сути, одинаковы, отличаются только типы и количество входных аргументов. Поэтому, можно попытаться сделать шаблон. Возьмем наиболее простой случай, когда функция на вход принимает только один параметр (Листинг 25):

Листинг 25. Шаблон для инициатора синхронного вызова

```
template <typename CallbackArgument>
void run(CallbackArgument callbackHandler)
{
  int eventID = 0;
  //Some actions
  callbackHandler(eventID);
}
```

Получившийся шаблон подходит для реализации вызовов с помощью функциональных объектов (в Листинг 25 это строка номер 4), а также для лямбда-выражений. В последнем случае в качестве типа аргумента будет подставлен тип лямбда-выражения, определяемый компилятором.

Что же нам делать для остальных реализаций? Для указателей на функцию и указателей на статический метод (строки 1 и 2) можно сделать отдельный шаблон с двумя параметрами (Листинг 26):

Листинг 26. Шаблон для инициатора с двумя параметрами

```
template <typename CallbackArgument, typename Context>
void run(CallbackArgument callbackHandler, Context* context)
{
  int eventID = 0;
  //Some actions
  callbackHandler(eventID, context);
}
```

Однако такое решение противоречит идее обобщенного кода: для нового типа данных мы реализуем новый код, который дублирует предыдущий, за исключением самого вызова. Как следствие, если в коде инициатора нужно сделать изменения, их придется переносить на все объявленные функции. Но это еще не все: указанное решение не покрывает случая использования указателей на метод-член класса, синтаксис вызова которого отличается от синтаксиса вызова внешней функции. Таким образом, придется реализовать еще один шаблон функции для вызова метода класса. При этом он должен будет иметь другое имя, иначе возникнет конфликт с предыдущим определением: количество входных параметров одинаково, и компилятор не знает, какую реализацию шаблона подставлять при инстанциировании.

Вот если бы мы могли для всех аргументов использовать единый общий параметр, тогда все реализации могли быть описаны с помощью одного единственного шаблона. Решить эту задачу можно путем преобразования вызовов.

4.2.2. Преобразование вызовов

Для преобразования вызовов используется функциональный объект, в котором хранятся данные, необходимые для осуществления обратного вызова. Объявляется перегруженный опе-

ратор, который принимает информацию вызова. Реализация оператора выполняет требуемый вызов, передавая ему на вход полученную информацию вызова и, дополнительно, хранимые данные[18].

Вначале рассмотрим вызовы через указатели на функцию. Создадим шаблон для функционального объекта, в котором будем хранить указатель на функцию и контекст. Перегрузим оператор вызова функции, в реализации которого по хранимому указателю вызовем функцию-обработчик и передадим ей хранимый контекст (Листинг 27).

Листинг 27. Функциональный объект для вызова функции с передачей контекста

```
template<typename Function, typename Context>  // (1)
class CallbackConverter  // (2)
{
public:
  CallbackConverter (Function argFunction = nullptr, Context argCon
  {
    ptrFunction = argFunction; context = argContext;
  }

  void operator() (int eventID)      // (4)
  {
     ptrFunction(eventID, context);  // (5)
  }
private:
  Function ptrFunction;  // (6)
  Context context;       // (7)
};
```

В строке 1 объявлен шаблон с двумя параметрами – тип указателя на функцию и тип для контекста. В строке 2 объявлено имя класса. В строке 3 объявлен конструктор, который будет сохранять требуемые значения – указатель на функцию и указатель на контекст, переменные для хранения объявлены в строках 6 и 7. В строке 4 осуществляется перегрузка оператора вызова функции, который делает обратный вызов, передавая информацию и сохраненный контекст.

Рассмотренный шаблон также будет работать для указателей на статический метод класса, только необходимо объявить соответствующие типы указателей.

Для указателей на метод-член класса сделаем специализацию шаблона, как это показано в Листинг 28.

Листинг 28. Функциональный объект для вызова метода класса

```
  template<typename ClassName>  // (1)
  class CallbackConverter <void(ClassName::*)(int), ClassName>
// (2)
```

[18] Здесь функциональный объект реализует паттерн «адаптер». Для знакомства с паттернами вообще, и с паттерном «адаптер» в частности можно порекомендовать книгу « Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. Приемы объектно-ориентированного проектирования. Паттерны проектирования».

```
    {
    public:
      using ClassMethod = void(ClassName::*)(int);  // (3)

        CallbackConverter(ClassMethod  methodPointer  =  nullptr,
ClassName* classPointer = nullptr)  // (4)
      {
        ptrClass = classPointer; ptrMethod = methodPointer;
      }

      void operator()(int eventID)       // (5)
      {
        ptrClass->*ptrMethod)(eventID);  // (6)
      }
    private:
      ClassName* ptrClass;    // (7)
      ClassMethod ptrMethod;  // (8)
    };
```

В строке 1 объявлен шаблон с параметром – именем класса. В строке 2 объявлена специализация шаблона из Листинг 27. Именно эта специализация будет выбрана компилятором, если шаблон инстанциируется указателем на метод класса и указателем на класс. В строке 3 объявлен тип – указатель на метод класса. Этот тип выводится из имени класса, поэтому в шаблоне одного параметра – имени класса – будет достаточно. В строке 4 объявляется конструктор, который будет сохранять требуемые значения – указатель на экземпляр класса и указатель на метод, переменные для хранения объявлены в строках 7 и 8. В строке 5 перегружается оператор вызова функции, который вызывает метод класса.

4.2.3. Исполнитель

Итак, определив объекты для преобразования вызовов, мы теперь можем использовать в шаблоне-инициаторе, определенном в Листинг 25 п. 4.2.1, любые типы аргументов обратного вызова. Пример приведен в Листинг 29.

Листинг 29. Исполнитель для шаблона-инициатора синхронного вызова

```
    class Executor  // (1)
    {
    public:
      static void staticCallbackHandler(int eventID, Executor* executor
      void callbackHandler(int eventID) {}
      void operator() (int eventID) {}
    };

    void ExternalHandler(int eventID, void* somePointer)
    {
      Executor* ptrClass = (Executor*)somePointer;
    }
```

```
    int main()
    {
      Executor executor;
      int capturedValue = 0;

      // (2) External function
      using FunctionPointer = void(*)(int, void*);
      using FunctionConverter = CallbackConverter<FunctionPointer, void
      run(FunctionConverter(ExternalHandler, &executor));

      // (3) Static method
      using StaticPointer = void(*)(int, Executor*);
      using StaticConverter = CallbackConverter<StaticPointer, Executor
      run(StaticConverter(Executor::staticCallbackHandler, &executor));

      // (4) Member merthod
      using MethodPointer = void(Executor::*)(int);
      using MethodConverter = CallbackConverter<MethodPointer, Executor
     run(MethodConverter(&Executor::callbackHandler, &executor));

      // (5) Functional object
      run(executor);

      // (6) lambda-expression
          auto   lambda   =   [capturedValue](int   eventID)   {/
*it will be called by initiator*/};
      run(lambda);
    }
```

В строке 1 объявлен класс исполнителя, в котором определены все необходимые типы вызовов: статический метод, метод-член, перегруженный оператор. Для вызовов 2, 3 и 4 в качестве аргумента передается функциональный объект для преобразования, который инстанциируется соответствующими типами. В остальных случаях нужный аргумент передается непосредственно, преобразования вызовов там не нужно. При использовании лямбда-выражения (строка 6) компилятор неявно определит его тип и подставит его в функцию шаблона-инициатора как аргумент.

При использовании преобразования вызовов можно использовать сокращенную запись без дополнительного объявления промежуточных типов, в этом случае код получается более компактным, но более запутанным (см. Листинг 30)

Листинг 30. Преобразование вызовов без объявления промежуточных типов

```
    // (2) External function
    run(CallbackConverter<void(*)
(int, void*), void*>(ExternalHandler, &executor));

    // (3) Static method
```

```
    run(CallbackConverter<void(*)
(int, Executor*), Executor*>(Executor::staticCallbackHandler, &executor

    // (4) Member merthod
    run(CallbackConverter<void(Executor::*)
(int), Executor>(&Executor::callbackHandler , &executor));

    // (6) lambda-expression
    run([capturedValue](int              eventID)              {/
*it will be called by initiator*/});
```

4.3. Вызовы в алгоритмах

4.3.1. Описание проблемы

Алгоритмы – краеугольный камень информатики, они встречаются практически во всех ее разделах. Таким образом, проектирование и разработка алгоритмов – одна из важнейших задач как в теоретической науке, так и в инженерной практике.

В реализации алгоритмов одной из трудностей, встающей перед разработчиком, является адаптация для конкретной структуры данных. Это связано с тем, что алгоритмы задают последовательность операций, но не определяют данные, с которыми работают. Предполагается, что алгоритм работает с любой структурой данных.

Например, предположим, что мы написали код для алгоритма сортировки. Естественно предположить, что он будет сортировать числа. Но вот появилась новая задача: отсортировать строки. По сравнению с исходной реализацией у нас теперь другая структура данных (строки) и новые правила сравнения (строки сравниваются совсем не так, как числа). А ведь в будущем, возможно, появятся более сложные случаи – например, сортировка структур по отдельным полям… Как написать универсальный код, работающий с любыми типами данных?

4.3.2. Параметризация типов

Обозначенная выше проблема в рамках параметрического полиморфизма решается просто: код оформляется в виде шаблона, параметрами шаблона выступают типы данных. При инстанциировании шаблона генерируется код, в который подставляются соответствующие типы.

Поясним сказанное на примере. Предположим, мы реализовали алгоритм сортировки пузырьком (Листинг 31).

Листинг 31. Сортировка массива методом пузырька

```
void sort_bubble(int* data, size_t size)
{
  for (size_t i = 0; i < size - 1; i++)
  {
    for (size_t j = 0; j < size - i - 1; j++)
    {
      if (data[j + 1] < data[j])
      {
        int temp = data[j];
        data[j] = data[j + 1];
        data[j + 1] = temp;
      }
    }
  }
}
```

Описанный код работает с числами. Параметризуем типы (Листинг 32):

Листинг 32. Параметризация типов для сортировки пузырьком

```
template <typename Data>                    // (1)
void sort_bubble(Data* data, size_t size)   // (2)
{
  for (size_t i = 0; i < size - 1; i++)
  {
    for (size_t j = 0; j < size - i - 1; j++)
    {
      if (data[j + 1] < data[j])
      {
        Data temp = data[j]; // (3)
        data[j] = data[j + 1];
        data[j + 1] = temp;
      }
    }
  }
}
```

По сравнению с предыдущим листингом изменений здесь совсем немного: в строке 1 объявлен параметр шаблона для типа данных, в реализации функции вместо типа данных подставляется параметр шаблона (строки 2 и 3). Теперь мы можем делать сортировку для любого типа данных: мы просто вызываем функцию и передаем ей требуемую переменную-массив, а компилятор сгенерирует код для соответствующего массива.

4.3.3. Объявление предикатов

После описанной модификации первоначального кода у нас остается одна проблема: как выполнять операции сравнения для нечисловых данных, например, структур? Ведь алгоритм не знает, да и не должен знать, по каким правилам нужно их сравнивать. Выход очевидный – делегировать эти операции создателю данных. Для этого будем использовать обратный вызов «вычисление по запросу» (п. 1.2.2). Параметрами вызова будут экземпляры данных, а возвращать он будет результат сравнения. Оформленный таким образом вызов называется предикатом.

> **Предикат** – это выражение, принимающее одну или более величину и возвращающее результат булевого типа.

Объявим предикат как дополнительный параметр шаблона (Листинг 33).

Листинг 33. Шаблон с объявлением предиката

```
template <typename Data, typename Predicate>            // (1)
void sort_bubble(Data* data, size_t size, Predicate less) //
(2)
{
  for (size_t i = 0; i < size - 1; i++)
  {
    for (size_t j = 0; j < size - i - 1; j++)
```

```
        {
          if (less (data[j + 1], data[j])) // (3)
          {
            Data temp = data[j];
            data[j] = data[j + 1];
            data[j + 1] = temp;
          }
        }
      }
    }
```

По сравнению с предыдущим кодом из Листинг 32 изменения здесь следующие: в объявлении шаблона (строка 1) объявлен дополнительный параметр – предикат, в функции шаблона (строка 2) предикат объявляется как дополнительный входной параметр, в строке 3 вместо операции сравнения происходит вычисление предиката.

В качестве предикатов могут использоваться:

- глобальные функции;
- статические функции класса;
- перегруженные операторы;
- лямбда-выражения.

В Листинг 34 продемонстрировано использование предикатов различных типов.

Листинг 34. Сортировка данных с использованием предикатов различных типов

```
    struct DBRecord  // (1)
    {
      char firstName[50];
      char lastName[50];
    };

    bool CompareByFirstName(const DBRecord& rec1, const DBRecord&
rec2)  // (2)
    {
      return strcmp(rec1.firstName, rec2.firstName) < 0;
    }

    bool CompareByLastName(const DBRecord& rec1, const DBRecord&
rec2)  // (3)
    {
      return strcmp(rec1.lastName, rec2.lastName) < 0;
    }

    class SortRules  // (4)
    {
    public:
      enum {SORT_ASC = 1, SORT_DESC = 2} sortDirect;          // (5)
      enum { SORT_FIRST_NAME = 1, SORT_LAST_NAME = 2 } sortWhat; //
(6)
```

```
     bool operator () (const DBRecord& rec1, const DBRecord& rec2)
const // (7)
      {
        if (sortDirect == SORT_ASC)
        {
          if (sortWhat == SORT_FIRST_NAME)
          {
            return strcmp(rec1.firstName, rec2.firstName) < 0;
          }
        else
        {
          return strcmp(rec1.lastName, rec2.lastName) < 0;
        }
        }
        else
        {
          if (sortWhat == SORT_FIRST_NAME)
          {
            return strcmp(rec1.firstName, rec2.firstName) > 0;
          }
          else
          {
            return strcmp(rec1.lastName, rec2.lastName) > 0;
          }
        }
      }
    };

    int main()
    {
    DBRecord dbRecArray[10];  // (8)
    //Read from database

    sort_bubble(dbRecArray, 10, CompareByFirstName); // (9)
    sort_bubble(dbRecArray, 10, CompareByLastName);  // (10)

    sort_bubble(dbRecArray, 10, [](const DBRecord& rec1, const
DBRecord& rec2)  // (11)
    {
    return strcmp(rec1.firstName, rec2.firstName) < 0;
    });
    sort_bubble(dbRecArray, 10, [](const DBRecord& rec1, const
DBRecord& rec2)  // (12)
    {
    return strcmp(rec1.lastName, rec2.lastName) < 0;
    });

    SortRules rules; // (13)
```

```
rules.sortWhat = SortRules::SORT_LAST_NAME;  // (14)
rules.sortDirect = SortRules::SORT_ASC;      // (15)
sort_bubble(dbRecArray, 10, rules);          // (16)
}
```

В строке 8 объявлен массив структур, сами структуры объявлены в строке 1 (предположим, что это записи базы данных). В строке 9 и 10 происходит сортировка массива с использованием предикатов в виде внешней функции, в строках 11 и 12 – в виде лямбда-выражений.

В строке 13 объявлен предикат как экземпляр класса. Если посмотреть объявление класса (строка 4), то можно увидеть, что он позволяет осуществлять настройку правил: в строке 5 имеется переменная для настройки порядка сортировки (возрастание либо убывание), в строке 6 имеется переменная для настройки поля сортировки. В строке 7 реализован перегруженный оператор, который в соответствии с настроенными правилами вычисляет, является ли первый элемент меньше второго. В строках 14 и 15 производится настройка предиката, в строке 16 – сортировка в соответствии с заданными правилами.

4.3.4. Предикаты по умолчанию

Итак, мы рассмотрели, как с помощью предикатов реализуется операция вычисления меньшего из двух элементов. Но далеко не всегда требуется сортировать сложные структуры данных, зачастую это всего лишь обычные числовые значения. В этом случае придется объявлять предикат с тривиальной реализацией (сравнить два числа). Может также случиться, что у нас в объявлении элемента данных уже реализован перегруженный оператор сравнения, тогда в предикате придется дублировать его код. Всего этого можно избежать, если объявить предикат, который будет использоваться по умолчанию. Реализация приведена в Листинг 35.

Листинг 35. Шаблон с предикатом по умолчанию

```
template <typename Data>  // (1)
struct default_less
{
  bool operator()(const Data& x, const Data& y)  // (2)
  {
    return x < y;
  }
};

  template<typenameData,typenamePredicate=default_less<Data>>
// (3)
  void sort_bubble(Data* data, size_t size, Predicate less = Predicat
  {
    for (size_t i = 0; i < size - 1; i++)
    {
      for (size_t j = 0; j < size - i - 1; j++)
      {
        if (less (data[j + 1], data[j]))
        {
          Data temp = data[j];
          data[j] = data[j + 1];
```

```
            data[j + 1] = temp;
          }
        }
      }
    }
```

В строке 1 объявлен шаблон для структуры, реализующей предикат сравнения. В этой структуре перегружен оператор (строка 2), который возвращает результат сравнения двух аргументов. Он будет корректно работать как для чисел, так и для объектов, в которых перегружен оператор «меньше».

В строке 3 объявлен шаблон для функции сортировки. Первый параметр шаблона – это тип данных, которые необходимо сортировать, а второй параметр – это тип предиката. По умолчанию типом предиката является структура, объявленная выше, которая инстанциируется соответствующим типом данных.

В строке 4 объявлена функция шаблона. Первый параметр здесь – это данные для сортировки, а второй параметр – предикат для вычисления меньшего элемента. Если при вызове функции предикат не задан, то в качестве значения по умолчанию будет подставлена переменная – экземпляр структуры, объявленной в строке 1. Инстанциироваться эта структура будет типом **Data**, переданным как первый параметр шаблона.

Итак, на примере алгоритма сортировки мы рассмотрели, как реализуются предикаты для выбора меньшего элемента из двух. Подобным образом можно реализовать множество других операций: сравнения, сложения, вычисления хэш-суммы и т. п. Таким образом, предикаты предлагают удобный способ реализации арифметико-логических операций с нечисловыми типами данных. Частично снимается проблема монолитной архитектуры при использовании функциональных объектов: мы можем реализовать любое количество нужных объектов и подставлять их в шаблон по мере необходимости [19]. И в заключение отметим, что концепция предикатов широко используется в реализации алгоритмов стандартной библиотеки STL.

[19] Мы употребили термин «частично», потому что полной независимости здесь нет: при изменении функционального объекта нужно перекомпилировать как инициатор, так и исполнитель. Таким образом, независимость здесь обеспечивается только на уровне исходного кода.

4.4. Асинхронные вызовы

4.4.1. Инициатор

Также, как мы делали при анализе синхронных вызовов, проанализируем различные реализации инициатора асинхронных вызовов (Листинг 36, некоторые фрагменты кода пропущены, чтобы не загромождать описание).

Листинг 36. Реализации инициатора асинхронных вызовов для различных типов аргументов

```
class Executor;

class CallbackHandler
{
public:
  void operator() (int eventID);
};

//Pointer to function
class Initiator1
{
public:
  using ptr_callback = void(*) (int, void*);
  void setup(ptr_callback pPtrCallback, void* pContextData) ;

private:
  ptr_callback ptrCallback = nullptr;
  void* contextData = nullptr;
};

//Pointer to the class static method
class Initiator2
{
public:
  using ptr_callback_static = void(*) (int, Executor*);
  void setup(ptr_callback_static pPtrCallback, Executor* pContextDa

private:
      ptr_callback_static ptrCallback = nullptr;
      Executor* contextData = nullptr;
};

//Pointer to the class member method
class Initiator3
{
public:
```

```
  using ptr_callback_method = void(Executor::*)(int);

  void setup(Executor* argCallbackClass, ptr_ callback_method argCa

private:
  Executor* ptrCallbackClass = nullptr;
  ptr_ callback_method ptrCallbackMethod = nullptr;
};

//Functional object
class Initiator4
{
public:
  void setup(const CallbackHandler& callback);

private:
  CallbackHandler callbackObject;
};
```

Аналогично синхронным вызовам, можно заметить, что все реализации по своей сути практически одинаковы, отличается только тип и количество аргументов. Попробуем для класса сделать шаблон (Листинг 37).

Листинг 37. Шаблон для инициатора асинхронного вызова

```
template<typename CallbackArgument>
class Initiator
{
public:
  void setup(const CallbackArgument& argument)
  {
      callbackHandler = argument;
  }

  void run()
  {
    int eventID = 0;
    //Some actions
    callbackHandler(eventID);
  }

private:
  CallbackArgument callbackHandler;
};
```

Получившийся шаблон подходит для реализации с использованием функционального объекта. Для реализаций с использованием указателей на функцию, указателей на статический метод и на метод-член класса можно использовать шаблон для преобразования вызовов (см. п. 4.2.2). А вот реализация с помощью лямбда-выражений здесь работать не будет, потому что

хранить лямбда-выражение как аргумент, подобно обычной переменной, нельзя. Рассмотрим этот вопрос подробнее.

4.4.2. Хранение лямбда-выражений

Почему хранение лямбда-выражений является проблемой?

При объявлении лямбда-выражения компилятор генерирует функциональный объект, который называется объект-замыкание (closure type). Этот объект хранит в себе захваченные переменные и имеет перегруженный оператор вызова функции. Сигнатура оператора повторяет сигнатуру лямбда-выражения, а в теле оператора размещается код выражения. Пример объекта-замыкания приведен в Листинг 38.

Листинг 38. Лямбда-выражение и объект-замыкание

```
    int main()
    {
      int capture = 0;
      [capture](int eventID) {/*this is a body of lambda*/};

                                                              //The
following object will be generated implicitly by the compiler from lamb
      class Closure
      {
      public:
        Closure(int value) :capture(value) {}

        void operator() (int eventID)
        {
          /*this is a body of lambda*/
        }

        int capture; //captured value
      };
    }
```

Как видно из примера, в зависимости от состава захваченных переменных объект-замыкание будет иметь различный тип. То есть, этот тип заранее неизвестен, он будет сгенерирован компилятором. По этой причине тип лямбда-выражения не имеет заранее определенного имени, и мы не можем просто объявить переменную соответствующего типа и присвоить ей значение, как мы делаем, например, в случае использования числовых переменных.

Если лямбда-выражение не захватывает переменные, то стандарт допускает преобразование лямбда-выражения к указателю на функцию. В этом случае объект-замыкание не содержит переменных, что позволяет код лямбда-выражения оформить в виде статической функции и объявить соответствующий оператор преобразования. Таким образом, появляется возможность сохранить лямбда-выражение в переменной типа "указатель на функцию", как показано в Листинг 39.

Листинг 39. Объект-замыкание с преобразованием в указатель на функцию

```
    int main()
    {
      [](int eventID) {/*this is a body of lambda*/};  // (1)

                                                          //The
following object will be generated implicitly by the compiler from lamb
      class Closure  // (2)
      {
      public:
        void operator() (int eventID)  // (3)
        {
          call_invoker(eventID);
        }

              static  void    call_invoker(int  eventID)  {  /
*this is a body of lambda*/ }  // (4)

        using function_pointer = void(*)(int);  // (5)

        operator function_pointer() const       // (6)
        {
          return call_invoker;
        }
      };

      //Conversion the closure object to the function pointer
      Closure cl;  // (7)
      using pointer_to_function = void(*)(int);  // (8)
      pointer_to_function fptr = cl;             // (9)

      //Conversion a lambda to the function pointer
     fptr = [](int eventID) {/*this is a body of lambda*/};  // (10)
    }
```

В строке 1 объявлено лямбда-выражение, в строке 2 объявлен объект-замыкание. Подчеркнем: этот объект здесь всего лишь для демонстрации, чтобы показать, как он будет сгенерирован компилятором. В реальном коде такой объект объявлять не нужно, компилятор его создаст при объявлении лямбда-выражения.

В строке 3 объявлен перегруженный оператор, который вызывает статическую функцию 4. В той функции размещается код лямбда-выражения.

В строке 5 объявлен тип указателя на функцию, в строке 6 объявлен оператор преобразования типа. Реализация оператора возвращает указатель на статическую функцию 4.

В строках 7–9 показано, как осуществляется преобразование функционального объекта к указателю на функцию. В строке 7 объявлен объект-замыкание, в строке 8 объявлен тип указателя на функцию. В строке 9 объявляется переменная этого типа и вызывается перегруженный оператор присваивания 6, который возвращает указатель на функцию. Теперь в пере-

менной **fptr** будет храниться указатель на статическую функцию, которая была объявлена в соответствующем функциональном объекте.

В строке 10 продемонстрировано преобразование лямбда-выражения к указателю на функцию. Все действия, описанные выше с использованием функционального объекта, будут неявно сгенерированы компилятором.

Итак, если лямбда-выражение не захватывает переменные, то сохранить его как аргумент достаточно просто: объявляется указатель на функцию, которому присваивается соответствующее выражение. Однако в случае захвата переменных ситуация меняется. Теперь в объекте-замыкании будут храниться захваченные переменные, и компилятор не может код лямбда-выражения разместить в статической функции, ведь статическая функция не имеет доступа к членам класса. Поэтому указанный код вставляется в функцию-член класса. Казалось бы, почему не объявить указатель на функцию-член класса и присвоить ему значение? Проблема в том, что для этого необходимо знать тип класса, т. е. тип объекта-замыкания. А этот тип заранее неизвестен, он генерируется на этапе компиляции. Таким образом, здесь невозможно объявить указатель на метод и присвоить ему значение.

Если необходимо хранить лямбда-выражение в локальной переменной, можно использовать тип **auto**. Это означает, что компилятор подставит соответствующий тип, который будет сгенерирован из объявления лямбда-выражения (см. Листинг 40).

Листинг 40. Сохранение лямбда-выражения в локальной переменной

```
int capture = 10;

auto lambda = [capture](int eventID) {/
*this is a body of lambda*/};

lambda(10); //lambda call
```

Однако указанный способ не будет работать, когда требуется сохранить лямбда-выражение в классе. Мы не можем объявить переменную – член класса с типом **auto**, потому что это означало бы объявление переменной заранее не определенного типа, что не допускается.

Организовать хранение лямбда-выражения внутри класса можно с помощью шаблона, в котором тип выражения будет параметризован. Однако при инстанциировании шаблона переменной, предназначенной для хранения, значение должно быть присвоено в конструкторе. Это нельзя сделать позже, потому что в объекте-замыкании, генерируемым компилятором, запрещен оператор присваивания. Это и понятно: поскольку тип каждого объявленного лямбда-выражения является уникальным, то мы не можем ему ничего присваивать, кроме самого себя.

Добавим в реализацию инициатора, описанного в Листинг 37 п. 4.4.1, два конструктора. Один конструктор будет с переменной – аргументом обратного вызова для инициализации члена класса. Другой конструктор будет без аргументов (конструктор по умолчанию), чтобы оставить возможность отложенной настройки (Листинг 41).

Листинг 41. Инициатор с дополнительными конструкторами

```
template<typename CallbackArgument>
class Initiator
{
public:
  Initiator() {}
```

```
    Initiator(const CallbackArgument& argument) : callbackHandler(arg

    void setup(const CallbackArgument& argument)
    {
      callbackHandler = argument;
    }

    void run()
    {
      int eventID = 0;
      //Some actions
      callbackHandler(eventID);
    }

  private:
    CallbackArgument callbackHandler;
  };
```

Для любых типов аргументов обратного вызова, кроме лямбда-выражений, допускается использование обоих конструкторов. Для лямбда-выражений допускается использование только конструктора с аргументом, при попытке использования конструктора по умолчанию компилятор выдаст ошибку. Кроме того, в этом случае нельзя будет вызвать метод setup – также будет сгенерирована ошибка. Таким образом, использование инициатора с лямбда-выражением не предполагает динамической модификации: настройка происходит один раз в конструкторе при инстанциировании шаблона, и больше изменить ее нельзя[20].

А какой тип аргумента нам указывать при инстанциировании шаблона, ведь тип лямбда-выражения является анонимным? Для этой цели мы будем использовать ключевое слово **decltype**, которое возвращает тип объявленной переменной (см. Листинг 42).

Листинг 42.Инстанциирование шаблона асинхронного обратного вызова для лямбда-выражения

```
  int capture = 10;
  auto     lambda     =     [capture](int     eventID)     {/
*this is a body of lambda*/};
  Initiator<decltype(lambda)> callbackLambda1 (lambda); // Ok, initia
  Initiator<decltype(lambda)> callbackLambda = lambda; // Ok, implici
  Initiator<decltype(lambda)>     callbackLambda2;          //
Error: attempting to reference a deleted function
  callbackLambda.setup(lambda);                             //
Error:  'operator' =  attempting to reference a deleted function
  callbackLambda.run();
```

[20] Указанная проблема решается при использовании универсального аргумента, о чем пойдет речь в следующей главе

4.4.3. Исполнитель

В Листинг 43 приведены примеры реализации исполнителя для различных типов аргументов. Объявления класса **CallbackConverter** представлены в Листинг 27 и Листинг 28 п. 4.2.2, инициатор используется из Листинг 41 п. 4.4.2.

Листинг 43. Исполнитель для шаблона-инициатора с различными типами аргумента

```
class Executor  // (1)
{
public:
  static void staticCallbackHandler(int eventID, Executor* executor
  void callbackHandler(int eventID) {}
  void operator() (int eventID) {}
};

void ExternalHandler(int eventID, void* somePointer) {}  // (2)

int main()
{
  Executor executor;  // (3)
  int capturedValue = 0;

  // (4)  Pointer to the external function
  using PtrExtFunc = void(*) (int, void*);
// (5)
  usingCallbackExtFunction=CallbackConverter<PtrExtFunc,void*>;
// (6)
   Initiator<CallbackExtFunction> initExtFunction;
// (7)
    initExtFunction.setup(CallbackExtFunction(ExternalHandler, &execu

    // (9) Pointer to the static method
    using PtrStaticMethod = void(*) (int, Executor*);  // (10)
 usingCallbacStaticMethod=CallbackConverter<PtrStaticMethod,Executor*>;
// (11)
    Initiator<CallbacStaticMethod> initStaticMethod;
    initStaticMethod.setup(CallbacStaticMethod(Executor::staticCallba

    // (14) Pointer to the class member method
              using     PtrMethod     =     void(Executor::*)
(int);                                          // (15)
usingCallbackMemberMethod=CallbackConverter<Executor,void(Executor::*)
(int)>;     // (16)
    Initiator<CallbackMemberMethod> initMemberMethod;  // (17)
    initMemberMethod.setup(CallbackMemberMethod(&executor, &Executor:
```

```
      // (19) Functional object
      Initiator<Executor> initFunctionObject;  // (20)
      initFunctionObject.setup(executor);      // (21)

      // (22) Lambda-expression
          auto   lambda   =   [capturedValue](int   eventID)   {/
*Body of lambda*/};  // (23)
     Initiator<decltype(lambda)> initLambda ( lambda);
  // (24)
    }
```

В строке 1 объявлен класс – исполнитель, в котором определены необходимые нам типы вызовов: статический метод, метод-член, перегруженный оператор. В строке 2 объявлена внешняя функция, в строке 3 – экземпляр исполнителя.

В строке 4 показан обратный вызов через указатель на функцию. Объявлен тип указателя на функцию 5, тип функционального объекта для преобразования вызова 6, инстанциирование шаблона инициатора соответствующим типом 7, настройка инициатора 8. Запуск инициатора (метод **run**) не показан, чтобы не загромождать описание.

В строке 9 показан обратный вызов через указатель на статический метод класса. Похоже на предыдущий случай, только в качестве контекста используется указатель на класс. Объявлен тип указателя на статический метод 10, тип функционального объекта для преобразования вызова 11, инстанциирование инициатора соответствующего типа 12, настройка инициатора 13.

В строке 14 показан обратный вызов через указатель на метод-член класса. Объявлен тип указателя на метод 15, тип функционального объекта для преобразования вызова 16, инстанциирование инициатора соответствующим типом 17, настройка инициатора 18.

В строке 19 показан обратный вызов с помощью функционального объекта. Инстанциирование инициатора объявлено в строке 20, настройка инициатора – в строке 21.

В строке 22 показан обратный вызов с помощью лямбда-выражения. В строке 23 объявлено лямбда-выражение, которое запоминается в соответствующей переменной. В строке 24 инстанциирован инициатор типом лямбда-выражения. Инициатору в конструкторе передается переменная – объект указанного выражения.

Для случаев, когда используется преобразование вызовов (объявления 4, 9 и 14), можно использовать сокращенные объявления без использования промежуточных деклараций. Код в этом случае получается более компактным, но менее понятным (см. Листинг 44).

Листинг 44. Компактный способ объявлений при использовании преобразования вызовов

```
    int main
    {
      Executor executor;

      // (4) Pointer to the external function
                              Initiator<CallbackConverter<void(*)
(int, void*), void*>> initExtFunction;
                  initExtFunction.setup(CallbackConverter<void(*)
(int, void*), void*>(ExternalHandler, &executor));
```

```
      // (9) Pointer to the static method
                                Initiator<CallbackConverter<void(*)
(int, Executor*), Executor*>> initStaticMethod;
      initStaticMethod.setup(CallbackConverter<void(*)(int, Executor*),

      // (14) Pointer to the class member method
         Initiator<CallbackConverter<Executor,  void(Executor::*)
(int)>> initMemberMethod;
initMemberMethod.setup(CallbackConverter<Executor, void(Executor::*)
(int)>(&executor, &Executor::callbackHandler));
    }
```

Итак, как мы видим, для каждого типа аргумента обратного вызова нам приходится объявлять соответствующий инициатор. Может быть, можно сделать так, чтобы инициатор умел работать с различными типами аргументов? Для этого нужно спроектировать универсальный аргумент, чем мы и займемся в следующей главе.

4.5. Универсальный аргумент

4.5.1. Динамический полиморфизм

Для реализации универсального аргумента прежде всего необходимо обеспечить динамический полиморфизм, т. е. аргумент должен изменять свой тип в зависимости от задаваемого значения[21].

Как решается указанная задача в объектно-ориентированном дизайне? Объявляется базовый абстрактный класс, в котором описывается интерфейс в виде набора чисто виртуальных методов. Новый тип создается путем создания наследуемого класса, в котором объявляются нужные переменные и переопределяются методы. При инициализации создается класс нужного типа, и он сохраняется в переменной – указателе на базовый класс. Мы будем использовать аналогичный подход, только наследуемые типы будут создаваться динамически, используя параметры шаблона. Указанная техника называется «стирание типов»: при назначении нового типа аргумента предыдущий сохраненный уничтожается, и его место занимает новый[22].

Графическое изображение стирания типов изображено на Рис. 17. Рассмотрим начальное состояние а), показанное в верхней части рисунка. Имеется некоторый класс, назовем его **UniArgument**. В этом классе объявлен перегруженный оператор вызова функции *2*. Также здесь имеется указатель *3* типа **Callable***, который указывает на соответствующий экземпляр класса **Callable**. Класс **Callable** *4* объявлен внутри **UniArgument** и имеет виртуальный перегруженный оператор вызова функции с пустой реализацией.

Когда в **UniArgument** происходит вызов *1* перегруженного оператора *2*, последний через указатель *3* вызывает виртуальный перегруженный оператор класса **Callable**.

В нижней части рисунка б) показано, как назначается новый тип. Объявляется перегруженный оператор присваивания *10*, на входе он принимает аргумент обратного вызова 8. При вызове этого оператора старый экземпляр класса *4*, на который указывал указатель *3*, уничтожается в *11*, а вместо него создается новый класс **CallableObject** *5*, который наследуется от **Callable**. Внутри класса имеется поле 7, в которое записывается переданный аргумент *8*, тип этого поля совпадает с типом аргумента. В **CallableObject** переопределяется оператор вызова функции *6*, который, в свою очередь, осуществляет вызов через сохраненный аргумент *7*. Теперь указатель *3* указывает на новый созданный **CallableObject**, и при вызове *1* перегруженного оператора *2* будет вызываться перегруженный оператор указанного класса, который и выполнит обратный вызов.

---

[21] Термин «динамический полиморфизм» означает, что полиморфизм реализуется во время выполнения программы. В противоположность этому, статический полиморфизм реализуется на этапе компиляции программы. В строгом смысле этого термина динамический полиморфизм в C++ нереализуем, поскольку это язык со статической типизацией. Однако его можно смоделировать с помощью наследования и шаблонов, о чем пойдет речь далее.

[22] Для фундаментального изучения техники стирания типов можно порекомендовать книгу «Пикус Ф.Г.Идиомы и паттерны проектирования в современном C++», в которой указанной технике посвящена отдельная глава.

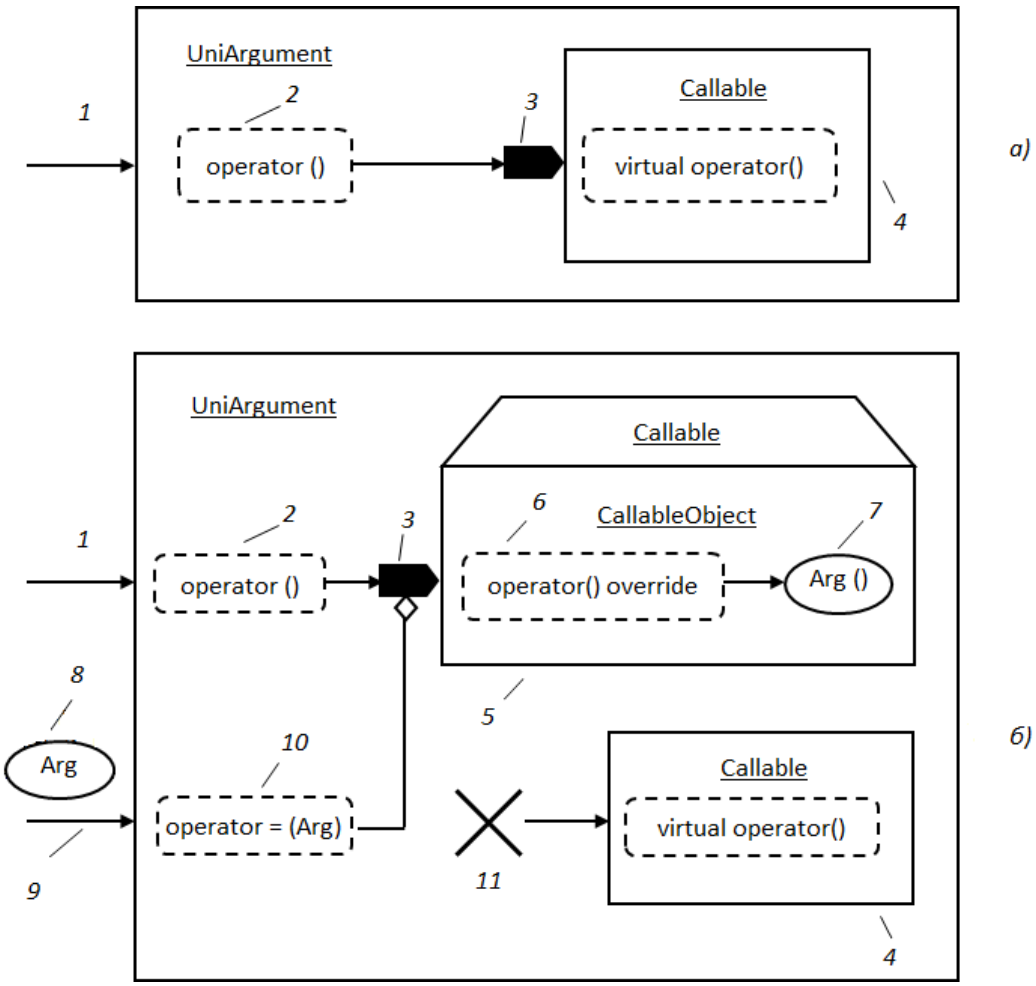

Рис. 17. Стирание типов: а) исходное состояние; б) состояние после назначения нового типа аргумента

Реализация рассмотренной схемы представлена в Листинг 45.

Листинг 45. Класс, реализующий стирание типов

```
class UniArgument  // (1)
{
private:
  class Callable   // (2)
  {
  public:
    virtual void operator()(int) = 0;          // (3)
  };

  std::unique_ptr<Callable> callablePointer;  // (4)

  template <typename ArgType>
  class CallableObject : public Callable      // (5)
  {
  public:
```

```
        CallableObject(ArgType argument) : storedArgument(argument) { }

        void operator() (int value) override  // (7)
        {
          storedArgument(value);  // (8)
        }
      private:
        ArgType storedArgument;   // (9)
      };

    public:
      void operator() (int value)  // (10)
      {
        callablePointer->operator()(value);  // (11)
      }

      template <typename ArgType>
      void operator = (ArgType argument)  // (12)
      {

  callablePointer.reset(new CallableObject<ArgType>(argument));  // (13
      }
    };
```

В строке 1 объявлен класс, реализующий универсальный аргумент. В строке 2 объявлен класс, который будет использоваться в качестве базового.

В базовом классе перегружен оператор вызова функции 3. Оператор объявлен чисто виртуальным, чтобы опустить его реализацию. Предполагается, что этот оператор будет выполнять обратный вызов, но аргумента вызова здесь нет, он будет храниться в наследуемом классе. Таким образом, реализация смысла не имеет. Более того, если, допустим, нам понадобится, чтобы оператор возвращал результат, то в нем должна присутствовать команда **return**, и какое тогда возвращать значение?

В строке 4 объявлен указатель на базовый класс, объявленный в 2.

В строке 5 объявлен шаблонный класс, который будет хранить переданный аргумент и вызывать его. Переменная для хранения аргумента объявлена в строке 9, тип переменной задается параметром шаблона. Аргумент назначается в конструкторе 6. Также в этом классе переопределяется оператор вызова функции 7, в котором происходит обратный вызов 8 через сохраненный аргумент.

В строке 10 объявлен перегруженный оператор основного класса, в котором вызывается соответствующий переопределенный оператор через указатель на базовый класс (строка 11).

В строке 12 объявлен шаблонный оператор присваивания, который настраивает аргумент. В реализации этого оператора 13 создается новый класс **CallableObject** нужного типа, в конструкторе этого класса переданный аргумент сохраняется, после чего переназначается указатель. Таким образом, при вызове оператора 10 будет вызван оператор соответствующего класса 11, и последний осуществит вызов через сохраненный аргумент.

Можно заметить, что универсальный аргумент не выполняет вызов сам по себе. По сути, он является своего рода оболочкой, которая перенаправляет вызов соответствующему объекту. Таким образом, у нас появляется новое понятие – объект вызова.

Объект вызова – это некоторая конструкция C++, поддерживающая интерфейс вызова в формате функции.

В соответствии с стандартом C++ на сегодняшний день[23], в качестве объектов вызова могут использоваться следующие конструкции:

- функции;
- методы класса;
- классы с перегруженным оператором вызова функции;
- лямбда-выражения.

В реализациях инициатора с помощью шаблонов, рассмотренных в предыдущих главах (см. п. 4.2.1, 4.4.1), аргумент вызова совпадает с объектом вызова. При использовании универсального аргумента эти сущности будут различаться: универсальный аргумент хранит в себе объект вызова.

Итак, мы реализовали универсальный аргумент, продемонстрируем теперь, как он может использоваться для реализации обратных вызовов (Листинг 46).

Листинг 46. Использование универсального аргумента

```
class Executor
{
public:
  static void staticCallbackHandler(int eventID, Executor* executor
  void callbackHandler(int eventID) {}
  void operator() (int eventID) {}
};

void ExternalHandler(int eventID, void* somePointer) {}

int main()
{
UniArgument argument;

Executor executor;
int capturedValue = 0;

using PtrExtFunc = void(*) (int, void*);
argument=CallbackConverter<PtrExtFunc,void*>(ExternalHandler,&executo
// (1)

using PtrStaticMethod = void(*) (int, Executor*);
CallbackConverter<PtrStaticMethod>(Executor::staticCallbackHandler,
//(2)

using PtrMemberMethod = void(Executor::*)(int);
argument = CallbackConverter<PtrMemberMethod, Executor>(&Executor::
```

[23] На момент написания книги это C++ 20.

```
    argument = executor;  // (4)

    argument      =      [capturedValue](int      eventID)      {/
*Body of lambda*/};  // (5)
    }
```

В строке 1 аргументу присваивается указатель на функцию, для преобразования вызовов используется класс **CallbackConverter** из Листинг 27 п. 4.2.2. Этот класс инстанциируется соответствующими типами, в конструкторе ему передается функция **ExternalHandler** и контекст, в качестве которого выступает указатель на класс **Executor**.

В строке 2 аргументу присваивается указатель на статический метод класса, что, в общем-то, идентично рассмотренному предыдущему случаю.

В строке 3 аргументу присваивается указатель на метод-член класса, для преобразования вызовов используется класс **CallbackConverter** из Листинг 28 п. 4.2.2. Этот класс инстанциируется соответствующими типами, в конструкторе ему передается указатель на класс и указатель на метод класса.

В строке 4 аргументу присваивается функциональный объект, в строке 5 – лямбда-выражение.

Отметим, что в универсальном аргументе лямбда-выражение сохраняется также просто, как и любой другой тип. Это связано с тем, что как оператор присваивания (**operator =** класса **UniArgument**, Листинг 45 п. 4.5.1), так и класс для хранения аргументов вызова (**CallableObject**, там же) реализованы в виде шаблонов. Когда мы вызываем указанный оператор, передавая ему лямбда-выражение, компилятор неявно выведет тип параметра шаблона из переданного аргумента, подобно тому, как это происходит в шаблонной функции для синхронных вызовов. В свою очередь, внутри оператора с помощью **new** динамически создается экземпляр **CallableObject**, инстанциированный соответствующим выведенным типом. Таким образом, явно указывать тип передаваемого аргумента не требуется, компилятор выводит его сам.

4.5.2. Настройка сигнатуры

До сих пор мы предполагали, что функция, реализующая обратный вызов, имеет тип **void** и на вход принимает только одно значение **eventID**, и исходя из этого, делали обратный вызов. А если выясняется, что функция должна иметь дополнительные параметры, нам придется изменять реализацию универсального аргумента и объектов, с ним связанных? А если нам необходимы инициаторы, которые используют функции с различными сигнатурами? Теперь что, для каждой сигнатуры придется реализовать отдельный аргумент? Есть другой путь: настройка сигнатуры вызова через параметры шаблона. Для ее реализации используется частичная специализация шаблона в сочетании с переменным числом параметров (partial template specialization, variadic templates), пример представлен в Листинг 47.

Листинг 47. Настройка сигнатуры

```
//General specialization
template <typename unused>  // (1)
class function;

//Partial specialization
template<typename Return, typename … ArgumentList >  // (2)
```

```
class function<Return(ArgumentList…)>
{
public:

  Return operator()(ArgumentList… arguments)  // (3)
  {
  }
};
```

В строке 1 объявлена общая специализация шаблона. Реализация класса здесь отсутствует, поскольку для каждой сигнатуры она будет различной. В строке 2 объявлен шаблон для частичной специализации, в котором два аргумента: тип возвращаемого значения и пакет параметров, передаваемых функции вызова.

В строке 3 объявлен перегруженный оператор, выступающий в качестве функции вызова. Сигнатура оператора содержит тип возвращаемого значения **Return** и пакет входных параметров **arguments**, которые разворачиваются в список аргументов. Таким образом, в зависимости от пакета и возвращаемого значения будет сгенерирована соответствующая специализация шаблона.

Описанная реализация всего лишь демонстрирует настройку сигнатуры. Практической пользы от нее немного, потому что тело перегруженного оператора пустое, и вызов осуществлен не будет. Используя описанную технику, добавим настройку сигнатуры к аргументу, реализующему стирание типов (Листинг 48).

Листинг 48. Стирание типов с настройкой сигнатуры

```
template <typename unused>
class UniArgument;

template<typename Return, typename … ArgumentList>
class UniArgument<Return(ArgumentList…)>  // (1)
{
private:
  struct Callable
  {
                          virtual      Return      operator()
(ArgumentList… arguments) = 0;  // (3)
  };

  std::unique_ptr<Callable> callablePointer;

  template <typename Argument>
  struct CallableObject : Callable
  {
    Argument storedArgument;

    CallableObject(Argument argument) : storedArgument(argument) {

   Return operator() (ArgumentList… arguments) override  // (8)
    {
```

```
        //return storedArgument(arguments…);
       return std::invoke(storedArgument, arguments…);    // (9)
      }
    };

  public:
    Return operator() (ArgumentList… arguments)         // (10)
    {
      return callablePointer->operator()(arguments…);  // (11)
    }

    template <typename Argument>
    void operator = (Argument argument)
    {
      callablePointer.reset(new CallableObject<Argument>(argument));
    }
  };
```

По сравнению с реализацией для фиксированной сигнатуры (Листинг 45 п. 4.5.1) изменения здесь следующие. Класс аргумента (строка 1) объявляется в виде шаблона. Параметрами шаблона выступают **Return** – тип значения, возвращаемого функцией, и **ArgumentList** – пакет параметров, определяющих типы передаваемых в функцию аргументов. При объявлении перегруженных операторов (строки 3, 8, 10), вместо конкретного типа возвращаемого значения подставляется параметр шаблона **Return**, вместо конкретных типов входных параметров подставляется **ArgumentList**. В местах, где происходит вызов оператора, пакет параметров раскрывается (строки 9 и 11), что означает, что вместо **arguments** будет подставлен список переменных с типами, заданными в пакете параметров.

Теперь в универсальном аргументе можно настраивать сигнатуру, как это продемонстрировано в Листинг 49.

Листинг 49. Использование аргумента с настройкой сигнатуры

```
  void         ExternalHandler1(int         eventID)         {/
*Do something*/}            // (1)
  int  ExternalHandler2(int eventID, int contextID) { return 0; }  //

  struct CallbackHandler  // (3)
  {
    void operator() (int eventID) {}
   bool operator() (int eventID, int contextID) { return false; }
  };

  int main()
  {
    int capturedValue = 100;
    CallbackHandler callbackObject;           // (4)

    UniArgument<void(int)> argument1;         // (5)
```

```
    UniArgument<bool(int, int)> argument2;    // (6)

    argument1 = ExternalHandler1;  // (7)
    argument2 = ExternalHandler2;  // (8)

    argument1 = callbackObject;    // (9)
    argument2 = callbackObject;    // (10)

        argument1   =   [capturedValue](int   eventID)   {/
*Do something*/};                              // (11)
    argument2 = [capturedValue](int eventID, int contextID) { /
*DoSomething*/return 0; };  // (12)

    argument1(3);               // (13)
    int res = argument2(4, 5);  // (14)

    return res;
  }
```

В строках 1 и 2 объявлены две внешние функции с различными сигнатурами. В строке 3 объявлен функциональный объект, в котором перегружены операторы вызова функции с такими же сигнатурами. В строке 4 объявлен экземпляр указанного объекта.

В строках 5 и 6 объявлены универсальные аргументы, в которых с помощью параметров шаблона настраивается нужная сигнатура. Далее этим аргументам будут присваиваться различные объекты вызова в зависимости от заданной сигнатуры.

В строках 7 и 8 в аргумент передаются внешние функции. В строках 9 и 10 передается функциональный объект, у которого, в зависимости от настроенной сигнатуры будет вызван соответствующий перегруженный оператор. В строках 11 и 12 передаются лямбда-выражения. В строках 13 и 14 осуществляются вызовы в соответствии с заданной сигнатурой.

4.5.3. Вызов метода класса

В текущей реализации универсальный аргумент может работать с любыми объектами вызова, за исключением методов класса. Это связано с тем, что вызов метода класса имеет другой синтаксис, отличный от вызова функции. Как добавить поддержку вызова методов? Можно предложить следующее решение: при настройке объекта назначать указатель на метод, аналогично обычной функции, а при вызове передавать экземпляр класса как дополнительный аргумент.

До появления стандарта C++17 реализация указанного способа была достаточно сложной: пришлось бы объявлять еще один объект, который наследовался от **Callable** и осуществлял вызов метода; для создания соответствующего объекта пришлось бы объявить дополнительный перегруженный оператор присваивания, который в качестве входного аргумента принимал указатель на метод. Но в новом стандарте появилась функция **std::invoke**, которая определяет тип принимаемого объекта вызова и осуществляет вызов для соответствующего типа. Таким образом, для поддержки вызова метода класса необходимо в реализации **CallableObject** изменить одну-единственную строчку:

```
Return operator() (ArgumentList… arguments) override  // (8)
{
  //return storedArgument(arguments…);
```

```
    return std::invoke(storedArgument, arguments…);  // (9)
}
```

На удивление просто, не правда ли?

Использование универсального аргумента для вызова метода класса представлено в Листинг 50.

Листинг 50. Использование универсального аргумента для вызова метода класса

```
struct CallbackHandler
{
  void handler1(int eventID) {};
  bool handler2(int eventID, int contextID) { return false; };
};

int main()
{
  CallbackHandler callbackObject;

  UniArgument<void(CallbackHandler*, int)> argument1;       // (1)
  UniArgument<bool(CallbackHandler*, int, int)> argument2;  // (2)

  argument1 = &CallbackHandler::handler1;  // (3)
  argument2 = &CallbackHandler::handler2;  // (4)
  argument1(&callbackObject, 100);         // (5)
  argument2(&callbackObject, 0, 1);        // (6)
}
```

В строках 1 и 2 объявлены универсальные аргументы для вызова соответствующих методов класса. Как видим, в сигнатуре функции первый параметр является типом класса, для которого будут вызываться соответствующие методы. В строках 3 и 4 производится настройка методов, в строках 5 и 6 – вызовы методов для экземпляра соответствующего класса.

Итак, универсальный аргумент практически готов. Нам осталось реализовать оператор копирования, оператор присваивания и некоторые другие операции. Но мы этим заниматься не будем: разработчики стандартной библиотеки уже обо всем позаботились, поэтому темой следующей главы будет обзор инструментов STL для организации обратных вызовов[24].

[24] «Зачем же мы тогда разрабатывали универсальный аргумент, если в STL все уже давно реализовано?» – может воскликнуть рассерженный читатель. Ну, во-первых, грамотный разработчик отличается от обычного разработчика тем, что он не только знает, как применять те или иные инструменты, но еще и понимает, как они работают. И, во-вторых, рассмотренные методы используются не только в проектировании обратных вызовов, они могут использоваться при решении самых различных задач.

4.6. Использование стандартной библиотеки

4.6.1. Организация вызовов

В стандартной библиотеке имеется полиморфный класс – оболочка **std::function**, предназначенная для организации вызовов различных типов. Этот класс идеально подходит на роль универсального аргумента. Кроме рассмотренных техник стирания типа и настройки сигнатуры, в нем реализовано множество других вещей: конструктор копирования, оператор присваивания, поддержка указателей на методы класса, проверка настройки аргумента, локальный буфер для хранения аргумента и многое другое. Мы не будем рассматривать реализацию **std::function**, потому что, во-первых, она достаточно сложная, а, во-вторых, может изменяться в зависимости от версии и платформы. При желании читатель сможет сделать это самостоятельно, проанализировав исходный код, мы же сосредоточимся на практическом использовании класса-оболочки.

Насколько сложна реализация **std::function**, настолько же просто ее использование. По аналогии с универсальным аргументом, рассмотренном в предыдущей главе, достаточно объявить экземпляр класса с нужной сигнатурой, после чего ему можно назначать различные объекты вызовов (Листинг 51).

Листинг 51. Использование std::function

```
void External(int eventID) {};

int main()
{
  struct Call
  {
    void operator() (int eventID) {};
  } objectCall;

  std::function<void(int)> fnt;

  fnt = External;
  fnt = objectCall;
  fnt = [](int evetID) {};

  fnt(0);
}
```

Полезной особенностью **std::function** является проверка настройки объекта вызова. Если объект не настроен, т. е. не было ни одного присваивания, то при попытке вызова будет выброшено исключение. Проверить, настроен ли объект, можно с помощью перегруженного оператора **bool**, пример приведен в Листинг 52.

Листинг 52. Проверка настройки аргумента

```
int main()
{
  std::function<void(int)> fnt;

  fnt(0); //Error: argument is not set. Exception will be thrown

  fnt = [](int) {};
  fnt(0); //Ok, argument is set

  //Check if the argument is set
  if (fnt)
  {
    fnt(0);
  }
}
```

4.6.2. Инициатор с универсальным аргументом

Для реализации инициатора с универсальным аргументом необходимо для хранения аргумента объявить соответствующую класс-оболочку **std::function** (Листинг 53).

Листинг 53. Инициатор с оболочкой std::function

```
class Initiator  // (1)
{
public:
  template<typename CallbackArgument>
  void setup(const CallbackArgument& argument)  // (2)
  {
    callbackHandler = argument;
  }

  void run()
  {
  int eventID = 0;
  //Some actions
  callbackHandler(eventID);
  }

private:
  std::function<void(int)> callbackHandler;  // (3)
};
```

Если сравнить реализацию инициатора с фиксированным типом аргумента (Листинг 37 п. 4.4.1) с приведенной, то можно заметить следующие отличия. В первом случае инициатор является шаблоном, здесь он объявляется обычным способом. Далее, хранимый аргумент 3 не

является переменной типа, задаваемого параметром шаблона, он объявлен как универсальный аргумент **std::function**. Метод настройки 2 объявлен как шаблон, параметром которого является тип назначаемого аргумента.

Описанный инициатор не работает с указателями на функцию и на метод класса: в первом случае необходимо передавать контекст, во втором случае необходимо передавать указатель на экземпляр класса и использовать другой синтаксис для вызова. Как уже рассматривалось в п. 4.2.2, в этих случаях необходимо преобразование вызовов. Однако, поскольку в универсальном аргументе сигнатура может настраиваться, в объекты преобразования также нужно ввести поддержку настройки сигнатуры.

4.6.3. Преобразование с настройкой сигнатуры

В п. 4.2.2 реализованы объекты преобразования, которые работали с фиксированной сигнатурой. Используя технику, описанную в Листинг 47 п. 4.5.2, модифицируем их таким образом, чтобы сигнатуру можно было настроить. Для этого в параметрах шаблона вместо задания типов указателей на функцию будем задавать параметры, определяющие сигнатуру, а типы указателей будем выводить из этих параметров.

Рассмотрим вначале указатели на функцию (Листинг 54).

Листинг 54. Преобразование вызовов с настройкой сигнатуры для указателей на функцию

```
template<typename unused>  // (1)
class CallbackConverter;

template<typename Context, typename Return, typename … ArgumentList
class CallbackConverter<Return(Context, ArgumentList…)>
{
public:

  using Function = Return(*)(Context, ArgumentList…);  // (4)

  CallbackConverter(Function argFunction = nullptr, Context argCont
  {
    ptrFunction = argFunction; context = argContext;
  }

 Return operator() (ArgumentList… arguments)              // (6)
  {
   ptrFunction(context, arguments…);                      // (7)
  }
private:
  Function ptrFunction;  // (8)
  Context context;       // (9)
};
```

В строке 1 вводится общая специализация шаблона. В строке 2 объявляется специализация для указателей на функцию, в которой задается тип передаваемого контекста и параметры сигнатуры. В строке 4 выводится тип указателя. В конструкторе 5 осуществляется настройка

указателей. В перегруженном операторе 6 осуществляется вызов 7, в который передаются соответствующие аргументы.

Аналогично выполняется специализация для вызова методов класса (Листинг 55).

Листинг 55. Преобразование вызовов с настройкой сигнатуры для указателей на метод класса.

```
    template<typename  ClassType,  typename  Return,  typename…
ArgumentList>  // (1)
    class                  CallbackConverter<Return(ClassType::*)
(ArgumentList…)>          // (2)
    {
    public:

            using    MemberPointer    =    Return(ClassType::*)
(ArgumentList…);  // (3)

      CallbackConverter(MemberPointer methodPointer = nullptr, ClassTyp
      {
          ptrClass = classPointer; ptrMethod = methodPointer;
      }

      Return operator()(ArgumentList… arguments)  // (5)
      {
        (ptrClass->*ptrMethod)(arguments…);       // (6)
      }
    private:
      ClassType* ptrClass;                          // (7)
      MemberPointer ptrMethod;                      // (8)
    };
```

Реализация практически повторяет предыдущую, за исключением того, что в объявлениях типов сигнатуры добавляется класс (строки 2 и 3), а перегруженный оператор вызывает метод класса (строка 6).

4.6.4. Исполнитель

Реализация исполнителя для инициатора с универсальным аргументом (см. Листинг 53 п. 4.6.2) приведена в Листинг 56, здесь используется **CallbackConverter** из Листинг 54 п. 4.6.3.

Листинг 56. Исполнитель для инициатора с оболочкой std::function

```
    class Executor
    {
    public:
      static void staticCallbackHandler(Executor* executor, int eventID
      void callbackHandler(int eventID) {}
```

```
    void operator() (int eventID) {}
};

void ExternalHandler(void* somePointer, int eventID) {}

int main()
{
    int capturedValue = 0;
    Initiator initiator;
    Executor executor;

    // Pointer to the external function
    initiator.setup(CallbackConverter<void(void*, int)>(ExternalHandl

    // Pointer to the static method
    initiator.setup(CallbackConverter<void(Executor*, int)>(Executor:

    // Pointer to the class member method
            initiator.setup(CallbackConverter<void(Executor::*)
(int)>(&Executor::callbackHandler, &executor));

    // Functional object
    initiator.setup(executor);

    // Lambda-expression
    initiator.setup([capturedValue](int eventID) {});
}
```

Если сравнить приведенную реализацию исполнителя для шаблона-инициатора с фиксированным типом аргумента (Листинг 43 и Листинг 44 п. 4.4.3) с приведенной, то можно заметить следующее. В первом случае для каждого типа аргумента приходится объявлять отдельный инициатор, инстанциируя его соответствующим типом. Здесь инициатор объявляется один раз, после чего тип аргумента вызова настраивается в процессе выполнения программы. В результате упрощается разработка, улучшается гибкость и прозрачность кода.

4.6.5. Инициатор для методов класса

До сих пор для вызова методов класса мы использовали преобразование вызовов. Однако, поскольку **std::function** непосредственно поддерживает вызов методов, появляется возможность реализовать специализированный инициатор для указанного случая. За основу возьмем инициатор из п. 4.6.2 и модифицируем его.

Как мы видели в реализации универсального аргумента (п. 4.5.3), для вызова метода класса первым параметром должен передаваться указатель на экземпляр класса. Поэтому, в инициатор необходимо добавить переменную для хранения этого указателя. Но поскольку тип класса заранее неизвестен, его следует задавать как параметр, т. е. инициатор должен быть объявлен в виде шаблона. Далее необходимо добавить метод для настройки указателя и, соответственно, при задании сигнатуры и выполнении вызова передавать дополнительный аргумент – указатель на экземпляр класса. Реализация приведена в Листинг 57.

Листинг 57. Инициатор с оболочкой std::function для вызова методов класса

```
template<typename ClassName>  // (1)
class InitiatorForClass
{
public:
  template<typename CallbackArgument>
  void setup(const CallbackArgument argument)  // (2)
  {
    callbackHandler = argument;
  }

  void setupInstance (ClassName* classObject)  // (3)
  {
    ptrClass = classObject;
  }

  void run()  // (4)
  {
      int eventID = 0;
      //Some actions
      callbackHandler(ptrClass, eventID);  // (5)
  }

private:
  std::function<void(ClassName*, int)> callbackHandler;  // (6)
  ClassName* ptrClass = nullptr;                         // (7)
};
```

В строке 1 объявлен шаблон класса. В строке 2 объявлен метод для настройки аргумента, в качестве которого выступает указатель на метод-член. В строке 3 объявлен метод для настройки экземпляра класса. Метод запуска 4 такой же, как и в исходном, за исключением того, что при вызове в аргумент дополнительно передается указатель на класс (строка 5). В строке 6 инстанциируется аргумент для вызова метода класса, в сигнатуре первым параметром выступает указатель на класс, задаваемый параметром шаблона-инициатора. В строке 7 объявлена переменная для хранения указателя на экземпляр класса.

Итак, модифицировав инициатор из Листинг 53 п. 4.6.2, мы реализовали отдельный инициатор для вызова методов-членов. Используя частичную специализацию шаблона, можно сделать так, чтобы оба инициатора объявлялись одинаковым способом (Листинг 58).

Листинг 58. Использование специализации шаблона-инициатора для вызова методов класса

```
template<typename… unused>  // (1)
class Initiator
{
```

```
  //… Implementation for origin initiator
};

template<typename ClassName>  // (2)
class Initiator<ClassName>
{
  //… Implementation for class method call initiator
};
```

В строке 1 объявлен исходный класс, но теперь он является шаблоном с пакетом параметров. Пакет параметров здесь не используется, он нужен только для дальнейшей специализации.

В строке 2 объявлен шаблон для вызова методов-членов. Поскольку его имя совпадает с именем предыдущего, компилятор будет считать, что здесь определяется не новый класс, а специализация объявленного ранее. В объявлении указан параметр, предполагается, что в этом качестве будет использоваться имя класса. Теперь, если при инстанциировании шаблона будет задаваться параметр, будет выбрана специализация для вызова методов-членов. При отсутствии параметров будет выбран исходный шаблон.

Использование двух типов инициатора (исходного и специализированного) для вызова методов класса приведено в Листинг 59, здесь используется преобразование вызовов из Листинг 54 п. 4.6.3.

Листинг 59. Использование инициатора с оболочкой std::function для вызова методов класса

```
class Executor
{
public:
  static void staticCallbackHandler(Executor* executor, int eventID
  void callbackHandler(int eventID) {}
  void operator() (int eventID) {}
};

int main()
{
  Executor executor;

  Initiator initiator;  // (1)
          initiator.setup(CallbackConverter<void(Executor::*)
(int)>(&Executor::callbackHandler, &executor));  // (2)
  initiator.run();

  Initiator<Executor> initiatorForClass;                // (3)
  initiatorForClass.setup(&Executor::callbackHandler);  // (4)
  initiatorForClass.setupInstance(&executor);           // (5)
  initiatorForClass.run();
}
```

В строке 1 объявлен исходный инициатор. В параметры шаблона мы не передаем никаких аргументов, т. е. шаблон инстанциируется подобно обычному классу. В строке 2 происходит настройка инициатора, в качестве аргумента передается объект для преобразования вызовов.

В строке 3 объявлен специализированный инициатор для вызова методов класса, он инстанциируется типом **Executor**. В строке 4 настраивается указатель на метод класса, в строке 5 настраивается указатель на экземпляр класса.

Какой инициатор лучше использовать для методов класса, исходный с преобразованием или модифицированный с непосредственным вызовом? Трудно однозначно ответить на этот вопрос. С одной стороны, использование специализированного класса противоречит идее обобщенного кода – в специализированном классе мы вынуждены повторять всю реализацию, даже в тех частях, где она совпадает с исходной. С другой стороны, упрощается работа с настройкой инициатора – нам не нужно использовать класс для преобразования, можно по отдельности изменять указатель на метод и указатель на экземпляр. В общем, выбор остается на усмотрение разработчика.

4.6.6. Перенаправление вызовов

Представьте следующую ситуацию: инициатор вызывает функцию с одной сигнатурой, а в клиенте реализован обработчик с другой сигнатурой. Например, в исполнителе реализована функция обработки нажатия кнопки, которая на вход принимает два параметра – идентификатор кнопки и текущее поле редактирования. В то же время инициатор вызывает функцию, передавая ей только один аргумент – идентификатор текущей нажатой кнопки, и он ничего не знает об остальных элементах управления. Можно ли сделать так, чтобы инициатор вызывал одну функцию, но при этом бы вызывалась другая функция, другими словами, происходило перенаправление вызова? В стандартной библиотеке для этого существуют специальные объекты связывания **std::bind**, которые при необходимости могут сохраняться в **std::function** подобно обычным функциональным объектам.

Графически использование связывания продемонстрировано на Рис. 18. Пусть инициатор вызывает функцию 1, которая на вход принимает аргумент 1. Исполнитель реализует обратный вызов с помощью функции 2, которая принимает на вход два аргумента. Вместо функции 1 инициатору назначается объект связывания, который имеет перегруженный оператор вызова функции с сигнатурой 1. Указанный объект хранит дополнительный параметр, значение которому присваивается во время инициализации. Перегруженный оператор, в свою очередь, вызывает функцию 2, первому аргументу передает сохраненный параметр, а второму аргументу передает значение аргумента из функции 1. Таким образом, осуществляется перенаправление вызова из функции 1 в функцию 2.

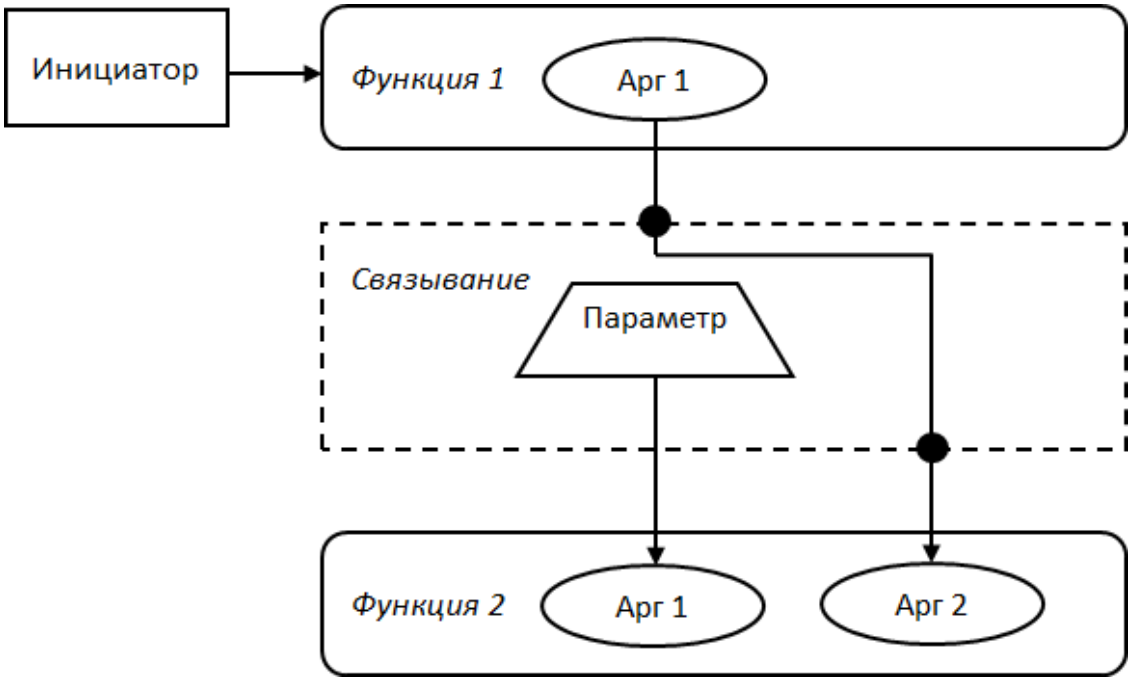

Рис. 18. Перенаправление вызовов

Использование перенаправления вызовов представлено в Листинг 60.

Листинг 60. Перенаправление вызовов

```
#include <functional>

void NativeHandler(int eventID)  // (1)
{
  //here eventID is 10
}

void AnotherHandler(int contextID, int eventID)  // (2)
{
  //here eventID is 1, contextID is 10;
}

int main()
{
  int eventID = 1; int contextID = 10;

  std::function<void(int)> fnt;  // (3)
  fnt = NativeHandler;           // (4)
 fnt(eventID);             // (5) NativeHandler will be called

  fnt = std::bind(AnotherHandler, contextID, std::placeholders::_1);
  fnt(eventID);  // (7) AnotherHandler will be called
}
```

В строке 1 объявлен исходный обработчик, в строке 2 – обработчик, в который будет перенаправляться вызов. В строке 3 объявлен универсальный аргумент с исходной сигнатурой. В строке 4 аргументу назначена функция, которая будет вызвана при выполнении вызова 5.

В строке 6 вызывается функция **bind**, которая из переданных аргументов формирует объект связывания. На вход **std::bind** передается имя новой функции-обработчика и аргументы, которые будут передаваться в эту функцию. Первому аргументу здесь будет назначено значение **contextID**, а второму аргументу будет назначено значение 1-го по порядку аргумента из исходной функции. Здесь конструкция **std::placeholders** определяет номер аргумента в исходной функции, который будет подставлен в качестве аргумента в перенаправляемую функцию.

Сформированный объект связывания сохраняется в универсальном аргументе. Если мы теперь выполним вызов (строка 7), то будет вызвана функция, назначенная этому объекту, и этой функции будут переданы соответствующие аргументы.

Аналогичным образом может быть объявлено перенаправление вызовов для методов-членов класса, но здесь должно соблюдаться следующее правило: первому аргументу новой функции должен быть назначен первый аргумент исходной функции, потому что он определяет экземпляр класса, для которого вызывается метод. Пример приведен в Листинг 61.

Листинг 61. Перенаправление вызовов для методов-членов класса

```
#include <functional>

class CallbackHandler
{
public:
  void NativeHandler(int eventID)
  {
    //eventID = 1;
  }
  void AnotherHandler(int contextID, int eventID)
  {
    //eventID = 1, contextID = 10;
  }
};

int main()
{
  using namespace std::placeholders; // (1)

  int eventID = 1; int contextID = 10;
  CallbackHandler handler;

  std::function<void(CallbackHandler*, int)> fnt;
  fnt = &CallbackHandler::NativeHandler;
  fnt(&handler, eventID); // NativeHandler will be called

  fnt = std::bind(&CallbackHandler::AnotherHandler, _1, contextID,
  fnt(&handler, eventID); // AnotherHandler will be called
}
```

Здесь в строке 1 мы использовали using namespace, что сокращает объявление позиций аргументов: как видно из строки 2, мы сразу пишем позицию без использования std::placeholders, что значительно компактнее и проще для восприятия. Здесь в исходной функции присутствует неявный параметр с номером 1, который определяет экземпляр класса. Этот параметр назначается первому (неявному) параметру новой функции, а второй параметр исходной функции **eventID** назначается последнему параметру новой функции.

В общем случае могут быть 4 варианта перенаправления вызовов:
- из функции в функцию (пример в Листинг 60);
- из функции в метод класса;
- из метода класса в другой метод этого же класса (пример в Листинг 61);
- из метода класса в метод другого класса;
- из метода класса в функцию.

Реализация указанных вариантов, по сути, одинакова, отличаются только объявления связывания. Сведем эти объявления в таблицу (Табл. 13).

Табл. 13. Связывания для различных вариантов перенаправления вызовов.

Аргумент вывзова	Обработчик	Связывание
Функция – Функция		
std::function<void(int)> fnt; fnt (eventID);	 ExternalHandler (int contextID, int eventID);	std::bind (ExternalHandler, contextID, _1);
Функция – метод		
std::function<void(int)> fnt; fnt (eventID);	class CallbackHandler { void AnotherHandler(int contextID, int eventID) ; } handler;	std::bind(&CallbackHandler::AnotherHandler, handler, contextID, _1)
Метод – метод		

class CallbackHandler
{
void NativeHandler(
int contextID);
} handler;
 	
std::function<void(CallbackHandler*, int)>
fnt;
fnt(&handler, eventID);
	class CallbackHandler
	{
	void AnotherHandler(
	
	int contextID,
	int eventID) ;
	} handler;
		std::bind(
		
		&CallbackHandler::AnotherHandler,
		_1,
		contextID,
		_2)

Метод – метод другого класса		

class CallbackHandler
{
void NativeHandler(
int contextID);
} handler;
 	
std::function<void(CallbackHandler*, int)>
fnt;
fnt(&handler, eventID);
	class CallbackHandler2
	{
	void AnotherHandler(
	
	int contextID,
	int eventID) ;
	} handler2;
		std::bind(
		
		&CallbackHandler2::AnotherHandler,
		handler2,
		contextID,
		_2);

Метод – функция		

class CallbackHandler
{
void NativeHandler(
int contextID);
} handler;
 	
std::function<void(CallbackHandler*, int)>;
fnt(&handler, eventID);
	
	void ExternalHandler(
	int contextID,
	int eventID);
		std::bind(
		ExternalHandler,
		contextID,
		_2)

Теперь перенаправление вызовов в исполнителе не представляет сложности: при настройке вместо объекта вызова нужно всего лишь подставить необходимое связывание. Пример для варианта «функция – функция» приведен в Листинг 62, здесь используется инициатор из Листинг 53.

Листинг 62. Перенаправление вызовов в исполнителе

```
void NativeHandler(int eventID)
{
  //here eventID is 10
}

void AnotherHandler(int contextID, int eventID)
{
```

```
  //here eventID is 10, contextID is 1;
}

int main()
{
  int eventID = 10; int contextID = 1;
  Initiator initiator;            // (1)

  initiator.setup(NativeHandler); // (2)
  initiator.setup(std::bind(AnotherHandler, contextID, std::placeho

  initiator.run(); // (4)
}
```

В строке 1 объявлен инициатор. В строке 2 происходит настройка инициатора с передачей ему указателя на функцию с «родной» сигнатурой, т. е. сигнатурой, для которой инициатор осуществляет вызов. Если бы мы после этого запустили инициатор путем вызова метода **run**, то инициатор вызывал бы функцию **NativeCallbackHandler**. В строке 3 вместо функции с «родной» сигнатурой мы подставляем объект связывания, который будет перенаправлять вызов в другую функцию. В строке 4 запускаем инициатор, в котором после вызова функции объекта связывания будет осуществлен вызов **AnotherCallbackHandler** с соответствующими параметрами. Аналогичным образом, подставляя нужные связывания из Табл. 13, осуществляется перенаправление вызовов для других вариантов.

Итак, использование объектов связывания предлагает универсальный способ преобразования вызовов: вместо объектов преобразования (п. 4.2.2, 4.6.3) в универсальный аргумент подставляется объект связывания, сгенерированный соответствующим вызовом **std::bind**.

4.6.7. Универсальный аргумент и производительность

Может показаться, что организация обратных вызовов с использованием **std::function** в качестве универсального аргумента является наилучшим решением, предлагающим простоту реализации в сочетании с максимальной гибкостью. В большинстве случаев это действительно так, однако **std::function** обладает недостатком, который может свести на нет все остальные достоинства: большие временные затраты для осуществления вызова по сравнению с другими способами реализации. Причины этого следующие:

1) при вызове происходит проверка, настроен ли аргумент;

2) вызов происходит через промежуточный объект с виртуальной функцией (см. 4.5.1) – расходуется дополнительное время для вызова этой функции;

3) поскольку промежуточный объект создается динамически, его адрес может изменяться, что требует загрузки адреса перед вызовом;

4) на этапе компиляции тип аргумента неизвестен, поэтому код обработки не может быть встроен в точку вызова.

Первые три причины вносят незначительный вклад в общее время, затрачиваемое на выполнение вызова, а вот четвертая может привести к резкому падению производительности. Мы уже рассматривали подобную проблему при анализе функциональных объектов (п. 2.4.6): при малом объеме кода обработчика время, затраченное на вызов функции, может превысить время выполнения тела функции.

Проведем эксперимент. Напишем программу, в которой циклически будут осуществляться вызовы различных типов для кода небольшого размера[25]. Поскольку код обработчика один и тот же, общее время, затраченное на выполнение вызова, будет прямо пропорционально времени, затраченному на организацию вызова. Запустим программу и выполним профилирование[26]. Результаты профилирования представлены в Табл. 14, графически они изображены на Рис. 19[27].

Табл. 14. Время, затраченное на выполнение вызовов различных типов для кода небольшого размера, млс.

Аргумент	Прямой вызов	Универсальный аргумент	Увеличение
Указатель на функцию	525.58	698.11	1.33
Указатель на метод	624.29	1 067.51	1.70
Функциональный объект	8.79	712.14	81.02
Лямбда-выражение	8.64	700.62	81.09

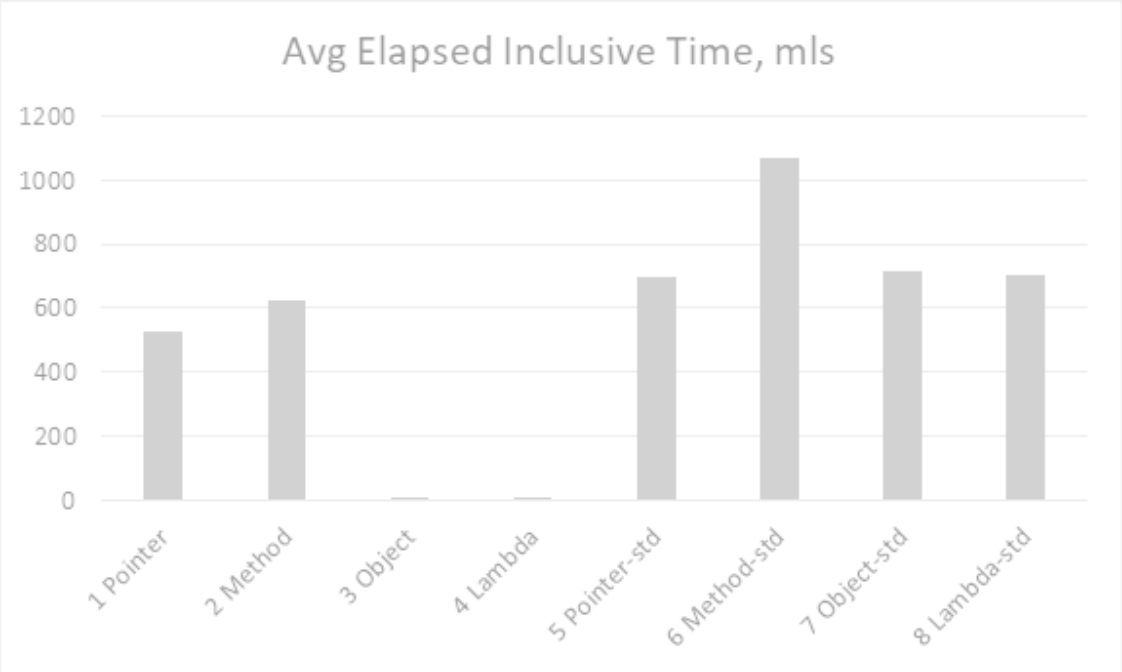

Рис. 19. Гистограмма результатов профилирования вызовов различных типов для кода небольшого размера

Проанализируем вначале результаты при организации вызовов напрямую, без использования универсального аргумента. Быстродействие для указателя на функцию и указателя

[25] Исходный код можно посмотреть здесь: **https://github.com/Tkachenko-vitaliy/Callbacks/tree/master/Profiling**.

[26] Указатели на статические методы классов в эксперименте не участвовали, потому что с точки зрения организации вызова они идентичны указателям на обычные функции. Профилирование выполнялось в среде Microsoft Visual Stidio.

[27] Если читатель попробует повторить эксперимент, то числовые значения, скорее всего, будут другими. Во-первых, они сильно зависят от используемого компилятора, точности профилировщика, производительности процессора. Во-вторых, в силу особенностей современных программно-аппаратных архитектур даже при запуске на одной и той же платформе результаты профилирования не будут повторяться, они плавают в некотором диапазоне значений. Заинтересованному читателю можно порекомендовать книгу «Крис Касперски. Техника оптимизации программ. Эффективное использование памяти», где подробно рассматривается этот вопрос.Тем не менее, указанные замечания не могут считаться основанием для сомнений в достоверности эксперимента, поскольку относительные значения всегда будут приблизительно одинаковыми независимо от используемых программно-аппаратных средств.

на метод различается незначительно, а вот при использовании функциональных объектов и лямбда-выражений оно вырастает на порядки[28], потому что код встраивается в точку вызова.

Посмотрим теперь результаты при использовании универсального аргумента. Если сравнить с вызовами напрямую, время выполнения ожидаемо увеличивается. Однако если для указателя на функцию и указателя на метод увеличение незначительно, то для функционального объекта и лямбда-выражения оно увеличивается настолько, что практически исчезает отличие от других способов. Теперь код обработчика не встраивается в точку вызова, и расходы на вызов функции во много раз превышают расходы на выполнение тела функции.

Модифицируем теперь код обработчика таким образом, чтобы оптимизатор не мог встроить его в точку вызова. Числовые значения замеров представлены в Табл. 15, графически они изображены на Рис. 20. Теперь картина получается иная: прямое использование функциональных объектов и лямбда-выражений не дают заметного выигрыша в производительности, а использование универсального аргумента увеличивает время выполнения незначительно.

Табл. 15. Время, затраченное на выполнение вызовов различных типов для кода большого размера, млс.

Аргумент	Прямой вызов	Универсальный аргумент	Увеличение
Указатель на функцию	1 769.64	2 435.98	1.37
Указатель на метод	1 794.78	2 134.78	1.19
Функциональный объект	1 084.23	1 822.01	1.68
Лямбда-выражение	848.00	1 744.65	2.06

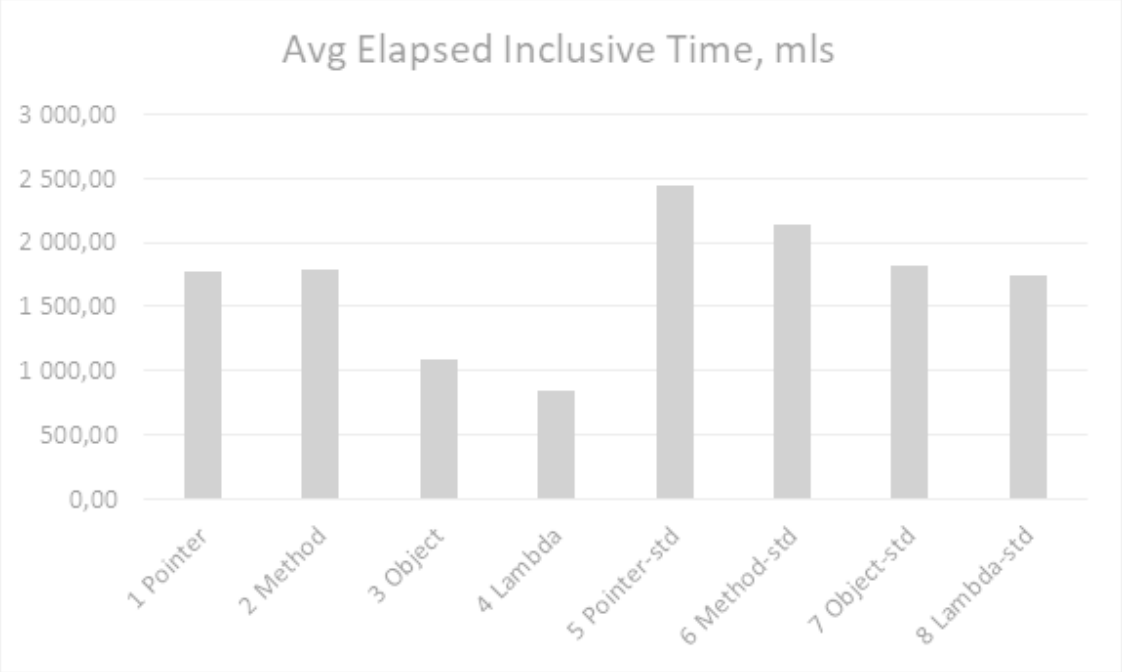

Рис. 20. Гистограмма результатов профилирования вызовов различных типов для кода большого размера

Какой код будет встраиваться в точку вызова, а какой нет? Однозначного ответа на этот вопрос дать невозможно. Алгоритмы работы оптимизатора не документируются и принимают во внимание множество факторов: количество команд в коде; количество точек вызова; нали-

[28] Снижается время выполнения – увеличивается быстродействие, т. е. эти показатели обратно пропорциональны.

чие рекурсивных вызовов; оценка степени увеличения результирующего кода после встраивания и т. п. Самый надежный способ – посмотреть дизассемблированный код, где однозначно видно, встроен ли код обработчика в точку вызова.

Исходя из изложенного, можно сделать следующий вывод:

> В системах, где предполагается интенсивное использование обратных вызовов и быстродействие является критически важным, использование универсального аргумента не является оптимальным выбором.

4.7. Проблемы, порождаемые шаблонами

4.7.1. Недостатки шаблонов

Как можно заметить из рассмотренных примеров, шаблоны являются мощным и эффективным инструментом реализации обобщенного кода. Но, как известно, не бывает ничего идеального, поэтому, конечно же, у них имеются недостатки.

Сложность разработки. При проектировании шаблонного кода операции зачастую задаются в декларативном виде, что приближает их к функциональному стилю. Использование пакетов параметров требует изощренных техник, весьма непохожих на классические приемы программирования.

Сложность понимания. Код, написанный с помощью шаблонов, гораздо труднее анализировать, чем обычный. В этом можно убедиться, просмотрев, к примеру, исходный код стандартной библиотеки STL.

Недружественность компилятора. Сообщения об ошибках, генерируемые при компиляции шаблонов, зачастую сложны и непонятны. Когда ошибка показывается где-то в недрах шаблонного кода, очень трудно бывает догадаться, возникает ли проблема из-за некорректной реализации этого кода либо из-за того, что структура данных, подставляемая в шаблон, не реализует предполагаемый интерфейс (например, требуется перегрузка некоторых операторов).

Тщательное тестирование. Шаблоны подчиняются концепции «компиляция по требованию», т. е. компилируются только те функции и методы, которые используются в коде. Поэтому, чтобы убедиться в отсутствии синтаксических и семантических ошибок, следует покрывать вызовами все функции и методы, объявленные в шаблоне. Причем желательно это делать на некотором наборе предполагаемых типов данных.

Большое время компиляции. Во-первых, компилятор осуществляет генерацию кода при каждом инстанциировании шаблона конкретным типом. Во-вторых, шаблоны для одних и тех же типов, инстанциируемые в разных участках программы, будут компилироваться заново. И, в-третьих, много времени тратится на компиляцию включаемых файлов: например, при каждом включении заголовочных файлов стандартной библиотеки все внутренние реализации шаблонов в этих файлах должны быть скомпилированы.

Склонность к разрастанию программного кода. Для каждого используемого типа будет сгенерирован отдельный код. Представим, к примеру, что мы используем шаблонную функцию с входным аргументом – числом, тип которого задается параметром шаблона. Если мы будем вызывать эту функцию с аргументами различных типов, допустим, **char**, **short**, **int**, **long**, для каждого типа будет сгенерирована отдельная функция, несмотря на то что используемые типы эквивалентны и можно обойтись одним-единственным типом **long**. Аналогичная ситуация возникает при специализации шаблонов: даже если мы делаем частичную специализацию с целью изменить поведение одного-единственного метода, нам придется повторить весь код, используемый в общей специализации.

4.7.2. Ограничения шаблонов

В общем-то, рассмотренные недостатки не так уж значительны, и преимуществ у шаблонов значительно больше. Тем не менее, они имеют фундаментальное ограничение, вытекающие из их внутренней природы: **шаблоны не создают предварительно откомпилированного кода**. По большому счету шаблон представляет собой не сам код, а правила для генерации кода. Пока шаблон не инстанциирован, его код отсутствует; после инстанциирования послед-

ний генерируется только для тех методов и функций, которые были вызваны. Из указанного ограничения вытекают следующие выводы.

Интерфейс шаблона не может быть отделен от реализации. И объявление шаблона, и его реализация должны находиться в одной области видимости (модель включения). Таким образом, при изменениях в реализации шаблона все компоненты, которые его используют, должны быть перекомпилированы.

Шаблоны не могут поставляться в виде статических или динамических библиотек , они должны поставляться только в виде исходного кода. Никакие сторонние приложения (за исключением компиляторов C++, разумеется) не могут использовать функциональность, реализованную на базе шаблонов.

По вышеуказанным причинам,[29]

С помощью шаблонов невозможно реализовать интерфейсы API.

[29] Здесь необходимо уточнить: речь идет только об интерфейсе API, т. е. его видимой части. В реализации API шаблоны использовать можно и нужно.

4.8. Итоги

Шаблоны обеспечивают параметрический полиморфизм, что позволяет писать обобщенный код, реализующий заданную функциональность без привязки к типам данных.

Инициатор для синхронных вызовов реализуется с помощью шаблонов функций, асинхронных – с помощью шаблонов классов.

В реализации шаблона инициатора тип объекта вызова задается параметром. Поскольку разные типы объектов требуют различное число параметров и используют неодинаковый синтаксис, для сохранения единой реализации используется преобразование вызовов.

Функциям, реализующим алгоритмы, зачастую требуются различные операции над данными. Поскольку в обобщенном коде типы данных заранее не известны, для реализации операций используются предикаты.

В асинхронных вызовах для каждого типа аргумента приходится инстанциировать соответствующий инициатор. Использование универсального аргумента позволяет реализовать единый класс для любых типов аргументов, однако в некоторых случаях это может привести к падению производительности.

В стандартной библиотеке STL имеются мощные средства для организации вызовов, реализующие универсальный аргумент, вызов методов класса, перенаправление вызовов.

Шаблонам присущи недостатки, большинство из которых незначительны и не перевешивают их достоинств. Однако шаблоны не предполагают предварительно откомпилированного кода, и по этой причине не могут использоваться в интерфейсах API.

5. Распределение вызовов

5.1. Постановка задачи

Под распределением вызовов понимается техника, в которой при вызове единственной функции осуществляется выполнение множества вызовов через соответствующие аргументы.

Графически задача распределения вызовов показана на Рис. 21. Компонент, осуществляющий вызов, называется источником; аргументы вызова называются получателями; компонент, осуществляющий распределение вызовов, называется распределитель; код, запускающий вызовы, называется распределяющая функция. При необходимости дополнительно в вызов могут передаваться какие-либо данные.

Распределитель может быть реализован в виде функции либо класса. Если распределитель реализован в виде функции, то он сам представляет собой распределяющую функцию. Если распределитель реализован в виде класса, то распределяющая функция представляет собой метод класса либо перегруженный оператор.

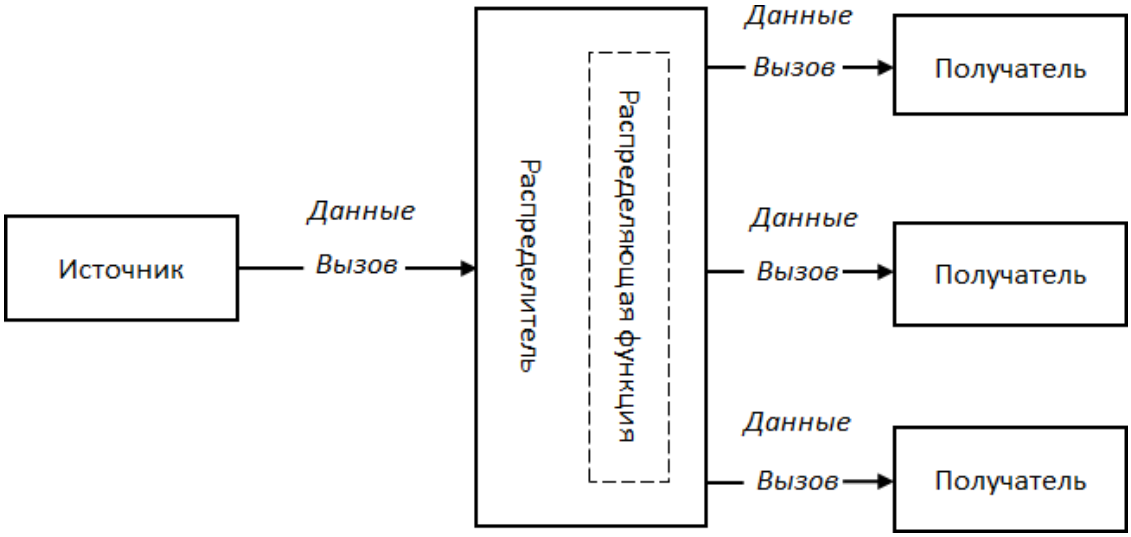

Рис. 21. Распределение вызовов

Как видим, постановка задачи звучит достаточно просто. Зачем же тогда ей посвящен отдельный раздел? Во-первых, распределение вызовов имеет важное прикладное значение: оно используется в самых различных приложениях, таких, как обработка команд, оповещение о событиях, синхронизация операций и др. Во-вторых, задача распределения вызовов совсем не такая простая, как это может показаться из формального описания. Для ее решения используются изощренные техники, призванные обеспечивать максимальную эффективность для самых различных требований.

Итак, рассмотрим, как реализуется распределение вызовов.

5.2. Статический набор получателей

5.2.1. Распределение в статическом наборе

Если типы и количество получателей известны на этапе компиляции и не планируется их изменение в процессе выполнения программы, то мы имеем статический набор получателей. В этом случае распределитель можно реализовать в виде шаблонной функции, которая в качестве входных аргументов будет принимать объекты вызова. Но поскольку типы объектов и их количество могут быть различными, логично в качестве входного параметра функции использовать пакет, задаваемый шаблоном.

Итак, нам необходимо выполнить вызов для каждого объекта, входящего в пакет. Для решения этой задачи используется техника рекурсивного развертывания пакета, суть которой заключается в следующем.

Объявляется функция, первым параметром которой выступает объект вызова, а вторым – пакет. Когда на вход данной функции поступает пакет, первый объект из него извлекается, происходит вызов этого объекта, а затем функция рекурсивно вызывается вновь с пакетом, содержащим еще не извлеченные объекты. Когда в результате рекурсивных вызовов все объекты будут извлечены, будет вызвана функция, на вход которой будет передан пустой пакет. Данная функция завершает рекурсивное выполнение.

Реализация описанной техники приведена в Листинг 63.

Листинг 63. Распределяющая функция для статического набора получателей

```
void Call()  // (1)
{
}

template < typename First, typename…Others>
void Call(First& first, Others…rest)  // (2)
{
  first();            // (3)
  Call(rest…);       // (4)
}

template <typename … CallObjects>
void Distribute(CallObjects… objects)  // (5)
{
  Call(objects…);  // (6)
}
```

Графически развертывание пакета параметров для трех аргументов изображено на Рис. 22. Процесс начинается с вызова распределяющей функции, которая объявлена в строке 5. Здесь используется пакет параметров **objects**, который содержит объекты вызова. Внутри этой функции, в строке 6, происходит первый вызов рекурсивной функции, которой на вход передаются соответствующий аргумент в виде пакета.

Рекурсивная функция **Call** объявлена в строке 2. Эта функция принимает два аргумента: первый параметр из пакета **first** и пакет остальных параметров **rest**. При первом вызове пакет

параметров из **Distribute** передается в эту функцию, и там происходит его распаковка: первый параметр извлекается и помещается в **first**, оставшаяся часть пакета записывается в **rest**. В строке 3 производится вызов, а пакет с оставшимися параметрами передается в рекурсивный вызов **Call** (строка 4).

Итак, на каждом шаге рекурсивного вызова из пакета извлекается очередной параметр, а размер исходного пакета уменьшается. Таким образом, в итоге все параметры будут извлечены, и пакет станет пустым. Эта ситуация обрабатывается путем объявления функции с пустым пакетом параметров, т. е. функции, которая на вход не принимает ни одного аргумента (строка 1). Тело этой функции пустое, в ней происходит возврат управления, и по цепочке рекурсивных вызовов управление возвращается в исходную точку в строке 6.

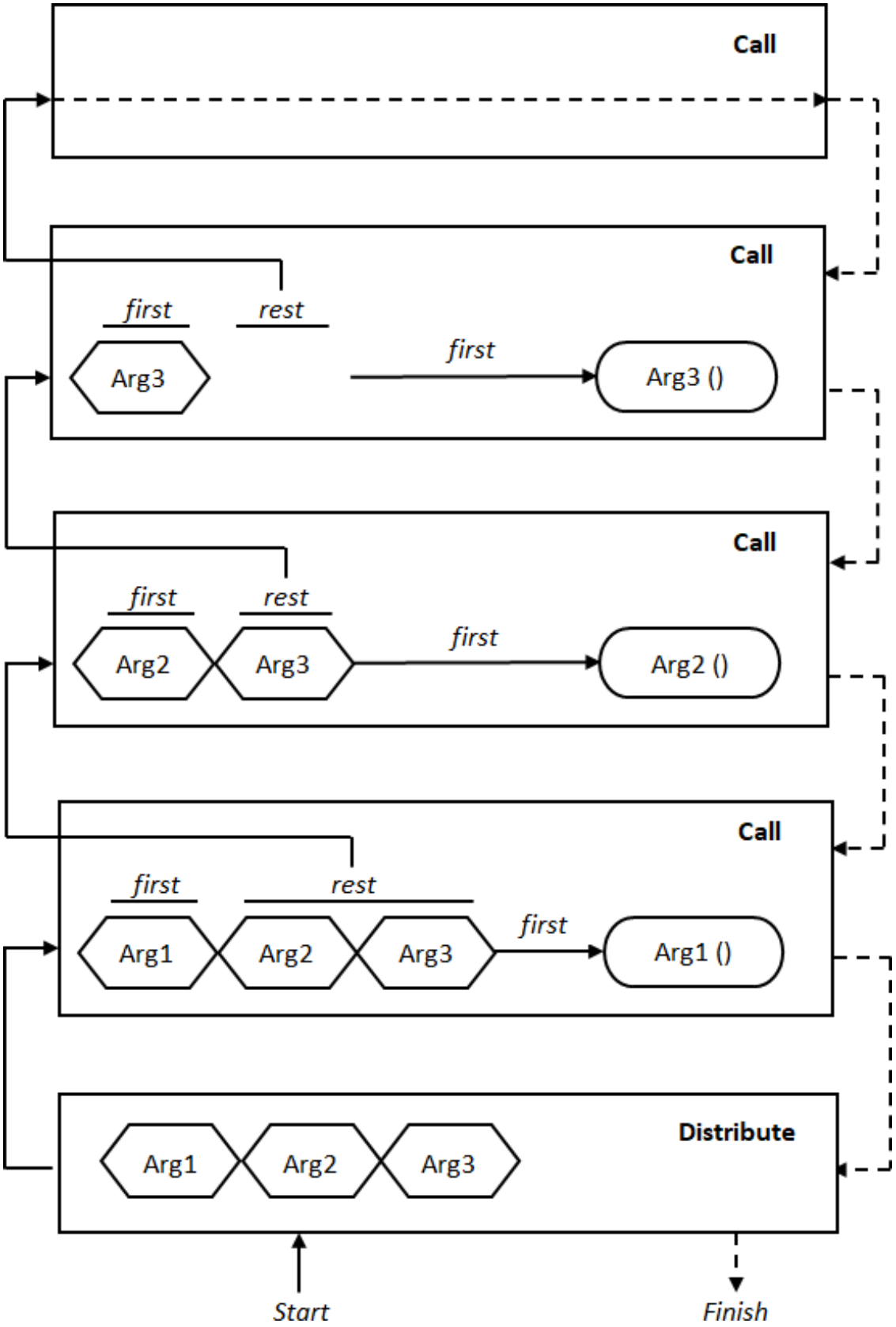

Рис. 22. Рекурсивное развертывание пакета параметров для трех аргументов

Использование распределения вызовов для статического набора получателей приведено в Листинг 64.

Листинг 64. Распределение вызова для статического набора

```
void ExternalHandler()  // (1)
{
}

struct FO
{
  void callbackHandler() {}
  void operator() () {}
};

int main()
{
  FO fo;                                  // (2)
  auto lambda = []() {};                  // (3)
  auto cb2cl = std::bind(&FO::callbackHandler, fo);  // (4)

  Distribute(ExternalHandler, fo, cb2cl, lambda);    // (5)
}
```

В строках 1, 2, 3, 4 объявлены соответствующие объекты вызова: внешняя функция, функциональный объект, лямбда-выражение, объект для вызова метода класса. Для вызова метода класса в строке 4 объявляется объект связывания (см. п. 4.6.6), в строке 5 происходит распределение вызовов.

5.2.2. Передача данных

Если в вызов необходимо передавать данные, то для этого в описанные выше функции необходимо ввести дополнительный параметр (Листинг 65).

Листинг 65. Распределяющая функция для статического набора получателей с передачей данных

```
template <typename CallData>  // (1)
void Call(CallData& data)
{
}

template <typename CallData, typename First, typename… Others>  // (2)
void Call(CallData data, First& first, Others&…rest)
{
  first(data);          // (3)
  Call(data, rest…);  // (4)
}
```

```
template <typename CallData, typename … CallObjects>  // (5)
void Distribute(CallData data, CallObjects… objects)
{
  Call(data, objects…);  // (6)
}
```

Приведенная реализация повторяет Листинг 63 п. 5.2.1, только теперь в функциях к объектам вызова добавляется параметр **data** для передачи данных.

Пример распределения для статического набора получателей с передачей данных представлен в Листинг 66.

Листинг 66. Распределение вызовов для статического набора получателей

```
void ExternalHandler(int eventID)  // (1)
{
}

struct FO
{
  void callbackHandler(int eventID) {}
  void operator() (int eventID) {}
};

int main()
{
  using namespace std::placeholders;

  FO fo;  // (2)
  auto lambda = [](int eventID) {};                    // (3)
  auto cb2cl = std::bind(&FO::callbackHandler, fo, _1);  // (4)

  int eventID = 0;  // (5)

  Distribute(eventID, ExternalHandler, fo, cb2cl, lambda);  // (6)
}
```

В строках 1, 2, 3, 4 объявлены соответствующие объекты вызова: внешняя функция, функциональный объект, лямбда-выражение, объект для вызова метода класса. Для вызова метода класса в строке 4 объявляется объект связывания (см. п. 4.6.6), в строке 5 объявляется переменная для передачи данных. В строке 6 происходит распределение вызовов, первым параметром передается аргумент данных **eventID**.

5.3. Настройка сигнатуры для передачи данных

5.3.1. Общая концепция

В рассмотренной выше реализации распределения с передачей данных (п. 5.2.2) есть один недостаток: данные, передаваемые в вызов, имеют заранее прописанную сигнатуру. В нашем случае предполагается, что это единственная числовая переменная. Если нам понадобится другая сигнатура, т. е. другой набор и типы переменных, нам придется повторять всю реализацию распределения, изменяя только сам вызов. Можно ли настроить сигнатуру, как это мы делали в универсальном аргументе? Тогда мы определяли сигнатуру с помощью пакета параметров, но теперь у нас пакет параметров используется для задания объектов вызова.

Получается, нам необходим еще один пакет параметров. В общем случае допускается объявлять шаблон функции с несколькими пакетами[30], однако в этом случае для вывода типов пакета используется схема раскрытия. По этой причине необходимо, чтобы все пакеты параметров раскрывались параллельно в рамках одной синтаксической конструкции (Листинг 67), что для нашей задачи не подходит: мы должны вначале раскрыть пакет объектов вызова, а затем для каждого элемента пакета раскрыть пакет сигнатуры. Здесь нужно какое-то другое решение.

Листинг 67. Пример шаблона функции с несколькими пакетами параметров

```
template<typename…First, typename…Second>
void init(std::pair<First,Second>…)
{
}

int main()
{
  init(std::make_pair(1, 2), std::make_pair(3,4), std::make_pair(0.
}
```

Поскольку пакет параметров в нашем случае может быть только один, необходима структура данных, в которую можно упаковать объекты различных типов. На эту роль лучше всего подойдет кортеж.

> **Кортеж** – это структура данных, которая используется для хранения объектов различных типов.

В STL кортеж реализуется шаблонным классом **std::tuple**, параметрами шаблона являются типы, которые будут храниться в кортеже. Этот класс как нельзя лучше подойдет для наших целей, потому что объекты вызова у нас также задаются параметрами шаблона.

Итак, у нас есть два набора: объекты вызова и данные, передаваемые в вызов. Какой набор упаковать в кортеж, а какой в пакет параметров? Рассмотрим различные способы упаковки наборов.

---

[30] Но не шаблон класса, в шаблонах классов пакет параметров может быть только один. Кроме того, если в шаблоне объявляется пакет параметров, он должен быть последним в списке параметров шаблона.

5.3.2. Способ 1: объекты в пакет, данные в кортеж

При использовании данного способа реализация распределения практически совпадает с описанной в Листинг 65 п. 5.2.2 с той разницей, что для передачи данных используется не переменная, а кортеж (Листинг 68).

Листинг 68. Распределение при упаковке объектов в пакет и данных в кортеж

```
    template <typename CallData>
    void Call(CallData& data)  // (1)
    {
    }

    template <typename CallData, typename First, typename…Others>
    void Call(CallData& data, First& first, Others&…rest)  // (2)
    {
      std::apply(first, data);  // (3)
      Call(data, rest…);       // (4)
    }

    template <typename… CallData, typename… CallObjects>
    void                          Distribute1(std::tuple<CallData…
> data, CallObjects… objects)  // (5)

    {
      Call(data, objects…);  // (6)
    }
```

Распределяющая функция объявлена в строке 5. Входными параметрами функции являются кортеж данных вызова **data** и пакет объектов вызова **objects**, типы их содержимого задаются параметрами шаблона. Внутри этой функции, в строке 6, происходит первый вызов рекурсивной функции, которой передаются соответствующие аргументы – кортеж и пакет.

Рекурсивная функция объявлена в строке 2. Эта функция извлекает очередной объект из пакета и осуществляет его вызов (строка 3). Здесь используется функция стандартной библиотеки **std::apply**, которая преобразует содержимое кортежа в список аргументов. Далее, в строке 4, пакет с оставшимися параметрами передается в рекурсивный вызов **Call**, и процесс повторяется до завершения рекурсии.

5.3.3. Способ 2: объекты в кортеж, данные в пакет

При использовании данного способа необходимо пройти по всем элементам кортежа и осуществить вызовы хранимых в нем объектов, передавая на вход пакет данных. Как осуществить обход содержимого кортежа?

Доступ к элементам кортежа осуществляется с помощью вызова

std::get<index>(tuple),

где **index** – это порядковый номер элемента (начиная с 0), **tuple** – имя переменной-кортежа. Проблема в том, что индексы должны быть заранее определены как числовые кон-

станты, использование переменной для задания индекса не допускается[31]. Поэтому здесь нельзя использовать ни циклы, ни функции с входным аргументом – индексом.

Можно попробовать объявить шаблон функции, в которой индекс задается параметром шаблона, а внутри функции изменить индекс и осуществить рекурсивный вызов. По идее, в этом случае для каждого индекса должна была бы сгенерироваться отдельная специализированная функция, однако стандарт не допускает специализацию шаблонов функций[32]. Но специализация шаблонов классов допустима, поэтому выходом будет обернуть функцию в класс – оболочку и уже для класса объявлять специализацию по индексам. Реализация приведена в Листинг 69.

Листинг 69. Распределение при упаковке объектов в кортеж и данных в пакет

```
    template<std::size_t  Index,  typename  CallObjects,  typename…
CallData>  // (1)
    struct TupleIterator
    {
        static  void  IterateTupleItem(CallObjects&  callObjects,
CallData…callData)  // (2)
      {
         const std::size_t idx = std::tuple_size_v<CallObjects> -
Index;  // (3)
        std::get<idx>(callObjects)(callData…);
  // (4)
             TupleIterator<Index  -  1,  CallObjects,  CallData…
>::IterateTupleItem(callObjects, callData…);  // (5)
      }
    };

    template<typename CallObjects, typename… CallData>  // (6)
    struct TupleIterator<0, CallObjects, CallData…>     // (7)
    {
        static  void  IterateTupleItem(CallObjects&  callObjects,
CallData… callData)  // (8)
      {
      }
    };

    template<typename…           CallObjects,           typename…
CallData>                          // (9)
    void    Distribute2(std::tuple<CallObjects…>    callObjects,
CallData… callData)  // (10)
    {
      TupleIterator  // (11)
      <
```

[31] Это связано с тем, что функция получения элемента кортежа по индексу объявлена как шаблон с параметром – числовым значением. Переменные не могут выступать параметрами шаблона.

[32] В отличие от шаблонов классов, шаблоны функций могут быть перегружены. Если бы допускалась специализация шаблонов функций, то возникала бы неопределенность выбора перегруженной и специализированной функции.

```
    sizeof…(CallObjects),       // (12)
    std::tuple<CallObjects…>,   // (13)
    CallData…                   // (14)
>
::IterateTupleItem(callObjects, callData…);  // (15)
}
```

В строке 1 объявляется шаблон структуры. Параметрами шаблона выступают индекс элемента кортежа, сам кортеж и пакет параметров, который определяет данные, передаваемые в вызываемый объект.

Внутри структуры в строке 2 объявлена функция, осуществляющая выполнение вызова для элемента кортежа. Входными параметрами этой функции будет кортеж объектов вызова и пакет данных вызова, элемент кортежа определяется индексом – параметром шаблона. Функция объявлена статической, чтобы не объявлять экземпляр структуры в процессе вызова. По сути дела, структура здесь не несет функциональной нагрузки, она выступает в качестве оболочки, чтобы обеспечить специализацию функции по индексу (поскольку непосредственная специализация шаблонов функций невозможна).

В строке 3 осуществляется пересчет индекса: от размера (количества элементов) кортежа отнимается текущий индекс. Это необходимо для того, чтобы обход кортежа осуществлялся в прямом порядке, от первого элемента к последнему. Если не выполнять пересчет индексов, то обход будет происходить в обратном порядке.

В строке 4 осуществляется вызов объекта. С помощью вызова **get** по пересчитанному индексу осуществляется доступ к соответствующему элементу кортежа. Для указанного элемента выполняется вызов, на вход ему передается пакет данных **callData**, распакованный в список аргументов.

В строке 5 происходит рекурсивный вызов. Объявляется структура с новым значением параметра-индекса, уменьшенным на единицу. Вызывается соответствующая функция с передачей кортежа объектов и пакета параметров, и процесс повторяется заново.

С каждой итерацией значение индекса уменьшается, и когда оно станет равным нулю, необходимо остановить итерации, поскольку все элементы кортежа будут посещены. Для этой цели в строке 6 объявлена специализация структуры для нулевого индекса. Тело функции в этой структуре пустое, таким образом, рекурсия будет завершена.

В строке 9 объявлен шаблон распределяющей функции. Этот шаблон имеет два пакета параметров: пакет объектов вызова и пакет данных вызова, типы содержимого пакетов будут выводиться из входных аргументов. В строке 10 объявляется сама функция, которая на вход принимает два аргумента: кортеж объектов вызова и пакет данных вызова.

В строке 11 запускается процесс итерации путем инстанциирования шаблона **TupleIterator**. Аргументами шаблона выступают: количество объектов вызова (строка 12), вычисляется с помощью операции **sizeof** применительно к соответствующему пакету параметров; кортеж объектов вызова (строка 13); данные, передаваемые в вызов (строка 14). В строке 15 вызывается стартовая функция итерации с передачей соответствующих аргументов. Как видим, начальное значение индекса равно количеству объектов вызова, которое затем с каждой новой итерацией будет уменьшаться на единицу, в то время как пересчитываемый индекс, соответственно, увеличивается.

5.3.4. Способ 3: объекты и данные в кортежах

При использовании данного способа реализация практически повторяет рассмотренную в предыдущем параграфе, только вместо пакета данных будет использоваться кортеж (Листинг 70).

Листинг 70. Распределение при упаковке объектов и данных в кортежи

```
    template<std::size_tIndex,typenameCallObjects,typenameCallData>
  // (1)
    struct TupleIterator3
    {
      static void IterateTupleItem(CallObjects& callObjects, CallData&
      {
     const std::size_t idx = std::tuple_size_v<CallObjects> - Index;
  // (3)
          std::apply(std::get<idx>(callObjects), callData);
          TupleIterator3<Index - 1, CallObjects, CallData>::IterateTupl
      }
    };

    template<typename CallObjects, typename CallData>  // (6)
    struct TupleIterator3<0, CallObjects, CallData>    // (7)
    {
      static void IterateTupleItem(CallObjects& callObjects, CallData&
      {
      }
    };

    template<typename… CallObjects, typename… CallData>
    void                        Distribute3(std::tuple<CallObjects…
> callObjects, std::tuple<CallData…> callData)  // (10)
    {
      TupleIterator3               // (11)
      <
      sizeof…(CallObjects),       // (12)
      std::tuple<CallObjects…>,  // (13)
      std::tuple<CallData…>      // (14)
      >
      ::IterateTupleItem(callObjects, callData);  // (15)
    }
```

По сравнению с Листинг 69 п. 5.3.3 изменения здесь следующие. Входными параметрами распределяющей функции (строка 10) являются кортеж объектов и кортеж данных (ранее параметр для данных задавался пакетом). В объявлениях шаблонов структур для обхода кортежа (строки 1, 6) параметр, определяющий данные вызова, объявляется как тип (ранее это был пакет). Вызов объекта (строка 4) осуществляется через **std::apply** (ранее объект вызывался

непосредственно). И еще здесь изменены имена структур, чтобы избежать конфликта имен с предыдущей реализацией.

5.3.5. Сравнение способов

В Листинг 71 приведен пример распределения вызовов с использованием различных способов настройки сигнатуры, в качестве данных выступают два числовых значения.

Листинг 71. Распределение вызовов с заданной сигнатурой

```
    void ExternalHandler(int eventID, int contextID) {}

    struct FO
    {
    void callbackHandler(int eventID, int contextID) {}
    void operator() (int eventID, int contextID) {}
    };

    int main()
    {
    int eventID = 0, contextID = 1;

    FO fo;
    auto lambda = [](int eventID, int contextID) {};
    auto cb2cl = std::bind(&FO::callbackHandler, fo, _1, _2);

    Distribute1(std::tuple(eventID, contextID), ExternalHandler,
fo, cb2cl, lambda);
    Distribute2(std::tuple(ExternalHandler, fo, cb2cl, lambda),
eventID, contextID);
    Distribute3(std::tuple(ExternalHandler, fo, cb2cl, lambda),
std::tuple(eventID, contextID));
    }
```

С точки зрения эффективности все три способа, в общем-то, равноценны. С точки зрения дизайна можно сказать следующее: первый способ самый простой в реализации; второй способ позволяет легко модифицировать код для сбора дополнительной информации при выполнении вызовов; третий способ позволяет передавать дополнительные параметры в функцию распределения, если это необходимо.

5.3.6. Настройка сигнатуры для перенаправления

В рассмотренных выше примерах мы предполагали, что все получатели используют одну и ту же сигнатуру вызова. Но что делать, если они имеют разные сигнатуры? Нам необходимо разработать какой-то объект, который бы обеспечивал следующее: настройку входной сигнатуры, в которую передаются данные вызова; настройку выходной сигнатуры, которая определяется получателем; преобразование одной сигнатуры в другую. По сути дела, необходимо обеспечить перенаправление вызовов, что решается с помощью инструментов STL, а именно – объектов связывания (см. п. 4.6.2). В этом случае в функцию распределителя вме-

сто объекта-получателя передается объект-связывание, который осуществляет перенаправление вызова с заданной сигнатурой. Пример реализации приведен в Листинг 72; здесь в качестве распределяющей функции используется реализация из Листинг 69 п. 5.3.3.

Листинг 72. Перенаправление вызовов с настройкой сигнатуры

```
    void NativeHandler(int eventID)
    {
    }

    void ExternalHandler(int eventID, int contextID)
    {
    }

    struct FO
    {
      void operator() (int eventID, int contextID) {}
      void callbackHandler(int eventID, int contextID) {}
    };

    int main()
    {
      int eventID = 0, contextID = 0;
      FO fo;
      auto lambda = [](int eventID, int contextID) {};

      Distribute2(std::tuple(   // (1)
        NativeHandler,          // (2)
              std::bind(ExternalHandler,   std::placeholders::_1,
contextID),           // (3)
       std::bind(&FO:: callbackHandler, fo, std::placeholders::_1,
contextID), // (4)
           std::bind(&FO::operator(),  fo,  std::placeholders::_1,
contextID),       // (5)
                     std::bind(lambda,    std::placeholders::_1,
contextID)                    // (6)
        ),
        eventID // (7)
      );
    }
```

Входными аргументами распределяющей функции служат кортеж объектов вызова (объявлен в строке 1) и данные вызова (строка 7). В строке 2 в кортеж передается объект вызова с сигнатурой, совпадающей с исходной. В строке 3 передается объект связывания (результат вызова **std::bind**), в котором исходный вызов перенаправляется в назначенную функцию **ExternalHandler**. В строке 4 объект связывания перенаправляет вызов в метод-член структуры, в строке 5 – в перегруженный оператор, в строке 6 – в лямбда-выражение.

5.4. Возврат результатов выполнения

5.4.1. Получение возвращаемых значений

До сих пор мы считали, что функции, реализующие код вызова, не возвращают результатов. Однако в некоторых случаях необходимо получить результаты выполнения вызовов. Очевидно, что в этом случае их должна вернуть распределяющая функция. Как же сформировать возвращаемые значение?

Поскольку возвращаемые значения могут иметь различные типы, напрашивается сохранять их в кортеже, который затем будет возвращаться как результат работы распределяющей функции. Но мы же не знаем заранее типы возвращаемых значений, их определяют объекты вызова. Какие тогда типы задавать при инстанциировании переменной-кортежа? Можно предложить следующее решение: при объявлении кортежа не указывать явно хранимые в нем типы, а в конструктор в качестве входных аргументов передать результаты выполнения вызовов. В этом случае типы элементов кортежа будут выведены автоматически.

Но сформировать набор результатов выполнения не так-то просто. Мы не можем перечислить в списке аргументов запрос объекта по индексу и его вызов, ведь количество объектов заранее не известно. Поэтому предварительно необходимо сформировать последовательность индексов, которая разворачивается в контексте запроса и вызова объекта. Реализация приведена в Листинг 73.

Листинг 73. Распределение вызовов с возвратом результатов

```
    template <typename… CallObjects, std::size_t… indices, typename…
CallData>                // (1)
    auto              DistributeReturnImpl(std::tuple<CallObjects…
>& callObjects, std::index_sequence<indices…>, CallData… callData)
  // (2)
    {
      return std::tuple(std::get<indices>(callObjects)(callData…)
…);                           // (3)
    }

    template<typename…            CallObjects,            typename…
CallData>                                  // (4)
    auto                  DistributeReturn(std::tuple<CallObjects…
> callObjects, CallData… callData)  // (5)
    {
      return DistributeReturnImpl(
  // (6)
        callObjects,
                                  std::make_index_sequence<sizeof…
(CallObjects)> (),                          // (8)
        callData…);
    }
```

Шаблон распределяющей функции объявлен в строке 4, параметрами шаблона являются пакет объектов вызова и пакет данных вызова. Сама функция объявлена в строке 5, входными параметрами являются кортеж вызываемых объектов, параметризованный пакетом объектов, и пакет данных вызова. Возвращаемое значение функции объявлено как **auto**, что означает, что оно будет выводиться из возвращаемого значения.

Для использования рассматриваемого распределения появляется требование, чтобы все объекты вызова возвращали результаты. Это связано с тем, что кортеж не может хранить типы **void**. Для вызовов, которые не возвращают результат, можно использовать любой из способов, описанный в главе 5.3.

В строке 6 вызывается вспомогательная функция, которой передается кортеж объектов вызова 7, последовательность индексов 8, данные вызова 9. Последовательность индексов формируется с помощью конструкции **std::make_index_sequence,** которой на вход в качестве значения передается размер пакета вызываемых объектов (определяется с помощью **sizeof**…).

В строке 1 объявлен шаблон вспомогательной функции, параметрами шаблона выступают пакет объектов вызова **CallObjects**, пакет индексов **Indices** и пакет данных вызова **CallData**. Сама функция объявлена в строке 2, ее входными параметрами являются: кортеж вызываемых объектов, параметризованный пакетом объектов вызова; последовательность индексов, параметризованная пакетом индексов; пакет данных вызова. Данная функция возвращает кортеж, сформированный по результатам вызова. Для получения элемента кортежа используется вызов **std::get**, на вход которому передается индекс элемента, и затем происходит вызов полученного элемента, на вход которому передаются данные **callData**. А поскольку вместо конкретного индекса мы используем последовательность индексов, она будет развернута в набор вызовов **get** с соответствующими индексами, таким образом, осуществляя вызовы для все элементов кортежа в соответствии с их индексами. Графически рассмотренная операция для трех объектов изображена на Рис. 23.

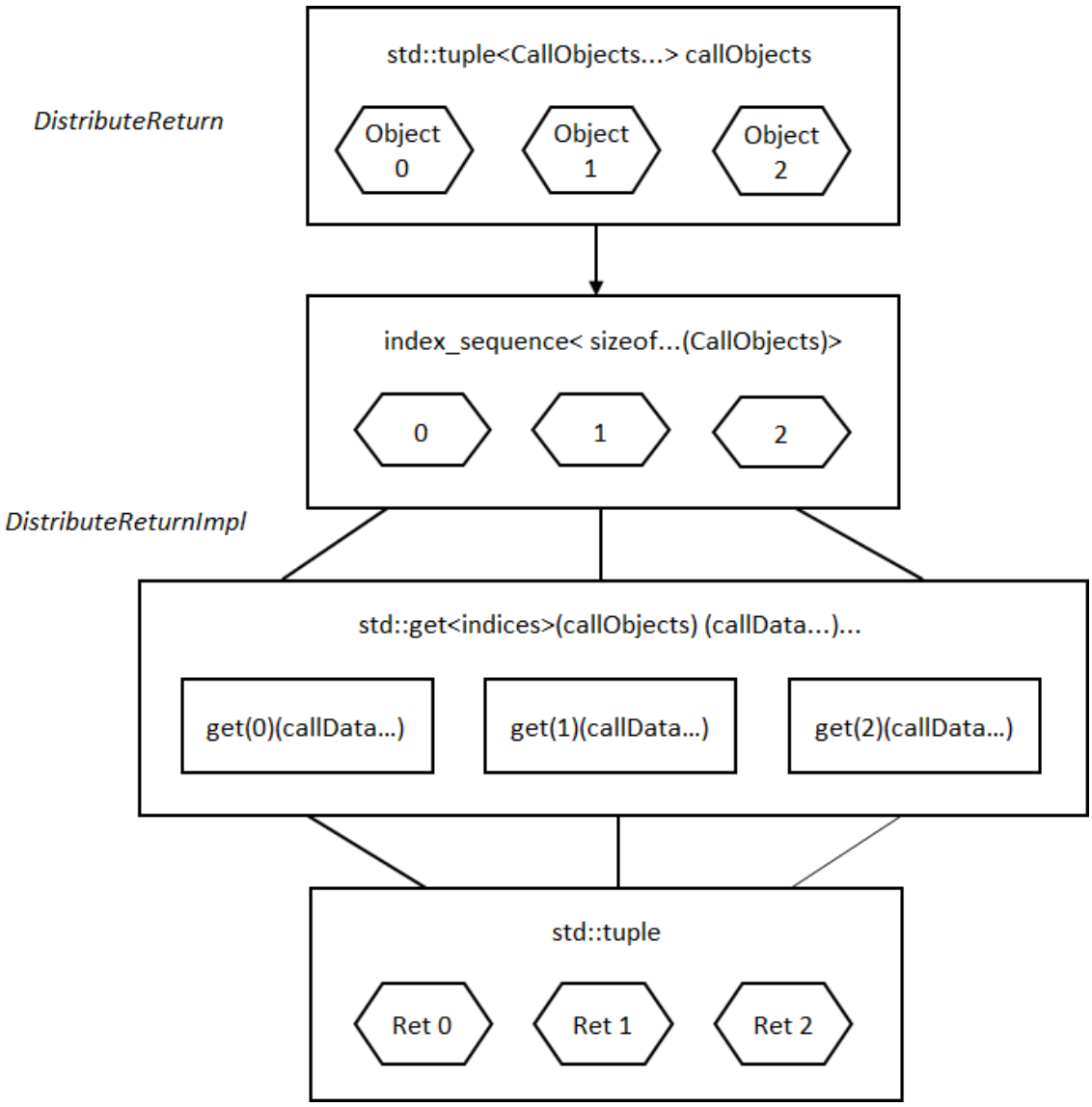

Рис. 23. Формирование кортежа возвращаемых значений

5.4.2. Анализ результатов

Итак, мы получили возвращаемые значения в виде кортежа. Как нам проанализировать полученные результаты? Существуют следующие способы анализа содержимого кортежа:

- доступ к элементам кортежа по индексу с помощью **std::get**;
- обход кортежа;
- использование структурных привязок.

Пример анализа значений, возвращаемых распределением вызовов, приведен в Листинг 74.

Листинг 74. Анализ возвращаемых значений

```
struct FO
{
  int operator() (int eventID)
  {
```

```
    return 10;
  }

};

struct SResult
{
  unsigned int code;
  const char* description;
};

SResult ExternalHandler(int eventID)
{
  return SResult{ 1, "this is an error" };
}

int main()
{
  FO fo;
  int eventID = 0;
  auto lambda = [](int eventID) { return 0.0; };

  auto results = DistributeReturn( std::tuple(fo, ExternalHandler,

  int foRes = std::get<0>(results);             // (2)
  SResult ExtRes = std::get<1>(results);        // (3)
  double lambdaRes = std::get<2>(results);      // (4)

  auto [foRes1, ExtRes1, lambdaRes1] = results; // (5)
auto [foRes2, ExtRes2, lambdaRes2] = DistributeReturn(std::tuple(fo, ExternalHandler, lambda), eventID);
// (6)
}
```

После выполнения распределения в строке 1 в переменную **results** помещен кортеж с результатами выполнения вызова. В строках 2, 3, 4 показано получение результатов с помощью запроса элементов кортежа по индексу, в строке 5 показано использование структурных привязок. В строке 6 показано, как можно использовать структурные привязки без промежуточной переменной **results**. Обход кортежа здесь не рассматривается, поскольку он был подробно описан в п. 5.3.3.

5.5. Распределитель для статического набора

5.5.1. Распределение без возврата результатов

До сих пор мы выполняли распределение с помощью функции, что вызывает определенные неудобства. Во-первых, вызов распределяющей функции получается громоздким, потому что приходится перечислять все объекты, участвующие в распределении. Во-вторых, требуются дополнительные операции, потому что в зависимости от способа настройки либо объекты вызова, либо аргументы сигнатуры необходимо упаковать в кортеж. Хорошим решением было бы предварительно сохранить нужные объекты, для чего нам понадобится распределитель в виде класса. Реализация приведена в Листинг 75.

Листинг 75. Распределитель для статического набора получателей

```
template<typename… CallObjects>  // (1)
class StaticDistributorVoid
{
public:
  StaticDistributorVoid (CallObjects… objects) : callObjects(object
  auto& tuple() {  return callObjects; }  // (3)

  template<typename… CallData>          // (4)
  void operator() (CallData… callData)
  {
    Distribute2(callObjects, callData…);
  }
private:
  std::tuple<CallObjects…> callObjects;  // (5)
};
```

В строке 1 объявлен шаблон класса, параметром которого выступает пакет объектов вызова. Кортеж для хранения объектов объявлен в строке 5, он инициализируется в конструкторе 2. Для доступа к кортежу реализован метод 3, который позволяет, если необходимо, изменить его содержимое.

В строке 4 объявлен перегруженный оператор, который осуществляет распределение. Этот оператор вызывает распределяющую функцию (реализацию см. Листинг 69 п. 5.3.3), которую при желании можно сделать членом класса.

Пример использования распределителя приведен в Листинг 76.

Листинг 76. Использование распределителя для статического набора

```
struct FO
{
  void operator() (int eventID) {}
  void callbackHandler(int eventID) {}
```

```
};

void ExternalHandler(int eventID) {}

int main()
{
  FO fo;
  int eventID = 0;
  auto lambda = [](int eventID) {};
  auto callbackToMethod = std::bind(&FO::callbackHandler, fo, std::

  StaticDistributorVoid distributor(ExternalHandler, fo, callbackTo

  distributor(eventID);  // (2)
}
```

Как видим, использование очень простое: в строке 1 объявляется распределитель, в конструктор передаются объекты вызова, через перегруженный оператор 2 производятся вызовы сохраненных объектов.

5.5.2. Распределение с возвратом результатов

Если нужно получить значения, возвращаемые вызовами, то в распределителе необходимо модифицировать перегруженный оператор (Листинг 77).

Листинг 77. Распределитель для статического набора с возвратом результатов

```
template<typename… CallObjects>  // (1)
class StaticDistributorReturn
{
public:
  StaticDistributorReturn(CallObjects… objects) : callObjects(objec
  auto& tuple() { return callObjects; }  // (3)

  template<typename… CallData>          // (4)
  auto operator() (CallData… callData)
  {
      return DistributeReturn(callObjects, callData…);
  }
private:
  std::tuple<CallObjects…> callObjects;  // (5)
};
```

В строке 4 объявлен перегруженный оператор с возвращаемым типом **auto**. Указанный тип будет выведен из значения, возвращаемого соответствующей распределяющей функцией. (реализацию см. в Листинг 73 п. 5.4.1).

Пример использования распределителя приведен в Листинг 78.

Листинг 78. Использование распределителя для статического набора с возвратом результатов

```
struct FO
{
  int operator() (int eventID) { return 10; }
  int callbackHandler(int eventID) { return 0; }
};

struct SResult
{
  unsigned int code;
  const char* description;
};

SResult ExternalHandler(int eventID)
{
  return SResult{ 1, "this is an error" };
}

int main()
{
  FO fo;
  int eventID = 0;
  auto lambda = [](int eventID) { return 0.0; };
  auto callbackToMethod = std::bind(&FO::callbackHandler, fo, std::

  StaticDistributorReturn distributor(ExternalHandler, fo, callback

  auto [resExtHandler, resFoOperator, resFoMethod, resLambda] = dis
}
```

В строке 1 объявляется распределитель, в конструктор передаются объекты вызова. Через перегруженный оператор 2 производятся вызовы хранимых объектов, результаты возвращаются с помощью структурных привязок.

К сожалению, мы не можем использовать рассмотренную реализацию для объектов, которые не возвращают результатов. Это связано с тем, что результаты выполнения вызовов возвращаются через кортеж, а он не может хранить типы **void**. Для таких вызовов нужно использовать реализацию, рассмотренную в предыдущем параграфе.

5.5.3. Параметризация возвращаемого значения

Итак, у нас имеется отдельная реализация распределителя для случая, когда результаты вызовов не требуются, и отдельная реализация для случая, когда необходимо получать возвращаемые значения. Обе реализации одинаковы, за исключением перегруженного оператора. Как сделать общую реализацию для обеих случаев? Разместить два перегруженных оператора в одном классе не получится, потому что они различаются только типом возвращаемого зна-

чения. Можно предложить следующее решение: ввести в шаблон дополнительный параметр, который указывает, нужно ли возвращать результаты выполнения вызовов, и в зависимости от этого по-разному формировать перегруженный оператор с помощью условной компиляции. Реализация приведена в Листинг 79.

Листинг 79. Условная компиляция в зависимости от типа возвращаемого значения

```
    template<typename… CallObjects>  // (1)
    class StaticDistributor
    {
    public:
                StaticDistributor(CallObjects…      objects)      :
callObjects(objects…) {}  // (2)
      auto& tuple() { return callObjects; }  // (3)

      template<typename… CallData>
      auto operator() (CallData… callData)  // (4)
      {

    #define callObject std::get<0>(callObjects)           // (5)
    #define callObjType decltype(callObject)              // (6)
    #define callObjInstance std::declval<callObjType>()  // (7)
    #define testCall callObjInstance(callData…)         // (8)
    #define retType decltype(testCall)                    // (9)

                  //if       constexpr       (std::is_same_v<void,
decltype(std::declval<decltype(std::get<0>(callObjects))>()
(callData…))>)  // (10)
      if constexpr (std::is_same_v<void, retType>)         // (11)
        return Distribute2(callObjects, callData…);       // (12)
      else
        return DistributeReturn(callObjects, callData…);  // (13)
      }
    private:
      std::tuple<CallObjects…> callObjects;
    };
```

В строках 1 – 4 код идентичен реализации распределителя в предыдущих случаях (Листинг 75 п. 5.5.1, Листинг 77 п. 5.5.2). Интерес представляет реализация перегруженного оператора (строка 4).

Макросы в строках 5 – 9 предназначены только для облегчения понимания кода, без них конструкция получается запутанной (строка 10).

В строке 5 мы получаем объект вызова, для которого будет проверяться, возвращает ли он значение. Мы запрашиваем нулевой элемент кортежа, поскольку предполагается, что кортеж содержит хотя-бы один объект (иначе зачем распределять вызовы для пустого кортежа?).

В строке 6 определяется тип объекта, который мы запросили. В строке 7 объявляется мета-экземпляр объекта соответствующего типа. Мы говорим «мета-экземпляр», потому что реально объект не создается, но его характеристики используются компилятором для анализа.

Конструкция **declval** необходима, чтобы не было ошибки в случае, если объект не имеет конструктора по умолчанию.

В строке 8 производится мета-вызов с передачей параметров. Мета-вызов здесь имеет тот же смысл, что и мета-экземпляр, т. е. в реальности вызов не производится, а используется для анализа. В строке 9 определяется тип значения, возвращаемого мета-вызовом.

В строке 11 проверяется, является ли тип возвращаемого значения void, и в этом случае вызывается распределяющая функция без возврата результатов (строка 12). В противном случае вызывается распределяющая функция, возвращающая результаты (строка 13).

Использование распределителя с условной компиляцией приведено в Листинг 80.

Листинг 80. Условная компиляция в зависимости от типа возвращаемого значения

```
struct FOReturn
{
  int operator() (int eventID) {return 10;}
};

struct FOVoid
{
  void operator() (int eventID) {  /*do something*/  }
};

struct SResult
{
  unsigned int code;
  const char* description;
};

SResult ExternalReturn(int eventID)
{
  return SResult{ 1, "this is an error" };
}

void ExternalVoid(int eventID)
{
}

int main()
{
  int eventID = 0;
  FOReturn foRet;
  FOVoid   foVoid;

  auto lambdaRet = [](int eventID) { return 0.0; };
  auto lambdaVoid = [](int eventID) {};

  using FunPtrRet = SResult(*)(int);
```

```
    using LambdaTypeRet = decltype(lambdaRet);
    using FunPtrVoid = void(*)(int);
    using LambdaTypeVoid = decltype(lambdaVoid);

       StaticDistributor<FOReturn,  FunPtrRet,  LambdaTypeRet>
distributor1(foRet, ExternalReturn, lambdaRet);  // (1)
       StaticDistributor<FOVoid,  FunPtrVoid,  LambdaTypeVoid>
distributor2(foVoid, ExternalVoid, lambdaVoid);  // (2)

    auto results = distributor1(eventID);
    distributor2(eventID);
  }
```

Как видим, в обоих случаях объявляется один и тот же распределитель, а из свойств объектов распределения будет генерироваться соответствующий перегруженный оператор.

5.6. Динамический набор получателей

5.6.1. Распределение в динамическом наборе

В предыдущих параграфах мы рассматривали статический набор получателей, когда типы и количество получателей определены на этапе компиляции и остаются неизменными. Теперь рассмотрим динамический набор, когда типы и количество получателей заранее неизвестны и изменяются в процессе выполнения программы. В какой-то степени реализация здесь получается проще: у нас не будет специализаций, рекурсий, выведения типов и прочей так называемой «шаблонной магии», все решается обычными методами классического программирования.

Итак, поскольку количество объектов заранее не определено, для их хранения необходим динамический контейнер. Однако он не может хранить объекты непосредственно, поскольку они могут иметь разные типы, а динамический контейнер работает с данными одного строго определенного типа. Выходом будет хранить универсальные аргументы, а уже в них сохранять объекты вызова. Структурная схема изображена на Рис. 24.

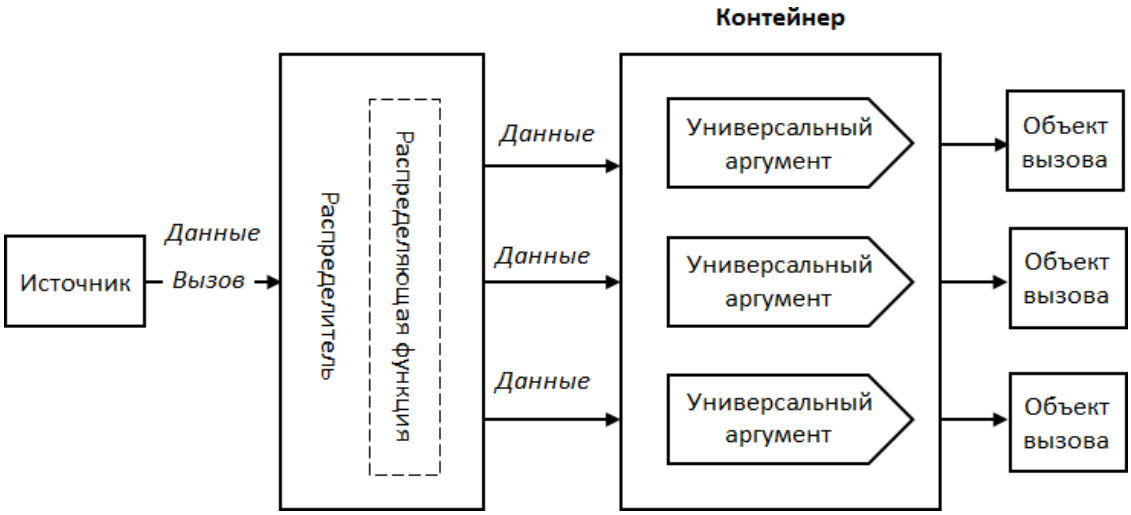

Рис. 24. Структурная схема распределителя для динамического набора получателей

Оптимальным решением будет реализация распределителя в виде класса, который, кроме выполнения распределения, будет поддерживать операции с контейнером. Конечно же, проектировать динамический контейнер и универсальный аргумент не нужно – в STL имеется все необходимое. Контейнер, в общем-то, можно использовать любой, а на роль универсального аргумента нет ничего лучше, чем **std::function**. Реализация приведена в Листинг 81.

Листинг 81. Распределитель для динамического набора получателей

```
template<typename unused> class DynamicDistributor;  // (1)

template<typename Return, typename… ArgumentList>  // (2)
class DynamicDistributor<Return(ArgumentList…)>
{
public:
```

```
  template <typename CallObject>
  void addCallObject(CallObject object)         // (3)
  {
    callObjects.push_back(object);
  }

  void operator ()(ArgumentList… arguments)  // (4)
  {
    for (auto& callObject : callObjects)
    {
      callObject(arguments…);
    }
  }
private:
  std::list< std::function<Return(ArgumentList …)> > callObjects;
};
```

В строке 1 объявлена общая специализация шаблона. Реализация класса здесь отсутствует, поскольку для каждой сигнатуры она будет различной. В строке 2 объявлен шаблон для частичной специализации, в котором два аргумента: тип возвращаемого значения и пакет параметров, передаваемых на вход вызова. Подобную конструкцию мы использовали, когда рассматривали настройку сигнатуры для универсального аргумента (п. 4.5.2).

В строке 3 объявлен метод, который добавляет объект вызова в контейнер, сам контейнер объявлен в строке 5. Тип контейнера мы выбираем список, поскольку он не перемещает элементов при вставке/удалении, а произвольный доступ здесь не требуется. Типом хранимых данных в контейнере является объект **std::function**, аргументы которого задаются исходя из параметров в объявлении шаблона класса.

В строке 4 объявлен перегруженный оператор, который осуществляет распределение вызовов, т. е. является распределяющей функцией. Он обходит элементы контейнера и осуществляет вызов в соответствии с списком аргументов, типы которых задаются в пакете параметров шаблона.

5.6.2. Получение возвращаемых значений

Как получить возвращаемые значения для динамического набора? На момент вызова распределяющей функции количество получателей может быть любым, и, соответственно, число возвращаемых значений заранее не определено. Использовать динамический контейнер как возвращаемое значение функции является плохой идеей: во-первых, заполнение контейнера и создание его копии в стеке требует значительного расхода времени и увеличивает фрагментацию памяти; во-вторых, если возвращаемое значение не используется, то все вышеописанное будет работать «вхолостую», выполняя совершенно ненужные операции. Использовать контейнер как входной параметр – это тоже идея не очень: мы вынуждаем привязаться к контейнеру определенного типа, а если нам результаты нужно хранить в других структурах? А если нам вообще их не нужно хранить, а нужно всего лишь проверить? Вопросы, вопросы… Можно предложить следующее решение: для возврата результата использовать обратный вызов, а пользователь сам решает, что делать с возвращаемыми значениями. Реализация приведена в Листинг 82.

Листинг 82. Возврат значений для динамического набора получателей

```
template <typename Return, typename… ArgumentList>
class DynamicDistributor<Return(ArgumentList…)>
{
                                        /**********************……
**********************************/

  template<typename CallbackReturn >  // (1)
                                    void                operator()
(CallbackReturn callbackReturn, ArgumentList… arguments)
  {
    for (auto& callObject : callObjects)
    {
        callbackReturn(callObject(arguments…));  // (2)
    }
  }
private:
  std::list< std::function<Return(ArgumentList …)> > callObjects;
};
```

Реализация совпадает с Листинг 82 п. 5.6.1, только добавляется еще один перегруженный оператор. Его шаблон объявлен строке 1, параметром шаблона является тип аргумента, через который будет выполняться обратный вызов. В строке 2 происходит вызов объекта, результат возвращается через аргумент, переданный как входной параметр функции.

Пример распределения вызовов для динамического набора получателей приведен в Листинг 83.

Листинг 83. Распределение вызовов для динамического набора получателей

```
struct FO
{
  int operator() (int eventID) { return 10; }
  int callbackHandler(int eventID) { return 100; }
};

int ExternalHandler(int eventID)
{
  return 0;
}

int main()
{
  int eventID = 0;

  FO fo;
  auto lambda = [](int eventID) { return 0; };
```

```
    auto binding = std::bind(&FO::callbackHandler, fo, std::placehold

    DynamicDistributor<int(int)> distributor;        // (1)

    distributor.addCallObject(fo);                   // (2)
    distributor.addCallObject(ExternalHandler);      // (3)
    distributor.addCallObject(binding);              // (4)
    distributor.addCallObject(lambda);               // (5)

    distributor(eventID);                            // (6)

    auto onReturnValue = [](int callResult) {};      // (7)
    distributor(onReturnValue, eventID);             // (8)
}
```

В строке 1 инстанциирован класс распределителя с заданной сигнатурой функции. В строке 2, 3, 4, 5 в распределитель добавляются объекты вызова различного типа. В строке 6 запускается распределение вызовов, в результате которого будут вызваны добавленные объекты. В строке 7 объявлено лямбда-выражение для получения результатов, при вызове соответствующего оператора 8 это выражение будет вызвано для каждого возвращаемого значения.

Касательно модификации содержимого контейнера наш распределитель поддерживает только одну операцию – добавление получателя. Ни удаление, ни модификация получателей не поддерживается. Это связано с тем, что получатели не идентифицированы, и поэтому невозможно узнать, в каком элементе контейнера хранится соответствующий объект вызова[33]. Далее мы рассмотрим, как можно решить указанную проблему.

[33] Справедливости надо отметить, что идентификация получателей все-таки возможна. Для этого можно использовать, например, итератор контейнера либо указатель на объект std::function, либо, например, динамически присваивать объекту контейнера какое-нибудь значение. Однако это было бы плохим решением в силу целого ряда причин:1) нарушается важнейший принцип проектирования – разделение интерфейса и реализации. Мы жестко завязываемся на структуру хранения объектов вызовов, поэтому архитектура получается монолитной;2) идентификаторы объектов не несут никакой смысловой нагрузки, это просто некие абстрактные значения;3) идентификаторы не детерминированы, при добавлении объекта в контейнер идентификатор получит произвольное значение;4) идентификаторам объектов невозможно назначить заранее заданные значения;5) в силу вышеуказанных причин невозможно реализовать логические протоколы обмена.

5.7. Адресное распределение

5.7.1. Понятие адресного распределения

До сих пор мы предполагали, что вызовы должны быть сделаны для всех получателей. Однако зачастую требуется распределять вызовы не всем, а только некоторым получателям из списка.

Как это реализовать? Прежде всего, необходимо как-то идентифицировать получателей, для чего вводится понятие адреса. Каждому получателю присваивается адрес, и с каждым адресом связывается универсальный аргумент, который хранит объект вызова. Таким образом, зная адреса получателей, можно осуществлять вызовы только для конкретных объектов. Попутно решается задача изменения списка получателей: по заданному адресу возможно удаление/изменение соответствующего аргумента.

Что может быть адресом? Все что угодно: числа, строки, структуры и т. п. Единственное требование, предъявляемое к адресу, заключается в том, что он должен быть уникальным, в противном случае невозможно однозначно идентифицировать получателя. Мы сделаем тип адреса параметром шаблона, а пользователь сам решит, что использовать в качестве адреса.

Теперь в функцию распределителя, помимо данных, будет передаваться адрес. Источник должен найти аргумент, которому соответствует полученный адрес, и выполнить для него вызов. Для поиска необходимо сравнивать адреса, но ведь мы не знаем их типы: теперь это параметр шаблона, и тип используемого адреса станет известен только после инстанциирования. По этой причине мы не можем производить сравнение адресов напрямую, для этого необходимо использовать предикаты (см. п. 4.3.3).

Какой выбрать контейнер? На эту роль лучше других подойдет **std::map**. Во-первых, не нужно вводить новую структуру для хранения адреса и аргумента, контейнер реализует ее естественным образом в виде пары «ключ-значение». И, во-вторых, **std::map** осуществляет быстрый поиск по ключу, в качестве которого выступает адрес. Структурная схема изображена на Рис. 25.

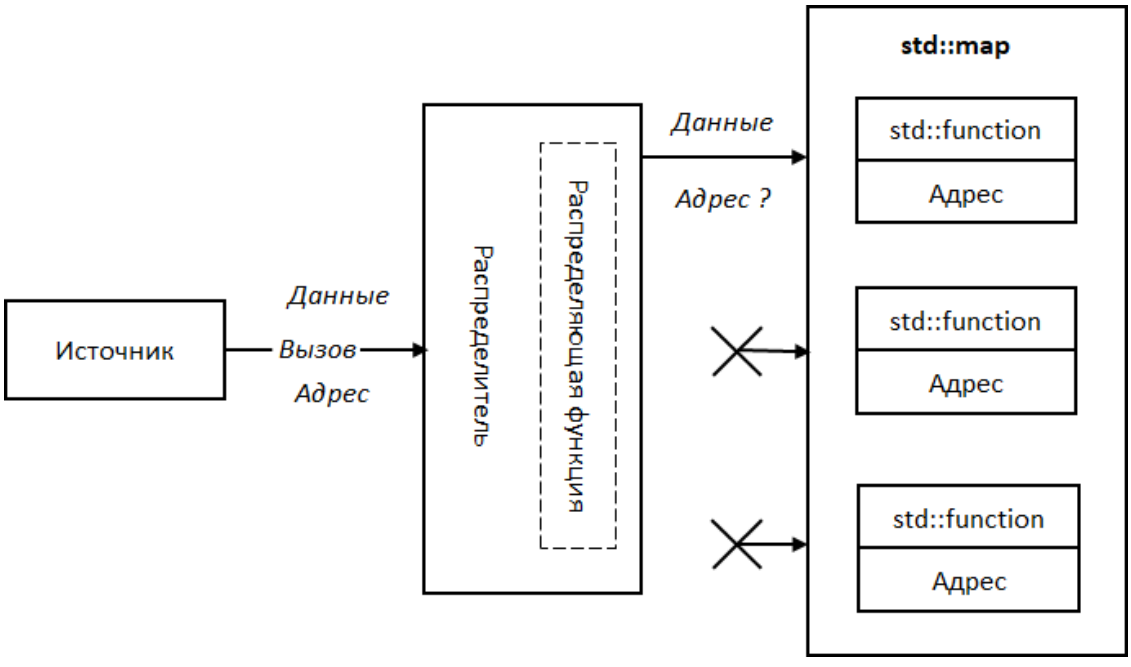

Рис. 25. Структурная схема адресного распределения

5.7.2. Адресный распределитель

Реализация адресного распределителя приведена в Листинг 84.

Листинг 84. Распределитель для адресного набора получателей

```
  template<typename Address, typename AddressCompare, typename Function> class AddressDistributor;
// (1)

  template<typename Address, typename AddressCompare, typename Return, typename... ArgumentList>
// (2)
  class AddressDistributor<Address,AddressCompare,Return(ArgumentList…)>
// (3)
  {
  public:
    template<typename CallObject>  // (4)
    void addReceiver(Address address, CallObject object)
    {
      callObjects.insert({ address,object } );
    }

    void deleteReceiver(Address address)  // (5)
    {
      callObjects.erase(address);
    }

                                    Return                operator()
(Address address, ArgumentList… arguments)  // (6)
    {
      auto iterator = callObjects.find(address);  // (7)
      if (iterator != callObjects.end())
      {
        return iterator->second(arguments…);    // (8)
      }
      else
      {
        throw std::invalid_argument("Invalid receiver address");  //
      }
    }
  private:
    std::map< Address, std::function<Return(ArgumentList…)>, AddressC
  };
```

В строке 1 объявлена общая специализация шаблона, параметрами выступают адрес получателя **Address**, предикат для сравнения **AddressCompare** и сигнатура распределяющей функции **Function**. Реализация здесь отсутствует, поскольку для каждой сигнатуры требуется

отдельная специализация – аналогично настройке сигнатуры для универсального аргумента (п. 4.5.2).

В строке 2 объявлена частичная специализация, в которой дополнительно представлены параметр для возвращаемого значения **Return** и пакет параметров **ArgumentList** для аргументов функции. В строке 3 объявлен класс, который специализируется сигнатурой из указанных параметров.

В строке 4 объявлен шаблон метода для добавления получателя, который принимает адрес **address**, вызываемый объект **object** и добавляет их в контейнер. В строке 5 объявлен метод для удаления получателя. Оба метода работают с контейнером, который объявлен в строке 10. Контейнер объявлен как **std::map**, ключом является адрес, а значением – объект **std::function** с заданной сигнатурой.

В строке 6 объявлен перегруженный оператор, который осуществляет распределение вызовов, т. е. является распределяющей функцией. Он пробегает по всем элементам контейнера и осуществляет вызов в соответствии с списком аргументов, типы которых задаются в пакете параметров шаблона класса. Поскольку мы используем адресное распределение, т. е. предполагается, что вызов попадает только одному получателю, то мы операторе можем вернуть результат вызова.

В строке 7 происходит поиск получателя по адресу. Если получатель найден, то происходит вызов объекта (строка 8). Если получатель не найден, то генерируется исключение (строка 9), иначе какой результат нам возвратить?

5.7.3. Использование адресного распределения

Пример использования адресного распределения приведен в Листинг 85.

Листинг 85. Использование адресного распределения

```
struct FO
{
  int operator() (int eventID)
  {
    return 10;
  }
};

int ExternalHandler(int eventID)
{
  return 0;
}

struct ReceiverAddress  // (1)
{
  ReceiverAddress(int idGroup = 0, int idNumber = 0)
  {
    group = idGroup; number = idNumber;
  }

  int group;
  int number;
```

```
};

template<>
struct std::less<ReceiverAddress>  // (2)
{
  bool operator() (const ReceiverAddress& addr1, const ReceiverAddr
  {
    if (addr1.group < addr2.group)
    {
      return true;
    }
    else
    {
      if (addr1.group == addr2.group)
        return addr1.number < addr2.number;
      else
        return false;
      }
  }
};

int main()
{
  int eventID = 0;
  FO fo;
  auto lambda = [](int eventID) { return 0; };

  AddressDistributor<ReceiverAddress, std::less<ReceiverAddress>, i

  distributor.addReceiver({ 1,1 }, fo);               // (4)
  distributor.addReceiver({ 2,2 }, ExternalHandler);  // (5)
  distributor.addReceiver({ 3,3 }, lambda);           // (6)

  distributor({ 1,1 }, eventID);  // (7)
  distributor({ 2,2 }, eventID);  // (8)
  distributor({ 3,3 }, eventID);  // (9)
}
```

В строке 1 объявлена структура для адреса, которая состоит из двух полей: идентификатор группы и номер получателя в группе. Сравнить эти две структуры напрямую нельзя, поэтому потребуется реализовать предикат.

В строке 2 объявлен функциональный объект, реализующий предикат для сравнения адресов. Почему именно в таком виде? Дело в том, что **std::map** требует, чтобы в качестве предиката использовался именно функциональный объект, мы не можем для этого использовать внешнюю функцию или лямбда-выражение. Это связано с тем, что в контейнере предикат хранится в виде переменной с конструктором, тип переменной определяется параметром шаблона. А наличие конструктора может обеспечить только функциональный объект.

Указанный подход имеет как достоинства, так и недостатки. С одной стороны, нам достаточно всего лишь объявить тип объекта в параметре шаблона, а затем про него можно забыть.

Объект не нужно ни настраивать, ни передавать в конструктор как входной аргумент. С другой стороны, было бы удобно использовать в качестве предиката что-либо другое, например, лямбда-выражение или внешнюю функцию. Но в этом случае предикат пришлось бы инициализировать в конструкторе, причем ему нельзя было бы назначить значение по умолчанию. В любом случае, мы вынуждены следовать заданной реализации, поэтому предикат объявляем как функциональный объект.

В STL уже объявлен шаблон структуры для предикатов **std::less**, параметром которого выступает тип данных, которые необходимо сравнить. Этот предикат принимает на вход две переменные и возвращает **true**, если первая меньше второй[34]. **std::less** реализует арифметическое сравнение, поэтому для типов, которые поддерживают арифметические операции, предикат объявлять не нужно, он будет сгенерирован компилятором. Однако в нашем случае данные арифметически сравниваться не могут, поэтому мы специализируем этот шаблон своим типом (строка 2) и реализуем перегруженный оператор, который будет сравнивать две структуры. При инстанциировании контейнера компилятор сам выберет подходящую специализацию предиката, исходя из типа хранимых элементов.

В строке 3 объявлен объект распределителя путем инстанциирования соответствующего шаблона. Аргументами шаблона выступают тип адреса, предикат для сравнения и сигнатура для вызова объектов. В строках 4, 5, 6 в распределитель добавляются объекты вызова различных типов, в строках 7, 8, 9 эти объекты будут вызваны в соответствии с их адресами.

[34] Контейнер std::map требует именно такой предикат, less, который возвращает истину в случае, если первый элемент меньше второго. Другие контейнеры могут требовать иные предикаты, например, проверку на равенство equal.

5.8. Итоги

Под распределением вызовов понимается техника, в которой при вызове единственной функции осуществляется выполнение множества вызовов через соответствующие аргументы. Структурно распределение состоит из следующих компонентов: источник, получатель, распределитель, распределяющая функция.

Если типы и количество получателей известны на этапе компиляции и не планируется их изменение в процессе выполнения программы, то мы имеем статический набор получателей. Распределитель для статического набора можно реализовать в виде функции, в этом случае распределитель структурно совпадает с распределяющей функцией.

В общем случае распределяющая функция принимает набор объектов и набор данных вызова. Эти наборы могут упаковываться в кортеж и пакет параметров в различных комбинациях. С точки зрения дизайна каждый способ упаковки имеет свои преимущества и недостатки, с точки зрения эффективности они равноценны.

Если требуются результаты выполнения вызовов, то они реализуются с помощью отдельной распределяющей функции, которая возвращает результаты в виде кортежа.

Зачастую бывает удобно реализовать распределитель для статического набора в виде класса, в котором объекты вызова хранятся в кортеже, а распределяющей функцией выступает перегруженный оператор. Здесь возникает проблема, как использовать класс с возвратом результатов выполнения и без возврата: перегруженный оператор имеет одинаковый набор входных параметров, различается только наличие и отсутствие возвращаемого значения. Выходом будет реализация двух отдельных классов либо общий класс с дополнительным параметром – индикатором. Во втором случае теряется возможность автоматического вывода типа.

Если типы и количество получателей заранее неизвестны и изменяются в процессе выполнения программы, то мы имеем динамический набор получателей. Он реализуется в виде класса с контейнером, в котором хранятся универсальные аргументы.

Если необходима передача вызовов не всем получателям, а только некоторым, то используется адресное распределение. Поскольку тип используемого адреса заранее не определен, то для сравнения адресов нужно использовать предикаты.

На этом изложение теоретического материала можно считать законченным. Далее рассмотрим, как обратные вызовы используются в практике разработки ПО.

6. Практическое использование обратных вызовов

Итак, мы изучили теоретические основы проектирования обратных вызовов, теперь пришло время продемонстрировать, как они используются в реальных системах. Для иллюстрации мы воспользуемся примером разработки модуля управления датчиками из проекта «автоматизированная система управления технологическими процессами», в котором когда-то принимал участие автор. Данный пример адаптирован, в нем опущены многие детали, которые не имеют отношения к рассматриваемой теме. Мы пройдемся через основные этапы проектирования и проследим, как обратные вызовы используются в реальных инженерных задачах.

Подробное описание всех компонентов модуля заняло бы слишком много места и навряд ли имеет практическую ценность, поэтому мы будем рассматривать самые общие принципы функционирования с акцентом на использование обратных вызовов. Полностью проект можно посмотреть здесь: **https://github.com/Tkachenko-vitaliy/Callbacks/tree/master/Sensor**.

6.1. Разработка архитектуры

6.1.1. Техническое задание

Первый вопрос, который должен быть задан перед началом разработки чего бы то ни было, звучит следующим образом: что мы будем разрабатывать и что мы хотим в итоге получить? Этот вопрос совсем не тривиальный, как может показаться вначале. Без ясного осознания конечной цели, без четкого понимания свойств и характеристик, которыми должна обладать проектируемая система, разработка может растянуться до бесконечности: происходят постоянные переделки, доработки, хаотичная реализация все новых и новых функций с не очень понятной ценностью, и т. п. В итоге, вместо результата мы сосредотачиваемся на процессе, а конечная цель пропадает где-то за горизонтом. Не сталкивались с такими проектами? Что ж, вам крупно повезло; чтобы также везло в дальнейшем, и подобные проекты в вашей карьере отсутствовали, любое проектирование нужно начинать с постановки целей, которые выражаются в требованиях, предъявляемых к системе. В нашем случае они будут следующими.

Разработать модуль управления датчиками, который должен обеспечивать:

1. Настройку конфигурации датчиков и возможность ее изменения в процессе работы.
2. Отслеживание состояния и определение неисправности датчиков.
3. Считывание показаний отдельных датчиков.
4. Считывание показаний всех работоспособных датчиков.
5. Асинхронный опрос показаний.
6. Возможность получения минимальных и максимальных значений для группы датчиков.
7. Настройка пороговых значений показателей и уведомление при их превышении.
8. Возможность работы как с реальными физическими датчиками, так и с их программными моделями.

6.1.2. Сценарий функционирования

Базовый сценарий функционирования модуля следующий.

Основным компонентом, поставляющим информацию, являются датчики. Они могут производить измерения трех типов: текущее, сглаженное и производное. Для идентификации датчикам присваиваются уникальные номера.

Перед началом работы производится настройка, т. е. определяется состав датчиков, с которых будут сниматься показания. Настройка не статическая, она может изменяться в процессе работы.

В любой момент приложение может запросить показания датчиков как в синхронном, так и в асинхронном режиме. Показания возвращаются только для функционирующих датчиков, в приложении должна иметься возможность проверить их работоспособность.

Коммуникация с датчиками осуществляется через протокол USB либо Ethernet путем пересылки / получения команд в соответствии с заданным протоколом.

В процессе работы модуль должен отслеживать и уведомлять приложение о том, что некоторые показатели превышают заданные пороговое значение. Состав измеряемых значений и их предельные величины настраиваются приложением.

В соответствии с описанием структура системы может быть представлена следующим образом (Рис. 26).

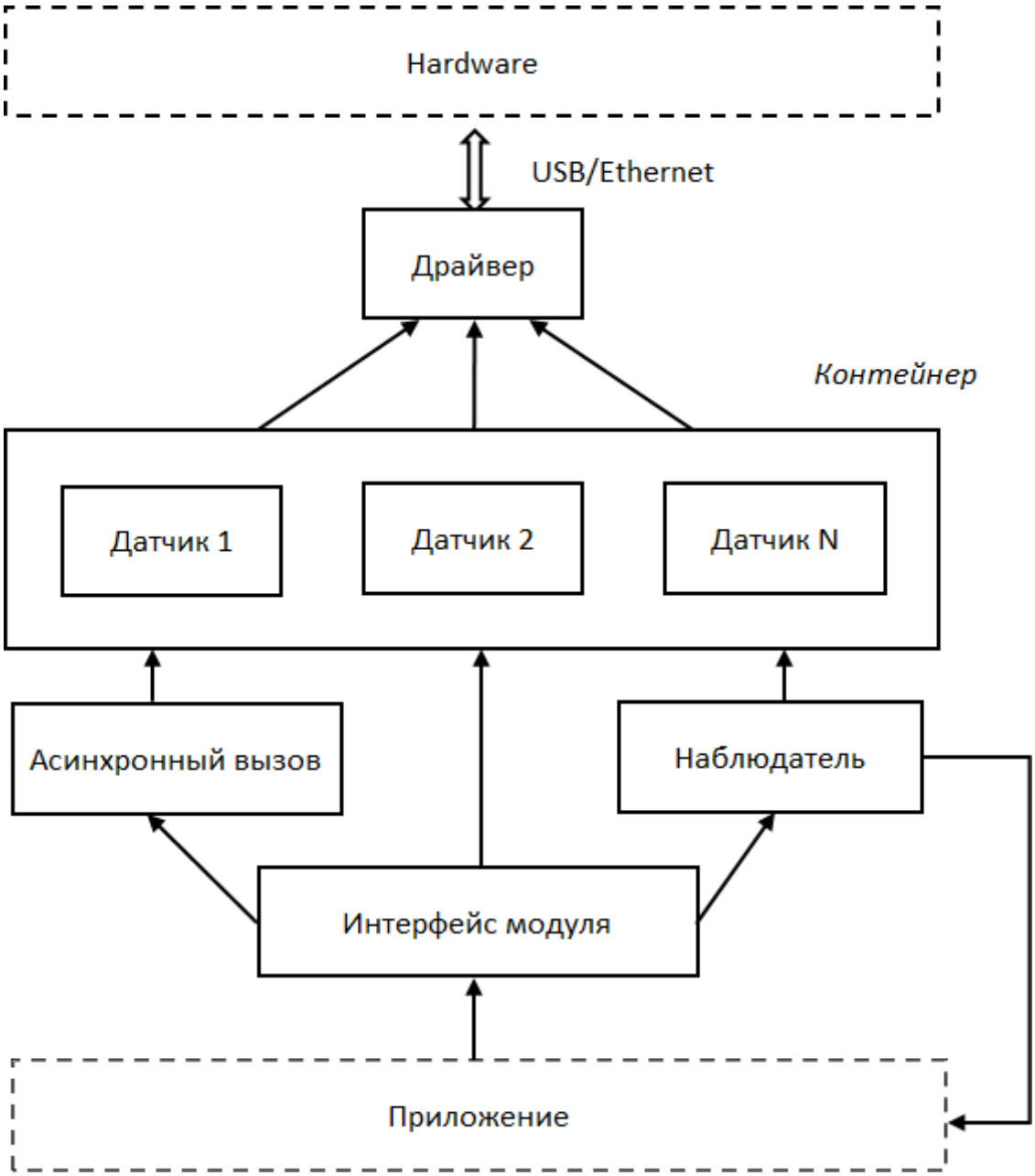

Рис. 26. Структурная схема

Приложение через интерфейс обращается к функциям модуля. В зависимости от вызываемой функции интерфейс обращается к соответствующим компонентам и возвращает результат.

Компонент «Асинхронный вызов» предназначен для выполнения асинхронных вызовов. «Наблюдатель» предназначен для отслеживания пороговых значений. «Контейнер» хранит список датчиков. Компонент «Датчик» через компонент «драйвер» обращается к аппаратному обеспечению.

6.1.3. Декомпозиция системы

Итак, в соответствии методологией объектно-ориентированного анализа необходимо определить состав классов и связи между ними, отражающие предметную область. Нам будут необходимы следующие классы:

- класс для работы с датчиком;
- контейнер для хранения указанных классов;
- драйвер, обеспечивающий низкоуровневое взаимодействие с аппаратурой;
- очередь для выполнения асинхронных запросов;
- класс для отслеживания пороговых значений;
- интерфейсный класс, который будет взаимодействовать с приложением для вызовов соответствующих функций модуля.

Обобщенная диаграмма классов модуля представлена на Рис. 27[35].

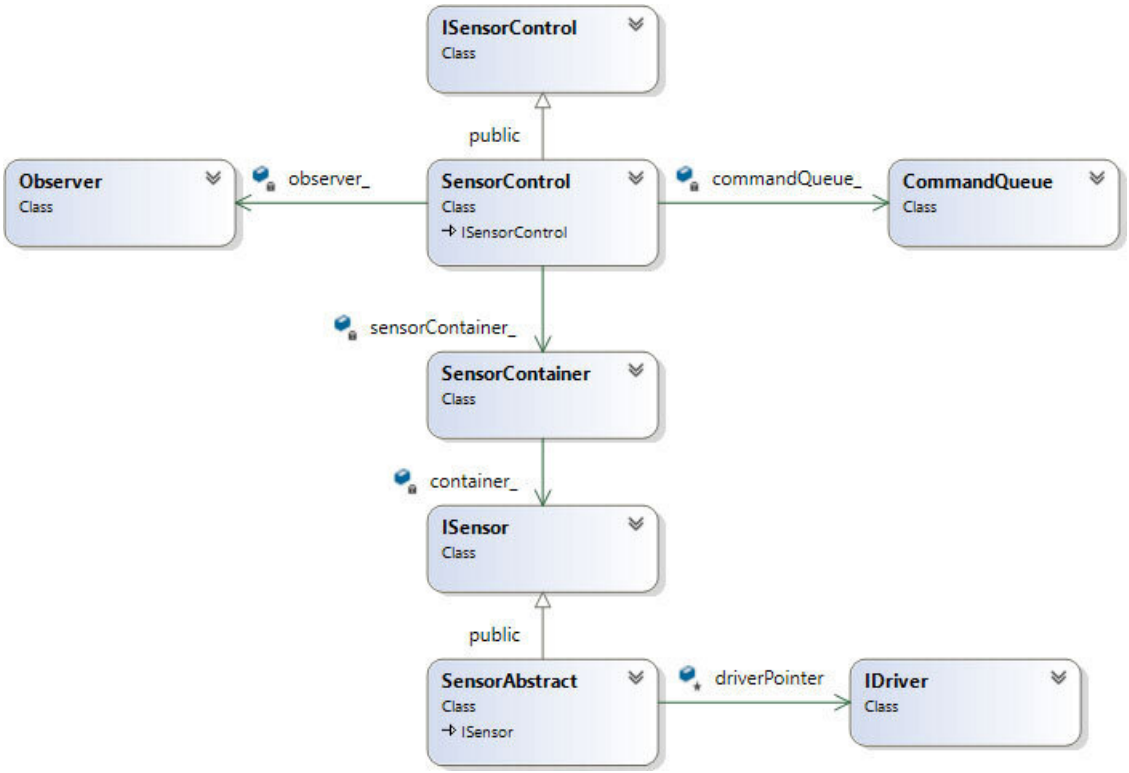

Рис. 27. Обобщенная диаграмма классов

Класс **ISensorControl** объявляет интерфейс модуля, класс **SensorControl** реализует указанный интерфейс. **SensorControl** содержит классы **Observer** (отслеживает пороговые значения), **CommandQueue** (очередь комманд для асинхронных запросов), **SensorContainer** (реализует контейнер для хранения классов для работы с датчиком).

Интерфейс для работы с датчиками объявлен в классе **ISensor**, обощенная реализация интерфейса осуществляется в классе **SensorAbstract**. Указанный класс хранит указатель на **IDriver**, который используется для получения значений датчиков. В классе **IDriver** объявляется интерфейс для взаимодействия с аппаратурой.

[35] Диаграмма классов изображена в формате UML. Читателям, которые не знакомы с указанным графическим языком моделирования, можно порекомендовать книгу «Леоненков А. В. Самоучитель UML 2».

6.2. Реализация классов

6.2.1. Общие определения

В Листинг 86 представлены общие объявления типов.

Листинг 86. Общие объявления типов (SensorDef.h)

```
    namespace sensor
    {
    class ISensor;
    class IDriver;

    using SensorNumber = unsigned int;       // (1)
    using SensorValue = double;              // (2)
    using CheckAlertTimeout = unsigned int;  // (3)

    enum class SensorType : uint32_t  // (4)
    {
      Spot = 0,
      Smooth = 1,
      Derivative = 2,
    };

    enum class DriverType : uint32_t  // (5)
    {
      Simulation = 0,
      Usb = 1,
      Ethernet = 2
    };

    enum class AlertRule : uint32_t // (6)
    {
      More = 0,
      Less = 1
    };

    using SensorPointer = std::shared_ptr<ISensor>;  // (7)
    using DriverPointer = std::shared_ptr<IDriver>;  // (8)
    using SensorValueCallback = std::function<void(SensorNumber,
SensorValue)>;               // (9)
    using                  SensorAlertCallback                  =
std::function<CheckAlertTimeout(SensorNumber, SensorValue)>;   //
(10)

    }; //namespace sensor
```

В строке 1 объявлен тип для номера датчика, в строке 2 объявлен тип значения, возвращаемого датчиком. В строке 3 объявлен тип значения интервала опроса датчиков для сигнализации пороговых значений.

В строке 4 объявлены идентификаторы типов датчиков, в строке 5 объявлены идентификаторы драйверов. В строке 6 объявлены идентификаторы правил для задания пороговых значений (сигнализация превышения или опускания ниже заданного значения).

В строке 7 объявлен тип для хранения указателей классов датчиков, в строке 8 – тип для хранения указателей классов драйверов. В строке 9 объявлен тип обратного вызова, в который передается значение датчика, в строке 10 – тип обратного вызова, в который передается значение датчика в случае срабатывания сигнализации порогового значения.

6.2.2. Обработка ошибок

В процессе работы любой программы могут ситуации, приводящие к ошибкам. Причины ошибок могут быть самыми различными: неправильные действия пользователя, некорректная работа ПО, сбои в работе оборудования и т. п. Таким образом, возникает необходимость реализации подсистемы обработки ошибок, которая осуществляет восстановление работоспособности компонента после возникновения ошибочной ситуации и уведомление об этом пользователя.

В общем случае существуют две модели обработки ошибок: анализ кодов возврата и использование исключений. Несмотря на то, что использование исключений в последнее время подвергается серьезной критике, вплоть до того, что в новых языках программирования от них избавляются, в C++ указанный механизм остается востребованным, и мы также им воспользуемся. Объявления для формирования исключений представлены в Листинг 87.

Листинг 87. Исключения для обработки ошибок (Errors.h)

```
namespace sensor
{

enum class SensorError: uint32_t  // (1)
{
  NoError = 0,
  NotInitialized = 1,
  UnknownSensorType = 2,
  UnknownSensorNumber = 3,
  SensorIsNotOperable = 4,
  DriverIsNotSet = 5,
  InvalidArgument = 6,
  NotSupportedOperation = 7,
  InitDriverError = 8
};

class sensor_exception : public std::exception    // (2)
{
public:
  sensor_exception(SensorError error);
  SensorError code() const;
  virtual const char* what() const;
```

```
    static void throw_exception(SensorError error);  // (3)

  private:
    SensorError code_;
  };

  }; //namespace sensor
```

В строке 1 объявлены коды возможных ошибок, в строке 2 объявлен класс исключений. Если при выполнении где-то в коде возникает ошибка, то в этом месте нужно вызвать метод, объявленный в строке 3. Указанный метод выбросит исключение с соответствующим кодом.

6.2.3. Драйвер

Драйвер предназначен для взаимодействия с аппаратным обеспечением. Класс, представляющий обобщенный интерфейс для работы с драйвером, приведен в Листинг 88.

Листинг 88. Интерфейс для работы с драйвером (DriverInterface.h)

```
  namespace sensor
  {

  class IDriver
  {
  public:
    virtual void initialize() = 0;                      // (1)
    virtual void activate(SensorNumber number) = 0;     // (2)
    virtual bool isOperable(SensorNumber number) = 0;  // (3)

    virtual SensorValue readSpot(SensorNumber number) = 0;
// (4)
    virtual SensorValue readSmooth(SensorNumber number) = 0;
// (5)
     virtual SensorValue readDerivative(SensorNumber number) =
0;  // (6)

    virtual ~IDriver() = default;

    static DriverPointer createDriver(DriverType type);  // (7)
  };

  }; //namespace sensor
```

В строке 1 объявлен метод для инициализации драйвера. В строке 2 объявлен метод для активации датчика. В строке 3 объявлен метод, возвращающий признак работоспособности датчика. В строках 4, 5 и 6 объявлены методы для чтения соответственно текущих, сглаженных и производных значений. Метод в строке 7 представляет собой фабрику классов, в котором происходит создание класса соответствующего типа.

От общего интерфейса наследуются классы, реализующие драйверы различных типов. В нашей системе реализованы три типа драйверов: драйвер для работы с шиной USB; драйвер для работы через сеть Ethernet; имитируемый драйвер. Диаграмма классов изображена на Рис. 28.

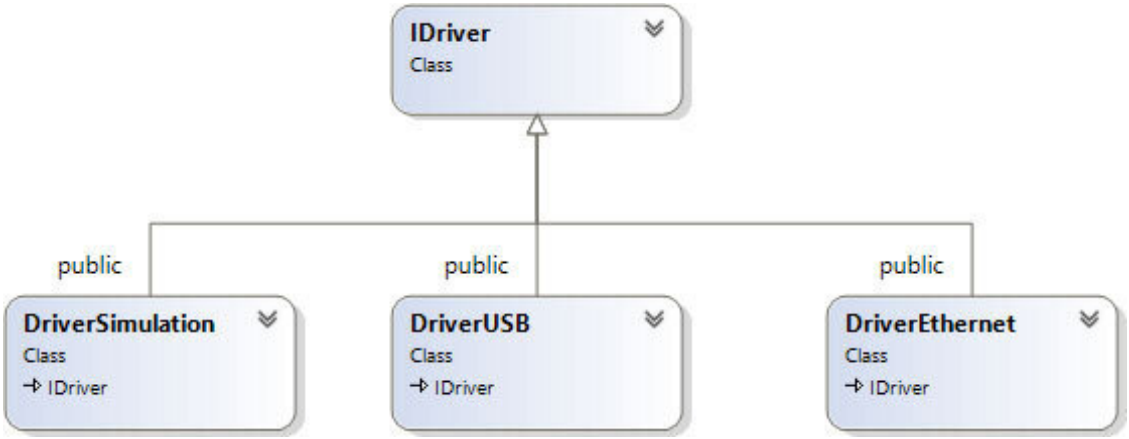

Рис. 28. Диаграмма классов, реализующих драйверы

Драйверы для работы с физическими устройствами формируют команды, посылают их через соответствующие протоколы и возвращают результаты. Реализацию этих драйверов мы рассматривать не будем, поскольку работа с hardware – это отдельная тема, для изучения которой требуется не одна книга. Для нас представляет интерес реализация имитируемого драйвера.

Очевидно, что имитируемый драйвер должен возвращать заранее заданные значения. Самое простое решение, лежащее на поверхности, заключается в том, чтобы хранить эти значения в глобальных или статических переменных и возвращать их в соответствующих методах. Однако в этом случае имитация будет очень примитивной: для всех датчиков будет возвращаться одно и то же значение. Можно хранить свое возвращаемое значение в каждом классе датчика, что больше похоже на работу в реальной системе, но это также не лишено недостатков: мы не можем моделировать изменения показателей в динамике. Лучшим решением было бы предоставить возможность пользователю возвратить значение в момент запроса, для чего нам, конечно же, понадобится обратный вызов. Обратный вызов будет использоваться по схеме «Запрос данных» (см. п. 1.2.1).

Итак, для реализации интерфейса имитируемого драйвера нам понадобятся дополнительные методы и определения (см. Листинг 89).

Листинг 89. Имитируемый драйвер (DriverImpl.h)

```
    class DriverSimulation : public IDriver
    {
    public:
        enum ReadType { READ_SPOT = 0, READ_SMOOTH = 1,
READ_DERIVATIVE = 2 };  // (1)

      using OnReadValue = std::function<SensorValue(SensorNumber,
ReadType)>;  // (2)
                          using          OnOperable          =
std::function<bool(SensorNumber)>;                    // (3)
```

```
    void initialize() override;
    void activate(SensorNumber number) override;
    bool isOperable(SensorNumber number) override;

    void setDefaultValue(SensorValue value);   // (4)
    void setDefaultOperable(bool isOperable);  // (5)
    void setReadValue(OnReadValue value);      // (6)
    void setOperable(OnOperable operable);     // (7)

     SensorValue readSpot(SensorNumber number) override;
// (8)
     SensorValue readSmooth(SensorNumber number) override;
// (9)
    SensorValue readDerivative(SensorNumber number) override; //
(10)

    static IDriver* create();

  protected:
    DriverSimulation();

  private:
    OnReadValue getValue_;          // (11)
    OnOperable getOperable_;        // (12)
    SensorValue defaultValue_ = 0; // (13)
    bool defaultOperable_ = true;  // (14)
  };
```

В строке 1 объявляется перечисление для указания используемого метода чтения показателей. В строке 2 и 3 объявляются типы для обратных вызовов. Переменные соответствующих типов для хранения вызовов объявлены в строках 11 и 12. Настройка вызовов производится в методах 6 и 7. Кроме того, объявляются переменные для хранения значений по умолчанию (строки 13 и 14), эти переменные настраиваются в методах 4 и 5.

Реализацию чтения показателей продемонстрируем на примере получения текущего значения датчика (Листинг 90).

Листинг 90. Чтение текущего значения датчика в имитируемом драйвере (DriverImpl.cpp)

```
  SensorValue DriverSimulation::readSpot(SensorNumber number)
  {
    if (getValue_)  // (1)
    {
      return getValue_(number, READ_SPOT);  // (2)
    }
    else
    {
      return defaultValue_;  // (3)
```

```
    }
}
```

В строке 1 проверяется, настроен ли обратный вызов. Если настроен, то через него запрашивается значение для соответствующего датчика. Информацией вызова здесь является номер датчика и метод чтения показателей (строка 2). Если обратный вызов не настроен, то возвращается значение по умолчанию (строка 3).

6.2.4. Датчик

Обобщенный интерфейсный класс для работы с датчиком приведен в Листинг 91.

Листинг 91. Интерфейсный класс для роботы с датчиком (SensorInterface.h)

```
namespace sensor
{

class ISensor
{
public:
  virtual void setDriver(DriverPointer driverPointer) = 0;  //
(1)
  virtual DriverPointer getDriver() = 0;  // (2)

  virtual double getValue() = 0;  // (3)
  virtual bool isOperable() = 0;  // (4)

  virtual ~ISensor() = default;

      static  SensorPointer  createSensor(SensorType  type,
SensorNumber number, DriverPointer driverPointer);  // (5)

};

}; //namespace sensor
```

В строке 1 объявлен метод для настройки драйвера, с которым будет работать датчик. Получить используемый драйвер можно с помощью метода 2. В строках 3 и 4 объявлены методы для получения текущего значения датчика и определения его работоспособности. В строке 5 объявлен метод для создания экземпляра класса соответствующего типа.

В соответствии с требованиями нам необходимо реализовать датчики, которые бы возвращали текущие, сглаженные и производные значения показателей. Для каждого способа реализован отдельный класс; диаграмма классов изображена на Рис. 29.

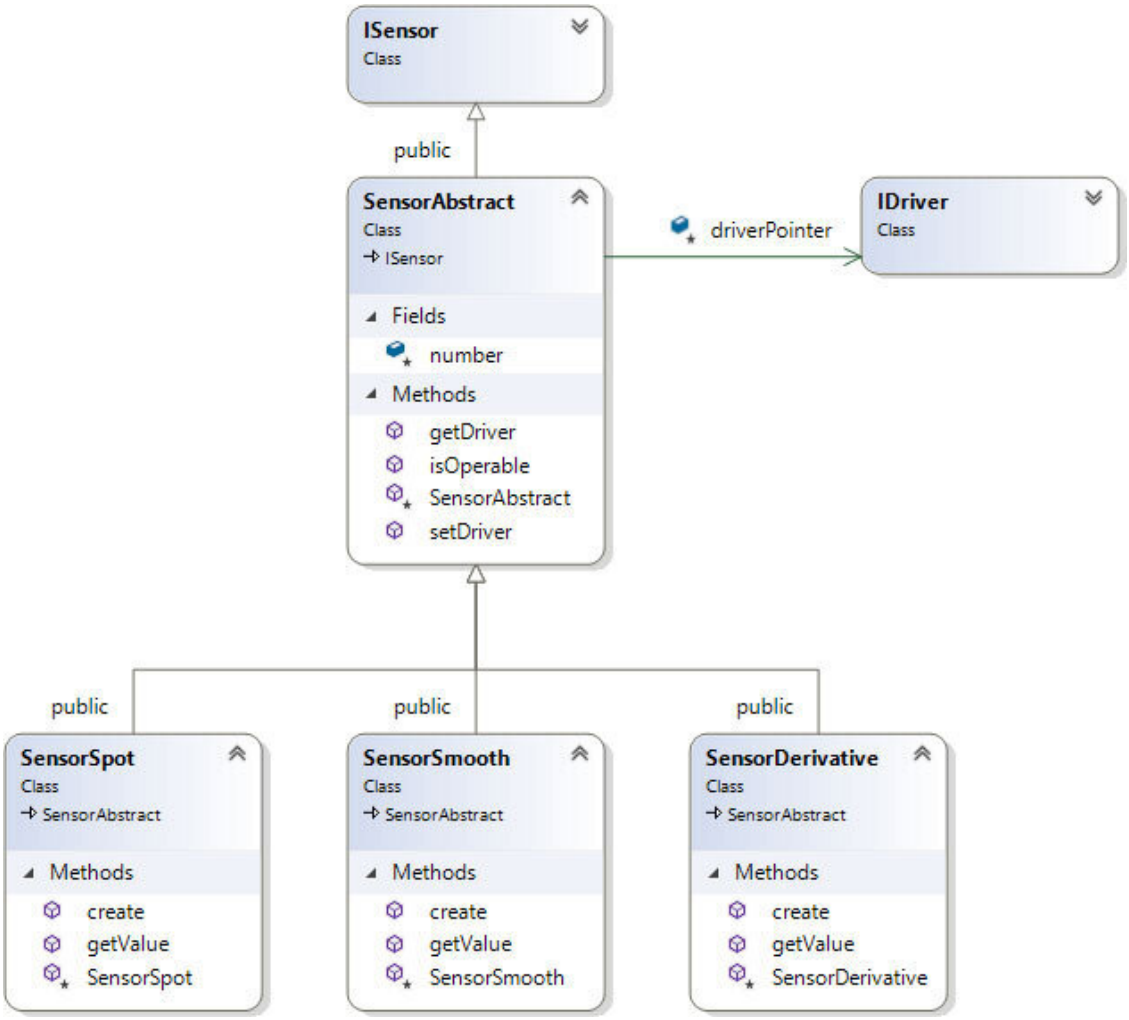

Рис. 29. Диаграмма классов, реализующих управление датчиками

Как видно из диаграммы, при вызове метода для получения значения датчик обращается к драйверу, вызывая соответствующие методы. В зависимости от настроенного драйвера будут возвращаться реальные или имитируемые значения.

6.2.5. Контейнер

Контейнер предназначен для хранения экземпляров классов для управления датчиками. Объявление класса приведено в Листинг 92.

Листинг 92. Объявление контейнера (SensorContainer.h)

```
namespace sensor
{
  class ISensor;

  class SensorContainer
  {
  public:
    void addSensor(SensorNumber number, SensorPointer sensor);  //
```

```
      void deleteSensor(SensorNumber number);                // (2)
       SensorPointer checkSensorExist(SensorNumber number);
// (3)
       SensorPointer findSensor(SensorNumber number);                //

       template<typename CallbackIterate>
       void forEachSensor(CallbackIterate&& callback)  // (5)
       {
         for (auto item : container_)                 // (6)
         {
           callback(item.first, item.second);
         }
       }
  private:
       std::map<SensorNumber, SensorPointer> container_;  // (7)
    };

  };
```

Хранилище объектов реализовано в виде двоичного дерева (строка 7). Ключом здесь выступает номер датчика, содержимым является указатель на класс управления датчиком. Методы для добавления и удаления указателей объявлены в строках 1 и 2.

Метод в строке 3 возвращает указатель на объект класса, если последний с заданным номером содержится в хранилище, в противном случае возвращается нулевой указатель. Метод в строке 4 возвращает указатель на объект класса для соответствующего номера; если объект отсутствует, то генерируется исключение.

Метод 5 предназначен для итерации по всем хранимым объектам. Здесь используется обратный синхронный вызов (см. п. 1.4.1) по схеме «перебор элементов» (см. п. 1.2.3). Реализация осуществляет перебор всех элементов хранилища, для каждого элемента выполняется соответствующий вызов. Метод реализован в виде шаблона, что позволяет его использование для различных типов объектов. Входным параметром метода выступает объект вызова, объявленный как ссылка на r-value. Такое объявление позволяет передавать выражения или временные копии объектов.

6.2.6. Асинхронные запросы

Для реализации асинхронных запросов объявляется очередь, в которую помещаются все поступающие запросы. Обработка очереди происходит в отдельном потоке. Поток извлекает очередной запрос и для него выполняет обратный вызов. Объявление класса для выполнения асинхронных вызовов приведено в Листинг 93.

Листинг 93. Класс для выполнения асинхронных вызовов (CommandQueue.h)

```
class CommandQueue
{
public:
  void start();  // (1)
  void stop();   // (2)
  void addCommand(SensorNumber number, SensorPointer pointer, Senso
```

```
private:
  struct Command  // (4)
  {
    SensorNumber number;
    SensorPointer pointer;
    SensorValueCallback callback;
  };

  std::queue<Command> commandQueue_ ;     // (5)
  std::condition_variable conditional_;  // (6)
  std::mutex mutex_;                      // (7)
  std::thread queueThread_;               // (8)
  bool exit_;                             // (9)

  void readCommand();  // (10)
};
```

В строке 4 объявлена структура, в которой будут храниться данные для выполнения вызова: номер датчика, указатель на класс датчика и объект вызова. В строке 5 объявлен контейнер, который будет хранить указанные структуры. В строках 6 и 7 объявлены переменные для синхронизации операций записи/чтения очереди, в строке 8 объявлен класс для запуска потока обработки очереди, в строке 9 объявлен индикатор для завершения работы потока.

В строке 1 объявлен метод, который запускает поток обработки очереди, в строке 2 объявлен метод для остановки этого потока. Метод, объявленный в строке 3, добавляет переданные данные в очередь путем создания экземпляра структуры 4 и размещения ее в контейнере 5.

Обработка очереди реализована в методе, объявленном в строке 10. Поток обработки очереди вызывает этот метод, который, в свою очередь, ожидает поступления записей и обрабатывает их. Реализация приведена в Листинг 95.

Листинг 94. Обработка очереди запросов (CommandQueue.cpp)

```
void CommandQueue::readCommand()
{
  while (!exit_)  // (1)
  {
    std::unique_lock<std::mutex> lock(mutex_);  // (2)

                              conditional_.wait(lock,       [this]
() {return commandQueue_.size() > 0 || exit_ == true; });  // (3)

     while (commandQueue_.size() > 0 && exit_ == false)     // (4)
      {
       Command cmd = commandQueue_.front();                // (5)
        commandQueue_.pop();                               // (6)
        lock.unlock();                                     // (7)
      cmd.callback(cmd.number, cmd.pointer->getValue());  // (8)
        lock.lock();  // (9)
      }
```

```
    }
}
```

Пока не установлен индикатор завершения (устанавливается в методе **stop**), выполняется цикл 1. Вначале блокируется мьютекс 2 (это необходимо для корректной работы условной переменной), затем осуществляется ожидание условной переменной 3. Когда метод **addCommand** сформировал новую запись и добавил ее в контейнер, он инициирует срабатывание условной переменной, и поток выполнения переходит к циклу 4 (мьютекс при этом оказывается заблокирован). Этот цикл работает, пока очередь не опустеет либо будет установлен индикатор выхода.

В строке 5 из контейнера извлекается очередная запись, в строке 6 эта запись удаляется из контейнера. В строке 7 снимается блокировка мьютекса, что позволяет добавлять в контейнер новые записи, пока идет обработка очередной команды. В строке 8 осуществляется обратный вызов, в строке 9 мьютекс блокируется вновь, и далее повторяется цикл 4.

6.2.7. Наблюдатель

Объявление класса наблюдателя приведено в Листинг 95.

Листинг 95. Наблюдатель – класс для отслеживания пороговых значений (Observer.h)

```
class Observer
{
public:
  void start();  // (1)
  void stop();   // (2)
  void addAlert(SensorNumber number, SensorPointer pointer, SensorA
  void deleteAlert(SensorNumber number);  // (4)
private:

  struct Alert  // (5)
  {
    Alert() {}
    Alert(SensorAlertCallback callback, SensorValue alertValue, Ale
    callback(callback), alertValue(alertValue), alertRule(alertRule
    {
    }

    SensorAlertCallback callback;
    SensorValue alertValue;
    AlertRule alertRule;
    SensorPointer sensor;
    CheckAlertTimeout checkTimeout;
    CheckAlertTimeout currentTimeout;
  };

  std::map<SensorNumber, Alert> containerAlert;  // (6)
  std::thread pollThread_;                       // (7)
```

```
  bool exit_;                                          // (8)
  std::mutex mutex_;                                   // (9)

  void poll();                                         // (10)
};
```

В строке 1 объявлен метод для запуска процесса отслеживания пороговых значений, в строке 2 – метод для останова. Метод в строке 3 добавляет датчик для отслеживания, метод 4 – удаляет.

В строке 5 объявлена структура, в которой хранятся данные, необходимые для отслеживания показаний датчика. В строке 6 объявлен контейнер для хранения указанных структур; метод **addAlert** добавляет запись в контейнер, метод **deleteAlert** удаляет ее. В строке 7 объявлен класс для запуска потока для отслеживания, в строке 8 объявлен индикатор выхода, в строке 9 объявлен мьютекс для синхронизации.

Отслеживание показаний реализовано в методе, объявленном в строке 10. Поток отслеживания вызывает этот метод, который циклически опрашивает назначенные датчики и в случае превышения пороговых значений осуществляет обратный вызов. Реализация приведена в Листинг 96.

Листинг 96. Отслеживание пороговых значений

```
void Observer::poll()
{
  using namespace std::chrono_literals;

  while (!exit_)  // (1)
  {
    std::this_thread::sleep_for(1s);           // (2)
    std::lock_guard<std::mutex> lock(mutex_);  // (3)

    for (auto& item : containerAlert)  // (4)
    {
      Alert& alert = item.second;
      alert.currentTimeout++;          // (5)
                              if     (alert.checkTimeout     !
= 0 && alert.currentTimeout >= alert.checkTimeout)  // (6)
      {
          bool triggerAlert = false;

          if (alert.alertRule == AlertRule::More)  // (7)
          {
                               triggerAlert   =   alert.sensor-
>getValue() > alert.alertValue;
          }
          else               // (8)
          {
                               triggerAlert   =   alert.sensor-
>getValue() < alert.alertValue;
          }
```

```
            if (triggerAlert)  // (9)
            {
                alert.checkTimeout = alert.callback(item.first, alert
            }

            alert.currentTimeout = 0;  // (11)
          }
      }
    }
  }
```

В строке 1 объявлен цикл опроса, который выполняется, пока не выставлен индикатор завершения (выставляется в методе **stop**). В строке 2 поток засыпает на 1 секунду, т. е. интервал опроса равен 1 секунде. В строке 3 блокируется мьютекс, чтобы избежать коллизий добавления/удаления элементов в контейнере.

В строке 4 осуществляется опрос элементов, хранящихся в контейнере. Текущее время опроса в строке 5 увеличивается на единицу. Если уведомление разрешено, о чем говорит ненулевое значение **timeout**, и время последнего опроса превысило назначенное время (строка 6), то тогда проверяется, имелось ли превышение пороговых значений в соответствии с назначенными правилами (строки 6, 7). Если превышение зафиксировано (строка 9), то осуществляется обратный вызов (строка 10). Этот вызов возвращает следующий интервал опроса, после чего текущее время сбрасывается (строка 11).

6.2.8. Интерфейсный класс

Класс, объявляющий интерфейс для взаимодействия с приложением, представлен в Листинг 97.

Листинг 97. Интерфейсный класс (ControlInterface.h)

```
namespace sensor
{

  class ISensorControl
  {
  public:
    virtual ~ ISensorControl () = default;

    virtual void initialize() = 0;  // (1)
    virtual void shutDown() = 0;    // (2)
   virtual void assignDriver(DriverPointer driver) = 0;  // (3)
   virtual DriverPointer getAssignedDriver() = 0;       // (4)
    virtual DriverPointer getSensorDriver(SensorNumber number) = 0;
    virtual void addSensor(SensorType type, SensorNumber number) =
    virtual void deleteSensor(SensorNumber number) = 0;           /
    virtual bool isSensorExist(SensorNumber number) = 0;          /
    virtual bool isSensorOperable(SensorNumber number) = 0;       /
   virtual SensorValue getSensorValue(SensorNumber number) = 0;
// (10)
```

```
    virtual void querySensorValue(SensorNumber number, SensorValueC
    virtual void readSensorValues(SensorValueCallback callback) = 0
    virtual SensorValue getMinValue(SensorNumber first, SensorNumbe
    virtual SensorValue getMaxValue(SensorNumber first, SensorNumbe
    virtual void setAlert(SensorNumber number, SensorAlertCallback
     virtual void resetAlert(SensorNumber number) = 0;  // (16)

      static ISensorControl* createControl();           // (17)
  };
};
```

В строке 1 и 2 объявлены методы для запуска и останова. В строках 3 и 4 объявлены методы для назначения и получения драйвера. Этот драйвер должен быть создан и назначен в самом начале работы, поскольку он будет передаваться новым создаваемым датчикам. Узнать назначенный драйвер для соответствующего датчика можно в методе 5.

В строках 6 и 7 объявлены методы для добавления и удаления датчика. В методе 8 можно проверить, существует ли датчик с переданным номером, в методе 9 можно проверить, является ли датчик работоспособным.

В строке 10 объявлен метод для чтения текущего показания датчика. В методе 11 осуществляется асинхронный запрос показания датчика, значение будет возвращаться через передаваемый обратный вызов. В строке 12 осуществляется опрос показаний всех работоспособных датчиков, значения также возвращаются через обратный вызов. С помощью методов, объявленных в строках 13 и 14, можно получить минимальное и максимальное значение для набора датчиков с номерами из указанного диапазона.

В строке 15 назначается отслеживание пороговых значений, в строке 16 отслеживание выключается. С помощью метода, объявленного в строке 17, можно создать экземпляр соответствующего интерфейсного класса.

Класс, реализующий интерфейс, приведен в Листинг 98.

Листинг 98 Класс, реализующий интерфейс (SensorControl.h)

```
namespace sensor
{

class ISensor;
class IDriver;
class CommandQueue;
class AlertControl;
class SensorContainer;

class SensorControl: public ISensorControl
{
  public:
    SensorControl();
    ~SensorControl();
    void initialize() override;
    /* Other Interface methods - they are not displayed here*/
```

```
  private:
    SensorContainer* sensorContainer_;    // (1)
    CommandQueue* commandQueue_;          // (2)
    AlertControl* alertControl_;          // (3)
    bool isInitialized_;                  // (4)
    DriverPointer driver_;                // (5)

    void checkInitialize();  // (6)
    void checkDriver();      // (7)
};

}; //namespace sensor
```

В строке 1 объявлен контейнер для хранения датчиков, в строке 2 – класс для выполнения асинхронных запросов, в строке 3 – класс для отслеживания пороговых значений. Соответствующие указатели создаются в конструкторе и уничтожаются в деструкторе. Индикатор 4 указывает, была ли выполнена инициализация.

В строке 6 объявлен вспомогательный метод, который проверяет, была ли выполнена инициализация (если нет, выбрасывает исключение). В строке 7 аналогичный метод проверяет, был ли установлен драйвер.

Рассмотрим, как здесь используются обратные вызовы. Для начала самый простой случай – чтение показаний работоспособных датчиков (Листинг 99).

Листинг 99. Обратные вызовы в классе, реализующем интерфейс (SensorControl.cpp)

```
void SensorControl::readSensorValues(SensorValueCallback callback)
{
  checkInitialize();  // (1)

                      sensorContainer_->forEachSensor([callback]
(SensorNumber number, SensorPointer sensor)  // (2)
      {
        if (sensor->isOperable())  // (3)
        {
          callback(number, sensor->getValue());  // (4)
        }
      }
  );
}
```

В строке 1 производится проверка, инициализирован ли класс. Если класс не проинициализирован, то функция выбросит исключение.

В строке 2 происходит перебор элементов контейнера, в качестве обратного вызова используется лямбда-выражение. Контейнер будет вызывать лямбда-выражение, в которое он будет передавать номер датчика и указатель на экземпляр класса. В теле выражения проверяется, является ли датчик работоспособным (строка 3), и если да, то выполняется соответствующий обратный вызов (строка 4).

Рассмотрим теперь поиск максимального и минимального значения для заданного диапазона номеров датчиков. Вначале разработаем вспомогательный класс, который будет последовательно принимать на вход показания датчиков и искать среди них максимальное и минимальное значение (Листинг 100).

Листинг 100. Класс для анализа минимального и максимального значения (SensorControl.cpp)

```
    class FindMinMaxValue
    {
    public:
      enum MinMaxSign { MIN_VALUE = 0, MAX_VALUE = 1 };  // (1)

      FindMinMaxValue(SensorNumber first, SensorNumber last, MinMaxSign
        sign_(sign), first_(first), last_(last), count_(0)
      {
        if (sign == MIN_VALUE)
        {
         result_ = std::numeric_limits<SensorValue>::max();  // (3)
        }
        else
        {
         result_ = std::numeric_limits<SensorValue>::min();  // (4)
        }


  arrayFunMinMax_[MIN_VALUE] = &FindMinMaxValue::CompareMin;  // (5)

  arrayFunMinMax_[MAX_VALUE] = &FindMinMaxValue::CompareMax;  // (6)
      }

                                      void                operator()
(SensorNumber number, SensorPointer sensor)                   // (7)
      {
                                         if         (         sensor-
>isOperable() && (number >= first_ && number <= last_) )  // (8)
        {
                              (this->*arrayFunMinMax_[sign_])(sensor-
>getValue());               // (9)
           count_++;
  // (10)
        }
      }
      SensorValue result() { return result_; }  // (11)
      size_t count() { return count_; }         // (12)
    private:
      SensorNumber first; // (13)
      SensorNumber last;  // (14)
```

```
    MinMaxSign sign;    // (15)
    SensorValue result; // (16)
    size_t count;       // (17)

        using FunMinMax = void (FindMinMaxValue::*)
(SensorValue value);  // (18)

    void CompareMin(SensorValue value)  // (19)
    {
        if (result_ > value)
        {
            result_ = value;
        }
    }

    void CompareMax(SensorValue value)  // (20)
    {
      if (result_ < value)
      {
        result_ = value;
      }
    }

    FunMinMax arrayFunMinMax_[2];       // (21)
  };
```

В строке 2 объявлен конструктор, который принимает на вход следующие параметры: минимальное значение диапазона номеров; максимальное значение диапазона номеров; параметр, указывающий, что необходим поиск минимального либо максимального значения. В конструкторе инициализируются переменные класса: минимальное значение диапазона (объявлено в строке 13); максимальное значение диапазона (объявлено в 14); параметр для поиска (объявлено в 15); итоговый результат (объявлено в 16); количество датчиков, которые участвовали в поиске (объявлено в 17). В зависимости от переданного параметра начальный результат инициализируется соответственно максимальным либо минимальным значением (строки 3 и 4). Кроме того, инициализируется массив указателей на функцию (строки 5 и 6, объявление в 21). Данные функции предназначены для сравнения и запоминания максимального либо минимального значений (объявлены в 19 и 20).

Анализ очередного значения происходит в перегруженном операторе 7. На вход подаются номер датчика и указатель на датчик. Если датчик работоспособный и его номер попадает в заданный диапазон номеров (строка 8), то в зависимости от параметра поиска через указатель вызывается соответствующая функция для анализа (строка 9), а также увеличивается счетчик просмотренных датчиков (строка 10). Функции 11 и 12 возвращают итоговые результаты.

Итак, класс для анализа готов. Теперь можно вызвать метод для итерации по элементам контейнера, и в качестве обратного вызова передать экземпляр соответствующего вспомогательного класса. Метод будет вызывать перегруженный оператор, и таким образом, мы узнаем минимальное либо максимальное значение (Листинг 101).

Листинг 101. Поиск минимального и максимального значений (SensorControl.cpp)

```
SensorValue SensorControl::getMinValue(SensorNumber first, SensorNu
{
  checkInitialize();

  FindMinMaxValue fmv(first, last, FindMinMaxValue::MIN_VALUE);
  sensorContainer_->forEachSensor(fmv);
  return fmv.result();
}

SensorValue SensorControl::getMaxValue(SensorNumber first, SensorNu
{
  checkInitialize();

  FindMinMaxValue fmv(first, last, FindMinMaxValue::MAX_VALUE);
  sensorContainer_->forEachSensor(fmv);
  return fmv.result();
}
```

6.3. Разработка системного API

6.3.1. API как оболочка

Уже после того, как классы модуля были разработаны, протестированы и начали использоваться в системе, появилось новое требование – ввести поддержку системного API. Как известно, в интерфейсах системных API можно использовать только внешние функции и простые структуры данных в стиле C; классы и другие специфические конструкции C++ использовать нельзя (см. п. 1.4.2). Так что же, все теперь придется переписывать? Можно предложить следующее решение: использовать интерфейс API как оболочку для вызова методов класса. Концептуальный пример приведен в Листинг 102.

Листинг 102. Концептуальный пример реализации API как оболочки

```
using ControlPointer = std::unique_ptr<sensor::ISensorControl>;
ControlPointer g_SensorControl(sensor::ISensorControl::createContro

void initialize () // This function is declared in the header file
{
  g_SensorControl->initialize();
}
```

Однако не все так просто, перед нами встают следующие проблемы.

1. В исходной реализации мы использовали специфические типы C++, такие, как std::function, smart pointers и т. п., что не допускается в интерфейсах системных API. Какие типы использовать взамен?

2. Для обработки ошибок в исходной реализации мы использовали исключения. Как сейчас обрабатывать ошибки, ведь в интерфейсах API исключения недопустимы?

3. В исходной реализации мы в каждом потоке могли объявить отдельный интерфейсный класс и работать с ним независимо от остальных потоков. Как теперь обеспечить многопоточную работу, ведь отдельные потоки вызывают одни и те же интерфейсные функции?

4. В исходной реализации драйвер настраивался путем создания нового класса и передаче его в интерфейсный класс. Как теперь настраивать драйвер, если в интерфейсах API нельзя использовать классы?

5. Как организовать обратные вызовы?

Рассмотрим, как эти проблемы можно решить.

6.3.2. Объявления типов

В исходной реализации общие типы объявлены в *SensorDef.h*, но мы не можем просто перенести их в интерфейс API из-за использования специфических конструкций C++. Поэтому нам придется повторить эти объявления в стиле C с использованием простых типов, которые можно будет использовать в интерфейсных функциях. Объявления представлены в Листинг 103.

Листинг 103. Объявления типов для интерфейса API (SensorLib.h)

```
    #ifdef _WINDOWS  // (1)
      #ifdef LIB_EXPORTS
        #define LIB_API __declspec(dllexport)
      #else
        #define LIB_API __declspec(dllimport)
      #endif
      #else
        #define LIB_API
    #endif

    typedef uint32_t SensorNumber;       // (2)
    typedef double SensorValue;          // (3)
    typedef uint32_t CheckAlertTimeout;  // (4)

    typedef uint32_t SensorType;         // (5)
    typedef uint32_t DriverType;         // (6)
    typedef uint32_t AlertRule;          // (7)

    typedef                              void(*SensorValueCallback)
(SensorNumber, SensorValue, void*);                // (8)
    typedef               CheckAlertTimeout(*SensorAlertCallback)
(SensorNumber, SensorValue, void*);  // (9)
    typedef                      SensorValue(*OnSimulateReadValue)
(SensorNumber, int, void*);                  // (10)
    typedef               int                (*OnSimulateOperable)
(SensorNumber, void*);                              // (11)

    enum eSensorType  // (12)
    {
      SENSOR_SPOT = 0,
      SENSOR_SMOOTH = 1,
      SENSOR_DERIVATIVE = 2,
    };

    enum eDriverType  // (13)
    {
      DRIVER_SIMULATION = 0,
      DRIVER_USB = 1,
      DRIVER_ETHERNET = 2
    };

    enum  eAlertRule  // (14)
    {
      ALERT_MORE = 0,
      ALERT_LESS = 1
    };
```

В строке 1 объявлены определения для экспортируемых функций. Эти объявления необходимы для компиляции динамической библиотеки в среде Windows, для других платформ они неактуальны.

В строках 2–4 объявлены типы, которые будут использоваться для входных параметров интерфейсных функций. Это те же объявления, которые использовались в исходной реализации (SensorDef.h, см. п. 6.2.2).

В строках 5–7 вместо перечислений C++ объявляются простые числовые типы. В экспортируемых функциях нежелательно использовать перечисления как типы входных параметров, потому что размер этих типов в C явно не определен. Вместо этого перечисления используются в качестве числовые констант, они объявлены соответственно в строках 12–14.

В строках 8–11 объявлены типы указателей на функцию для выполнения обратных вызовов. Как видим, в отличие от исходной реализации здесь присутствует дополнительный параметр для указания контекста вызова.

6.3.3. Интерфейс API и обработка ошибок

Исходя из концепции «API как оболочка», сигнатура интерфейсных функций API должна повторять сигнатуру методов интерфейсного класса. Однако здесь мы сталкиваемся с некоторыми проблемами, одна из которых – это обработка ошибок.

В исходной реализации мы обрабатывали ошибки с помощью исключений. Теперь исключения использовать нельзя, в системных API они недопустимы. Тем не менее, вызываемая функция должна как-то уведомить о возникновении ошибки, для чего могут использоваться следующие способы:

1) функция возвращает результат, для которого некоторое предопределенное значение говорит о том, что произошла ошибка. Код ошибки возвращается с помощью отдельного вызова;

2) код ошибки возвращается через дополнительный параметр функции;

3) все функции возвращают результат выполнения, который является кодом ошибки.

Ни один способов не является идеальным, каждый имеет свои достоинства и недостатки. Так, в первом способе возникают сложности, если результат, возвращаемый функцией, не имеет значений, которые недопустимы и могут сигнализировать о возникновении ошибки[36]. Во втором способе для всех вызовов придется использовать дополнительную переменную – код ошибки, даже если он нас не интересует. В третьем способе, если функция возвращает результат, то для него приходится использовать отдельный входной параметр, что не всегда удобно.

В нашем случае мы выберем третий способ, исходя из следующих соображений: объявления функций будут выглядеть единообразно; возникновение ошибки можно узнать непосредственно в момент вызова, (например, в операторе if); если функция не возвращает значений, то ей не нужно передавать никакие дополнительные параметры. Объявления интерфейсных функций с возвратом ошибок представлены в Листинг 104.

Листинг 104. Интерфейс системного API (SensorLib.h)

```
typedef unsigned int ErrorCode;
```

[36] Например, функция возвращает только положительные значения, в этом случае можно считать, что отрицательное значение сигнализирует о возникновении ошибки. Но если функция может возвращать значения с любым знаком, то неясно, какое из них назначить индикатором ошибки.

```
LIB_API ErrorCode initialize();
LIB_API ErrorCode shutDown();
LIB_API ErrorCode assignDriver(DriverType type);
LIB_API ErrorCode getAssignedDriver(DriverType* type);
LIB_API ErrorCode getSensorDriver(SensorNumber number, DriverType*
LIB_API ErrorCode addSensor(SensorType type, SensorNumber number);
LIB_API ErrorCode deleteSensor(SensorNumber number);
LIB_API ErrorCode isSensorExist(SensorNumber number, int* isExist);
LIB_API ErrorCode isSensorOperable(SensorNumber number, int* isOper
LIB_API ErrorCode getSensorValue(SensorNumber number, SensorValue*
LIB_API ErrorCode querySensorValue(SensorNumber number, SensorValue
LIB_API ErrorCode readSensorValues(SensorValueCallback callback, vo
LIB_API ErrorCode getMinValue(SensorNumber first, SensorNumber last
LIB_API ErrorCode getMaxValue(SensorNumber first, SensorNumber last
LIB_API ErrorCode setAlert(SensorNumber number, SensorAlertCallback
LIB_API ErrorCode resetAlert(SensorNumber number);
LIB_API ErrorCode setSimulateReadCallback(OnSimulateReadValue callb
LIB_API ErrorCode setSimulateOperableCallback(OnSimulateOperable ca
```

В реализации этих функций мы будем возвращать код ошибки, получая его из перехваченного исключения. В качестве примера рассмотрим реализацию функции для получения значения датчика (Листинг 105).

Листинг 105. Функция для получения значения датчика

```
ErrorCode getSensorValue(SensorNumber number, SensorValue* value)
{
  ErrorCode error = ERROR_NO;  // (1)

  try
  {
    *value = g_SensorControl->getSensorValue(number);  // (2)
  }
  catch (sensor::sensor_exception& e)  // (3)
  {
    error = e.code();                  // (4)
  }
  return error;                        // (5)
}
```

В строке 1 объявляем переменную – код возврата. В строке 2 осуществляем вызов метода класса, который заключен в блок try. В строке 3 осуществляется перехват исключения, в строке 4 присваивается код ошибки, который возвращается в строке 5.

Итак, мы придумали, как в интерфейсных функциях осуществлять обработку ошибок. Теперь перед нами встает следующая проблема: как настраивать типы драйверов, ведь в исходной реализации для этого используются классы? Прежде чем перейти к решению этой задачи,

остановимся на реализации многопоточной работы, поскольку используемые там конструкции нам понадобятся в дальнейшем.

6.3.4. Многопоточная работа

В исходной реализации в каждом потоке мы могли создать свой экземпляр класса **ISensorControl** и работать с ним независимо. В случае API это не работает, потому что экземпляр класса в реализации интерфейса объявляется глобальным, и все интерфейсные функции обращаются к одному и тому же экземпляру класса. Выходом здесь будет выделение отдельной области памяти для экземпляра класса в рамках одного потока, т. е. использование локальной памяти потока.

До появления стандарта C++ 11 использовать локальную память потока было непросто: для этого требовалось явное обращение к функциям операционной системы, что усложняло реализацию и делало код платформенно-зависимым. В C++ 11 появилось ключевое слово **thread_local**, и это сильно упростило жизнь: если в объявлении переменной добавить указанный спецификатор, то она становится локальной в рамках потока, т. е. каждый новый создаваемый поток будет иметь независимый экземпляр соответствующей переменной. Таким образом, достаточно экземпляр интерфейсного класса **ISensorControl** объявить как **thread_local**, и теперь для каждого потока будет существовать отдельный независимый экземпляр класса (Листинг 106).

Листинг 106. Объявление экземпляра класса как локального для текущего выполняемого потока (SensorLib.cpp)

```
using ControlPointer = std::unique_ptr<sensor::ISensorControl>;
thread_local ControlPointer g_SensorControl(sensor::ISensorControl:
```

6.3.5. Настройка драйвера

В исходной реализации в начале работы мы создавали необходимый класс драйвера, который затем передавали интерфейсному классу (Листинг 107). Но в интерфейсах системных API мы классы использовать не можем, как поступить в этом случае? Можно предложить следующее решение: класс драйвера создавать внутри API, а в функцию настройки передавать идентификатор, в соответствии с которым будет создан соответствующий драйвер (Листинг 108).

Листинг 107. Настройка драйвера в исходной реализации

```
ISensorControl sensorControl = ISensorControl::createControl;
DriverPointer driver = IDriver::createDriver(DRIVER_SIMULATION);
driver->initialize();
sensorControl->assignDriver(driver);
```

Листинг 108. Настройка драйвера в системном API (SensorLib.h)

```
thread_local sensor::DriverPointer g_DriverSimulation;  // (1)
thread_local sensor::DriverPointer g_DriverUSB;         // (2)
```

```
    thread_local sensor::DriverPointer g_DriverEthernet;    // (3)

    void CreateDriver(sensor::DriverType driverType, sensor::DriverPoin
    {
      if (!driverPointer)
      {
       driverPointer = sensor::IDriver::createDriver(driverType);
        driverPointer->initialize();
      }

      g_SensorControl->assignDriver(driverPointer);
    }

    ErrorCode assignDriver(DriverType driverType)  // (5)
    {
      ErrorCode error = ERROR_NO;

      try
      {
        EnumConverter<sensor::DriverType> conv;
                                                        conv.convert
(driverType, {sensor::DriverType::Simulation, sensor::DriverType::Usb,

        if (conv.error())
        {
          return ERROR_INVALID_ARGUMENT;
        }
        switch (conv.result())  // (7)
        {
          case sensor::DriverType::Simulation:
          {
            CreateDriver(sensor::DriverType::Simulation, g_DriverSimula
          }
          break;

          case sensor::DriverType::Usb:
          {
            CreateDriver(sensor::DriverType::Usb, g_DriverUSB);
          }
          break;

          case sensor::DriverType::Ethernet:
          {
            CreateDriver(sensor::DriverType::Ethernet, g_DriverEthernet
          }
          break;
        }
      }
      catch (sensor::sensor_exception& e)
```

```
    {
      error = static_cast<ErrorCode>(e.code());
    }

    return error;

  }
```

В строках 1–3 объявляются указатели для хранения классов всех возможных типов драйверов. В строке 4 объявлена вспомогательная функция для создания драйвера. Эта функция проверяет, создан ли драйвер соответствующего типа, при необходимости создает, инициализирует и передает его в интерфейсный класс.

В строке 5 приведена реализация интерфейсной функции для настройки драйвера. В строке 6 конвертируется переданное числовое значение в перечисление C++ (будет рассмотрено ниже). В строке 7 объявлен оператор **switch**, в котором анализируется полученное значение перечисления, и вызывается вспомогательная функция с соответствующими параметрами.

В функции API для задания типа драйвера используются числовые значения, а в интерфейсном классе используются перечисления C++. Для того, чтобы сконвертировать числовое значение в перечисление, используется вспомогательный класс **EnumConverter** (Листинг 109)

Листинг 109. Конвертер числовых значений в перечисление (EnumConverter.h)

```
template <typename Enum>  // (1)
class EnumConverter
{
public:
  template<typename ConvValueType>
  void convert(ConvValueType value, std::initializer_list<Enum> lis
  {
      isError_ = true;
      for (Enum item : list)                              // (4)
      {
        if (static_cast<ConvValueType>(item) == value)  // (5)
          {
             result_ = item;                              // (6)
              isError_ = false;
              break;
          }
      }
  };

  bool error() const { return isError_; }
  Enum result() const { return result_; }
private:
  bool isError_;
  Enum result_;
};
```

В строке 1 объявлен шаблонный класс, параметром которого является тип перечисления. Конвертация происходит в функции 2, которая объявлена в виде шаблона, параметром шаблона является тип числового значения для конвертации. Функция принимает число, которое должно быть сконвертировано, а также список значений перечисления (строка 3). Реализация пробегает по всем элементам списка (строка 4) и, если какой-то из элементов списка перечисления равен переданному значению, запоминает это значение перечисления в качестве результата (строки 5,6).

6.3.6. Обратные вызовы

Касательно обратных вызовов мы имеем следующую ситуацию. В системном API контекст вызова передается с помощью указателей на данные, по-другому организация передачи контекста здесь невозможна (см. п. 2.1.2). В интерфейсном классе указатель на данные не используется, поскольку в C++ имеется множество гораздо более изящных способов передачи контекста. Вот тут-то нам и понадобится перенаправление вызовов (см. п. 4.6.2). Реализация одной из интерфейсных функций API, использующей перенаправление вызовов, приведена в Листинг 110.

Листинг 110. Перенаправление вызовов в реализации интерфейсной функции (SensorLib.cpp)

```
    ErrorCode readSensorValues(SensorValueCallback callback, void* pCon
    {
      ErrorCode error = ERROR_NO;

      try
      {
        using namespace std::placeholders;
                                                  g_SensorControl-
>readSensorValues(std::bind(callback,_1,_2,pContextData));   // (1)
      }
      catch (sensor::sensor_exception& e)
      {
        error = e.code();
      }
      return error;
    }
```

В общем-то, вся реализация заключается в вызове метода интерфейсного класса (строка 1), в который вместо непосредственно обратного вызова передается объект связывания. Функция обратного вызова, объявленная в интерфейсе API, принимает 3 входных параметра: номер датчика, значение датчика и указатель на контекст. Когда будет происходить обратный вызов, то объект связывания вызовет назначенную функцию, в которую передаст первые два параметра исходной функции, а в третий параметр будет передан переданный указатель на контекст.

6.4. Итоги

На примере разработки модуля управления датчиками кратко описаны типовые этапы проектирования: описание технического задания; оформление сценариев функционирования системы; декомпозиция и формирование архитектуры. Затем рассмотрена реализация классов с акцентом на использовании обратных вызовов. И в заключение показан процесс создания системного API и трудности, с которыми сталкивается разработчик при реализации концепции «API как оболочка». Как можно увидеть в рассмотренном примере, в практике разработки ПО существует множество ситуаций, когда целесообразно использовать обратные вызовы как элементы дизайна компонентов системы.

Заключение

Итак, наше повествование подходит к концу, пора подвести некоторые итоги.

Обратный вызов – это паттерн, в котором какой-либо исполняемый код как аргумент передается в другой код. Ожидается, что через сохраненный аргумент исполняемый код будет запущен в какой-то момент времени. Типовые задачи, решаемые с помощью обратных вызовов, следующие: запрос данных; вычисления по запросу; перебор элементов; уведомления о событиях.

В C++ обратные вызовы реализуются с помощью следующих механизмов: указатель на функцию; указатель на статический метод класса; указатель на метод-член класса; функциональный объект; лямбда-выражение. Все они имеют свои достоинства и недостатки, и нельзя однозначно сказать, какой является наилучшим, все зависит от поставленных задач и требований к проектируемой системе.

Чтобы определить реализацию, наиболее подходящую для конкретной ситуации, целесообразно использовать метод интегральных оценок, который предлагает простые и эффективные процедуры выбора оптимального решения. Указанный метод можно использовать не только применительно к обратным вызовам, но также и в других случаях, когда необходимо выбрать наиболее подходящее архитектурное решение из множества возможных.

Шаблоны, основанные на принципах параметрического полиморфизма, открывают разработчику новые горизонты. С их помощью появляется возможность создавать обобщенный код, т. е. код, независимый от данных. Применительно к обратным вызовам шаблоны позволяют без особых усилий обеспечить требования, которые крайне сложно, а иногда и невозможно реализовать обычными средствами C++. К ним относятся возможность хранения аргумента настраиваемого типа (универсальный аргумент), настройка сигнатуры вызова, выполнение вызовов для набора аргументов различных типов и т. п.

Платой за использование шаблонов является сложность анализа кода, увеличение времени компиляции, склонность к разрастанию кода, необходимость тестирования на наборах данных различных типов. Кроме того, шаблоны не предоставляют предварительно откомпилированного кода, что делает невозможным их использование в интерфейсах API.

Уникальная особенность обратных вызовов проявляется в том, что они дают возможность динамической модификации поведения программы во время выполнения. Это обуславливает их широкое применение на практике, как было продемонстрировано в примере разработки модуля управления датчиками.

Список литературы и интернет-источников

Здесь приводится список упомянутых книг, а также литература и интернет-ресурсы для углубленного изучения рассмотренных тем.

1. Басс Л., Клементс П., Кацман Р. Архитектура программного обеспечения на практике. Спб, Питер, 2006. – 574 с.

Фундаментальное введение в теорию и практику построения программной архитектуры систем. Приведены методики сравнительного анализа архитектурных решений.

2. Вандевурд Д., Джосаттис Н., Грегор Д. Шаблоны C++. Справочник разработчика. Спб, Альфа-книга, 2018. – 848 с.

Максимально полно охватывает разнообразные аспекты использования шаблонов в C++, подходит как как в качестве справочного, так и учебного пособия.

3. Галовиц Я. C++ 17 STL. Стандартная библиотека шаблонов. Спб., Питер, 2018. – 432 с.

Отличная книга для изучения стандартной библиотеки STL.

4. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. «Приемы объектно-ориентированного проектирования. Паттерны проектирования». Спб, Питер, 2020. – 368 с.

Рассматриваются наиболее употребительные паттерны проектирования и их использование в решении задач. Не привязана к конкретному языку программирования. Отличается легкостью и доступностью изложения.

5. Касперски Крис. Техника оптимизации программ. Эффективное использование памяти. Спб, БХВ-Петербург, 2003. – 560 с.

Подробно рассмотрены принципы профилирования и оптимизации программ. Несмотря на то, что книга выпущена довольно давно, подходы, описанные в ней, остаются актуальными до сих пор.

6. Леоненков А. В. Самоучитель UML 2. СПб, БХВ-Петербург, 2007. – 576 с.

Простое и доступное изложение основ UML.

7. Орлов С. А. Программная инженерия. Технологии разработки программного обеспечения. Спб, Питер, 2018. – 640 с.

Рассматривается методология разработки программного обеспечения, организации и процессы проектирования больших программных систем.

8. Пикус Ф. Г. Идиомы и паттерны проектирования в современном C++. М, ДМК Пресс, 2020. – 452 с.

Рассматриваются реализации различных паттернов проектирования с использованием современных средств C++. Книга достаточно сложная, предполагается, что читатель хорошо владеет C++, имеет опыт обобщенного программирования.

9. Пирс Бенджамин. Типы в языках программирования. М., Лямбда-пресс, 2011. – 656 с.

Академическое изложение теории типов, довольно сложный математический аппарат. Книга скорее ориентирована на теорию, чем на практическое применение, но есть интересные темы о классификации типов и видах полиморфизма.

10. Эванс Эрик. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. М., Вильямс, 2011. – 448 с.

Описаны фундаментальные принципы трансляции предметной области в программные модели, чем, в общем-то, в той или иной степени занимается каждый разработчик.

11. Lambda Expressions in C++.
https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=vs-2019

Кратко и наглядно описан синтаксис лямбда-выражений.

12. Template Specialization (C++).

https://docs.microsoft.com/en-us/cpp/cpp/template-specialization-cpp?view=vs-2019

Множество развернутых примеров, демонстрирующих использование частичной специализацию шаблонов.