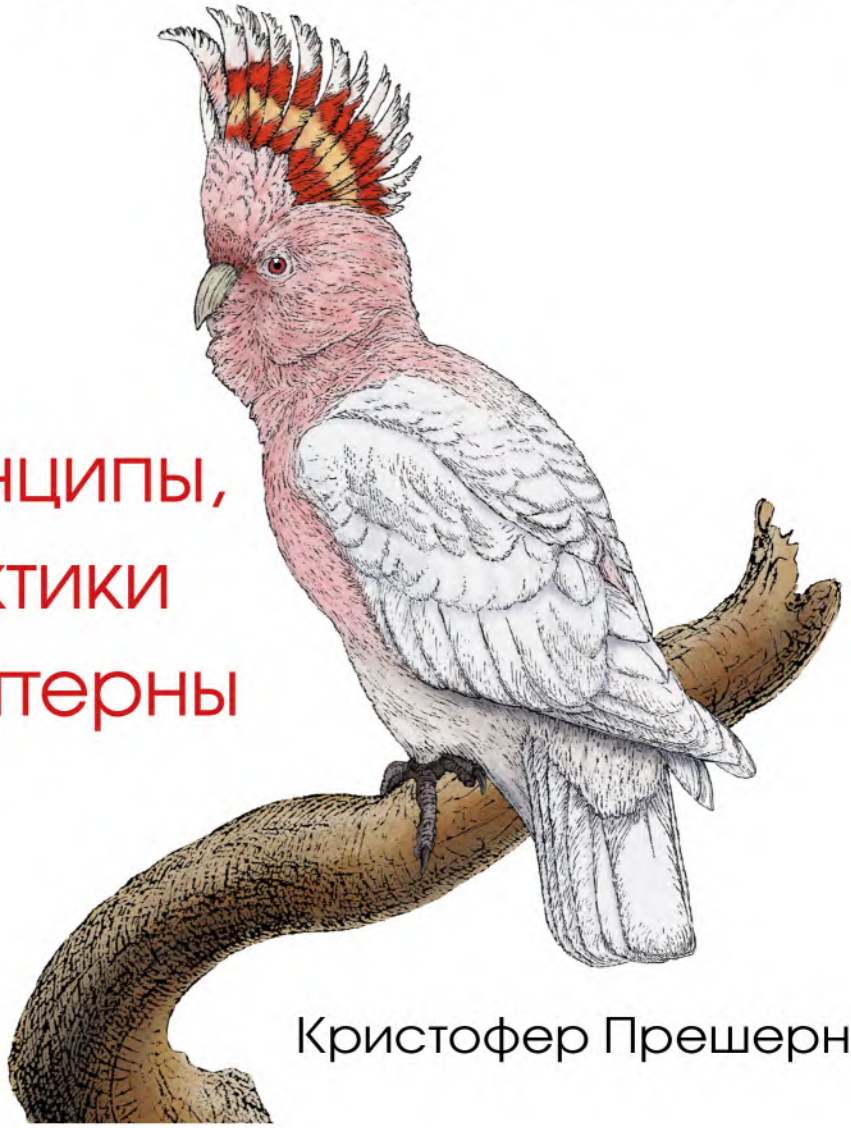
O'REILLY®

Язык C. Мастерство программирования

Принципы, практики и паттерны



Кристофер Прешерн

BOOKS.KZ

Прешерн К.

Язык С
Мастерство программирования

Принципы, практики и паттерны

Fluent C

Principles, Practices, and Patterns

Christopher Preschern

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Язык C

Мастерство программирования

Принципы, практики и паттерны

Прешерн К.

BOOKS.KZ
2023

УДК 004.4
ББК 32.372
П71

П71 Прешерн К.

Язык С. Мастерство программирования. Принципы, практики и паттерны / пер. с англ. А. Н. Слинкина – М.: ДМК Пресс, 2023. – 300 с.: ил.

ISBN 978-6-01810-340-7

В этом практическом руководстве начинающие и опытные программисты на С найдут наставления по принятию проектных решений, включая пошаговое применение паттернов к сквозным примерам.

Автор, один из ведущих членов сообщества паттернов проектирования, объясняет, как организовать программу на С, как обрабатывать ошибки и проектировать гибкие интерфейсы. В части I вы научитесь реализовывать проверенные практикой подходы к программированию на языке С; часть II показывает, как паттерны программирования на С применяются к реализации более крупных программ.

ISBN 978-1-49210-973-3 (англ.)
ISBN 978-6-01810-340-7 (казах.)

Оглавление

Предисловие

Вы купили эту книгу, чтобы поднять свои навыки программирования на новый уровень. И это правильно, потому что вам, безусловно, пригодятся излагаемые в ней практические знания. Если у вас имеется большой опыт программирования на C, то вы в деталях узнаете, как принимаются хорошие проектные решения и какие у них есть плюсы и минусы. Если вы только начинаете знакомиться с C, то найдете здесь руководство по принятию решений и на примерах кода увидите, как эти решения применяются для построения больших программ.

В книге есть ответы на вопросы о том, как структурировать C-программу, как обрабатывать ошибки и как проектировать гибкие интерфейсы. Когда вы больше узнаёте о программировании на C, начинают возникать разные вопросы, например:

- следует ли возвращать имеющуюся информацию об ошибке?
- следует ли использовать для этой цели глобальную переменную `errno`?
- что лучше: немного функций с большим числом параметров или наоборот?
- как построить гибкий интерфейс?
- как реализовать базовые вещи, например итератор?

Для объектно ориентированных языков на большую часть этих вопросов почти исчерпывающий ответ дает книга «банды четырех»: *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software»*[1]. Паттерны проектирования дают программисту проверенные опытом рекомендации, как должны взаимодействовать между собой объекты и как они связаны отношением владения. Кроме того, они показывают, как следует группировать объекты.

Однако на процедурных языках типа C большинство этих паттернов проектирования невозможно реализовать так, как описано «бандой четырех». В C нет встроенных объектно ориентированных механизмов. Наследование или полиморфизм можно эмулировать, но это не лучшее решение, потому что такой код будет непонятен программистам, привыкшим к программированию на C, но не владеющим программированием на объектно ориентированных языках типа C++ и незнакомым с использованием таких концепций, как наследование и полиморфизм. Такие программисты хотели бы придерживаться стиля программирования на C, к которому привыкли. Однако к нему применимы не все объектно ориентированные рекомендации или, по крайней мере, конкретная реализация идеи паттерна проектирования не годится для не объектно ориентированного языка.

[1] *Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес.* «Паттерны объектно ориентированного проектирования». Питер, 2022.

Итак, ситуация выглядит следующим образом: мы хотим писать на C, но не можем напрямую использовать большую часть знаний, документированных в виде паттернов проектирования. В этой книге показано, как преодолеть разрыв и практически реализовать эти знания на языке программирования C.

Зачем я написал эту книгу

Теперь я хочу рассказать, почему знания, собранные в этой книге, оказались столь важными для меня и почему их так трудно отыскать.

В школе я изучал C в качестве первого языка программирования. Как и любой начинающий программист на C, я удивлялся, почему нумерация элементов массива начинается с 0, и наугад пытался поместить операторы `*` и `&` в нужное место, чтобы заставить работать магию указателей.

В университете я узнал, как в действительности работают синтаксические конструкции C и как они транслируются в аппаратные биты и байты. Вооруженный этими знаниями, я смог писать небольшие программы, которые работали очень хорошо. Но я по-прежнему не понимал, почему более длинный код выглядит именно так, а не иначе, и, уж конечно, не мог сам придумать решения вроде:

```
typedef struct INTERNAL_DRIVER_STRUCT* DRIVER_HANDLE;
typedef void (*DriverSend_FP)(char byte);
typedef char (*DriverReceive_FP)();
typedef void (*DriverIOCTL_FP)(int ioctl, void* context);

struct DriverFunctions
{
  DriverSend_FP fpSend;
  DriverReceive_FP fpReceive;
  DriverIOCTL_FP fpIOCTL;
};

DRIVER_HANDLE driverCreate(void* initArg, struct DriverFunctions f);
void driverDestroy(DRIVER_HANDLE h);
void sendByte(DRIVER_HANDLE h, char byte);
char receiveByte(DRIVER_HANDLE h);
void driverIOCTL(DRIVER_HANDLE h, int ioctl, void* context);
```

При изучении этого кода возникает много вопросов:

- зачем нужны указатели на функции в `struct`?
- зачем функциям нужен этот `DRIVER_HANDLE`?
- что такое IOCTL и почему бы не написать вместо этого отдельные функции?
- зачем нужны явные функции создания и уничтожения?

Эти вопросы появились, когда я начал писать производственные приложения.

Я то и дело сталкивался с ситуациями, когда понимал, что мне не хватает знаний о C; например, как реализовать итератор или как обрабатывать ошибки в функциях. Я осознавал, что синтаксис-то я освоил, но понятия не имею, как им правильно воспользоваться. Я пытался чего-то добиться, но все получалось коряво или не получалось вовсе. Мне были необходимы рекомендации, показывающие, как решать конкретные задачи на языке C. Например:

- как проще всего захватывать и освобождать ресурсы?
- можно ли использовать `goto` для обработки ошибок?
- следует ли сразу проектировать интерфейс гибким или лучше изменять его, когда возникнет необходимость?
- следует ли использовать макрос `assert`, или нужно возвращать код ошибки?
- как реализовать итератор на C?

Для меня оказалось неожиданным открытием, что, хотя у моих опытных коллег было много различных ответов на эти вопросы, никто не смог направить меня туда, где такие проектные решения были документированы вместе с описанием их плюсов и минусов.

Поэтому я обратился к интернету и снова испытал удивление: оказалось очень трудно найти убедительные ответы на эти вопросы, хотя язык C существует уже не один десяток лет. Я обнаружил, что, несмотря на изобилие литературы по основам и синтаксису языка C, нет почти ничего о продвинутом программировании и о том, как писать на C красивый код, который выдержит испытание производственным приложением.

И вот тут в игру вступает эта книга. Она поможет вам отточить свои навыки программирования на C и перейти от простеньких программок к большим системам, в которых ошибки обрабатываются должным образом и которые обладают достаточной гибкостью, чтобы быть готовыми к будущим изменениям требований и проекта. В этой книге используется концепция паттернов проектирования, чтобы познакомить вас со всеми шагами принятия решений и оценкой их достоинств и недостатков. Эти паттерны применяются к сквозным примерам когда, чтобы показать, как код эволюционирует и почему принимает именно такую, а не иную конечную форму.

Основы паттернов

Рекомендации по проектированию в этой книге приводятся в форме паттернов. Идея представлять знания и передовые практики в виде паттернов исходит от архитектора Кристофера Александра, который высказал ее в книге «The Timeless Way of Building» (Oxford University Press, 1979). Он использует небольшие проверенные временем фрагменты для решения важнейшей проблемы в своей области: как проектировать и возводить города. Подход на основе паттернов переняли разработчики программного обеспечения, и теперь проводятся конференции типа Pattern Languages of Programs (PLoP), имеющие целью расширить наши знания о паттернах. В особенности книга «банды четы-

рех» «Design Patterns: Elements of Reusable Object-Oriented Software» (Prentice Hall, 1997) оказала значительное влияние и познакомила разработчиков ПО с концепцией паттернов проектирования.

Но что же такое паттерн? Определений много, и, если эта тема вас сильно интересует, почитайте книгу Frank Buschmann et al. «Pattern-Oriented Software Architecture: On Patterns and Pattern Languages» (Wiley, 2007), где приведены точные описания и детали. А для наших целей достаточно считать, что паттерн дает проверенное временем решение какой-то практической задачи. Представленные в этой книге паттерны имеют структуру, описанную в табл. P.1.

Таблица P.1. Структура паттернов, представленных в этой книге

Часть паттерна	Описание
Название	Это легко запоминаемое имя паттерна. Предполагается, что программисты будут использовать его в повседневном общении (как в случае паттернов из книги «банды четырех», когда можно услышать, например, такую фразу: «И абстрактная фабрика создает объект»). Названия паттернов в этой книге начинаются с заглавной буквы
Контекст	Определяет обстановку, в которой действует паттерн. Говорит, при каких условиях паттерн можно применить
Проблема	Сообщает информацию о задаче, которую мы хотим решить. Эта часть начинается с основной постановки, выделенной полужирным шрифтом, после чего детально описывается, почему данную проблему трудно решить. (В других форматах паттернов детали вынесены в отдельную часть, называемую «Движущие силы».)
Решение	В этой части приводятся рекомендации по решению проблемы. В начале формулируется основная идея решения, выделенная полужирным шрифтом, а затем следуют детали. Также включен пример кода, содержащий конкретную реализацию
Последствия	В этом разделе перечисляются плюсы и минусы описанного решения. Применяя паттерн, вы должны быть уверены, что последствия вас устраивают
Известные примеры использования	Примеры использования убеждают в том, что предложенное решение действительно работает в реальных приложениях. Кроме того, они дают конкретные примеры, помогающие понять, как применяется паттерн

Основное преимущество представления рекомендаций в виде паттернов заключается в том, что паттерны можно применять один за другим. Для крупной проблемы проектирования трудно найти конкретную рекомендацию и конкретное решение именно этой проблемы. Вместо этого большая и весьма специфическая проблема разбивается на много меньших и более общих проблем, а затем эти проблемы решаются по очереди путем применения раз-

личных паттернов. Мы просто сравниваем ситуацию с описанием паттерна и применяем тот паттерн, который отвечает проблеме и имеет устраивающие нас последствия. Эти последствия могут порождать новую проблему, которая решается применением другого паттерна. Таким образом, мы проектируем программу постепенно, не стараясь сразу выложить на стол полный проект еще до того, как написана первая строчка кода.

Как читать эту книгу

Вы должны быть знакомы с основами программирования на C. Вы должны знать синтаксис и семантику C – например, эта книга не расскажет вам о том, что такое указатель и как им пользоваться. Приводятся рекомендации только по вопросам более высокого порядка.

Главы книги независимы. Вы можете читать их в произвольном порядке или выбирать только те, которые вас интересуют. В следующем разделе приведен краткий обзор всех паттернов, что позволит перейти сразу к тем, что вам интересны. Так что если вы точно знаете, что ищете, то можете начать отсюда.

Если вы не ищете какой-то конкретный паттерн, а хотите получить общее представление о проектировании программ на C, то прочитайте часть I от начала до конца. Каждая глава этой части посвящена одной теме, начиная с таких простых, как обработка ошибок и управление памятью, и кончая такими более продвинутыми и специальными, как проектирование интерфейсов или платформенно независимого кода. В каждой главе описываются относящиеся к ее теме паттерны и сквозной пример кода, демонстрирующий их применение.

Во второй части книги приведено два больших примера, иллюстрирующих применение многих паттернов из первой части. Здесь вы увидите, как большая программа строится с использованием паттернов.

Краткий обзор паттернов

В табл. P.2–P.10 перечислены все паттерны. Каждая строка содержит краткое описание проблемы, после которого идет ключевое слово «Поэтому» и краткое описание решения.

Таблица P.2. Паттерны для обработки ошибок

Название паттерна	Краткое описание
Разбиение функции	На функции лежит несколько обязанностей, что затрудняет ее чтение и сопровождение. Поэтому разбейте ее на части. Выделите часть функции, которая кажется полезной сама по себе, создайте из нее новую функцию и вызовите ее
Проверка условий	Функцию трудно читать и сопровождать, потому что проверка предусловий совмещена в ней с основной логикой. Поэтому сначала проверьте выполнение обязательных предусловий и сразу же верните управление, если они не выполняются

Название паттерна	Краткое описание
Принцип самурая	Возвращая информацию об ошибке, вы предполагаете, что вызывающая сторона проверит эту информацию. Однако она может попросту опустить проверку, и ошибка останется незамеченной. Поэтому либо возвращайте управление, если все хорошо, либо не возвращайте вовсе. Если ошибку невозможно обработать, то аварийно завершайте программу
Переход к обработке ошибки	Код становится трудно читать, если он захватывает и освобождает несколько ресурсов в разных точках функции. Поэтому соберите все освобождение ресурсов и обработку ошибок в конце функции. Если ресурс невозможно захватить, то используйте предложение `goto` для перехода в точку освобождения ресурсов
Запись об очистке	Трудно сделать кусок кода удобным для чтения и сопровождения, если он захватывает и освобождает несколько ресурсов, особенно когда эти ресурсы зависят друг от друга. Поэтому после успешного вызова функций захвата ресурсов запоминайте, какие функции требуется вызвать для очистки. И вызывайте те, которые были зарегистрированы
Объектная обработка ошибок	Наличие нескольких обязанностей у одной функции, например захват ресурса, освобождение ресурса и его использование, затрудняет реализацию, чтение, сопровождение и тестирование функции. Поэтому поместите инициализацию и очистку в разные функции по аналогии с идеей конструкторов и деструкторов в объектно ориентированном программировании

Таблица P.3. Паттерны для возврата информации об ошибке

Название паттерна	Краткое описание
Возврат кода состояния	Вам нужен механизм возврата информации о состоянии вызывающей стороне, чтобы та могла на нее отреагировать. Механизм должен быть простым в использовании, а вызывающая сторона не должна путать разные ошибки. Поэтому используйте возвращаемое значение функции для возврата информации о состоянии. Возвращайте значение, представляющее конкретное состояние. Вызывающая и вызываемая сторона должны одинаково интерпретировать возвращаемые значения
Возврат существенной информации об ошибке	С одной стороны, вызывающая сторона должна иметь возможность реагировать на ошибки, а с другой стороны, чем больше информации об ошибке вы возвращаете, тем длиннее становится ваш код и код для обработки ошибки. Поэтому возвращайте только ту информацию об ошибке, которая существенна для вызывающей стороны. Информация существенна, если вызывающая сторона может на нее отреагировать

Название паттерна	Краткое описание
Специальное возвращаемое значение	Вы хотите вернуть информацию об ошибке, но явно возвращать коды состояния не годится, потому что тогда нельзя использовать возвращаемое функцией значение для возврата других данных. Если использовать выходные параметры, то вызывать функцию станет труднее. Поэтому используйте возвращаемое значение для возврата вычисленных функцией данных, но зарезервируйте одно или несколько специальных значений на случай ошибки
Протоколирование ошибок	Вы хотите быть уверены, что в случае ошибки сумеете легко определить ее причину. Но не хотите ради этого усложнять код обработки ошибок. Поэтому используйте разные каналы: один для предоставления информации об ошибке, существенной для вызывающей стороны, а другой для предоставления информации, существенной для разработчика. Например, записывайте отладочную информацию об ошибке в файл журнала и не возвращайте ее вызывающей стороне

Таблица P.4. Паттерны для управления памятью

Название паттерна	Краткое описание
Сначала стек	Любому программисту часто приходится принимать решение о классе хранения и области памяти (стек, куча…) для размещения переменных. Если для каждой переменной по новой взвешивать все плюсы и минусы различных вариантов, то ни на что другое не останется времени. Поэтому по умолчанию размещайте переменные в стеке, это даст возможность воспользоваться механизмом автоматической очистки памяти
Вечная память	Хранить большие объемы данных и передавать их между вызовами функций трудно, потому что нужно гарантировать, что памяти для данных достаточно и что она не стирается между вызовами. Поэтому размещайте данные в памяти, которая остается доступной в течение всего времени работы программы
Отложенная очистка	Необходимость в динамической памяти возникает, если нужна память большого и заранее неизвестного размера. Но проблема очистки динамической памяти трудна и является источником многих ошибок. Поэтому только выделяйте динамическую память, а ее освобождение оставьте операционной системе в момент завершения программы
Единоличное владение	За удобство динамической памяти приходится расплачиваться необходимостью ее освобождения. В больших программах трудно гарантировать, что вся динамически выделенная память надлежащим образом освобождается. Поэтому уже в момент выделения памяти ясно и недвусмысленно определите, где она должна быть освобождена и кто за это отвечает

Название паттерна	Краткое описание
Обертка выделения	Любое выделение динамической памяти может завершиться неудачно, поэтому следует проверять результат выделения в своем коде и реагировать соответственно. Это громоздко, потому что такие проверки приходится делать во многих местах программы. Поэтому оберните вызовы функций выделения и освобождения памяти и реализуйте логику обработки ошибок или дополнительного управления памятью в этих обертках
Проверка указателя	Программные ошибки, связанные с доступом по недействительному указателю, ведут к неопределенному поведению программы, и отлаживать их трудно. Но поскольку код часто работает с указателями, велики шансы появления таких ошибок. Поэтому явно делайте недействительными неинициализированные или освобожденные указатели и всегда проверяйте действительность перед доступом по ним
Пул памяти	Частое выделение и освобождение памяти из кучи приводит к фрагментации памяти. Поэтому выделите большой участок памяти на все время работы программы. Во время выполнения получайте блоки фиксированного размера из этого пула памяти, а не непосредственно из кучи

Таблица P.5. Паттерны для возврата данных из C-функций

Название паттерна	Краткое описание
Возвращаемое значение	Части, на которые вы хотите разбить функцию, не являются независимыми. Как обычно бывает в процедурном программировании, одна часть производит результат, необходимый другой части. Части разбиваемой функции должны разделять какие-то данные. Поэтому используйте механизм C, предназначенный для получения информации о результате вызова функции: возвращаемое значение. Механизм возврата данных в C копирует результат функции и дает вызывающей стороне доступ к копии
Выходные параметры	Язык C поддерживает возврат только одного значения из функции, и вернуть несколько элементов информации затруднительно. Поэтому возвращайте все данные с помощью единственного вызова функции, эмулируя передачу аргументов по ссылке с помощью указателей
Агрегат	Язык C поддерживает возврат только одного значения из функции, и вернуть несколько элементов информации затруднительно. Поэтому поместите все данные в специально определенный тип. Пусть этот агрегат содержит все связанные данные, которые вы хотите использовать сообща. Определите этот тип в интерфейсе своего компонента, так чтобы вызывающая сторона могла обращаться ко всем данным, хранящимся в экземпляре агрегата

Название паттерна	Краткое описание
Неизменяемый экземпляр	Вы хотите передать информацию, хранящуюся в больших порциях неизменяемых данных, из своего компонента вызывающей стороне. Поэтому разместите экземпляр (например, `struct`), содержащий разделяемые данные, в статической памяти. Предоставляйте эти данные нуждающимся в них пользователям, но организуйте дело так, чтобы они не могли изменить данные
Буфер, принадлежащий вызывающей стороне	Вы хотите передать сложные или большие данные известного размера вызывающей стороне, и эти данные не являются неизменяемыми (т. е. могут быть изменены во время выполнения). Поэтому потребуйте, чтобы вызывающая сторона предоставила буфер и его размер функции, возвращающей данные. Внутри функции скопируйте данные в буфер при условии, что его размер достаточен
Вызываемая сторона выделяет память	Вы хотите передать сложные или большие данные известного размера вызывающей стороне, и эти данные не являются неизменяемыми (т. е. могут быть изменены во время выполнения). Поэтому выделите буфер нужного размера в самой функции, возвращающей данные. Скопируйте данные в буфер и верните указатель на этот буфер

Таблица P.6. Паттерны, относящиеся ко времени жизни данных и владению ими

Название паттерна	Краткое описание
Программный модуль без состояния	Вы хотите предоставить логически связанную функциональность вызывающей стороне и максимально упростить ее использование. Поэтому делайте функции простыми и не храните информацию о состоянии в реализации. Поместите все связанные функции в один заголовочный файл и предоставьте вызывающей стороне этот интерфейс к своему программному модулю
Программный модуль с глобальным состоянием	Вы хотите структурировать логически связанный код, нуждающийся в общей информации о состоянии, и максимально упростить его использование. Поэтому заведите один глобальный экземпляр, чтобы все связанные функции могли разделять его. Поместите все функции, работающие с этим экземпляром, в один заголовочный файл и предоставьте вызывающей стороне этот интерфейс к своему программному модулю
Экземпляр, принадлежащий вызывающей стороне	Вы хотите предоставить нескольким вызывающим сторонам или потокам доступ к некоторой функциональности с помощью функций, которые зависят друг от друга. При этом взаимодействие вызывающей стороны с вашими функциями приводит к образованию информации о состоянии. Поэтому потребуйте, чтобы вызывающая сторона передавала вашим функциям экземпляр, используемый для хранения ресурса и информации о состоянии. Предоставьте явные функции для создания и уничтожения таких экземпляров, чтобы вызывающая сторона могла контролировать время их существования

Название паттерна	Краткое описание
Разделяемый экземпляр	Вы хотите предоставить нескольким вызывающим сторонам или потокам доступ к некоторой функциональности с помощью функций, которые зависят друг от друга. При этом взаимодействие вызывающей стороны с вашими функциями приводит к образованию информации о состоянии, которую все вызывающие стороны хотят разделять. Поэтому потребуйте, чтобы вызывающая сторона передавала вашим функциям экземпляр, используемый для хранения ресурса и информации о состоянии. Используйте один и тот же экземпляр для нескольких вызывающих сторон и оставьте владение этим экземпляром за своим программным модулем

Таблица P.7. Паттерны для создания гибких API

Название паттерна	Краткое описание
Заголовочные файлы	Вы хотите, чтобы реализованная вами функциональность была доступна коду, реализованному другими лицами, но при этом хотите скрыть детали реализации от вызывающей стороны. Поэтому включите в свой API объявления функций, реализующих функциональность, предоставляемую пользователям. Скройте все внутренние функции, внутренние данные и определения (реализации) функций в файлах реализации и не передавайте эти файлы пользователям
Описатель	Вам нужно разделить информацию о состоянии или работать с разделяемыми ресурсами в реализациях своих функций, но вы не хотите, чтобы вызывающая сторона видела эту информацию и разделяемые ресурсы. Поэтому заведите функцию, создающую контекст, с которым будет работать вызывающая сторона, и возвращающую абстрактный указатель на внутренние данные в этом контексте. Потребуйте, чтобы вызывающая сторона передавала этот указатель всем вашим функциям, которые смогут тогда воспользоваться внутренними данными для хранения информации о состоянии и ресурсов
Динамический интерфейс	Должна быть возможность вызывать реализации с немного различающимся поведением, но при этом не хотелось бы дублировать код, даже код управления логикой реализации и объявления интерфейса. Поэтому определите общий интерфейс для различающейся функциональности в своем API и потребуйте, чтобы вызывающая сторона предоставила функцию обратного вызова для варианта этой функциональности, которую вы сможете вызвать из реализации своей функции

Название паттерна	Краткое описание
Управление функцией	Вы хотите вызывать реализации с немного различающимся поведением, но не хотите дублировать код, даже код управления логикой реализации и объявления интерфейса. Поэтому добавьте в функцию параметр, в котором функции передается метаинформация об этом вызове, определяющая, какая именно функциональность требуется

Таблица P.8. Паттерны для создания гибких интерфейсов итератора

Название паттерна	Краткое описание
Доступ по индексу	Вы хотите, чтобы пользователь мог удобно перебирать элементы вашей структуры данных, при этом нужно сохранить возможность изменения внутреннего устройства этой структуры, не изменяя пользовательский код. Поэтому предоставьте функцию, которая принимает индекс для адресации элемента в вашей структуре данных и возвращает его содержимое. Пользователь вызывает эту функцию в цикле для обхода всех элементов
Курсор	Вы хотите предоставить пользователю такой интерфейс итератора, который был бы устойчив относительно изменения элементов в процессе итерирования и который позволил бы изменять впоследствии структуру данных, не внося изменений в пользовательский код. Поэтому создайте экземпляр курсора, указывающий на элемент структуры данных. Функция итерирования принимает этот экземпляр итератора в качестве аргумента, получает элемент, на который указывает итератор, и модифицирует экземпляр итератора, так чтобы он указывал на следующий элемент. Пользователь в цикле вызывает эту функцию, чтоб получать элементы по одному
Итератор обратного вызова	Вы хотите предоставить устойчивый интерфейс итерирования, который не требовал бы от пользователя реализации цикла для обхода всех элементов и позволял бы в будущем вносить изменения в структуру данных, не изменяя пользовательский код. Поэтому воспользуйтесь своей уже существующей структурой данных и реализованными вами операциями для обхода всех элементов в ней и вызывайте предоставленную пользователем функцию для каждого элемента в процессе этого обхода. Эта пользовательская функция принимает содержимое элемента в качестве параметра и выполняет операции над этим элементом. Чтобы начать итерирование, пользователь вызывает всего одну функцию, а весь обход производится внутри вашей реализации

Таблица P.9. Паттерны для организации файлов в модульных программах

Название паттерна	Краткое описание
Охрана включения	Включить один и тот же заголовочный файл легко, но это приведет к ошибкам компиляции, если в файле имеются определения типов или макросы определенного вида, поскольку в процессе компиляции они будут определены повторно. Поэтому защищайте содержимое заголовочных файлов от повторного включения, чтобы разработчику, пользующемуся ими, не нужно было думать, сколько раз включен файл. Для этого можно использовать директиву `#ifdef` или `#pragma once`
Каталоги программных модулей	Разнесение кода по нескольким файлам увеличивает количество файлов в кодовой базе. Если хранить все файлы в одном каталоге, то становится трудно обозревать их, особенно когда кодовая база велика. Поэтому помещайте тесно связанные заголовочные файлы и файлы реализации в один каталог. Назовите этот каталог, так чтобы имя отражало функциональность, предоставляемую заголовочными файлами
Глобальный каталог include	Чтобы включить файлы из других программных модулей, приходится использовать относительные пути вида *`../othersoftwaremodule/file.h`*. Вы должны знать точное местоположение других заголовочных файлов. Поэтому заведите в кодовой базе один глобальный каталог, содержащий API всех программных модулей. Добавьте этот каталог в глобальный список путей к включаемым файлам, поддерживаемый вашим инструментарием
Автономные компоненты	Структура каталогов не позволяет увидеть зависимости в коде. Любой программный модуль может просто включить заголовочные файлы из любого другого программного модуля, поэтому проверить зависимости с помощью компилятора невозможно. Поэтому идентифицируйте программные модули, которые содержат схожую функциональность и должны развертываться вместе. Поместите эти модули в общий каталог и заведите подкаталог для тех заголовочных файлов, которые нужны вызывающей стороне
Копия API	Вы хотите разрабатывать, присваивать номера версий и развертывать части кодовой базы независимо друг от друга. Однако для этого необходимо иметь четко определенные интерфейсы между частями кода и возможность хранить этот код в различных репозиториях. Поэтому, чтобы использовать функциональность другого компонента, скопируйте его API. Отдельно соберите этот компонент и скопируйте артефакты сборки и его публичные заголовочные файлы. Поместите эти файлы в каталог внутри своего компонента и сконфигурируйте этот каталог как путь к глобальному include

Таблица P.10. Паттерны, позволяющие избежать ада `#ifdef`

Название паттерна	Краткое описание
Избегание вариантов	Использование разных функций для каждой платформы затрудняет чтение и написание кода. От программиста требуется понимать, правильно использовать и тестировать эти многочисленные функции, чтобы обеспечить одинаковую функциональность на нескольких платформах. Поэтому пользуйтесь стандартизованными функциями, имеющимися на всех платформах. Если стандартизованных функций нет, то подумайте, не стоит ли отказаться от соответствующей функциональности
Изолированные примитивы	Варианты кода, организованные с помощью директив `#ifdef`, делают код нечитаемым. Очень трудно следить за потоком программы, который реализован несколько раз для разных платформ. Поэтому изолируйте свои варианты кода. В файле реализации поместите многовариантный код в отдельные функции и вызывайте эти функции из основной программы, которая таким образом будет содержать только платформенно независимый код
Атомарные примитивы	Функцию, которая содержит варианты и вызывается из основной программы, все равно трудно понять, потому что код, содержащий многочисленные `#ifdef`, был помещен туда только для того, чтобы избавиться от него в основной программе, но проще он от этого не стал. Поэтому делайте примитивы атомарными. В каждой функции обрабатывайте только один вариант. Если, например, нужно обработать несколько вариантов операционных систем и оборудования, то заведите для каждого свою функцию
Уровень абстракции	Вы хотите использовать функциональность, которая отвечает за платформенные варианты в нескольких местах кодовой базы, но не хотите дублировать код этой функциональности. Поэтому предоставьте API для каждого вида функциональности, требующего платформенно зависимого кода. В заголовочном файле объявляйте только платформенно независимые функции, а весь платформенно зависимый код, заключенный внутри `#ifdef`, помещайте в файл реализации. Вызывающая сторона должна будет включить только ваш заголовочный файл, но не платформенно зависимые файлы
Разделение реализаций вариантов	Платформенно зависимые реализации по-прежнему содержат директивы `#ifdef`, чтобы различать варианты кода. Из-за этого трудно понять, какую часть кода следует собирать для какой платформы. Поэтому помещайте реализацию каждого варианта в отдельный файл и пофайлово выбирайте, что хотите компилировать для какой платформы

Графические выделения

В книге применяются следующие графические выделения.

Курсив

Новые термины, URL-адреса, адреса электронной почты, имена и расширения файлов.

Полужирный

Выделение формулировки проблемы и ее решения для каждого паттерна.

`Моноширинный`

Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные окружения, предложения и ключевые слова языка.

Так обозначается замечание общего характера.

Так обозначается предупреждение или предостережение.

О примерах кода

Примеры кода в этой книге представляют собой короткие фрагменты, акцентированные на какой-то одной идее, с целью продемонстрировать паттерны и их применение. Сами по себе эти фрагменты не компилируются, потому что некоторые вещи в них опущены (в частности, заголовочные файлы). Полный код, который компилируется, можно скачать с GitHub по адресу https://github.com/christopher-preschern/fluent-c.

Вопросы технического характера, а также замечания по примерам кода следует отправлять по адресу bookquestions@oreilly.com.

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешения необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, никто не возбраняет включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров из книг издательства O'Reilly разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем ссылки на наши издания. В ссылке обычно указывается название книги, имя автора, издательство и ISBN, например: «Fluent C by Christopher Preschern (O'Reilly). Copyright 2023 Christopher Preschern, 978-1-492-09733-4».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу **permissions@oreilly.com**.

Все паттерны в этой книги взяты из существующего кода, в котором они применяются. В списке ниже приведены ссылки на эти примеры кода:

- игра NetHack (https://oreil.ly/nxO5w);
- проект OpenWrt (https://oreil.ly/qeppo);
- библиотека OpenSSL (https://oreil.ly/zzsMO);
- сетевой анализатор Wireshark (https://oreil.ly/M55B5);
- портлендский репозиторий паттернов (https://oreil.ly/wkZzb);
- система управления версиями Git (https://oreil.ly/7F9Oz);
- переносимая среда исполнения Apache (https://oreil.ly/ysaM6);
- веб-сервер Apache (https://oreil.ly/W6SMn);
- операционная система B&R Automation Runtime (проприетарный закрытый код компании B&R Industrial Automation GmbH);
- визуальный редактор системы автоматизации B&R Visual Components, проприетарный закрытый код компании B&R Industrial Automation GmbH);
- система управления данными NetDRMS (https://oreil.ly/eR0EV);
- платформа программирования и численных расчетов MATLAB (https://oreil.ly/UpvJK);
- библиотека GLib (https://oreil.ly/QoUwT);
- веб-анализатор реального времени GoAccess (https://oreil.ly/L1Eij);
- программа физических расчетов Cloudy (https://oreil.ly/phLBb);
- собрание компиляторов GNU (GCC) (https://oreil.ly/KK4jY);
- система баз данных MySQL (https://oreil.ly/YKXxs);
- диспетчер памяти ION для Android (https://oreil.ly/2JV7h);
- Windows API (https://oreil.ly/nnzyX);
- Apple Cocoa API (https://oreil.ly/sQual);
- операционная система реального времени VxWorks (https://oreil.ly/UMUaj);
- текстовый редактор sam (https://oreil.ly/k3SQl);
- функции из стандартной библиотеки C: реализация glibc (https://oreil.ly/9Qr95);
- проект Subversion (https://oreil.ly/8Yz5R);
- инструмент исследования сети Nmap (https://oreil.ly/sg9sz);
- монитор производительности в реальном времени и система визуализации Netdata (https://oreil.ly/1sDZz);

- файловая система OpenZFS (https://oreil.ly/VWeQL);
- каркас обратной разработки Radare (https://oreil.ly/TUYfh);
- цифровые обучающие программы Education First (https://www.ef.com);
- текстовый редактор VIM (https:/gitub.com/vim/vim);
- программа построения графиков GNUplot (https://oreil.ly/PIQPj);
- движок базы данных SQLite (https://oreil.ly/5Knfz);
- программа сжатия данных gzip (https://oreil.ly/it40Z);
- веб-сервер lighttpd (https://github.com/lighttpd);
- начальный загрузчик U-Boot (https://oreil.ly/IKVYV);
- система моделирования дискретных событий (https://oreil.ly/NJnCH);
- платформа Nokia Maemo (https://oreil.ly/RwDtt).

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North Sebastopol, CA 95472
800-998-9938 (в США и Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс).

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: https://www.oreil.ly/fluent-c.

Замечания и вопросы технического характера следует отправлять по адресу bookquestions@oreilly.com.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу http://www.oreilly.com.

Ищите нас в LinkedIn: https://linkedin.com/company/oreilly-media.

Следите за нашей лентой в Twitter: http://twitter.com/oreillymedia.

Смотрите нас на YouTube: http://www.youtube.com/oreillymedia.

Благодарности

Я хочу поблагодарить свою жену Силке, которая теперь даже знает, что такое паттерны :-), и свою дочь Илви. Обе они делают меня счастливее и обе следят за тем, чтобы я не сидел за компьютером все время, а наслаждался жизнью.

Эта книга не увидела бы свет без помощи многих энтузиастов паттернов. Я благодарен всем участникам семинара Writers' Workshops на Европейской конференции по языкам паттернов в программах за отзывы о паттернах. В частности, я хочу выразить благодарность следующим лицам, которые дали очень полезные отзывы во время так называемого пастырского процесса на этой конференции, среди них Яри Раухамяки, Тобиас Раутер, Андреа Холлер,

Джеймс Коплиен, Уве Здун, Томас Разер, Иден Бэртон, Клаудиус Линк, Валентино Вранич и Сумит Калра. Отдельное спасибо моим коллегам по работе, в особенности Томасу Гавловецу, который проследил за тем, чтобы все детали программирования на C в моих паттернах были правильны. Роберт Ханмер, Майкл Вейсс, Дэвид Гриффитс и Томас Круг потратили немало времени на рецензирование этой книги и поделились со мной мыслями о том, как сделать ее лучше, – большое вам спасибо! Также я признателен всему коллективу издательства O'Reilly, помогавшему мне в работе над этой книгой. Особенно хочу поблагодарить редактора-консультанта Корбина Коллинза и выпускающего редактора Джонатона Оуэна.

Текст этой книги основан на следующих статьях, которые были приняты на Европейской конференции по языкам паттернов в программах и опубликованы в изданиях ACM. Эти статьи можно скачать бесплатно на сайте http://www.preschern.com.

- «A Pattern Story About C Programming», EuroPLoP '21: 26th European Conference on Pattern Languages of Programs, July 2015, article no. 53, 1–10, https://dl.acm.org/doi/10.1145/3489449.3489978.
- «Patterns for Organizing Files in Modular C Programs», EuroPLoP '20: Proceedings of the European Conference on Pattern Languages of Programs, July 2020, article no. 1, 1–15, https://dl.acm.org/doi/10.1145/3424771.3424772.
- «Patterns to Escape the #ifdef Hell», EuroPLop '19: Proceedings of the 24th European Conference on Pattern Languages of Programs, July 2019, article no. 2, 1–12, https://dl.acm.org/doi/10.1145/3361149.3361151.
- «Patterns for Returning Error Information in C», EuroPLop '19: Proceedings of the 24th European Conference on Pattern Languages of Programs, July 2019, article no. 3, 1–14, https://dl.acm.org/doi/10.1145/3361149.3361152.
- «Patterns for Returning Data from C Functions», EuroPLop '19: Proceedings of the 24th European Conference on Pattern Languages of Programs, July 2019, article no. 37, 1–13, https://dl.acm.org/doi/10.1145/3361149.3361188.
- «C Patterns on Data Lifetime and Ownership», EuroPLop '19: Proceedings of the 24th European Conference on Pattern Languages of Programs, July 2019, article no. 36, 1–13, https://dl.acm.org/doi/10.1145/3361149.3361187.
- «Patterns for C Iterator Interfaces», EuroPLoP '17: Proceedings of the 22nd European Conference on Pattern Languages of Programs, July 2017, article no. 8, 1–14, https://dl.acm.org/doi/10.1145/3147704.3147714.
- «API Patterns in C», EuroPlop '16: Proceedings of the 21st European Conference on Pattern Languages of Programs, July 2016, article no. 7, 1–11, https://dl.acm.org/doi/10.1145/3011784.3011791.
- «Idioms for Error Handling in C», EuroPLoP '15: Proceedings of the 20th European Conference on Pattern Languages of Programs, July 2015, article no. 53, 1–10, https://dl.acm.org/doi/10.1145/2855321.2855377.

Часть I

Паттерны на C

Паттерны облегчают нам жизнь. Они избавляют нас от необходимости всякий раз самостоятельно придумывать проектные решения. Паттерны предлагают проверенные практикой решения, и в первой части книги вы найдете такие решения и узнаете о последствиях их применения. Каждая из последующих глав посвящена какому-то одному вопросу программирования на C, в ней представлены соответствующие паттерны и демонстрируется их использование на сквозном примере.

Глава 1

Обработка ошибок

Обработка ошибок – важная часть написания программ; если это сделано неправильно, то программу становится трудно развивать и сопровождать. В таких языках программирования, как C++ и Java, имеются «исключения» и «деструкторы», упрощающие обработку ошибок. В C подобных механизмов нет, а сведения о хороших способах обработки ошибок в программах на C разбросаны по всему интернету.

В этой главе представлены коллективные знания о правильной обработке ошибок в форме паттернов и сквозного примера применения этих паттернов. Паттерны предлагают проверенные практикой проектные решения вместе с указаниями на то, когда их применять и к каким последствиям это приводит. С точки зрения программиста, паттерны избавляют от бремени принятия многих детальных решений и позволяют положиться на знания, заключенные в паттерне, и взять его за основу для написания хорошего кода.

На рис. 1.1 приведен обзор паттернов, рассматриваемых в этой главе, и связи между ними, а в табл. 1.1 дано краткое описание паттернов.

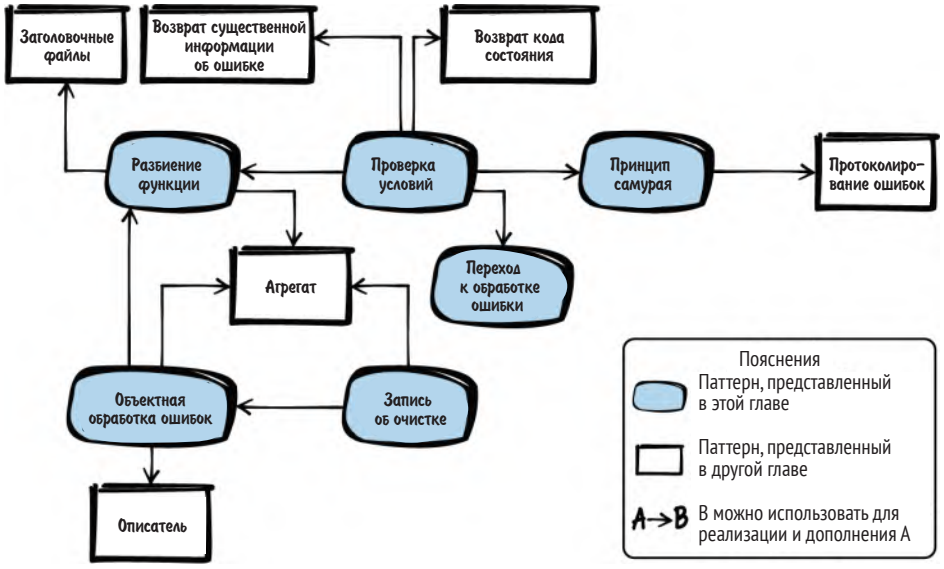


Рис. 1.1. Обзор паттернов для обработки ошибок

Таблица 1.1. Паттерны для обработки ошибок

Название паттерна	Краткое описание
Разбиение функции	На функции лежит несколько обязанностей, что затрудняет ее чтение и сопровождение. Поэтому разбейте ее на части. Выделите часть функции, которая кажется полезной сама по себе, создайте из нее новую функцию и вызовите ее
Проверка условий	Функцию трудно читать и сопровождать, потому что проверка предусловий совмещена в ней с основной логикой. Поэтому сначала проверьте выполнение обязательных предусловий и сразу же верните управление, если они не выполняются
Принцип самурая	Возвращая информацию об ошибке, вы предполагаете, что вызывающая сторона проверит эту информацию. Однако она может попросту опустить проверку, и ошибка останется незамеченной. Поэтому либо возвращайте управление, если все хорошо, либо не возвращайте вовсе. Если ошибку невозможно обработать, то аварийно завершайте программу
Переход к обработке ошибки	Код становится трудно читать, если он захватывает и освобождает несколько ресурсов в разных точках функции. Поэтому соберите все освобождение ресурсов и обработку ошибок в конце функции. Если ресурс невозможно захватить, то используйте предложение `goto` для перехода в точку освобождения ресурсов
Запись об очистке	Трудно сделать кусок кода удобным для чтения и сопровождения, если он захватывает и освобождает несколько ресурсов, особенно когда эти ресурсы зависят друг от друга. Поэтому после успешного вызова функций захвата ресурсов запоминайте, какие функции требуется вызвать для очистки. И вызывайте те, которые были зарегистрированы
Объектная обработка ошибок	Наличие нескольких обязанностей у одной функции, например захват ресурса, освобождение ресурса и его использование, затрудняет реализацию, чтение, сопровождение и тестирование функции. Поэтому поместите инициализацию и очистку в разные функции по аналогии с идеей конструкторов и деструкторов в объектно ориентированном программировании

Сквозной пример

Вы хотите написать функцию, которая ищет в файле определенные ключевые слова и возвращает информацию о том, какие слова были найдены.

Стандартный способ сообщить об ошибке в C – использовать возвращаемое функцией значение. Для передачи дополнительной информации в унаследованных функциях часто записывают конкретный код ошибки в переменную `errno` (см. файл *errno.h*). Вызывающая сторона может затем проверить `errno`, чтобы узнать, какая произошла ошибка.

Однако в показанном ниже коде мы просто используем возвращаемое значение вместо `errno`, поскольку очень подробная информация об ошибке не нужна. В итоге получается такая первоначальная версия кода:

```
int parseFile(char* file_name)
{
 int return_value = ERROR;
 FILE* file_pointer = 0;
 char* buffer = 0;

 if(file_name!=NULL)
 {
  if(file_pointer=fopen(file_name, "r"))
  {
   if(buffer=malloc(BUFFER_SIZE))
   {
    /* разобрать содержимое файла */
    return_value = NO_KEYWORD_FOUND;
    while(fgets(buffer, BUFFER_SIZE, file_pointer)!=NULL)
    {
     if(strcmp("KEYWORD_ONE\n", buffer)==0)
     {
      return_value = KEYWORD_ONE_FOUND_FIRST;
      break;
     }
     if(strcmp("KEYWORD_TWO\n", buffer)==0)
     {
      return_value = KEYWORD_TWO_FOUND_FIRST;
      break;
     }
    }
    free(buffer);
   }
   fclose(file_pointer);
  }
 }
 return return_value;
}
```

В коде мы должны проверять возвращаемые значения после вызовов функций, чтобы узнать, была ли ошибка. Поэтому возникают глубоко вложенные предложения `if`. Это создает следующие проблемы:

- функция оказывается длинной, в ее коде перемешаны обработка ошибок, инициализация, очистка и собственно логика функции. Поэтому код трудно сопровождать;
- основной код, который читает и интерпретирует данные файла, глубоко вложен, поэтому трудно следить за логикой программы;

- функции очистки далеко отстоят от функций инициализации, поэтому можно легко забыть о необходимости какой-то очистки. В особенности это относится к функциям, содержащим несколько предложений `return`.

Чтобы улучшить ситуацию, произведем для начала «Разбиение функции».

Разбиение функции

Контекст

Имеется функция, выполняющая несколько действий. Например, она выделяет ресурс (скажем, динамическую память или описатель файла), использует этот ресурс и освобождает его.

Проблема

У функции несколько обязанностей, что затрудняет ее чтение и сопровождение.

Такая функция могла бы нести ответственность за выделение ресурсов, действия с этими ресурсами и их очистку. Быть может, очистка даже разбросана по всей функции и дублируется в нескольких местах. Особенно трудно функцию становится читать из-за обработки ошибок выделения ресурса, потому что очень часто при этом появляются вложенные предложения `if`.

Если выделять, очищать и использовать несколько ресурсов в одной функции, то легко можно забыть об очистке какого-то ресурса, особенно если код впоследствии изменяется. Например, если в середину кода будет добавлено предложение `return`, то вполне можно забыть об очистке уже выделенных к этому моменту ресурсов.

Решение

Разбейте функцию на части. Выделите часть функции, которая представляется полезной сама по себе, создайте из нее новую функцию и придумайте для нее имя.

Чтобы понять, какую часть функции выделять, просто подумайте, можно ли дать ей осмысленное имя и действительно ли такое разбиение изолирует какую-то обязанность. Например, таким образом можно было бы выделить функцию, содержащую только функциональный код, и функцию, содержащую только код обработки ошибок.

Хороший признак необходимости разбиения функции на части – наличие кода очистки в нескольких местах. В таком случае было бы гораздо лучше поместить в одну функцию код, который выделяет и освобождает ресурсы, а в другую – код, который использует эти ресурсы. Тогда функция, использующая ресурсы, вполне может содержать несколько предложений `return` без необходимости очищать ресурсы перед каждым из них, потому что очистка производится в другой функции. Это показано в следующем коде:

```
void someFunction()
{
```

```
 char* buffer = malloc(LARGE_SIZE);
 if(buffer)
 {
  mainFunctionality(buffer);
 }
 free(buffer);
}

void mainFunctionality()
{
  // здесь должна быть реализация
}
```

Теперь у нас две функции вместо одной. Конечно, это означает, что вызывающая функция больше не автономна, а зависит от другой функции. И вам придется решить, куда эту функцию поместить. Первая мысль – поместить ее в один файл с вызывающей, но если две функции не являются тесно связанными, то можно поместить вызываемую функцию в отдельный файл и завести заголовочный файл с объявлением этой функции.

Последствия

Код стал лучше, потому что две короткие функции проще читать и сопровождать, чем одну длинную. В частности, код проще читать, потому что функции очистки расположены ближе к функциям, нуждающимся в очистке, и потому что выделение и очистка ресурсов не смешиваются с основной логикой программы. Поэтому в будущем основную логику будет легче сопровождать и расширять.

Теперь вызываемая функция может содержать несколько предложений `return`, так как ей не нужно заботиться об очистке ресурсов перед каждым `return`. Очистка производится в одном месте вызывающей функцией.

Если в вызываемой функции используется несколько ресурсов, то все они должны быть ей переданы. При наличии большого числа параметров код функции становится трудно читать, а если по неосторожности передать параметры не в том порядке, то произойдет ошибка. Во избежание такого развития события можно использовать паттерн «Агрегат».

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Практически в любом коде на C есть части, в которых этот паттерн применяется, и части, где он не применяется; последние сопровождать труднее. Согласно книге *Robert C. Martin «Clean Code: A Handbook of Agile Software» Craftsmanship*»[1] (Prentice Hall, 2008), у каждой функции должна быть ровно одна обязанность (принцип единственной обязанности), поэтому обработка ресурса и прочая программная логика всегда должны разноситься по разным функциям.

[1] *Роберт Мартин*. Чистый код. Создание анализ и рефакторинг. Питер, 2022.

- В портлендском репозитории паттернов этот паттерн называется «Оберткой функции» (Function Wrapper).
- В объектно ориентированном программировании паттерн «Шаблонный метод» также описывает способ структурирования кода путем его разбиения на части.
- Критерии того, когда и в каких местах разбивать функцию, описаны в книге Martin Fowler «Refactoring: Improving the Design of Existing Code»[1] (Addison-Wesley, 1999) в виде паттерна «Извлечение метода».
- В игре NetHack этот паттерн применяется в функции `read_config_file`: там обрабатываются ресурсы и вызывается функция `parse_conf_file`, которая работает с этими ресурсами.
- В коде OpenWrt этот паттерн используется в нескольких местах для работы с буферами. Например, код, отвечающий за вычисление хеша MD5, выделяет буфер, передает его другой функции, которая работает с этим буфером, а затем освобождает этот буфер.

Применение к сквозному примеру

Наш код уже выглядит гораздо лучше. Вместо одной гигантской функции мы имеем две большие функции с различными обязанностями. Одна функция отвечает за получение и освобождение ресурсов, а другая – за поиск ключевых слов:

```
int searchFileForKeywords(char* buffer, FILE* file_pointer)
{
 while(fgets(buffer, BUFFER_SIZE, file_pointer)!=NULL)
 {
  if(strcmp("KEYWORD_ONE\n", buffer)==0)
  {
   return KEYWORD_ONE_FOUND_FIRST;
  }
  if(strcmp("KEYWORD_TWO\n", buffer)==0)
  {
   return KEYWORD_TWO_FOUND_FIRST;
  }
 }
 return NO_KEYWORD_FOUND;
}

int parseFile(char* file_name)
{
 int return_value = ERROR;
 FILE* file_pointer = 0;
 char* buffer = 0;
```

[1] *Мартин Фаулер.* Рефакторинг. Улучшение проекта существующего кода. Диалектика-Вильмс, 2019.

```
if(file_name!=NULL)
{
 if(file_pointer=fopen(file_name, "r"))
 {
  if(buffer=malloc(BUFFER_SIZE))
  {
   return_value = searchFileForKeywords(buffer, file_pointer);
   free(buffer);
  }
  fclose(file_pointer);
 }
}
return return_value;
}
```

Глубина вложенности `if` уменьшилась, но в функции `parseFile` все еще остались три предложения `if` для проверки ошибок выделения ресурсов – это слишком много. Эту функцию можно сделать чище, воспользовавшись паттерном «Проверка условий».

Проверка условий

Контекст

Имеется функция, выполняющая задачу, которая может быть успешно завершена только при определенных условиях (например, правильных входных параметрах).

Проблема

Функцию трудно читать и сопровождать, потому что проверка предусловий перемешана с основной логикой.

Выделение ресурсов всегда должно сопровождаться их освобождением. Если вы выделили ресурс, а позже поняли, что какое-то предусловие функции не удовлетворяется, то этот ресурс придется освободить.
Трудно понять логику программы, если проверки нескольких предусловий разбросаны по функции, особенно если они реализованы во вложенных предложениях `if`. Когда таких проверок много, функция становится очень длинной, что само по себе признак кода с душком.

Код с душком

Говорят, что код «дурно пахнет», если он плохо структурирован или написан так, что его трудно сопровождать. Примерами кода с душком являются очень длинные функции или дублирование кода. Другие примеры и контрмеры описаны в книге Martin Fowler «Refactoring: Improving the Design of Existing Code» (Addison-Wesley, 1999).

Решение

Проверяйте все обязательные предусловия и немедленно возвращайте управление, если они не выполнены.

Например, следует проверять допустимость входных параметров и находится ли программа в состоянии, допускающем выполнение остальной части функции. Тщательно обдумывайте, какие нужно установить предусловия вызова функции. С одной стороны, вы облегчите себе жизнь, если будете очень строго относиться к входным параметрам функции, но, с другой стороны, вызывающей стороне будет проще, если вы станете относиться к параметрам более снисходительно (закон Постеля формулирует это так: «Будь консервативен в собственных действиях и либерален к тому, что принимаешь от других»).

Если имеется много предусловий, то можно завести отдельную функцию для их проверки. В любом случае выполняйте проверки до выделения ресурсов, потому что вернуть управление из функции гораздо проще, когда не нужно производить никакой очистки.

Четко описывайте предусловия своей функции в ее интерфейсе. Лучшее место для документирования этого поведения – заголовочный файл, в котором функция объявлена.

Если вызывающей стороне важно знать, какое предусловие не выполнено, то можно предоставить ей информацию об ошибке. Например, можно вернуть код состояния, следя за тем, чтобы возвращать только существенную информацию об ошибке. В коде ниже приведен пример без возврата информации об ошибке.

someFile.h

```
/* эта функция работает с параметром 'user_input', который должен быть отличен
   от NULL */
void someFunction(char* user_input);
```

someFile.c

```
void someFunction(char* user_input)
{
 if(user_input == NULL)
 {
  return;
 }
 operateOnData(user_input);
}
```

Последствия

Код функции, которая немедленно возвращает управление, если предусловия не выполнены, проще читать, чем вложенные предложения `if`. Сразу видно, что в этом случае выполнение функции прерывается. Поэтому предусловия очень четко отделены от остального кода.

Но в некоторых рекомендациях по кодированию категорически запрещается возвращать управление в середине функции. Например, чтобы правильность кода можно было доказать формально, предложение `return` обычно допускается только в самом конце функции. В таком случае следует сохранять запись об очистке; это также предпочтительный вариант, если вы хотите организовать центральное место для обработки ошибок.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Паттерн «Проверка условий» описан в портлендском репозитории паттернов.
- В статье Klaus Renzel «Error Detection» (Proceedings of the 2nd EuroPLoP conference, 1997) описан очень похожий паттерн «Обнаружение ошибок», который предлагает включать проверки пред- и постусловий.
- В игре NetHack этот паттерн используется в нескольких местах, например в функции `placebc`. Эта функция в качестве наказания набрасывает цепь на героя NetHack, что уменьшает скорость его передвижения. Она возвращает управление немедленно, если нет доступных объектов цепи.
- Этот паттерн используется в коде OpenSSL. Например, функция `SSL_new` немедленно возвращает управление, если входные параметры недопустимы.
- Функция `capture_stats` в Wireshark, отвечающая за сбор статистики при анализе сетевых пакетов, сначала проверяет, допустимы ли входные параметры, и сразу же возвращает управление, если это не так.

Применение к сквозному примеру

В коде ниже показано, как функция `parseFile` применяет описанный паттерн для проверки предусловий:

```
int parseFile(char* file_name)
{
 int return_value = ERROR;
 FILE* file_pointer = 0;
 char* buffer = 0;

 if(file_name==NULL) ❶
 {
  return ERROR;
 }
 if(file_pointer=fopen(file_name, "r"))
 {
  if(buffer=malloc(BUFFER_SIZE))
  {
   return_value = searchFileForKeywords(buffer, file_pointer);
   free(buffer);
```

```
    }
    fclose(file_pointer);
  }
  return return_value;
}
```

❶ Если переданы недопустимые параметры, то мы сразу же возвращаем управление, и никакая очистка не нужна, потому что ресурсы еще не выделялись.

Для реализации паттерна «Проверка условий» здесь используется возврат кода состояния. Если параметр равен `NULL`, то возвращается константа `ERROR`. Вызывающая сторона может проверить возвращенное значение и узнать, что функции был передан недопустимый параметр `NULL`. Но обычно это признак программной ошибки, а проверять программные ошибки и распространять эту информацию по всему коду – неудачная идея. В таком случае проще применить «Принцип самурая».

Принцип самурая

Контекст

Имеется код, включающий сложную обработку ошибок, и некоторые ошибки очень серьезны. Ваша система не выполняет действий, связанных с технической или физической безопасностью, и высокая доступность не стоит на первом месте.

Проблема

Возвращая информацию об ошибке, вы предполагаете, что вызывающая сторона проверит ее. Однако вызывающая сторона может опустить проверку, и ошибка останется незамеченной.

В C необязательно проверять возвращенное функцией значение, и вызывающая сторона может просто проигнорировать его. Если возникшая внутри функции ошибка серьезна и вызывающая сторона не может ее корректно обработать, то не нужно оставлять решение на ее усмотрение. Вместо этого лучше самостоятельно позаботиться о том, чтобы нужное действие было точно предпринято.

Даже если вызывающая сторона обработает ошибочную ситуацию, часто случается, что программа все равно аварийно завершается, или происходит еще какая-то ошибка. Ошибка может проявиться где-то в другом месте, быть может, на несколько уровней выше, где она не обработана должным образом. В таком случае обработка ошибки лишь маскирует ее, что сильно затрудняет отладку с целью найти истинную причину ошибки.

Некоторые ошибки в вашем коде могут встречаться очень редко. В таком случае возвращать коды состояний и обрабатывать их на вызывающей стороне значит делать код менее понятным, потому что это отвлекает внимание читателя от основной логики программы и настоящей цели, которую преследу-

ет вызывающая функция. На вызывающей стороне приходится писать много строк кода для обработки чрезвычайно редких ситуаций.

Возврат информации о таких ошибках также поднимает вопрос о том, как именно ее возвращать. Использование для этой цели возвращаемого значения или выходных параметров усложняет сигнатуру функции и затрудняет чтение кода. Не хочется заводить дополнительные параметры только для возврата информации об ошибке.

Решение

Возвращаться из функции только с победой или не возвращаться вовсе («Принцип самурая»). Если вы точно знаете, что обработать ошибку невозможно, то завершайте программу.

Не используйте для возврата информации об ошибке выходные параметры или возвращаемое значение. Вся информация уже в наличии, так что обработайте ошибку немедленно. Если ошибка произошла, дайте программе возможность «грохнуться». Но делайте это упорядоченным образом, воспользовавшись макросом `assert`. Кстати, `assert` позволяет включить отладочную информацию, как показано ниже:

```
void someFunction()
{
 assert(checkPreconditions() && "Предусловия не выполнены");
 mainFunctionality();
}
```

Здесь в `assert` проверяется условие, и если оно не выполнено, то на `stderr` выводится сообщение, включающее строку справа от оператора `&&`, и программа завершается. Можно завершить программу и менее структурированным способом, если не проверять указатели на `NULL` и обращаться к памяти по таким указателям. Просто сделайте так, чтобы программа гарантированно завершалась в месте возникновения ошибки.

Очень часто проверки условий – отличное место для завершения программы в случае ошибок. Например, если вы точно знаете, что имеет место программная ошибка (вызывающая сторона передала вам указатель `NULL`), то завершите программу и запишите в журнал отладочную информацию, вместо того чтобы возвращать информацию об ошибке вызывающей стороне. Но не надо завершать программу из-за любой ошибки. Например, такие ошибки, как неправильный ввод данных пользователем, очевидно, не должны приводит к аварийному завершению.

Вызывающая сторона должна быть хорошо осведомлена о поведении вашей функции, поэтому в описании ее API следует документировать случаи, когда функция завершает программу. Например, в документации может быть написано, что программа «грохается», если функции передан нулевой указатель в качестве параметра.

Разумеется, «Принцип самурая» применим не ко всем ошибкам и не ко всем видам программ. Нельзя аварийно завершать программу при получе-

нии неожиданных данных от пользователя. Но в случае программной ошибки «быстрый отказ» с немедленным завершением программы, возможно, имеет смысл. Тогда программисту будет совсем просто отыскать ошибку.

Тем не менее такой отказ необязательно показывать пользователю. Если ваша программа – всего лишь некритическая часть большего приложения, то можно позволить ей завершиться. Но в контексте всего приложения ваша программа может завершиться «по-тихому», не тревожа ни остальное приложение, ни пользователя.

Макросы assert в выпускных версиях

При использовании `assert` часто возникает вопрос, следует ли активировать их только в отладочных версиях исполняемых файлов или также в выпускных. Деактивировать `assert` можно, определив в своем коде макрос `NDEBUG` перед включением *assert.h* или прямо в цепочке инструментов. Основной аргумент в пользу деактивации `assert` в выпускных версиях – то, что ошибки, для которых используется `assert` при тестировании отладочных версий, и так обрабатываются, поэтому не нужно оставлять возможность случайного аварийного завершения из-за присутствия `assert` в выпускных версиях. Основной аргумент в пользу того, чтобы оставлять `assert` активными и в выпускных версиях, заключается в том, что их в любом случае следует использовать для критических ошибок, которые невозможно обработать корректно, и что такие ошибки никогда не должны оставаться незамеченными, даже в исполняемых файлах, поставляемых заказчикам.

Последствия

Ошибка не может остаться незамеченной, если она должным образом обрабатывается в месте возникновения. На вызывающую сторону не возлагается бремя проверки этой ошибки, поэтому ее код становится проще. Но зато теперь вызывающая сторона не может выбирать способ реакции на ошибку.

В некоторых случаях аварийное завершение приложения приемлемо, потому что быстрый отказ лучше непредсказуемого поведения в будущем. Но все равно нужно думать о том, как представить такую ошибку пользователю. Быть может, пользователь должен увидеть ее как сообщение об аварийном завершении на экране. Но во встраиваемых приложениях, которые используют датчики и приводы для взаимодействия с окружающей средой, нужно быть осторожнее и принять во внимание, какое влияние завершившаяся программа окажет на свое окружение и можно ли его считать приемлемым. Во многих случаях приложение должно быть надежным, и простой останов не годится.

Останов программы и протоколирование ошибки прямо в точке возникновения упрощает ее поиск и исправление, потому что ошибка ничем не замаскирована. Поэтому в перспективе применение этого паттерна позволит сделать ваши программы надежнее, уменьшив число ошибок.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Похожий паттерн, предлагающий добавлять отладочную строку в макрос `assert`, называется Контекстом утверждения и описан в книге *A*dam Tornhill «Patterns in C» (Leanpub, 2014).
- В сетевом анализаторе Wireshark этот паттерн применяется повсеместно. Например, в функции `register_capture_dissector` макрос `assert` используется для проверки единственности регистрации диссектора.
- В исходном коде проекта Git используются макросы `assert`. Например, в функциях для хранения SHA1-хешей `assert` проверяет правильность пути к файлу, в котором должен храниться хеш.
- В коде OpenWrt, отвечающем за обработку больших чисел, `assert` используется для проверки предусловий в функциях.
- Похожий паттерн под названием «Дай ему упасть» представлен Пекка Алхо и Яри Раухамяки в статье «Patterns for Light-Weight Fault Tolerance and Decoupled Design in Distributed Control Systems» (https://oreil.ly/x0tQW). Паттерн ориентирован на распределенные системы управления и предлагает позволить аварийное завершение с последующим быстрым перезапуском одиночным отказоустойчивым процессам.
- В стандартной библиотеке C функция `strcpy` не проверяет корректность входных данных. Если вы передадите ей нулевой указатель, то функция «грохнется».

Применение к сквозному примеру

Теперь функция `parseFile` выглядит гораздо лучше. Вместо того чтобы возвращать код ошибки, мы включили простой макрос `assert`. В результате код оказался короче, а вызывающей стороне не приходится нести бремя проверки возвращенного значения:

```
int parseFile(char* file_name)
{
 int return_value = ERROR;
 FILE* file_pointer = 0;
 char* buffer = 0;

 assert(file_name!=NULL && "Недопустимое имя файла");
 if(file_pointer=fopen(file_name, "r"))
 {
  if(buffer=malloc(BUFFER_SIZE))
  {
   return_value = searchFileForKeywords(buffer, file_pointer);
   free(buffer);
  }
  fclose(file_pointer);
 }
```

```
 return return_value;
}
```

Предложения `if`, не требующие очистки ресурсов, устранены, но код все равно содержит вложенные `if` для всего, что требует очистки. Кроме того, не обрабатывается ситуация, когда вызов `malloc` завершается ошибкой. Все это можно исправить, воспользовавшись паттерном «Переход к обработке ошибки».

Переход к обработке ошибки

Контекст

Имеется функция, которая захватывает и освобождает несколько ресурсов. Быть может, вы уже пытались уменьшить ее сложность, применяя паттерны «Проверка условий», «Разбиение функции» и «Принцип самурая», но все же остались глубоко вложенные `if`, особенно в связи с выделением ресурсов. Возможно даже, что код очистки ресурсов дублируется.

Проблема

Код функции становится трудно читать и сопровождать, если он захватывает и освобождает несколько ресурсов в разных местах.

Затруднения вызваны тем, что выделение любого ресурса может завершиться неудачно, а освобождать ресурс нужно, только если он был выделен успешно. Чтобы обеспечить это, необходимо много предложений `if`, но при плохой реализации наличие вложенных `if` в одной функции затрудняет чтение и сопровождение кода.

Поскольку ресурсы необходимо освобождать, возврат в середине функции, если что-то пошло не так, – не самая лучшая мысль. Ведь все уже захваченные ресурсы нужно будет освобождать перед каждым предложением `return`. Поэтому в коде образуется несколько мест, где освобождается один и тот же ресурс, но мы не хотим дублировать код обработки ошибок и очистки.

Решение

Поместите всю очистку ресурсов и обработку ошибок в конец функции. Если ресурс не удалось захватить, перейдите на код очистки с помощью предложения `goto`.

Захватывайте ресурсы в том порядке, в каком считаете нужным, а в конце функции освобождайте их в обратном порядке. Для каждого ресурса заведите отдельную метку, на которую можно будет перейти, чтобы очистить его. В случае ошибки или при невозможности захватить ресурс просто перейдите на эту метку, но не делайте несколько переходов и всегда переходите только вперед, как показано в следующем коде[1]:

```
void someFunction()
{
 if(!allocateResource1())
```

[1] Код неправилен. Если Resource1 не удалось захватить, то его не нужно и очищать. И точно так же для Resource 2. Метки расставлены неверно. – *Прим. перев.*

```
 {
  goto cleanup1;
 }
 if(!allocateResource2())
 {
  goto cleanup2;
 }
 mainFunctionality();
cleanup2:
 cleanupResource2();
cleanup1:
 cleanupResource1();
}
```

Если принятый в вашей организации стандарт кодирования запрещает использование `goto`, то его можно эмулировать, поместив код внутрь цикла `do{ ... }while(0);`. В случае ошибки выполните `break`, чтобы выйти из цикла туда, где находится код обработки ошибок. Однако обычно в таком обходном маневре нет ничего хорошего, потому что если стандарт кодирования не разрешает использовать `goto`, то не нужно эмулировать его, только чтобы настоять на своем собственном стиле программирования. В качестве альтернативы `goto` можно использовать паттерн «Запись об очистке».

В любом случае использование `goto` может указывать на то, что ваша функция уже слишком сложна, и лучше бы разбить ее на части, воспользовавшись, например, паттерном «Объектная обработка ошибок».

goto: добро или зло?

Много спорят о том, хорошо или плохо использовать предложение `goto`. Самая знаменитая статья о вреде использования `goto` принадлежит перу Эдсгера В. Дейкстры, который аргументирует свою точку зрения тем, что `goto` мешает следить за потоком выполнения программы. Это правда, если `goto` используется для переходов вперед и назад, но в C `goto` невозможно использовать так же безответственно, как в тех языках, о которых писал Дейкстра (в C `goto` не может выводить за пределы функции).

Последствия

Функция имеет единственную точку возврата, а основной поток программы отделен от обработки ошибок и очистки ресурсов. Чтобы достичь этого результата, не понадобилось вложенных предложений `if`, но не все привыкли и готовы читать код с предложениями `goto`.

Допуская предложения `goto`, контролируйте себя, потому что может возникнуть соблазн использовать их не только для обработки ошибок и очистки, а вот тогда код точно станет нечитаемым. Кроме того, внимательно следите за

соответствием между метками и функциями очистки. Типичная ошибка – помещать функцию очистки не под той меткой[1].

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В ядре Linux в основном используется обработка ошибок с применением `goto`. Например, в книге Alessandro Rubini and Jonathan Corbet «Linux Device Drivers» (O'Reilly, 2001) описывается такой подход для программирования драйверов устройств в Linux.
- В книге C. Seacord «The CERT C Coding Standard by Robert» (Addison-Wesley Professional, 2014) рекомендуется использовать `goto` для обработки ошибок.
- Эмуляция `goto` с помощью цикла `do-while` описана в портлендском репозитории паттернов под названием Тривиальный цикл do-while.
- В коде OpenSSL используется `goto`. Например, в функциях для обработки сертификатов X509 `goto` применяется для перехода вперед на центральный обработчик ошибок.
- В коде Wireshark `goto` используется в функции `main` для перехода на центральный обработчик ошибок, расположенный в конце функции.

Применение к сквозному примеру

Хотя довольно много людей категорически не одобряют использование `goto`, обработка ошибок стала лучше по сравнению с предыдущим примером. В следующем коде нет вложенных предложений `if`, и код очистки четко отделен от основного потока программы:

```
int parseFile(char* file_name)
{
 int return_value = ERROR;
 FILE* file_pointer = 0;
 char* buffer = 0;

 assert(file_name!=NULL && "Недопустимое имя файла");
 if(!(file_pointer=fopen(file_name, "r")))
 {
  goto error_fileopen;
 }
 if(!(buffer=malloc(BUFFER_SIZE)))
 {
  goto error_malloc;
 }
 return_value = searchFileForKeywords(buffer, file_pointer);
 free(buffer);
```

[1] Ну в точности так, как в примере, приведенном самим автором. Врачу, исцелися сам. – *Прим. перев.*

```
error_malloc:
 fclose(file_pointer);
error_fileopen:
 return return_value;
}
```

А теперь допустим, что вы сами не любите `goto` или принятые в организации стандарты кодирование запрещают их использование, но очищать ресурс все равно надо. Что ж, есть альтернативы. Например, можно использовать паттерн «Запись об очистке».

Запись об очистке

Контекст

Имеется функция, которая захватывает и освобождает несколько ресурсов. Быть может, вы уже пытались уменьшить ее сложность, применяя паттерны «Проверка условий», «Разбиение функции» или «Принцип самурая», но все же остались глубоко вложенные `if`, связанные с выделением ресурсов. Возможно даже, что код очистки ресурсов дублируется. Принятые в организации стандарты кодирования запрещают использовать паттерн «Переход к обработке ошибки», или вы сами не хотите использовать `goto`.

Проблема

Код функции становится трудно читать и сопровождать, если он захватывает и освобождает несколько ресурсов, особенно когда эти ресурсы зависят друг от друга.

Затруднения вызваны тем, что выделение любого ресурса может завершиться неудачно, а освобождать ресурс нужно, только если он был выделен успешно. Чтобы обеспечить это, необходимо много предложений `if`, но при плохой реализации наличие вложенных `if` в одной функции затрудняет чтение и сопровождение кода.

Поскольку ресурсы необходимо освобождать, возврат в середине функции, если что-то пошло не так, – не самая лучшая мысль. Ведь все уже захваченные ресурсы нужно будет освобождать перед каждым предложением `return`. Поэтому в коде образуется несколько мест, где освобождается один и тот же ресурс, но мы не хотим дублировать код обработки ошибок и очистки.

Решение

Вызывайте функции захвата ресурсов, пока они завершаются успешно, и сохраняйте информацию о том, какие ресурсы нуждаются в очистке. В зависимости от того, что сохранено, вызывайте функции очистки.

В языке C для реализации этой идеи можно воспользоваться ленивым вычислением предложения `if`. Просто вызывайте функции одну за другой в одном предложении `if`, пока функции завершаются успешно. Для каждого вызова функции сохраняйте захваченный ресурс в переменной. Код, работающий

с ресурсами, поместите в тело предложения `if`, а все освобождение ресурсов – после предложения `if`. При этом освобождать нужно только те ресурсы, которые были успешно захвачены. Пример приведен ниже:

```
void someFunction()
{
 if((r1=allocateResource1()) && (r2=allocateResource2()))
 {
  mainFunctionality();
 }
 if(r1) ❶
 {
  cleanupResource1();
 }
 if(r2) ❶
 {
  cleanupResource2();
 }
}
```

❶ Чтобы код было проще читать, эти проверки можно поместить внутрь функций очистки. Это разумно, если функции очистке все равно нужно передавать указатель на ресурс.

Последствия

Теперь не осталось ни одного вложенного `if`, а в конце функции имеется центральная точка для очистки ресурсов. Код стало значительно проще читать, потому что основному потоку программы больше не мешает обработка ошибок.

Кроме того, функцию проще читать, потому что в ней всего одна точка выхода. Однако из-за того, что появилось много переменных для учета того, какие ресурсы были успешно захвачены, код усложнился. Быть может, паттерн «Агрегат» поможет структурировать эти переменные.

Если захватывается много ресурсов, то в одном предложении `if` вызывается много функций. Такое предложение очень трудно читать и еще труднее отлаживать. Поэтому когда захватывается много ресурсов, гораздо лучше применить «Объектную обработку ошибок».

Еще одна причина применить «Объектную обработку ошибок» вместо «Записи об очистке» заключается в том, что показанный выше код все еще сложен, так как состоит из единственной функции, которая содержит как основную функциональность, так и логику захвата и освобождения ресурсов. То есть у одной функции несколько обязанностей.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В портлендском репозитории паттернов представлено похожее решение, в котором каждая вызванная функция регистрирует обработчик очистки в списке обратных вызовов. Для очистки вызываются все функ-

ции из этого списка.

- В функции `dh_key2buf` из библиотеки OpenSSL ленивое вычисление `if` применяется для отслеживания выделенных байтов, которые освобождаются впоследствии.
- В функции `cap_open_socket` из сетевого анализатора Wireshark ленивое вычисление `if` используется для сохранения выделенных ресурсов в переменных. На этапе очистки эти переменные проверяются, и если ресурс был выделен успешно, то он освобождается.
- В исходном коде OpenWrt функция `nvram_commit` выделяет ресурсы внутри предложения `if` и сохраняет их в переменных прямо в том же `if`.

Применение к сквозному примеру

Теперь вместо `goto` и вложенных `if` мы имеем единственное предложение `if`. Преимущество отказа от `goto` в показанном ниже коде заключается в том, что обработка ошибок четко отделена от основного потока программы:

```
int parseFile(char* file_name)
{
 int return_value = ERROR;
 FILE* file_pointer = 0;
 char* buffer = 0;

 assert(file_name!=NULL && "Недопустимое имя файла");
 if((file_pointer=fopen(file_name, "r")) &&
   (buffer=malloc(BUFFER_SIZE)))
 {
  return_value = searchFileForKeywords(buffer, file_pointer);
 }
 if(file_pointer)
 {
  fclose(file_pointer);
 }
 if(buffer)
 {
  free(buffer);
 }
 return return_value;
}
```

И все же код выглядит коряво. У этой функции слишком много обязанностей: выделение ресурсов, освобождение ресурсов, работа с файлом и обработка ошибок. Эти обязанности следует разнести по разным функциям, воспользовавшись паттерном «Объектная обработка ошибок».

Объектная обработка ошибок

Контекст

Имеется функция, которая захватывает и освобождает несколько ресурсов. Быть может, вы уже пытались уменьшить ее сложность, применяя паттерны «Проверка условий», «Разбиение функции» или «Принцип самурая», но все же остались глубоко вложенные `if`, связанные с выделением ресурсов. Возможно даже, что код очистки ресурсов дублируется. Но, быть может, вы уже избавились от вложенных предложений `if`, применив паттерн «Переход к обработке ошибки» или «Запись об очистке».

Проблема

Наличие нескольких обязанностей у одной функции, например выделение, освобождение и использование ресурса, затрудняет реализацию, чтение, сопровождение и тестирование.

Затруднения вызваны тем, что выделение любого ресурса может завершиться неудачно, а освобождать ресурс нужно, только если он был выделен успешно. Чтобы обеспечить это, необходимо много предложений `if`, но при плохой реализации наличие вложенных `if` в одной функции затрудняет чтение и сопровождение кода.

Поскольку ресурсы необходимо освобождать, возврат в середине функции, если что-то пошло не так, – не самая лучшая мысль. Ведь все уже захваченные ресурсы нужно будет освобождать перед каждым предложением `return`. Поэтому в коде образуется несколько мест, где освобождается один и тот же ресурс, но мы не хотим дублировать код обработки ошибок и очистки.

Даже после применения паттернов «Запись об очистке» или «Переход к обработке ошибки» функцию все еще трудно читать, так как в ней смешаны разные обязанности. Функция отвечает за захват нескольких ресурсов, обработку ошибок и освобождение ресурсов. Однако каждая функция должна отвечать за что-то одно.

Решение

Поместите инициализацию и очистку в разные функции, как это делается в конструкторах и деструкторах в объектно ориентированном программировании.

В главной функции просто вызовите одну функцию, которая захватит все ресурсы, другую функцию, которая будет работать с этими ресурсами, и третью функцию, которая освободит все ресурсы.

Если захваченные ресурсы не глобальны, то их нужно передавать между функциями. Когда ресурсов несколько, можно передавать содержащий их «Агрегат». Если вы хотите скрыть сами ресурсы от вызывающей стороны, то можете использовать «Описатель» для передачи функциям информации о ресурсах.

Если выделение ресурса завершилось неудачно, сохраните информацию об этом в переменной (например, указатель `NULL` свидетельствует об ошибке вы-

деления памяти). При использовании или очистке ресурсов сначала проверяйте, действителен ли ресурс. Эту проверку производите не в главной функции, а в вызываемых, тогда код главной функции будет гораздо легче читать.

```
void someFunction()
{
 allocateResources();
 mainFunctionality();
 cleanupResources();
}
```

Последствия

Теперь код функции легко читать. Хотя выделение, освобождение и использование нескольких ресурсов никуда не делись, эти разные по характеру задачи разнесены по нескольким функциям.

Заведение похожих на объекты экземпляров, которые передаются между функциями, называется «объектным» стилем программирования. При таком подходе процедурное программирование больше напоминает объектно ориентированное, поэтому код, написанный в этом стиле, выглядит более знакомым программистам, привыкшим к объектной ориентированности.

В главной функции больше нет причин для появления нескольких предложений `return`, потому что не осталось вложенных `if`, реализующих логику выделения и освобождения ресурсов. Но, конечно, эта логика не исчезла совсем, а просто перенесена в отдельные функции, чтобы избежать смешения с операциями над ресурсами.

Вместо одной функции мы теперь имеем несколько. Это может снизить производительность, но обычно снижение настолько незначительно, что в большинстве приложений им можно пренебречь.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Эта форма очистки применяется в объектно ориентированном программировании, где неявно вызываются конструкторы и деструкторы.
- Этот паттерн используется в коде OpenSSL. Например, выделение и освобождение буферов реализовано функциями `BUF_MEM_new` и `BUF_MEM_free`, которые вызываются повсеместно для работы с буферами.
- Функция `show_help` в исходном коде OpenWrt отображает справочную информацию в контекстном меню. Она вызывает функцию инициализации, чтобы создать структуру `struct`, затем производит действия с этой структурой и напоследок вызывает функцию для ее освобождения.
- Функция `cmd__windows_named_pipe` из проекта Git использует паттерн «Описатель» для создания канала, затем работает с этим каналом и вызывает отдельную функцию для его очистки.

Применение к сквозному примеру

Наконец-то мы пришли к следующему коду, в котором функция `parseFile` вызывает другие функции для создания и очистки экземпляра анализатора:

```
typedef struct
{
 FILE* file_pointer;
 char* buffer;
} FileParser;

int parseFile(char* file_name)
{
 int return_value;
 FileParser* parser = createParser(file_name);
 return_value = searchFileForKeywords(parser);
 cleanupParser(parser);
 return return_value;
}

int searchFileForKeywords(FileParser* parser)
{
 if(parser == NULL)
 {
  return ERROR;
 }
 while(fgets(parser->buffer, BUFFER_SIZE, parser->file_pointer)!=NULL)
 {
  if(strcmp("KEYWORD_ONE\n", parser->buffer)==0)
  {
   return KEYWORD_ONE_FOUND_FIRST;
  }
  if(strcmp("KEYWORD_TWO\n", parser->buffer)==0)
  {
   return KEYWORD_TWO_FOUND_FIRST;
  }
 }
 return NO_KEYWORD_FOUND;
}

FileParser* createParser(char* file_name)
{
 assert(file_name!=NULL && "Недопустимое имя файла");
 FileParser* parser = malloc(sizeof(FileParser));
 if(parser)
 {
  parser->file_pointer=fopen(file_name, "r");
```

```
    parser->buffer = malloc(BUFFER_SIZE);
    if(!parser->file_pointer || !parser->buffer)
    {
     cleanupParser(parser);
     return NULL;
    }
   }
   return parser;
  }

  void cleanupParser(FileParser* parser)
  {
   if(parser)
   {
    if(parser->buffer)
    {
     free(parser->buffer);
    }
    if(parser->file_pointer)
    {
     fclose(parser->file_pointer);
    }
    free(parser);
   }
  }
```

В этом коде больше нет каскада `if` в основном потоке программы. Поэтому функцию `parseFile` гораздо проще читать, отлаживать и сопровождать. Главная функция не занимается выделением ресурсов, освобождением ресурсов и обработкой ошибок. Все эти детали перенесены в отдельные функции, каждая из которых отвечает за что-то одно.

Оцените, насколько элегантнее стал код по сравнению с первоначальной версией. Шаг за шагом применение паттернов делало код проще для чтения и сопровождения. На каждом шаге мы удаляли каскад `if` и улучшали методику обработки ошибок.

Резюме

В этой главе было показано, как выполнять обработку ошибок в программах на C. Паттерн «Разбиение функции» рекомендует разделять функцию на меньшие части, чтобы упростить обработку ошибок в этих частях. Паттерн «Проверка условий» рекомендует проверять предусловия функции и возвращать управление немедленно, если они не выполнены. Это оставляет на долю остального кода меньше забот об обработке ошибок. Вместо того чтобы возвращать управление из функции, можно сразу завершить программу, следуя «Принципу самурая». Что до более сложной обработки ошибок – особенно в сочетании с захватом и освобождением нескольких ресурсов, – в вашем распоря-

жении имеется несколько паттернов на выбор. «Переход к обработке ошибки» рекомендует совершать прямой переход в точку обработки ошибки. Паттерн «Запись об очистке» рекомендует вместо перехода сохранять информацию о том, какие ресурсы нуждаются в очистке, и выполнять эту очистку в конце функции. Метод захвата ресурсов, более близкий к объектно ориентированному программированию, предлагает паттерн «Объектная обработка ошибок», в котором используются отдельные функции инициализации и очистки по аналогии с идеей конструкторов и деструкторов.

Имея в своем арсенале эти паттерны обработки ошибок, вы сможете писать небольшие программы, которые обрабатывают ошибки, не жертвуя удобством сопровождения кода.

Для дополнительного чтения

Если у вас проснулся аппетит, то вот еще несколько ресурсов, которые расширят ваши знания об обработке ошибок.

- В портлендском репозитории паттернов (https://oreil.ly/qFLDa) предлагается много паттернов с обсуждениями на тему обработки ошибок и не только. В большинстве паттернов обработки ошибок речь идет об обработке исключений и использовании утверждений, но есть и несколько паттернов для C.
- Исчерпывающий обзор обработки ошибок вообще имеется в магистерской диссертации Томаса Аглассингера «Error Handling in Structured and Object-Oriented Programming Languages» (Университет Оулу, 1999). Описываются различные типы ошибок, обсуждаются механизмы обработки ошибок в языках программирования C, Basic, Java и Eiffel, и даются рекомендации, в частности, выполнять очистку ресурсов в порядке, противоположном их выделению. В диссертации упоминаются также сторонние решения в форме библиотеки, предлагающей улучшенные средства обработки ошибок для C, например обработку исключений с помощью команд `setjmp` и `longjmp`.
- Пятнадцать объектно ориентированных паттернов обработки ошибок, адаптированных для деловых систем, представлены в статье Klaus Renzel «Error Handling for Business Information Systems» (https://oreil.ly/bQnfx), и большая их часть применима не только к объектно ориентированным программам. Представленные паттерны охватывают обнаружение, протоколирование и обработку ошибок.
- Реализации некоторых паттернов проектирования из книги «банды четырех», включающие фрагменты кода на C, приведены в книге Adam Tornhill «Patterns in C» (Leanpub, 2014). Там приводится описание передовых практик в форме паттернов C, некоторые из которых относятся к обработке ошибок.
- Собрание паттернов для обработки и протоколирования ошибок представлено в статьях Andy Longshaw and Eoin Woods (https://oreil.ly/7Yj8h) «Patterns for Generation, Handling and Management of Errors» и «More

Patterns for the Generation, Handling and Management of Errors». Большинство из них ориентировано на обработку ошибок в форме исключений.

Что дальше

В следующей главе показано, как обрабатывать ошибки в более крупных программах, которые возвращают информацию об ошибке в интерфейсах к другим функциям. Паттерны рекомендуют, какого рода информацию об ошибке возвращать и как именно.

Глава 2

Возврат информации об ошибке

Предыдущая глава была посвящена обработке ошибок. Здесь мы продолжим это обсуждение, но сместим акцент на то, как информировать пользователей об обнаруженных ошибках.

В любой большой программе программист должен решить, как реагировать на ошибки, возникающие в его собственном коде, как реагировать на ошибки в чужом коде, как передавать информацию об ошибке и как представлять ее пользователям.

В большинстве объектно ориентированных языков имеется удобный механизм исключений, дающий программисту дополнительный канал возврата информации об ошибке, но в C такого встроенного механизма нет. Существуют способы эмулировать обработку исключений и даже наследование исключений в C, например описанные в книге Axel-Tobias Schreiner «Object-Oriented Programming with ANSI-C». Но для программистов, работающих с унаследованным кодом на C или желающих придерживаться свойственного C стиля, такие механизмы исключений неприемлемы. Взамен им нужны рекомендации по применению механизмов обработки ошибок, которые уже имеются в C.

В этой главе как раз и даются рекомендации о передаче информации об ошибках между функциями и интерфейсами. На рис. 2.1 приведен обзор паттернов, рассматриваемых в этой главе и связей между ними, а в табл. 2.1 – краткое описание этих паттернов.

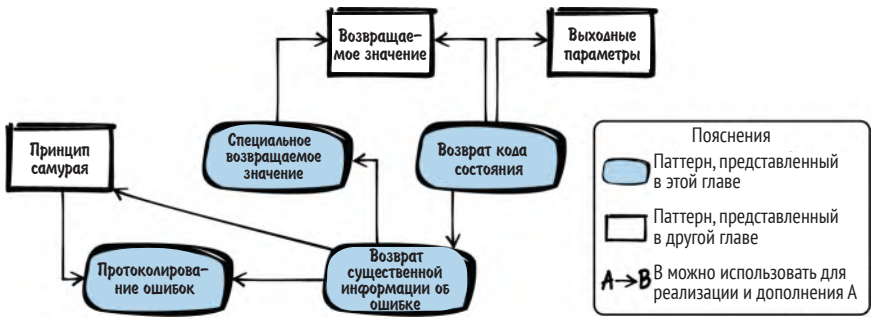


Рис. 2.1. Обзор паттернов для возврата информации об ошибках

Таблица 2.1. Паттерны для возврата информации об ошибках

Название паттерна	Краткое описание
Возврат кода состояния	Вам нужен механизм возврата информации о состоянии вызывающей стороне, чтобы та могла на нее отреагировать. Механизм должен быть простым в использовании, а вызывающая сторона не должна путать разные ошибки. Поэтому используйте возвращаемое значение функции для возврата информации о состоянии. Возвращайте значение, представляющее конкретное состояние. Вызывающая и вызываемая стороны должны одинаково интерпретировать возвращаемые значения
Возврат существенной информации об ошибке	С одной стороны, вызывающая сторона должна иметь возможность реагировать на ошибки, а с другой стороны, чем больше информации об ошибке вы возвращаете, тем длиннее становится ваш код и код для обработки ошибки. Поэтому возвращайте только ту информацию об ошибке, которая существенна для вызывающей стороны. Информация существенна, если вызывающая сторона может на нее отреагировать
Специальное возвращаемое значение	Вы хотите вернуть информацию об ошибке, но явно возвращать коды состояния не годится, потому что тогда нельзя использовать возвращаемое функцией значения для возврата других данных. Если использовать выходные параметры, то вызывать функцию станет труднее. Поэтому используйте возвращаемое значение для возврата вычисленных функцией данных, но зарезервируйте одно или несколько специальных значений на случай ошибки
Протоколирование ошибок	Вы хотите быть уверены, что в случае ошибки сумеете легко определить ее причину. Но не хотите ради этого усложнять код обработки ошибок. Поэтому используйте разные каналы: один для предоставления информации об ошибке, существенной для вызывающей стороны, а другой для предоставления информации, существенной для разработчика. Например, записывайте отладочную информацию об ошибке в файл журнала и не возвращайте ее вызывающей стороне

Сквозной пример

Требуется реализовать программный модуль, предоставляющий возможность хранить строковые значения строковых ключей. Иными словами, нужна функциональность, похожая на реестр Windows. Для простоты в следующем коде не реализованы иерархические связи между ключами и обсуждаются только функции создания элементов реестра.

API реестра

```
/* Описатель раздела реестра */
typedef struct Key* RegKey;

/* Создать новый раздел реестра с именем 'key_name' */
RegKey createKey(char* key_name);

/* Сохранить значение 'value' в разделе с именем ''key' */
void storeValue(RegKey key, char* value);

/* Сделать раздел доступным для чтения (другими функциями,
   которые в этом примере не рассматриваются) */
void publishKey(RegKey key);
```

Реализация реестра

```
#define STRING_SIZE 100
#define MAX_KEYS 40
struct Key
{
 char key_name[STRING_SIZE];
 char key_value[STRING_SIZE];
};

/* глобальный на уровне файла массив, содержащий все разделы реестра */
static struct Key* key_list[MAX_KEYS];
RegKey createKey(char* key_name)
{
 RegKey newKey = calloc(1, sizeof(struct Key));
 strcpy(newKey->key_name, key_name);
 return newKey;
}

void storeValue(RegKey key, char* value)
{
 strcpy(key->key_value, value);
}

void publishKey(RegKey key)
{
 int i;
 for(i=0; i<MAX_KEYS; i++)
 {
  if(key_list[i] == NULL)
  {
   key_list[i] = key;
   return;
```

```
  }
 }
}
```

Здесь мы не уверены, как следует передавать вызывающей стороне информацию о внутренних ошибках или, скажем, о недопустимых параметрах функции. Вызывающая сторона не знает, завершился ли вызов успешно, и в итоге получается такой код:

```
RegKey my_key = createKey("myKey");
storeValue(my_key, "A");
publishKey(my_key);
```

На вызывающей стороне код очень короткий и легко читается, но она не знает, имела ли место ошибка, и не может на нее отреагировать. Чтобы дать ей такую возможность, мы должны включить обработку ошибок в собственный код и передавать вызывающей стороне информацию об ошибках. Первое, что приходит на ум, – сообщать вызывающей стороне обо всех ошибках в программном модуле. Для этого мы будем возвращать коды состояния.

Возврат кода состояния

Контекст

Вы реализуете программный модуль, который выполняет какую-то обработку ошибок, и хотите возвращать вызывающей стороне информацию об ошибках и другую информацию о состоянии.

Проблема

Необходим механизм возврата информации о состоянии вызывающей стороне, чтобы она могла отреагировать. Механизм должен быть простым в использовании, и у вызывающей стороны не должно возникать трудностей в различении возможных ошибочных ситуаций.

На заре развития C информация об ошибках передавалась в виде кода ошибки в глобальной переменной `errno`. Вызывающая сторона должна была сбросить эту переменную, затем вызвать функцию, а та в случае ошибки устанавливала `errno`, ожидая, что вызывающая сторона проверит ее после вызова.

Однако нам хотелось бы иметь способ возврата информации о состоянии, который был бы проще использования `errno`. Вызывающая сторона должна по сигнатуре функции понимать, как будет возвращена информация о состоянии и какого рода информацию ожидать.

Кроме того, механизм возврата информации о состоянии должен быть безопасен в многопоточном окружении, и только вызванная функция должна иметь возможность повлиять на эту информацию. Иными словами, механизм не должен препятствовать реентерабельности функции.

Решение

Используйте возвращаемое значение функции для возврата информации о состоянии. Возвращайте значение, отражающее конкретное состояние. И вызывающая, и вызываемая сторона должны одинаково понимать, что означает возвращенное значение.

Обычно возвращаемое значение – числовой идентификатор. Вызывающая сторона может проверить этот идентификатор и действовать соответственно. Если функция должна возвращать еще какие-то результаты, передавайте их вызывающей стороне в виде выходных параметров.

Определите в своем API идентификаторы состояний с помощью `enum` или `#define`. Если кодов состояния много или программный модуль содержит более одного заголовочного файла, то можно завести отдельный заголовочный файл, который содержит только коды состояния и включается во все остальные заголовочные файлы.

Придумайте осмысленные имена идентификаторов состояний и документируйте их назначение в комментариях. Имена должны быть единообразными.

Ниже приведен пример использования кодов состояний.

Использование кодов состояний в вызывающем коде

```
ErrorCode status = func();
if(status == MAJOR_ERROR)
{
 /* завершить программу */
}
else if(status == MINOR_ERROR)
{
 /* обработать ошибку */
}
else if(status == OK)
{
 /* продолжить выполнение */
}
```

API вызываемой стороны, возвращающей коды состояний

```
typedef enum
{
 MINOR_ERROR,
 MAJOR_ERROR,
 OK
} ErrorCode;

ErrorCode func();
```

Реализация вызываемой стороны, возвращающей коды состояний

```
ErrorCode func()
{
 if(minorErrorOccurs())
 {
  return MINOR_ERROR;
 }
 else if(majorErrorOccurs())
 {
  return MAJOR_ERROR;
 }
 else
 {
  return OK;
 }
}
```

Последствия

Теперь у нас есть способ возврата информации о состоянии, позволяющий вызывающей стороне очень просто проверить, была ли при выполнении ошибка. По сравнению с `errno`, вызывающей стороне не нужно раздельно устанавливать и проверять значение переменной в дополнение к вызову функции. Вместо этого она проверяет само возвращенное функцией значение.

Возврат кода состояния можно безопасно использовать в многопоточной среде. Вызывающая сторона может быть уверена, что только вызванная функция и никто другой определяет возвращенное состояние.

Сигнатура функции недвусмысленно показывает, как возвращается информация о состоянии. Это ясно как вызывающей стороне, так и компилятору или средствам статического анализа, которые могут удостовериться, что вызывающая сторона проверила возвращенное значение, сравнив его со всеми возможными кодами состояний.

Поскольку теперь функция возвращает разные результаты при разных ошибках, эти результаты необходимо тестировать. По сравнению с функцией, не обрабатывающей ошибки, необходимо более полное тестирование. Кроме того, на вызывающую сторону возлагается бремя проверки всех возможных ошибок, что может привести к увеличению ее размера.

В C любая функция может возвращать только один объект того типа, который указан в сигнатуре функции, и наша функция теперь возвращает код состояния. Поэтому для возврата других значений придется использовать более сложную технику. Это можно сделать с помощью паттерна «Выходные параметры», у которого есть недостаток – необходимость дополнительного параметра, или же воспользовавшись «Агрегатом», содержащим как информацию о состоянии, так и другие вычисленные функцией результаты.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Microsoft использует `HRESULT` для возврата информации о состоянии. `HRESULT` – это уникальный код состояния. Уникальность хороша тем, что информацию о состоянии можно передавать по цепочке из многих функций, сохраняя возможность узнать о первоначальном источнике. Но для обеспечения уникальности требуются дополнительные усилия: нужно аккуратно присваивать числовые коды и следить за тем, кому какие коды состояния разрешено использовать. Еще одна особенность `HRESULT` состоит в том, что он кодирует определенную информацию, в частности серьезность ошибки, в коде состояния, используя для этого выделенные биты.
- В коде Apache Portable Runtime определен тип `apr_status_t` для возврата информации об ошибке. Функция, пользующаяся этим типом, возвращает `APR_SUCCESS` в случае успеха, а любое другое значение говорит об ошибке. Все значения кодов ошибок уникальные и определены директивами `#define`.
- В коде OpenSSL коды состояния определены в нескольких заголовочных файлах (*dsaerr.h*, *kdferr.h*...). Например, коды состояния `KDF_R_MISSING_PARAMETER` или `KDF_R_MISSING_SALT` сообщают вызывающей стороне об отсутствующих или недопустимых входных параметрах. В каждом файле определены коды состояний только для определенного множества функций, связанных с этим файлом. Коды состояний не являются глобально уникальными в пределах всего кода OpenSSL.
- В портлендском репозитории описан паттерн «Код ошибки». Он описывает идею возврата информацию об ошибке путем явного возврата значения.

Применение к сквозному примеру

Теперь мы передаем вызывающей стороне информацию о возникших в коде ошибках. В примере ниже проверяется, что могло бы пойти не так, и эта информация возвращается вызывающей стороне.

API реестра

```
/* Коды ошибок, возвращаемые реестром */
typedef enum
{
 OK,
 OUT_OF_MEMORY,
 INVALID_KEY,
 INVALID_STRING,
 STRING_TOO_LONG,
 CANNOT_ADD_KEY
```

```
} RegError;

/* Описатель для разделов реестра */
typedef struct Key* RegKey;

/* Создать новый раздел реестра с именем 'key_name'.
   Возвращает OK, если ошибок не было, INVALID_KEY; если параметр 'key'
   равен NULL; INVALID_STRING, если имя 'key_name' равно NULL;
   STRING_TOO_LONG, если имя 'key_name' слишком длинное; OUT_OF_MEMORY
   в случае нехватки памяти. */
RegError createKey(char* key_name, RegKey* key);

/* Сохранить значение 'value' в разделе с именем 'key'.
   Возвращает OK, если ошибок не было, INVALID_KEY; если параметр 'key'
   равен NULL; INVALID_STRING, если 'value' равно NULL;
   STRING_TOO_LONG, если имя 'key_name' слишком длинное. */
RegError storeValue(RegKey key, char* value);

/* Сделать раздел доступным для чтения.
   Возвращает OK, если ошибок не было, INVALID_KEY; если параметр 'key'
   равен NULL; CANNOT_ADD_KEY, если реестр заполнен и новых разделов
   опубликовать нельзя. */
RegError publishKey(RegKey key);
```

Реализация реестра

```
#define STRING_SIZE 100
#define MAX_KEYS 40

struct Key
{
 char key_name[STRING_SIZE];
 char key_value[STRING_SIZE];
};

/* глобальный в области видимости файла массив, содержащий
   все разделы реестра */
static struct Key* key_list[MAX_KEYS];

RegError createKey(char* key_name, RegKey* key)
{
 if(key == NULL)
 {
  return INVALID_KEY;
 }

 if(key_name == NULL)
```

```
 {
  return INVALID_STRING;
 }

 if(STRING_SIZE <= strlen(key_name))
 {
  return STRING_TOO_LONG;
 }

 RegKey newKey = calloc(1, sizeof(struct Key));
 if(newKey == NULL)
 {
  return OUT_OF_MEMORY;
 }

 strcpy(newKey->key_name, key_name);
 *key = newKey;
 return OK;
}

RegError storeValue(RegKey key, char* value)
{
 if(key == NULL)
 {
  return INVALID_KEY;
 }
 if(value == NULL)
 {
  return INVALID_STRING;
 }
 if(STRING_SIZE <= strlen(value))
 {
  return STRING_TOO_LONG;
 }
 strcpy(key->key_value, value);
 return OK;
}

RegError publishKey(RegKey key)
{
 int i;
 if(key == NULL)
 {
  return INVALID_KEY;
 }
 for(i=0; i<MAX_KEYS; i++)
 {
```

```
  if(key_list[i] == NULL)
  {
   key_list[i] = key;
   return OK;
  }
 }

 return CANNOT_ADD_KEY;
}
```

Теперь вызывающая программа может отреагировать на полученную информацию об ошибке и предоставить пользователю подробный отчет о том, что произошло.

Код вызывающей стороны

```
RegError err;
RegKey my_key;

err = createKey("myKey", &my_key);
if(err == INVALID_KEY || err == INVALID_STRING)
{
 printf("Внутренняя ошибка приложения\n");
}
if(err == STRING_TOO_LONG)
{
 printf("Слишком длинное имя раздела реестра\n");
}
if(err == OUT_OF_MEMORY)
{
 printf("Недостаточно памяти для создания раздела\n");
}

err = storeValue(my_key, "A");
if(err == INVALID_KEY || err == INVALID_STRING)
{
 printf("Внутренняя ошибка приложения\n");
}
if(err == STRING_TOO_LONG)
{
 printf("Значение слишком длинное для сохранения в этом разделе\n");
}

err = publishKey(my_key);
if(err == INVALID_KEY)
{
 printf("Внутренняя ошибка приложения\n");
}
```

```
if(err == CANNOT_ADD_KEY)
{
 printf("Раздел невозможно опубликовать, потому что реестр заполнен\n");
}
```

Теперь вызывающая сторона может реагировать на ошибки, но размер модуля, равно как и размер вызывающей стороны увеличился более чем в два раза. Код вызывающей стороны можно было бы немного почистить, написав отдельную функцию для отображения кода ошибки на текст сообщения, но большая часть кода, связанная с обработкой ошибок, все равно останется.

Как видим, обработка ошибок обходится недаром. Мы вложили немало усилий в ее реализацию. Это видно и в API реестра. Комментарии к функциям стали гораздо длиннее, потому что описывают возможные ошибочные ситуации. Вызывающая сторона должна также тщательно обдумывать, что делать при возникновении определенной ошибки.

Передавая настолько подробную информацию вызывающей стороне, вы возлагаете на нее бремя реагирования на ошибки и решения о том, какие ошибки существенны, а какие – нет. Поэтому нужно думать о том, чтобы, с одной стороны, предоставить вызывающей стороне необходимую информацию, а с другой стороны, не перегрузить ее лишней информацией.

Наш следующий шаг – учесть эти соображения в коде и предоставлять только ту информацию об ошибках, которая действительно полезна вызывающей стороне. Иначе говоря, возвращать только существенную информацию.

Возврат существенной информации об ошибке

Контекст

Вы пишете программный модуль, который выполняет какую-то обработку ошибок, и хотите возвращать информацию об ошибках вызывающей стороне.

Проблема

С одной стороны, вызывающая сторона должна иметь возможность реагировать на ошибки, а с другой стороны, чем больше информации возвращается, тем длиннее код обработки ошибок – ваш и вызывающей стороны. Длинный код труднее читать и сопровождать, к тому же повышается риск внесения новых ошибок.

Говоря о возврате информации об ошибке вызывающей стороне, помните, что обнаружение ошибки и возврат информации о ней не единственные ваши задачи. Необходимо также документировать в API, какие ошибки возвращаются. Если этого не сделать, то вызывающая сторона не будет знать, к чему готовиться. Документирование ошибок – обязательная работа. Чем больше типов ошибок, тем больше объем этой работы.

Если возвращать очень подробную, зависящую от реализации информацию об ошибках, да еще и расширять ее впоследствии при изменении реализации, то при любом таком изменении придется вносить изменения в семантику интерфейса и документировать их. Это может быть нежелательно для существу-

ющих клиентов, потому что они должны будут адаптировать свой код к вновь появившейся информации об ошибках.

Предоставление излишне детальной информации – не всегда благо и для вызывающей стороны. Любая возвращенная ей информация об ошибке означает дополнительную работу. Вызывающая сторона должна решить, существенна ли эта информация и как ее обработать.

Решение

Возвращайте информацию об ошибке, только если она существенна для вызывающей стороны. Информация об ошибке существенна, если вызывающая сторона может на нее отреагировать.

Если вызывающая сторона не может отреагировать на информацию об ошибке, то и не нужно предоставлять ей такую возможность (или бремя).

Существует несколько способов возвращать только существенную информацию. Крайний способ – не возвращать вообще ничего. Например, функции `cleanupMemory (void* handle)`, очищающей память, нет нужды сообщать вызывающей стороне об успехе или неудаче операции, потому что та в любом случае не сможет отреагировать на ошибку (повторный вызов функции очистки в большинстве случае не решает проблему). Поэтому функция не возвращает вообще ничего. Чтобы ошибка не осталась незамеченной, можно даже принудительно завершать программу («Принцип самурая»).

Или допустим, что единственная причина возвращать информацию об ошибке вызывающей стороне – дать ей возможность запротоколировать ошибку. В таком случае не возвращайте ничего, а просто протоколируйте ошибки самостоятельно, чтобы облегчить жизнь вызывающей стороне.

Если вы уже возвращаете коды состояния, то следует возвращать только информацию об ошибке, существенную для вызывающей стороны. Все прочие ошибки можно свести к одному коду внутренней ошибки. Кроме того, детальные коды ошибок от вызываемых вами функций необязательно возвращать в том же виде из своей функции. Их можно свести в один код внутренней ошибки, как показано ниже.

Код вызывающей стороны

```
ErrorCode status = func();
if(status == MAJOR_ERROR || status == UNKNOWN_ERROR)
{
 /* завершить программу */
}
else if(status == MINOR_ERROR)
{
 /* обработать ошибку */
}
else if(status == OK)
{
 /* продолжить выполнение */
```

```
}
API
typedef enum
{
 MINOR_ERROR,
 MAJOR_ERROR,
 UNKNOWN_ERROR,
 OK
} ErrorCode;

ErrorCode func();
Реализация
ErrorCode func()
{
 if(minorErrorOccurs())
 {
  return MINOR_ERROR;
 }
 else if(majorErrorOccurs())
 {
  return MAJOR_ERROR;
 }
 else if(internalError1Occurs() || internalError2Occurs())
 {
  return UNKNOWN_ERROR; ❶
 }
 else
 {
  return OK;
 }
}
```

❶ Если имеет место `internalError1Occurs` или `internalError2Occurs`, то мы возвращаем одну и ту же ошибку, потому что вызывающей стороне безразлично, какая из двух зависящих от реализации ошибок произошла. Вызывающая сторона все равно отреагирует на них одинаково (в примере выше завершит программу).

Если необходима более подробная информация для отладки, то можете запротоколировать ошибки. Если оказывается, что после сведения информации к существенной осталось не так уж много различных ошибок, то вместо кодов ошибок иногда выгоднее возвращать специальные значения.

Последствия

Отказ от возврата подробной информации о внутренних ошибках – большое облегчение для вызывающей стороны, которой больше не нужно думать о том, как обработать все возможные ошибки. И тогда более вероятно, что вызывающая сторона отреагирует на все возвращенные ошибки, потому что они суще-

ственны для нее. Да и тестировщики будут довольны, потому что чем меньше информации об ошибках возвращают функции, тем меньше ошибочных ситуаций необходимо тестировать.

Если используются очень строгие компиляторы или инструменты статического анализа, которые проверяют, что вызывающая сторона анализирует все возможные возвращаемые значения, то та не обязана явно обрабатывать несущественные ошибки (например, в предложении `switch` с большим числом проваливаний и одной центральной ветвью для обработки всех внутренних ошибок). Вместо этого она обрабатывает только один код внутренней ошибки, а если вы завершаете программу в случае ошибки, то может вообще ничего не обрабатывать.

Отказ от возврата подробной информации об ошибках лишает вызывающую сторону возможности показать эту информацию пользователю или сохранить ее для отладки. Но для целей отладки в любом случае было бы лучше протоколировать ошибки в том модуле, где они возникли, а не обременять этим вызывающую сторону.

Если вы возвращаете не всю информацию об ошибках, возникших в функции, а лишь ту, которая, по вашему мнению, может быть существенна для вызывающей стороны, то есть шанс, что вы ошибаетесь. Вы можете забыть о какой-то информации, необходимой вызывающей стороне, и это станет причиной для будущего запроса об изменении. Но если вы возвращаете коды состояния, то добавить дополнительные коды ошибок легко без изменения сигнатуры функции.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В коде, относящемся к безопасности, очень часто возвращают только существенную информацию об ошибках. Например, если функция аутентификации пользователя возвращает подробную информацию о том, что причиной отказа в аутентификации стало неверное имя пользователя или пароль, то вызывающая сторона могла бы воспользоваться этой функцией, чтобы узнать, какие имена пользователей уже заняты. Чтобы избежать открытия побочных каналов на основе этой информации, принято возвращать только двоичную информацию: успешно прошла аутентификация или нет. Например, функция `rbacAuthenticateUserPassword`, применяемая для аутентификации пользователей в операционной системе B&R Automation Runtime, возвращает значение типа `bool`, равное `true`, если аутентификация была успешной, и `false` в противном случае. Никакой подробной информации о причинах неудачной аутентификации не возвращается.
- Функция `FlushWinFile` в игре NetHack сбрасывает файл на диск, вызывая функцию Macintosh `FSWrite`, которая возвращает код ошибки. Однако NetHack игнорирует этот код, и `FlushWinFile` имеет тип `void`, потому что использующий эту функцию код не может адекватно отреагировать на ошибку. Поэтому информация об ошибке и не передается.

- Функция OpenSSL `EVP_CIPHER_do_all` инициализирует комплекты шифров с помощью внутренней функции `OPENSSL_init_crypto`, которая возвращает код состояния. Однако это детальная информация игнорируется, потому что `EVP_CIPHER_do_all` имеет тип `void`. Таким образом, функция-обертка изменяет стратегию возврата детальной информации об ошибке, склоняясь к возврату только существенной информации, – в данном случае не возвращается вообще ничего.

Применение к сквозному примеру

Если возвращается только существенная информация об ошибках, то код реестра принимает вид, показанный ниже. Для простоты оставлена только функция `createKey`:

Реализация функции `createKey`

```
RegError createKey(char* key_name, RegKey* key)
{
 if(key == NULL || key_name == NULL)
 {
  return INVALID_PARAMETER; ❶
 }

 if(STRING_SIZE <= strlen(key_name))
 {
  return STRING_TOO_LONG;
 }

 RegKey newKey = calloc(1, sizeof(struct Key));
 if(newKey == NULL)
 {
  return OUT_OF_MEMORY;
 }

 strcpy(newKey->key_name, key_name);
 *key = newKey;
 return OK;
}
```

❶ Вместо `INVALID_KEY` или `INVALID_STRING` мы теперь возвращаем единый код: `INVALID_PARAMETER`.

Теперь вызывающая сторона не может обработать разные ошибки задания параметров по-разному, а значит, ей и не нужно думать о различиях. Код вызывающей стороны стал проще, потому что нужно обрабатывать на одну ситуацию меньше.

Это хорошо. Действительно, что должна сделать вызывающая сторона, если функция вернула `INVALID_KEY` или `INVALID_STRING`? Повторно вызывать ее точно не имеет смысла. В обоих случаях вызывающая сторона должна просто сми-

риться с тем, что функция не отработала, и сообщить об этом пользователю или завершить программу. Поскольку у вызывающей стороны нет причин по-разному обрабатывать эти две ошибки, мы освободили ее от тяжести принятия решения. Теперь она должна анализировать только одну ситуацию и реагировать соответственно.

Чтобы еще упростить код, мы далее применим «Принцип самурая». Вместо того чтобы возвращать все ошибки, мы сведем обработку некоторых к завершению программы.

Объявление функции `createKey`

```
/* Создать новый раздел реестра с именем 'key_name' (должно быть
   отлично от NULL, макс. длина - STRING_SIZE символов). Сохраняет
   описатель раздела в переданном параметре 'key' (должен быть
   отличен от NULL). Возвращает OK в случае успеха и OUT_OF_MEMORY
   в случае нехватки памяти. */
RegError createKey(char* key_name, RegKey* key);
```

Реализация функции `createKey`

```
RegError createKey(char* key_name, RegKey* key)
{
 assert(key != NULL && key_name != NULL); ❶
 assert(STRING_SIZE > strlen(key_name)); ❶

 RegKey newKey = calloc(1, sizeof(struct Key));
 if(newKey == NULL)
 {
  return OUT_OF_MEMORY;
 }

 strcpy(newKey->key_name, key_name);
 *key = newKey;
 return OK;
}
```

❶ Вместо того чтобы возвращать `INVALID_PARAMETER` или `STRING_TOO_LONG`, мы теперь завершаем программу, если какой-либо из переданных параметров некорректен.

Завершение программы в случае слишком длинных строк на первый взгляд кажется чрезмерно суровой мерой. Но, как и нулевые указатели, слишком длинная строка является недопустимым параметром. Если реестр принимает строку не от пользователя посредством графического интерфейса, а от вызывающей программы, то программа завершается только в случае очевидной ошибки, и такое поведение вполне оправдано.

Далее мы видим, что функция `createKey` возвращает всего два разных кода: `OUT_OF_MEMORY` и `OK`. Код станет немного чище, если передавать эту информацию с помощью возврата специальных значений.

Специальное возвращаемое значение

Контекст

Имеется функция, которая вычисляет некоторый результат, и вы хотите передать вызывающей стороне информацию об ошибке в случае возникновения таковой. Но желательно возвращать только существенные ошибки.

Задача

Вы хотите возвращать информацию об ошибке, но не хотите явно возвращать коды состояний, потому что тогда функции труднее возвращать другие данные. Можно было бы добавить выходные параметры, но это усложнило бы вызов функции.

Полный отказ от возврата информации об ошибке – тоже не вариант. Какую-то информацию вызывающей стороне предоставить нужно, и вы хотите, чтобы та могла отреагировать на нее. Но вы хотите передавать вызывающей стороне не слишком много информации. Например, это может быть просто двоичный признак: успешно завершилась функция или неудачно. Возвращать код состояния для такой простой информации – пожалуй, перебор. Применить «Принцип самурая» и завершить программу нельзя, потому что возникающие в функции ошибки не настолько серьезны. Или, быть может, вы хотите, чтобы вызывающая сторона решала, как обрабатывать ошибки, потому что она может сделать это разумно.

Решение

Используйте возвращаемое функцией значение для возврата вычисленных ей данных, но зарезервируйте одно или несколько специальных значений для возврата информации об ошибках.

Например, если функция возвращает указатель, то в качестве специального значения, показывающего, что произошла ошибка, можно использовать `NULL`. Нулевой указатель по определению недопустим, поэтому есть уверенность, что это специальное значение не перепутаешь с правильным указателем, вычисленным функцией. Ниже показано, как возвращать информацию об ошибке при работе с указателями:

Реализация вызываемой функции

```
void* func()
{
 if(somethingGoesWrong())
 {
  return NULL;
 }
 else
 {
```

```
  return some_pointer;
 }
}
```

Код вызывающей стороны

```
pointer = func();
if(pointer != NULL)
{
 /* работа с указателем */
}
else
{
 /* обработка ошибки */
}
```

Вы должны обязательно документировать в API все специальные возвращаемые значения. Иногда имеется общепринятое соглашение о том, какие специальные значения соответствуют ошибкам. Например, очень часто для обозначения ошибок используются отрицательные целые числа. Но даже в таком случае семантику значений необходимо документировать.

Необходимо, чтобы специальное значение, обозначающее ошибку, не могло возникнуть в ситуации, когда ошибки нет. Например, если функция возвращает температуру по Цельсию в виде целого числа, то было бы неправильно следовать принятому в UNIX соглашению о том, что любое отрицательное число означает ошибку. Вместо этого можно было бы выбрать в качестве индикатора ошибки число –300, потому что физически невозможны температуры ниже –273 °C.

Последствия

Теперь функция может сообщать информацию об ошибке с помощью возвращаемого значения, даже если оно используется также для возврата результата функции. При этом нет нужды в дополнительных выходных параметрах.

Иногда специальных значений для кодирования информации об ошибке не так много. Например, если возвращается указатель, то только значение `NULL` можно использовать для обозначения ошибки. В результате мы можем сообщить вызывающей стороне только одно: была ошибка или нет. А значит, никакой детальной информации об ошибке передать не получится. Но можно найти и хорошую сторону в этой ситуации: нет соблазна вернуть ненужную информацию об ошибке. Во многих случаях достаточно сообщить, что ошибка имела место, а на более подробную информацию вызывающая сторона все равно не смогла бы отреагировать.

Если впоследствии вы поймете, что нужна более подробная информация об ошибке, то может оказаться, что предоставить ее невозможно, потому что не осталось неиспользованных специальных значений. Тогда придется изменять всю сигнатуру функции и использовать возврат кодов состояния для предо-

ставления дополнительной информации. Изменение сигнатуры неприемлемо, если API должен оставаться совместимым с существующими клиентами. Если вы предвидите изменения в будущем, то не используйте специальные возвращаемые значения, а сразу выбирайте возврат кодов состояния.

Иногда программисты предполагают, что нет нужды пояснять, какие возвращаемые значения означают ошибку, – мол, и так ясно. Например, для кого-то очевидно, что нулевой указатель – признак ошибки. А еще для кого-то – что –1 обозначает ошибку. Это создает опасную ситуацию, когда программист предполагает, что всем ясно, какие значения соответствуют ошибкам. Однако это не более чем предположение. В любом случае в API должно быть четко документировано, какие значения обозначают ошибки, – и не следует пренебрегать этим под предлогом, что и так все понятно.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Функция `getobj` в игре NetHack возвращает указатель на некоторый объект или `NULL` в случае ошибки. Чтобы сообщить о специальном случае – отсутствии объекта, который можно было бы вернуть, – функция возвращает указатель на глобальный объект `zeroobj;` он принадлежит тому же типу, что указан в сигнатуре функции, и известен вызывающей стороне. Вызывающая сторона может проверить, совпадает ли возвращенный указатель с указателем на этот глобальный объект, и таким образом отличить допустимый объект от объекта `zeroobj`, имеющего специальную семантику.
- Функция `getchar` из стандартной библиотеки C читает символ из `stdin`. Она возвращает значение типа `int`, позволяющее вернуть гораздо больше информации, чем просто символ. Если символов больше нет, функция возвращает константу `EOF`, обычно определенную как –1. Поскольку символы не могут принимать отрицательные значения, `EOF` легко отличима от нормальных результатов и потому может обозначать специальную ситуацию – отсутствие доступных символов.
- В большинстве функций UNIX или POSIX отрицательные числа используются для обозначения ошибки. Например, POSIX-функция `write` возвращает число записанных байтов или –1 в случае ошибки.

Применение к сквозному примеру

При использовании паттерна «Специальное возвращаемое значение» код выглядит, как показано ниже. Для простоты представлена только функция `createKey`:

Объявление функции `createKey`

```
/* Создать новый раздел реестра с именем 'key_name' (должно быть
   отлично от NULL, макс. длина - STRING_SIZE символов). Возвращает
   описатель раздела или NULL в случае ошибки. */
RegKey createKey(char* key_name);
```

Реализация функции `createKey`

```
RegKey createKey(char* key_name)
{
 assert(key_name != NULL);
 assert(STRING_SIZE > strlen(key_name));

 RegKey newKey = calloc(1, sizeof(struct Key));
 if(newKey == NULL)
 {
  return NULL;
 }

 strcpy(newKey->key_name, key_name);
 return newKey;
}
```

Функция `createKey` стала гораздо проще. Теперь она возвращает не коды состояний, а сразу описатель, и выходной параметр больше не нужен. Документация API функции тоже сократилась, поскольку не нужно описывать дополнительный параметр и включать длинное описание того, как результат функции будет возвращен вызывающей стороне.

Для вызывающей стороны все стало намного проще. Ей не нужно передавать описатель в выходном параметре, она получит его непосредственно в виде возвращаемого значения – поэтому код стал удобнее для чтения и сопровождения.

Однако теперь у нас появилась проблема, которой не было при возврате детальной информации об ошибке в виде кода состояния – мы знаем только, успешно или нет завершилась функция. Все внутренние сведения об ошибке отброшены, и если впоследствии они понадобятся, например для отладки, то получить их нет никакой возможности. Для решения этой проблемы можно использовать паттерн «Протоколирование ошибок».

Протоколирование ошибок

Контекст

Имеется функция, в которой обрабатываются ошибки. Вы хотите возвращать вызывающей стороне только существенную информацию об ошибках, чтобы она могла на них отреагировать, но при этом сохранить детальную информацию для отладки.

Проблема

Вы хотите быть уверены, что в случае ошибки легко сможете найти ее причину. Однако не хочется, чтобы из-за этого код обработки ошибок усложнился.

Один способ – возвращать очень подробную информацию об ошибке непосредственно вызывающей стороне. Для этого можно возвращать коды состояния, надеясь, что вызывающая сторона покажет код пользователю. Тот

может обратиться к вам (например, по горячей линии) и спросить, что значит код и как исправить ошибку. А вы, имея детальную информацию, используемую для отладки программы, можете определить, что случилось.

Однако у такого подхода есть крупный недостаток – вызывающая сторона, которой вообще нет дела до ваших ошибок, должна предъявлять информацию о них пользователю только для того, чтобы тот мог сообщить ее вам. А пользователю тоже такая подробная информация неинтересна.

Кроме того, возврат кодов состояний плох тем, что возвращаемое значение используется для возврата информации об ошибке, а собственно результаты функции приходится возвращать в выходных параметрах. Иногда, правда, можно воспользоваться специальными возвращаемыми значениями, но не всегда. А вы не хотите заводить дополнительные параметры только для того, чтобы предоставить информацию об ошибке, так как это усложняет код вызывающей стороны.

Решение

Используйте разные каналы для передачи информации об ошибке, существенной для вызывающего кода, и информации, существенной для разработчика. Например, записывайте отладочную информацию в файл журнала и не возвращайте детальные сведения об ошибке вызывающей стороне.

В случае ошибки пользователь программы должен будет предоставить вам отладочную информацию из журнала, чтобы вы могли легко определить причину ошибки. Например, ее можно было бы передать по электронной почте.

Альтернативно можно было бы записывать сведения об ошибке в журнал на стыке между вами и вызывающей стороной и возвращать вызывающей стороне только существенную информацию об ошибке. Например, вызывающую сторону можно было бы проинформировать о том, что произошла внутренняя ошибка, но не сообщать детали. Таким образом, вызывающая сторона могла бы обработать общую ошибку, ничего не зная о том, что делать с ее конкретными разновидностями, но вы при этом не потеряете ценную отладочную информацию.

Чтобы не потерять отладочную информацию, вы должны протоколировать сведения о программных и неожиданных ошибках. Очень полезно сохранять информацию о степени серьезности и месте ошибки, например имя исходного файла и номер строки в нем или трассу вызова. В языке C есть специальные макросы для получения номера текущей строки (`__LINE__`), имени текущей функции (`__func__`) и имени текущего файла (`__FILE__`). В следующем коде для протоколирования используется макрос `__func__`.

```
void someFunction()
{
 if(something_goes_wrong)
 {
  logInFile("что-то пошло не так", ERROR_CODE, __func__);
 }
}
```

Многострочные макросы

Погрузив предложения макроса в цикл `do/while`, вы сможете избежать проблем, демонстрируемых в следующем коде:

```
#define MACRO(x) \
x=1;             \
x=2;             \

if(x==0)
 MACRO(x)
```

Здесь нет фигурных скобок вокруг тела `if`, поэтому, читая код, можно подумать, что код макроса выполняется только при `x==0`. Но на самом деле после расширения макроса получается такой код:

```
if(x==0)
 x=1;
x=2;
```

Последняя строка не принадлежит телу `if`, а это совсем не то, что мы хотели. Чтобы избежать таких проблем, лучше всего помещать предложения макроса внутрь цикла `do/while`..

Для получения более детальной информации можно сохранить трассу вызовов функций вместе с возвращенными ими значениями. Это поможет воспроизвести ситуацию, в которой произошла ошибка, но, разумеется, влечет за собой дополнительные накладные расходы. Для трассировки значений, возвращенных функциями, можно использовать такой код:

```
#define RETURN(x)         \
do {                      \
 logInFile(__func__, x); \
 return x;               \
while (0)

int soneFunction()
{
 RETURN(-1);
}
```

Журнал ошибок можно хранить в файле, как показано выше. Но надо предусмотреть особые случаи, например отсутствие места для файла на диске или крах программы во время записи в файл. Обработать такие ситуации нелегко, но очень важно, если вы хотите иметь надежный механизм протоколирования, на который можно будет положиться для последующей отладки. Если данные в журналах некорректны, то в своей охоте за ошибками вы можете пойти по ложному следу.

Последствия

Отладочную информацию можно получить, не требуя от вызывающей стороны ее обработки или сквозной передачи и тем самым упростив ее код. Вместо этого вы предоставляете детальную информацию об ошибке самостоятельно.

В некоторых случаях желательно сохранить в журнале какую-то информацию об ошибке или ситуации, но вызывающей стороне это совершенно безразлично. Поэтому и возвращать ей какую-либо информацию об ошибке необязательно. Например, если в случае ошибки вы завершаете программу, то вызывающая сторона вообще никак не должна реагировать на ошибку, но все равно вы не потеряете ценную отладочную информацию, если будете протоколировать ошибки. Поэтому не нужны дополнительные параметры для возврата информации об ошибке, что упрощает вызов функции и помогает держать код вызывающей стороны в чистоте.

Вы не теряете ценную информацию об ошибке и можете использовать ее для отладки. Для ее сохранения вы предоставляете другой канал, например файлы журналов. Однако нужно подумать о том, как эти журналы получить. Можно было бы попросить пользователей отправить их электронной почтой или же реализовать какой-то более сложный автоматический механизм уведомления об ошибках. Но никакой подход не дает стопроцентной гарантии, что информация из журнала дойдет до вас. Если пользователь не хочет, он может воспрепятствовать этому.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В веб-сервере Apache функция `ap_log_error` записывает информацию об ошибках, связанных с запросами и подключениями, в журнал. В каждой записи журнала хранится имя файла и строка кода, в которой произошла ошибка, а также строка, переданная функцией вызывающей стороне. Журнал хранится в файле `error_log` на сервере.
- В операционной системе B&R Automation Runtime используется система протоколирования, позволяющая программистам сообщать информацию пользователям путем вызова функции `eventLogWrite` из любого места программы. Это позволяет предоставлять информацию, не передавая ее по всей цепочке вызовов в какой-то центральный компонент протоколирования.
- Паттерн «Контекст утверждения» из книги Adam Tornhill «Patterns in C» (Leanpub, 2014) рекомендует завершать программу в случае ошибок и протоколировать причину и место ошибки, добавляя строку в вызов `assert`. Если условие в `assert` не выполнено, то будет напечатана строка кода, содержащая `assert`, которая включает и добавленную строку.

Применение к сквозному примеру

После применения всех паттернов получается следующий окончательный код модуля реестра. Он предоставляет вызывающей стороне существенную информацию об ошибке, но не требует от нее обработки внутренних ошибок:

API реестра

```
/* макс. размер строковых параметров (включая завершающий NULL) */
#define STRING_SIZE 100

/* Коды ошибок, возвращаемые реестром */
typedef enum
{
  OK,
  CANNOT_ADD_KEY
} RegError;

/* Описатель раздела реестра */
typedef struct Key* RegKey;

/* Создать новый раздел реестра с именем 'key_name' (должно быть
   отлично от NULL, макс. размер STRING_SIZE символов). Возвращает
   описатель раздела или NULL в случае ошибки. */
RegKey createKey(char* key_name);

/* Сохранить переданное значение 'value' (должно быть отлично
   от NULL, макс. размер STRING_SIZE символов) в разделе с
   именем 'key' (должно быть отлично от NULL) */
void storeValue(RegKey key, char* value);

/* Сделать раздел с именем 'key' (должно быть отлично от NULL)
   доступным для чтения. Возвращает OK, если все хорошо, или
   CANNOT_ADD_KEY, если реестр заполнен и новые разделы
   опубликовать невозможно. */
RegError publishKey(RegKey key);
```

Реализация реестра

```
#define MAX_KEYS 40

struct Key
{
  char key_name[STRING_SIZE];
  char key_value[STRING_SIZE];
};

/* макрос для протоколирования отладочной информации и утверждения */
#define logAssert(X)                    \
if(!(X))                                \
{                                       \
 printf("Error at line %i", __LINE__); \
 assert(false); \
```

```
}

/* глобальный на уровне файла массив, содержащий все разделы реестра */
static struct Key* key_list[MAX_KEYS];

RegKey createKey(char* key_name)
{
 logAssert(key_name != NULL)
 logAssert(STRING_SIZE > strlen(key_name))

 RegKey newKey = calloc(1, sizeof(struct Key));
 if(newKey == NULL)
 {
  return NULL;
 }

 strcpy(newKey->key_name, key_name);
 return newKey;
}

void storeValue(RegKey key, char* value)
{
 logAssert(key != NULL && value != NULL)
 logAssert(STRING_SIZE > strlen(value))
 strcpy(key->key_value, value);
}

RegError publishKey(RegKey key)
{
 logAssert(key != NULL)
 int i;
 for(i=0; i<MAX_KEYS; i++)
 {
  if(key_list[i] == NULL)
  {
   key_list[i] = key;
   return OK;
  }
 }
 return CANNOT_ADD_KEY;
}
```

Этот код короче предыдущего по следующим причинам:

- в случае программных ошибок программа сразу завершается. Недопустимые параметры типа нулевых указателей не имеет смысла обрабатывать, ведь в документации API ясно сказано, что описатель должен быть отличен от NULL;

- возвращаются только ошибки, существенные для вызывающей стороны. Например, функция `createKey` возвращает не код состояния, а просто описатель, который будет равен `NULL` в случае ошибки, потому что вызывающей стороне более подробная информация об ошибке не нужна.

Хотя код стал короче, размер комментариев вырос. Теперь они более точно описывают, как функции ведут себя в случае ошибок. Проще стал не только ваш код, но и код вызывающей стороны, потому что ей больше не нужно принимать решения о том, как реагировать на разнообразные ошибки.

Код вызывающей стороны

```
RegKey my_key = createKey("myKey");
if(my_key == NULL)
{
 printf("Ошибка при создании раздела\n");
}

storeValue(my_key, "A");

RegError err = publishKey(my_key);
if(err == CANNOT_ADD_KEY)
{
 printf("Раздел невозможно опубликовать, потому что реестр заполнен\n");
}
```

Этот код короче предыдущего, потому что:

- не нужно проверять коды тех ошибок, которые приводят к завершению программы;
- функции, не возвращающие подробной информации об ошибках, теперь возвращают сам запрошенный элемент. Например, `createKey()` возвращает описатель, и вызывающая сторона не должна подготавливать выходной параметр;
- коды программных ошибок, например «передан недопустимый параметр», больше не возвращаются, а потому и не проверяются вызывающей стороной.

Финальный код сквозного примера показал, как важно обдумывать, какие виды ошибок следует обрабатывать в коде и как это делать. Просто возвращать все ошибки и требовать от вызывающей стороны их обработки – не лучшее решение. Возможно, вызывающей стороне неинтересна подробная информация об ошибке или она не хочет реагировать на ошибку. Быть может, ошибка настолько серьезна, что в точке ее возникновения следует принять решение о немедленном завершении программы. Такие меры упрощают код, поэтому их следует рассматривать при проектировании API компонента.

Резюме

В этой главе было показано, как обрабатывать ошибки, передавая информацию о них между функциями и различными частями программы. Паттерн «Возврат кода состояния» рекомендует возвращать вызывающей стороне числовые коды, представляющие ошибки. Паттерн «Возврат существенной информации об ошибке» рекомендует возвращать только информацию о тех ошибках, на которые вызывающая сторона может отреагировать, а паттерн «Специальное возвращаемое значение» предлагает один из способов сделать это. Паттерн «Протоколирование ошибок» рекомендует создавать дополнительный канал для передачи информации об ошибках, предназначенной не для вызывающей стороны, а для пользователя или для отладки.

Эти паттерны обогащают ваш арсенал инструментами для анализа ошибочных ситуаций и направляют в нужную сторону при реализации большого куска кода.

Для дополнительного чтения

Если у вас проснулся аппетит, то вот еще несколько ресурсов, которые расширят ваши знания о возврате информации об ошибках.

- Исчерпывающий обзор обработки ошибок вообще имеется в магистерской диссертации Томаса Аглассингера «Error Handling in Structured and Object-Oriented Programming Languages» (Университет Оулу, 1999). Приводятся рекомендации с примерами кода на разных языках программирования, включая C.
- В портлендском репозитории паттернов (https://oreil.ly/bs9FX) предлагается много паттернов с обсуждениями на тему обработки ошибок и не только. В большинстве паттернов обработки ошибок речь идет об обработке исключений и использовании утверждений, но представлены также идиомы для C.
- В статьях Andy Longshaw and Eoin Woods (https://oreil.ly/7Yj8h) «Patterns for Generation, Handling and Management of Errors» и «More Patterns for the Generation, Handling and Management of Errors» представлены паттерны протоколирования и обработки ошибок с упором на обработку ошибок в форме исключений.

Что дальше

В следующей главе речь пойдет о работе с динамической памятью. Для возврата более сложных данных из функции, а также для организации больших порций данных и управления временем их жизни динамическая память необходима, а вам нужны рекомендации по решению возникающих задач.

Глава 3

Управление памятью

Любая программа хранит какие-то значения в памяти для последующего использования. Эта функциональность настолько типична, что в современных языках программирования сделано все для того, чтобы пользоваться ей было как можно проще. Язык C++, равно как и другие объектно ориентированные языки, предлагает конструкторы и деструкторы, т. е. четко определенное место для выделения и освобождения памяти. В языке Java есть даже сборщик мусора, который следит за тем, чтобы уже не используемая память становилась доступной другим.

Язык C в этом отношение занимает особое место – программист должен управлять памятью вручную. Программист должен сам решить, хранить ли переменные в стеке, в куче или в статической памяти. Он же должен позаботиться об освобождении памяти, выделенной из кучи, и не существует никаких механизмов типа деструктора или встроенного сборщика мусора, который помог бы в решении этой задачи.

Рекомендации на эту тему разбросаны по всему интернету, из-за чего довольно трудно отвечать на вопросы типа «Должна ли эта переменная находиться в стеке или в куче?». Чтобы ответить на этот и другие вопросы, мы в этой главе представим паттерны, касающиеся работы с памятью в программах на C. Паттерны дают рекомендации, когда использовать стек, а когда кучу, а также когда и как освобождать память, взятую из кучи. Чтобы было проще освоить паттерны, они применяются к сквозному примеру.

На рис. 3.1 приведен обзор паттернов, рассматриваемых в этой главе, и связей между ними, а в табл. 3.1 – краткое описание этих паттернов.

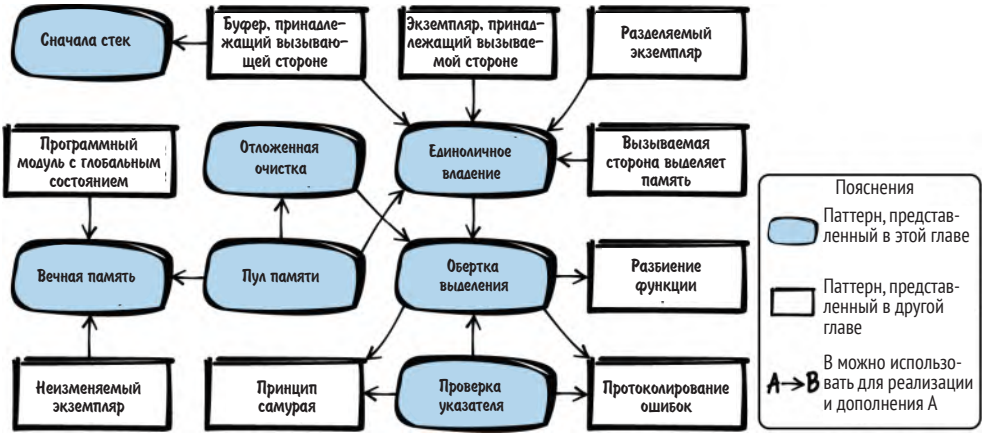


Рис. 3.1. Обзор паттернов для управления памятью

Таблица 3.1. Паттерны для управления памятью

Название паттерна	Краткое описание
Сначала стек	Любому программисту часто приходится принимать решение о классе хранения и области памяти (стек, куча...) для размещения переменных. Если для каждой переменной по новой взвешивать все плюсы и минусы различных вариантов, то ни на что другое не останется времени. Поэтому по умолчанию размещайте переменные в стеке, это даст возможность воспользоваться механизмом автоматической очистки памяти
Вечная память	Хранить большие объемы данных и передавать их между вызовами функций трудно, потому что нужно гарантировать, что памяти для данных достаточно и что она не стирается между вызовами. Поэтому размещайте данные в памяти, которая остается доступной в течение всего времени работы программы
Отложенная очистка	Необходимость в динамической памяти возникает, если нужна память большого и заранее неизвестного размера. Но проблема очистки динамической памяти трудна и является источником многих ошибок. Поэтому только выделяйте динамическую память, а ее освобождение оставьте операционной системе в момент завершения программы
Единоличное владение	За удобство динамической памяти приходится расплачиваться необходимостью ее освобождения. В больших программах трудно гарантировать, что вся динамически выделенная память надлежащим образом освобождается. Поэтому уже в момент выделения памяти ясно и недвусмысленно определите и документируйте, где она должна быть освобождена и кто за это отвечает

Название паттерна	Краткое описание
Обертка выделения	Любое выделение динамической памяти может завершиться неудачно, поэтому следует проверять результат выделения в своем коде и реагировать соответственно. Это громоздко, потому что такие проверки приходится делать во многих местах программы. Поэтому оберните вызовы функций выделения и освобождения памяти и реализуйте логику обработки ошибок или дополнительного управления памятью в этих обертках
Проверка указателя	Программные ошибки, связанные с доступом по недействительному указателю, ведут к неопределенному поведению программы, и отлаживать их трудно. Но поскольку код часто работает с указателями, велики шансы появления таких ошибок. Поэтому явно делайте недействительными неинициализированные или освобожденные указатели и всегда проверяйте действительность перед доступом по ним
Пул памяти	Частое выделение и освобождение памяти из кучи приводит фрагментации памяти. Поэтому выделите большой участок памяти на все время работы программы. Во время выполнения получайте блоки фиксированного размера из этого пула памяти, а не непосредственно из кучи

Хранение данных и проблемы с динамической памятью

В языке C данные могут размещаться в нескольких местах.

- В стеке. Это область памяти фиксированного размера, зарезервированная в каждом потоке (выделяется при создании потока). При вызове функции в потоке участок на вершине стека резервируется для параметров и автоматических переменных функции. По завершении вызова эта память автоматически освобождается. Чтобы поместить данные в стек, достаточно объявить переменные в той функции, где они используются. К этим переменным можно обращаться, пока они не выйдут из области видимости (при выходе из блока).

```
void main()
{
 int my_data;
}
```

  В статической памяти. Это область памяти фиксированного размера, которая выделяется и инициализируется на этапе компиляции. Чтобы разместить переменную в статической памяти, поместите перед ее объявлением ключевое слово `static`. Такие переменные доступны на протя-

жении всей работы программы. Это относится и к глобальным переменным, объявленным даже без ключевого слова `static`:

```
int my_global_data;
static int my_fileglobal_data;
void main()
{
 static int my_local_data;
}
```

- Если размер данных фиксирован и они неизменяемы, то их можно поместить прямо в той статической памяти, где хранится код. Очень часто фиксированные строки так и хранятся. Такие данные доступны на протяжении всей работы программы (даже если, как в примере ниже, указатель на них покидает область видимости):

```
void main()
{
 char* my_string = "Здравствуй, мир";
}
```

- Данные можно разместить в динамической памяти, выделенной из кучи. Куча – это глобальный пул памяти, доступный всем процессам в системе. Программист может выделять из нее память и освобождать ее в любой момент времени:

```
void main()
{
 void* my_data = malloc(1000);
 /* поработать с 1000 выделенных байт памяти */
 free(my_data);
}
```

Выделение динамической памяти – первое место, где все может пойти наперекосяк, и вся эта глава посвящена решению возникающих в этой связи проблем. Использование динамической памяти в программах на C сопряжено с многочисленными проблемами, которые необходимо решить или, по крайней мере, учитывать. Ниже кратко перечислены основные проблемы с динамической памятью.

- Выделенную память нужно в какой-то момент освободить. Если этого не сделать, то вы будете потреблять больше памяти, чем необходимо, и возникнет так называемая утечка памяти. Если это происходит часто и приложение работает долго, то в конце концов памяти не останется вовсе.

  Неоднократное освобождение одной и той же памяти – проблема, которая ведет к неопределенному поведению программы, что совсем плохо. В худшем случае в той строке кода, где допущена ошибка, ничего страшного не происходит, зато в какой-то заранее неизвестный момент времени позже программа «падает». Отлаживать такие ошибки – мука мученическая.

- Попытка доступа к уже освобожденной памяти – тоже проблема. Очень легко освободить память, а затем по ошибке разыменовать указатель на нее (так называемый висячий указатель). Этот опять-таки ведет к трудным для отладки ошибкам. В лучшем случае программа просто упадет, а в худшем она продолжит работать, но эта память будет принадлежать кому-то другому. Ошибки, связанные с использованием такой памяти, представляют угрозу безопасности и могут проявляться совершенно неожиданно в процессе последующего выполнения программы.

  Необходимо управлять временем жизни выделенных данных и их владением. Вы должны знать, кто и когда освобождает данные, а в C это может оказаться весьма не тривиальным делом. В C++ можно было бы просто выделить память для данных объекта в конструкторе и освободить ее в деструкторе. В сочетании с *умными указателями* это даже позволяет автоматически освобождать память объекта, когда он покидает область видимости. Но в C это невозможно, так как деструкторов в нем нет. Нас никто не уведомит о том, что указатель вышел из области видимости и память следует освободить.
- Для работы с кучей требуется больше времени, чем со стеком или со статической памятью. Выделение памяти из кучи необходимо защищать от условий гонки, потому что другие процессы, пользуются тем же самым пулом. Поэтому выделение происходит медленнее. Доступ к куче также медленнее, потому что, например, к памяти в стеке обращаются чаще, и потому она с большей вероятностью находится в кеше или в регистрах процессора.

  Серьезнейшую проблему представляет фрагментация динамической памяти, схематически изображенная на рис. 3.2. Если выделить блоки памяти A, B и C, а затем освободить блок B, то эта область памяти перестает быть непрерывной. Попытка выделить большой блок памяти D завершится неудачно, хотя в целом столько памяти есть. Но она не образует один непрерывный участок, поэтому вызов `malloc` завершается ошибкой. Фрагментация – огромная проблема в системах с ограниченной памятью, которые работают длительное время (как встраиваемые системы реального времени).

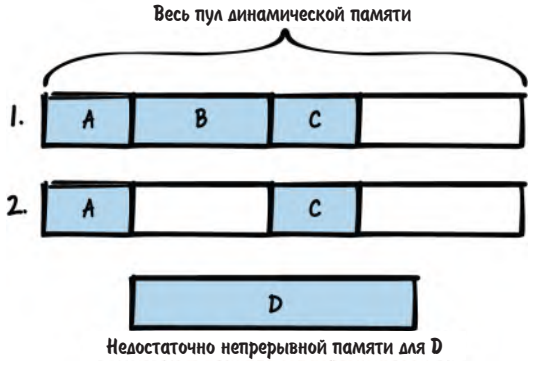


Рис. 3.2. Фрагментация памяти

Решать эти проблемы нелегко. Описанные в последующих разделах паттерны дают рекомендации, как избежать выделения динамической памяти или, если это невозможно, организовать работу с ней правильно.

Сквозной пример

Вы хотите реализовать простую программу, которая применяет к тексту шифр Цезаря. Этот шифр заменяет каждую букву другой, отстоящей от нее на фиксированное число позиций в алфавите. Например, если число позиций равно 3, то буква A будет заменена буквой D. Начнем с показанной ниже реализации шифра Цезаря.

```
/* Применяет шифр Цезаря с фиксированным ключом 3. Параметр 'text'
   должен содержать текст, написанный заглавными буквами. Параметр
   'length' должен содержать длину текста, не включая завершающий NULL. */
void caesar(char* text, int length)
{
 for(int i=0; i<length; i++)
 {
  text[i] = text[i]+3; ❶
  if(text[i] > 'Z')
  {
   text[i] = text[i] - 'Z' + 'A' - 1; ❷
  }
 }
}
```

❶ В C символы хранятся как числовые значения, и для сдвига символа вправо по алфавиту нужно прибавить к символу число.

❷ Если результат оказывается дальше буквы Z, то мы возвращаемся в начало алфавита.

Далее вы хотите проверить, работает ли функция, а для этого нужно подать ей на вход какой-то текст. Функция принимает указатель на строку. Но где хранить эту строку? Нужно ли выделить ее динамически, или лучше взять память из стека? Вы приходите к выводу, что проще всего воспользоваться паттерном «Сначала стек».

Сначала стек

Контекст

Вы хотите сохранить данные и обращаться к ним в будущем. Максимальный размер известен заранее, и он не особенно велик (всего несколько байтов).

Проблема

Решение о классе хранения и области памяти (стек, куча и т. д.) для размещения переменных каждому программисту приходится принимать часто. Ни на что другое не

хватит времени, если каждый раз взвешивать плюсы и минусы всех вариантов со всеми деталями.

Для хранения данных в программе на C есть масса возможностей, из которых самые распространенные – стек, статическая память и динамическая память. У каждой из них есть свои преимущества и недостатки, а решение о том, где хранить переменную, очень важно. Оно влияет на время жизни переменной, и от него зависит, освобождается память автоматически или это надо делать вручную.

От этого решения зависит также объем необходимых усилий и дисциплинированность программиста. Вы хотите максимально облегчить себе жизнь, поэтому если нет каких-то особых требований к хранению данных, то желательно использовать тот вид памяти, который требует наименьших усилий для выделения, освобождения и исправления потенциальных программных ошибок.

Решение

Просто помещайте переменные в стек по умолчанию, чтобы воспользоваться всеми выгодами автоматического освобождения занятой ими памяти.

Все переменные, объявленные внутри блока кода, по умолчанию являются *автоматическими*, т. е. размещаются в стеке и автоматически уничтожаются в конце блока (когда переменная покидает область видимости). Можно явно указать, что переменная является автоматической, поместив перед ней спецификатор класса хранения `auto`, но так почти никогда не делают, потому что это уже класс по умолчанию.

Память, выделенную в стеке, можно передавать другим функциям (например, буфер, принадлежащий вызывающей стороне), но никогда не возвращайте адрес такой переменной. В конце функции переменная выходит из области видимости и автоматически уничтожается. Возврат адреса такой переменной приводит к появлению висячего указателя, доступ по которому считается неопределенным поведением и может стать причиной краха программы.

Ниже показан очень простой пример использования переменных в стеке:

```
void someCode()
{
 /* Это автоматическая переменная, которая помещается в стек и выходит
    из области видимости в конце функции */
 int my_variable;
 {
  /* Это автоматическая переменная, которая помещается в стек и выходит
     из области видимости сразу после этого блока, т. е. после закрывающей
     скобки '}' */
  int my_array[10];
 }
}
```

Массивы переменной длины

В примере выше массив имеет фиксированную длину. Обычно в стеке размещают только данные фиксированного размера, известного на этапе компиляции, но можно также определить размер стековой переменной на этапе выполнения. Для этого предназначены функции типа `alloca()` (она не является частью стандарта C и может привести к переполнению стека, если выделить слишком много памяти) или массивы переменной длины (регулярные массивы, размер которых задается переменной), появившиеся в стандарте C99.

Последствия

Хранение данных в стеке упрощает доступ к ним. В отличие от динамически выделенной памяти не возникает нужды в работе с указателями. Это устраняет саму возможность программных ошибок, связанных с висячими указателями. Кроме того, куча не фрагментируется, а освободить память проще. Переменные потому и называются автоматическими, что автоматически уничтожаются. Не надо вручную освобождать память, и это устраняет риск утечек памяти или случайного освобождения памяти несколько раз. В общем и целом большинства трудных для отладки ошибок, связанных с некорректным использованием памяти, можно избежать, просто размещая переменные в стеке.

Выделение памяти в стеке и доступ к ней производятся очень быстро по сравнению с динамической памятью. При выделении не нужно работать со сложными структурами данных, применяемыми для управления доступной памятью. Также не нужно беспокоиться об исключении доступа со стороны других потоков, потому что у каждого потока свой стек. К тому же доступ к данным в стеке обычно очень быстрый, потому что эта память используется часто и, следовательно, с большой вероятностью находится в кеше процессора.

Однако у стека есть недостаток – он ограничен. Сравнительно с кучей стек очень мал (в зависимости от параметров, заданных при сборке программы, размер может составлять всего несколько килобайтов). Если поместить в стек слишком много данных, он переполнится, что приведет к краху программы. Проблема в том, что вы не знаете, сколько еще места осталось в стеке. Все зависит от того, сколько памяти уже занято вызванными ранее функциями. Поэтому вы не должны помещать в стек слишком много данных, и размер этих данных должен быть известен заранее.

Программные ошибки, связанные с буферами в стеке, могут быть источником серьезных проблем с безопасностью. Если буфер в стеке переполняется, то противник может легко эксплуатировать эту ошибку, чтобы перезаписать данные в стеке. Если ему удастся перезаписать адрес возврата из функции, то он сможет выполнить произвольный код.

Кроме того, размещение данных в стеке годится не всегда. Если требуется вернуть вызывающей стороне данные большого объема, например содержимое файла или сетевого буфера, то нельзя просто вернуть адрес массива в стеке, потому что он будет уничтожен после возврата из функции. Для возврата больших данных нужны другие подходы.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Почти в любой программе на C что-то хранится в стеке. В большинстве программ хранение в стеке подразумевается по умолчанию, потому что это самое простое решение.
- Спецификатор класса хранения `auto`, означающий, что переменная автоматическая и размещается в стеке, подразумевается по умолчанию (и по этой причине обычно опускается).
- В книге James Noble and Charles Weir «Small Memory Software: Patterns for Systems with Limited Memory» (Addison-Wesley, 2000) в описании паттерна Распределение памяти сказано, что при выборе места размещения переменной в памяти следует отдавать предпочтение простейшему решению – в случае языка C это стек.

Применение к сквозному примеру

Что ж, пока все просто. Мы выделяем память для размещения текста в стеке и передаем ее функции шифрования:

```
#define MAX_TEXT_SIZE 64

void encryptCaesarText()
{
 char text[MAX_TEXT_SIZE];
 strlcpy(text, "PLAINTEXT", MAX_TEXT_SIZE);
 caesar(text, strnlen(text, MAX_TEXT_SIZE));
 printf("Зашифрованный текст: %s\n", text);
}
```

Это было очень простое решение. Нам не пришлось иметь дело с выделением динамической памяти. Освобождать память тоже не нужно, потому что как только переменная `text` покинет область видимости, она освободится автоматически.

Далее мы хотим зашифровать большой текст. Текущее решение не позволяет это сделать, потому что память размещается в стеке, а он обычно недостаточно велик. В зависимости от платформы размер стека может составлять всего несколько килобайтов. А нам ведь нужно шифровать и более длинные тексты. Чтобы не связываться с динамической памятью, вы решаете попробовать паттерн «Вечная память».

Вечная память

Контекст

Имеются данные большого, но фиксированного размера, которые будут нужны программе в течение длительного времени.

Проблема

Хранить большой объем данных и передавать его между функциями трудно, потому что нужно выделить для данных память достаточного размера, время жизни которой охватывает все вызовы функций.

Использовать стек было бы удобно, потому что он берет на себя заботы об освобождении памяти. Но поместить данные в стек не получится, потому что он не позволяет передавать большие объемы данных между функциями. Это было бы неэффективно, потому что передача данных функции означает их копирование. Альтернатива – ручное выделение памяти в каждом месте программы, где она необходима, и освобождение сразу после того, как необходимость в этой памяти отпала, – сработала бы, но уж больно это хлопотно и чревато ошибками. В частности, следить за временем жизни всех данных и знать, когда и где они освобождаются, – непростая задача.

Если вы работаете с критически важным приложением, когда необходима уверенность, что память обязательно будет доступна, то ни стек, ни динамическая память не годятся, потому что память может закончиться в самый неподходящий момент. Но и в других приложениях могут быть части, где нельзя допустить нехватки памяти. Например, в коде протоколирования ошибок память должна быть доступна, иначе нельзя будет положиться на находящуюся в журнале информацию и определить место ошибки будет трудно.

Решение

Поместите данные в память, доступную на протяжении всего времени работы программы.

Самый распространенный способ сделать это – воспользоваться статической памятью. Либо задайте для переменной спецификатор класса хранения `static`, либо – если у переменной должна быть более широкая область видимости – объявите ее вне любой функции (но только если широкая область видимости в самом деле необходима). Статическая память выделяется в начале работы программы и доступна вплоть до ее завершения. Ниже приведен пример:

```
#define ARRAY_SIZE 1024

int global_array[ARRAY_SIZE]; /* переменная в статической памяти с глобальной
                                 областью видимости */
static int file_global_array[ARRAY_SIZE]; /* переменная в статической памяти с
                                 областью видимости, ограниченной этим файлом */
void someCode()
{
 static int local_array[ARRAY_SIZE]; /* переменная в статической памяти с
                                 областью видимости, ограниченной этой функцией */
}
```

В качестве альтернативы использованию статических переменных вы можете в самом начале программы вызвать функцию инициализации, которая

выделит память, а в конце программы – функцию очистки, которая эту память освободит. Тогда вы тоже получите область памяти, доступную на всем протяжении работы программы, но выделение и освобождение будет лежать на вас.

Неважно, выделяется память в начале программы вручную или используется статическая память, при доступе к ней нужна осторожность. Поскольку память выделена не в стеке, у каждого потока нет ее копии, которая принадлежала бы только ему. В многопоточной программе при доступе к такой памяти необходима синхронизация.

Ваши данные имеют фиксированный размер. В отличие от памяти, динамически выделяемой во время выполнения, размер «Вечной памяти» нельзя изменить.

Последствия

Вам нет нужды беспокоиться о времени жизни и о том, в каком месте освобождать память вручную. Правила просты: память существует на протяжении всего времени работы программы. А использование статической памяти вообще снимает с вас бремя выделения и освобождения памяти.

Теперь вы можете хранить в памяти большие объемы данных и передавать ее другим функциям. В отличие от паттерна «Сначала стек» вы даже можете передать данные вызывающей стороне.

Однако размер необходимой памяти необходимо знать еще на этапе компиляции или, самое позднее, в начале работы. Если размер памяти заранее неизвестен или может изменяться во время выполнения, то паттерн «Вечная память» – не лучший выбор, и следует вместо этого использовать память из кучи.

При использовании «Вечной памяти» программа может запускаться дольше, потому что в этот момент память необходимо выделить. Но это окупается, поскольку впоследствии выделять память уже не придется.

Для выделения и доступа к статической памяти не нужны сложные структуры данных, поддерживаемые операционной системой или средой выполнения для управления кучей. Поэтому память используется более эффективно. Еще оно важное преимущество «Вечной памяти» – отсутствие фрагментации кучи, потому что нет нужды постоянно выделять и освобождать память. Но у этого достоинства есть обратная сторона – память остается занятой все время, даже если она больше не нужна. Более гибкий подход, позволяющий избежать фрагментации памяти, – использовать «Пул памяти».

Одна из проблем «Вечной памяти» – тот факт, что все потоки пользуются одним и тем же ее экземпляром (если используются статические переменные). Поэтому нужно следить за тем, чтобы доступ к памяти из разных потоков не производился одновременно. Впрочем, в частном случае «Неизменяемого экземпляра» эта проблема не стоит.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В игре NetHack статические переменные используются для хранения данных, необходимых на всем протяжении игры. Например, информа-

ция об артефактах, найденных в игре, хранится в статическом массиве `artifact_names`.

- В сетевом анализаторе Wireshark в функции `cf_open_error_message` используется статический буфер для хранения сообщений об ошибках. Вообще, во многих программах для реализации протоколирования ошибок используется статическая память или память, выделенная в начале программы. Так делается, чтобы быть уверенным, что в случае ошибки хотя бы эта часть программы будет работать, а не откажет из-за нехватки памяти.
- В коде OpenSSL статический массив `OSSL_STORE_str_reasons` используется для хранения информации об ошибках, возникающих при работе с сертификатами.

Применение к сквозному примеру

Код почти не изменился. Мы только добавили ключевое слово `static` перед объявлением переменной `text` и увеличили длину текста:

```
#define MAX_TEXT_SIZE 1024

void encryptCaesarText()
{
 static char text[MAX_TEXT_SIZE];
 strlcpy(text, "LARGETEXTTHATCOULDBETHOUSANDCHARACTERSLONG", MAX_TEXT_SIZE);
 caesar(text, strnlen(text, MAX_TEXT_SIZE));
 printf("Зашифрованный текст: %s\n", text);
}
```

Теперь текст хранится не в стеке, а в статической памяти. Поступая так, следует помнить, что переменная существует в единственном экземпляре и сохраняет свое значение (даже если вход в функцию осуществляется несколько раз). Это может оказаться проблемой в многопоточной программе, потому что в этом случае при доступе к переменной нужно гарантировать взаимное исключение.

В настоящее время наше программа не многопоточная. Однако требования к системе изменились: теперь мы хотим прочитать текст из файла, зашифровать его и показать зашифрованный текст. Мы не знаем, какой длины будет текст, а он может оказаться очень длинным. Поэтому принимается решение использовать динамическое выделение памяти:

```
void encryptCaesarText()
{
 /* открыть файл (для простоты опускаем проверку ошибок) */
 FILE* f = fopen("my-file.txt", "r");

 /* получить длину файла */
 fseek(f, 0, SEEK_END);
 int size = ftell(f);
```

```
  /* выделить буфер */
  char* text = malloc(size);

  ...
}
```

Но как должен продолжаться этот код? Мы выделили память из кучи? А как ее освободить? Для начала мы решаем поручить освобождение памяти кому-то другому – операционной системе. Так мы приходим к паттерну «Отложенная очистка».

Отложенная очистка

Контекст

Вы хотите где-то в программе сохранить данные, и эти данные велики (возможно, вы даже не знаете их размер заранее). Размер данных изменяется редко, и данные нужны на протяжении всего времени работы программы. Программа работает недолго (не много дней без перезагрузки).

Проблема

Динамическая память необходима, если требуется много памяти, но точный размер заранее неизвестен. Однако ручное освобождение памяти – головная боль и источник многих программных ошибок.

Во многих ситуациях – например, если размер данных велик и заранее неизвестен – нельзя поместить данные в стек или в статическую память. Поэтому приходится пользоваться динамической памятью и заниматься ее выделением. Но тогда возникает вопрос, как освобождать эту память. Освобождение памяти – источник многих программных ошибок. Можно случайно освободить память слишком рано, что приведет к висячему указателю. Можно случайно освободить одну и ту же память дважды. Оба вида программных ошибок ведут к неопределенному поведению программы, например к краху в более поздний момент. Такие ошибки очень трудно отлаживать, и программисты на C тратят на это много времени.

По счастью, для большинства видов памяти существует тот или иной механизм автоматической очистки. Память в стеке автоматически освобождается при возврате из функции. Статическая память и куча автоматически освобождаются при завершении программы.

Решение

Выделите динамическую память и позвольте операционной системе освободить ее по завершении программы.

По завершении программы операционная система подчищает за вашим процессом, и большинство систем в том числе освобождают память, которую вы выделили, но не освободили сами. Воспользуйтесь этим и поручите ОС все

заботы об отслеживании того, какая память нуждается в освобождении, и пусть она же освобождает ее.

```
void someCode()
{
 char* memory = malloc(size);
 ...
 /* что-то сделать с памятью */
 ...
 /* об освобождении памяти беспокоиться не нужно */
}
```

На первый взгляд этот подход выглядит чересчур брутально. Вы осознанно допускаете утечки памяти. Однако именно такой стиль кодирования используется в других языках программирования, в которых имеется сборщик мусора. Можно было бы даже включить библиотеку сборки мусора в C, чтобы использовать все преимущества автоматического освобождения памяти (а вместе с ними и недостаток – менее предсказуемый момент очистки).

Сознательно допущенные утечки памяти – возможное решение для некоторых приложений, особенно тех, которые работают недолго и выделяют память не слишком часто. Но в других приложениях это не годится, и надо будет применять паттерн «Единоличное владение памятью» и заниматься ее освобождением. Простой способ освободить память, для которой раньше применялась отложенная очистка, – воспользоваться паттерном «Обертка выделения» и написать одну функцию, которая в конце программы освобождает всю выделенную память.

Последствия

Очевидное достоинство – возможность использовать динамическую память, не «заморачиваясь» ее освобождением. Это здорово облегчает жизнь программисту. К тому же не тратится процессорное время на освобождение памяти, что ускоряет процедуру остановки вашей программы.

Однако платить за это приходится тем, что другие работающие процессы не смогут получить память, которую вы не освободили. Быть может, вы и сами в какой-то момент не сможете выделить дополнительную память, потому что ее осталось мало, а вы не освободили ту память, которую могли бы освободить. В частности, если память выделяется слишком часто, то такой подход не годится. Вместо этого используйте паттерн «Единоличное владение» и освобождайте память самостоятельно.

Применяя этот паттерн, вы осознанно идете на утечку памяти. Возможно, это хорошо для вас, но может быть плохо для клиентов, вызывающих ваши функции. Если вы пишете библиотеку, которой могут пользоваться другие люди, наличие утечек памяти неприемлемо. Кроме того, если вы сами захотите писать чистый код в других местах программы и, к примеру, использовать такой инструмент отладки работы с памятью, как *valgrind*, то столкнетесь с проблемой интерпретации результатов в случае, когда какая-то часть программы написана коряво и не освобождает память.

Этот паттерн легко можно предъявить как оправдание небрежного отношения к освобождению памяти даже в тех случаях, когда это не оправдано. Поэтому следует дважды проверить, действительно ли контекст позволяет сознательно пренебречь освобождением памяти. Если есть шанс, что программа будет развиваться и очищать память все-таки потребуется, то начинать с «Отложенной очистки» не стоит, а лучше сразу прибегнуть к «Единоличному владению».

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В Wireshark функция `pcap_free_datalinks` при некоторых условиях сознательно не освобождает всю память. Причина в том, что часть кода Wireshark могла быть построена другим компилятором и скомпонована с другими библиотеками времени выполнения. Освобождение памяти, выделенной таким кодом, привело бы к краху программу. Поэтому память не освобождается вовсе.
- Драйверы устройств в операционной системе B&R Automation Runtime обычно не включают функцию очистки. Вся выделенная ими память никогда не освобождается, потому что драйверы никогда не выгружаются. Если нужно использовать другой драйвер, то перезагружается вся система. Поэтому явное освобождение памяти излишне.
- Код системы управления данными NetDRMS, в которой хранятся изображения Солнца для научных расчетов, не освобождает всю память в случае ошибки. Например, в случае ошибки функция `EmptyDir` не освобождает память и другие ресурсы, относящиеся к доступу к файлам, потому что это привело бы к еще более серьезной ошибке, и программа все равно аварийно завершилась бы.

В любом коде на C, где используется библиотека сборки мусора, применяется этот паттерн, а все недостатки утечки памяти компенсируются явной сборкой мусора.

Применение к сквозному примеру

В коде просто опущены все вызовы `free`. Кроме того, код реорганизован – доступ к файлам вынесен в отдельные функции.

```
/* Возвращает длину файла с именем 'filename' */
int getFileLength(char* filename)
{
 FILE* f = fopen(filename, "r");
 fseek(f, 0, SEEK_END);
 int file_length = ftell(f);
 fclose(f);
 return file_length;
```

```
}

/* Сохраняет содержимое файла с именем 'filename' в предоставленном буфере
   'buffer' (который должен иметь длину не менее 'file_length'). Файл должен
   содержать только заглавные буквы без символов новой строки (именно
   такие данные принимает наша функция caesar). */
void readFileContent(char* filename, char* buffer, int file_length)
{
 FILE* f = fopen(filename, "r");
 fseek(f, 0, SEEK_SET);
 int read_elements = fread(buffer, 1, file_length, f);
 buffer[read_elements] = '\0';
 fclose(f);
}

void encryptCaesarFile()
{
 char* text;
 int size = getFileLength("my-file.txt");
 if(size>0)
 {
  text = malloc(size);
  readFileContent("my-file.txt", text, size);
  caesar(text, strnlen(text, size));
  printf(«Зашифрованный текст: %s\n», text);
  /* здесь память не освобождается */
 }
}
```

Мы выделяем память, но не вызываем `free` для ее освобождения. Вместо этого мы позволяем указателям на память покинуть область видимости, создавая тем самым утечку памяти. Но это не проблема, потому что программа все равно завершается сразу после этого, и операционная система освобождает память.

Такой подход кажется небрежным, но в некоторых случаях вполне приемлем. Если память нужна на протяжении всего времени работы программы или если программа работает недолго и вы уверены, что ее код не будет развиваться или использоваться где-то еще, то возможность наплевать на освобождение памяти может стать решением, котором сильно облегчит вам жизнь. Но надо иметь уверенность в том, что программа не начнет эволюционировать и не превратится в долго работающую. Если такой уверенности нет, то нужно поискать другой подход.

Именно этим мы и займемся далее. Вы хотите зашифровать несколько файлов, а именно все файлы, находящиеся в текущем каталоге. Вы понимаете, что придется выделять память чаще и отказ от освобождения памяти больше не годится, потому что приведет к потреблению слишком большого объема

памяти. Это может создать проблемы для других программ или для вашей собственной.

Возникает вопрос, в каком месте кода следует освобождать память. Кто за это отвечает? Необходим паттерн «Единоличное владение».

Единоличное владение

Контекст

В программе имеется много данных заранее неизвестно размера, и для их хранения используется динамическая память. Память не нужна на протяжении всего времени работы программы, и вы часто вынуждены выделять память разного размера, поэтому прибегнуть к «Отложенной очистке» не получится.

Проблема

Использование динамической памяти дает большую власть, но вместе с ней приходит и большая ответственность – память нужно освобождать. В больших программах трудно гарантировать, что вся динамическая память будет надлежащим образом освобождена.

Освобождение динамической памяти сулит немало трудностей. Можно освободить ее слишком рано, когда кому-то еще нужен к ней доступ (висячий указатель). Или по ошибке освободить одну и ту же память несколько раз. Обе программных ошибки ведут к неопределенному поведению, например к краху программы в каком-то другом месте. К тому же они создают угрозу безопасности, которая может быть эксплуатирована противником. И ко всему прочему такие ошибки очень трудно отлаживать.

Но память все равно освобождать нужно, потому что иначе со временем программа захватит слишком много памяти, после чего она сама или другие процессы не смогут получить дополнительную память.

Решение

В тот момент, когда выделяете память, четко определите и документируйте, где она должна быть освобождена и кто это должен сделать.

В коде должно быть ясно документировано, кто владеет памятью и как долго она должна оставаться действительной. Лучше всего еще перед написанием первого вызова `malloc` спросить себя, где эта память будет освобождена. В комментариях к объявлению функции нужно указать, передает ли эта функция какие-нибудь буферы памяти и, если да, кто отвечает за их освобождение.

В других языках программирования, например в C++, для такой документации можно использовать программные конструкции. В таких конструкциях, как `unique_ptr` или `shared_ptr`, уже на уровне объявления функции видно, кто отвечает за освобождение памяти. Поскольку в C подобных конструкций нет, приходится тщательно документировать обязанности в форме комментариев.

Если возможно, возлагайте ответственность за выделение и освобождение на одну функцию, как в паттерне «Объектная обработка ошибок», где в коде

имеется ровно одна точка для вызова функций, подобных конструктору и деструктору:

```
#define DATA_SIZE 1024
void function()
{
 char* memory = malloc(DATA_SIZE);
 /* работа с памятью */
 free(memory);
}
```

Если ответственность за выделение и освобождение разнесена и владение памятью передается, то ситуация осложняется. В некоторых случаях без этого не обойтись, например если одна лишь выделяющая память функция знает размер данных, а эти данные необходимы другим функциям:

```
/* Выделяет и возвращает буфер, который должен быть освобожден вызывающей стороной */
char* functionA()
{
 char* memory = malloc(data_size); ❶
 /* заполнить память */
 return memory;
}

void functionB()
{
 char* memory = functionA();
 /* поработать с памятью */
 free(memory); ❷
}
```

❶ Вызываемая сторона выделяет память.
❷ Вызывающая сторона отвечает за освобождение памяти.

По возможности старайтесь избегать разнесения ответственности за выделение и освобождение памяти по разным функциям. Но в любом случае документируйте, кто отвечает за освобождение, чтобы тут не оставалось места для сомнений.

Есть и другой паттерн, описывающий более специальные ситуации, относящиеся к владению памятью: «Буфер» (или «Экземпляр»), принадлежащий вызывающей стороне, когда вызывающая сторона отвечает за выделение и освобождение памяти.

Последствия

Наконец, можно выделить память и корректно освободить ее. Это повышает гибкость. Можно временно заимствовать большие объемы памяти из кучи, а впоследствии вернуть ее, сделав доступной другим.

Но, конечно, за это удобство приходится расплачиваться. Вы должны заниматься освобождением памяти, что усложняет программирование. Даже при использовании «Единоличного владения» ошибки при работе с памятью случаются и приводят к ситуациям, которые трудно отлаживать. Кроме того, на освобождение памяти уходит время. Явное документирование того, где должна освобождаться память, помогает предотвратить некоторые ошибки и в целом делает код проще для понимания и сопровождения. Чтобы еще снизить количество ошибок, можете использовать паттерны «Обертка выделения» и «Проверка указателя».

При использовании выделения и освобождения динамической памяти нарастают проблемы фрагментации кучи и увеличивается время выделения и доступа к памяти. В некоторых приложениях это вообще не проблема, но есть и такие, где эти вопросы стоят очень остро. В таком случае может помочь параметр «Пул памяти».

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В книге Kamran Amini «Extreme C» (Packt, 2019) рекомендуется возлагать ответственность за освобождение памяти на функцию, которая ее выделила, и документировать в комментариях, какая функция или объект владеет памятью. Конечно, эта идея остается в силе и при использовании обертывающих функций. Тогда функция, вызвавшая обертку выделения, должна вызывать и обертку освобождения.
- В реализации функции `mexFunction` из среды численных расчетов MATLAB четко документировано, кто владеет памятью и должен ее освободить.
- В игре NetHack явно документировано, должны ли освобождать какую-то память стороны, вызывающие функции. Например, функция `nh_compose_ascii_screenshot` выделяет и возвращает строку, которая должна быть освобождена вызывающей стороной.
- Диссектор Wireshark для хешей «идентификаторов сообщества» в документации по входящим в него функциям ясно указывает, кто отвечает за освобождение памяти. Например, функция `communityid_calc` выделяет память и требует, чтобы ее освобождала вызывающая сторона.

Применение к сквозному примеру

Функциональность `encryptCaesarFile` осталась прежней. Изменилось только то, что теперь вы вызываете `free` для освобождения памяти и четко документируете, кто отвечает за ее освобождение. Кроме того, реализована функция `encryptDirectoryContent`, которая шифрует все файлы в текущем каталоге.

```
/* Получив имя файла 'filename', эта функция читает текст из указанного файла
   и печатает текст, зашифрованный шифром Цезаря. Эта функция отвечает за выделение
   и освобождение буферов, необходимых для хранения содержимого файла. */
void encryptCaesarFile(char* filename)
```

```
{
 char* text;
 int size = getFileLength(filename);
 if(size>0)
 {
  text = malloc(size);
  readFileContent(filename, text, size);
  caesar(text, strnlen(text, size));
  printf("Зашифрованный текст: %s\n", text);
  free(text);
 }
}

/* Для каждого файла в текущем каталоге эта функция читает текст из файла
   и печатает текст, зашифрованный шифром Цезаря. */
void encryptDirectoryContent()
{
 struct dirent *directory_entry;
 DIR *directory = opendir(".");
 while ((directory_entry = readdir(directory)) != NULL)
 {
  encryptCaesarFile(directory_entry->d_name);
 }
 closedir(directory);
}
```

Этот код печатает результаты шифрования всех файлов в текущем каталоге шифром Цезаря. Отметим, что он работает только в UNIX-системах, и для простоты не проверяется, что содержимое находящихся в каталоге файлов удовлетворяет предъявляемым требованиям.

Теперь память освобождается, когда в ней отпадает необходимость. Отметим, что не вся необходимая программе память выделяется одновременно. В каждый момент времени занята только память для хранения одного файла. Поэтому потребление памяти программой значительно сокращается, что особенно заметно, когда каталог содержит много файлов.

В показанном выше коде ошибки не обрабатываются. Например, что будет, если памяти не хватает? Программа просто «грохнется». В таких ситуациях какая-то обработка ошибок необходима, но каждый раз проверять указатели, возвращенные `malloc`, слишком утомительно. Нам нужна «Обертка выделения».

Обертка выделения

Контекст

Вы выделяете динамическую память в нескольких местах и хотите реагировать на такие ошибки, как нехватка памяти.

Проблема

Любое выделение динамической памяти может завершиться неудачно, поэтому нужно проверять результат и реагировать соответственно. Это утомительно, потому что в коде может быть много мест, где такие проверки необходимы.

Функция `malloc` возвращает `NULL`, если запрошенной памяти нет. С одной стороны, пренебрежение проверкой значения, возвращенного `malloc`, чревато крахом программы в случае, если возвращен указатель `NULL`. С другой стороны, если проверять возвращенное значение везде, где выделяется память, то код станет более сложным, а значит, его будет труднее читать и сопровождать.

Если такие проверки разбросаны по всей кодовой базе, а впоследствии вы захотите изменить поведение программы в случае ошибок выделения памяти, то придется просматривать много мест. Кроме того, простое добавление проверки ошибок в существующие функции нарушает принцип единственной обязанности, который гласит, что каждая функция должна отвечать только за что-то одно (а не за такие разные вещи, как логика программы и выделение памяти).

Кроме того, если вы впоследствии захотите изменить способ выделения памяти, например явно инициализировать всю выделенную память, то наличие многочисленных обращений к функциям выделения, разбросанных по всему коду, затруднит работу.

Решение

Оберните вызовы функций выделения и освобождения и реализуйте в этих обертках обработку ошибок или дополнительное управление памятью.

Реализуйте функции-обертки для вызовов `malloc` и `free` и, когда нужно выделить или освободить память, вызывайте только эти функции. Обертка будет служить единым центральным местом для обработки ошибок. Например, можно проверить возвращенный указатель на выделенную память (см. паттерн «Проверка указателя») и в случае ошибки завершить программу, как показано ниже.

```
void* checkedMalloc(size_t size)
{
 void* pointer = malloc(size);
 assert(pointer);
 return pointer;
}

#define DATA_SIZE 1024
void someFunction()
{
 char* memory = checkedMalloc(DATA_SIZE);
 /* поработать с памятью */
 free(memory);
}
```

Вместо завершения программы можно запротоколировать ошибки. Для протоколирования отладочной информации проще даже использовать макрос, а не функцию. Тогда можно, не утруждая вызывающую сторону, записать в журнал имя файла, имя функции и номер строки, в которой произошла ошибка. При наличии такой информации программисту будет очень легко найти место, где возникла ошибка. К тому же использование макроса вместо функции в качестве обертки позволяет сэкономить на одном вызове функции (но в большинстве случаев это неважно, потому что компилятор в любом случае встроит вызов). Применяя макросы для выделения и освобождения памяти, мы даже можем реализовать синтаксический аналог конструктора:

```
#define NEW(object, type)                  \
do {                                       \
 object = malloc(sizeof(type));            \
 if(!object)                               \
 {                                         \
  printf("Malloc Error: %s\n", __func__); \
  assert(false);                           \
 }                                         \
} while (0)

#define DELETE(object) free(object)

typedef struct{
 int x;
 int y;
}MyStruct;

void someFunction()
{
 MyStruct* myObject;
 NEW(myObject, MyStruct);
 /* поработать с объектом */
 DELETE(myObject);
}
```

Помимо обработки ошибок, в функциях-обертках можно делать и другие вещи. Например, можно следить, какую память выделяла программа, и сохранять эту информацию в списке вместе с именем файла и номером строки (для этого понадобилась бы также обертка для `free`, как в примере выше). Так вы сможете легко напечатать отладочную информацию, если захотите узнать, какая память выделена в данный момент (и какую вы, возможно, забыли освободить). Но если вас интересует такого рода информация, то проще воспользоваться инструментом для отладки работы с памятью типа valgrind. Кроме того, если вы будете следить за тем, какую память выделили, то сможете написать функцию, которая освобождает всю память разом, – это один из вариантов сделать программу чище в случае использования паттерна «Отложенная очистка».

Помещать все в одно место не всегда удобно. Быть может, в приложении есть некритические части, где возникновение ошибки необязательно должно приводить к завершению всего приложения. В таком случае имеет смысл завести несколько «Оберток выделения». Одна по-прежнему будет содержать `assert` и может использоваться в критических частях, ошибка в которых не позволяет приложению работать дальше, а другая – в некритических, где можно вернуть код состояния и корректно обработать ошибку.

Последствия

Обработка ошибок и другие операции управления памятью теперь сосредоточены в одном месте. В тех местах программы, где нужно выделить память, вы просто вызываете обертку, так что явно обрабатывать ошибки в этом месте уже не нужно. Но эта идея работает только для некоторых видов обработки ошибок. Если в случае ошибки нужно завершить программу, то это как раз то, что нужно, но если можно продолжить работу, сократив функциональность, то все равно придется вернуть из обертки какую-то информацию об ошибке и отреагировать на нее. Поэтому «Обертка выделения» не облегчает жизнь. Однако и при таком сценарии в обертке можно реализовать протоколирование, немного уменьшив объем работы.

Функция-обертка полезна при тестировании, потому что является центральным местом, где можно изменить поведение функции выделения памяти. Кроме того, в обертке можно реализовать имитацию (подменить обернутые вызовы какой-то другой тестовой функцией), не затрагивая другие вызовы `malloc` (скажем, в стороннем коде).

Отделение обработки ошибок от вызывающего кода с помощью обертки – рекомендуемая практика, потому что у вызывающей стороны не возникает искушения реализовать обработку ошибок прямо в коде, занимающемся реализацией другой логики. Выполнение нескольких вещей в одной функции (логики программы и развернутой обработки ошибок) нарушает принцип единственной обязанности.

Обертка выделения позволяет единообразно обрабатывать ошибки выделения памяти и в случае необходимости изменить способ обработки ошибок. Если вы решите протоколировать информацию об ошибке, то внести изменения надо будет только в одном месте. Если впоследствии вы захотите не вызывать `malloc` напрямую, а воспользоваться «Пулом памяти», то наличие обертки существенно упростит задачу.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В книге David R. Hanson «C Interfaces and Implementations» (Addison-Wesley, 1996) функция-обертка используется для выделения памяти в реализации пула памяти. Обертки просто вызывают `assert` для завершения программы в случае ошибки.
- Библиотека GLib предоставляет функции `g_malloc` и `g_free` среди прочих функций работы с памятью. Функция `g_malloc` хороша тем, что в случае ошибки останавливает программу («Принцип самурая»). Поэто-

му вызывающей стороне не нужно проверять значение, возвращенное при каждом выделении памяти.

- В анализаторе веб-журналов реального времени GoAccess реализована функция `xmalloc`, обертывающая вызовы `malloc` с целью обработки ошибок.
- «Обертка выделения» – частный случай паттерна «Декоратор», описанного в книгах Erich Gamma, Richard Helm, Ralph Johnson и John Vlissides «Design Patterns: Elements of Reusable Object-Oriented Software» (Prentice Hall, 1997).

Применение к сквозному примеру

Теперь вместо вызова самих функций `malloc` и `free` мы всюду в коде используем функции-обертки:

```
/* Выделяет память и вызывает assert, если памяти недостаточно */
void* safeMalloc(size_t size)
{
 void* pointer = malloc(size);
 assert(pointer); ❶
 return pointer;
}

/* Освобождает память, на которую указывает 'pointer' */
void safeFree(void *pointer)
{
 free(pointer);
}

/* Получив имя файла 'filename', эта функция читает текст из указанного файла
   и печатает текст, зашифрованный шифром Цезаря. Эта функция отвечает за выделение
   и освобождение буферов, необходимых для хранения содержимого файла. */
void encryptCaesarFile(char* filename)
{
 char* text;
 int size = getFileLength(filename);
 if(size>0)
 {
  text = safeMalloc(size);
  readFileContent(filename, text, size);
  caesar(text, strnlen(text, size));
  printf("Зашифрованный текст: %s\n", text);
  safeFree(text);
 }
}
```

❶ В случае ошибки выделения мы следуем «Принципу самурая» и завершаем программу. Для такого приложения это допустимая стратегия. Если невозможно обработать ошибку корректно, то завершение программы – правильный выбор.

Паттерн «Обертка выделения» хорош тем, что предлагает центральное место для обработки ошибок выделения памяти. Не нужно писать код для проверки указателя после каждого выделения. Имеется также обертка для освобождения памяти, которая может пригодиться в будущем, например, если вы решите вести учет памяти, выделенной приложением в каждый момент времени.

Сразу после выделения полученный указатель проверяется. И больше нигде это не делается, считается, что указатели, передаваемые между функциями, действительны. Это хорошо при условии, что в код не закрались ошибки, но если непреднамеренно производится доступ по недействительному указателю, то отладить ситуацию будет трудно. Чтобы улучшить код и подстраховаться, вы решаете воспользоваться паттерном «Проверка указателя».

Проверка указателя

Контекст

В вашей программе есть много мест, где память выделяется и освобождается, и много мест, где производится доступ к памяти или другим ресурсам по указателю.

Проблема

Программные ошибки, заключающиеся в доступе по недействительному указателю, приводят к неуправляемому поведению программы, их очень трудно отлаживать. Но поскольку работа с указателями встречается сплошь и рядом, велики шансы внести такие ошибки.

Программирование на C требует повсеместных операций с указателями, и чем больше в коде таких мест, тем больше шансов внести ошибки. Использование уже освобожденного или неинициализированного указателя ведет к ошибкам, которые трудно отлаживать. Любая ситуация такого рода очень серьезна. Результатом становится неуправляемое поведение программы и (если повезет) ее крах. Если же не повезет, то ошибка проявится в каком-то другом месте, возможно, далеко отстоящем от места возникновения, и вы потратите неделю, чтобы найти и исправить ее. Вы хотите, чтобы программа была более надежно защищена от таких ошибок, чтобы они были менее серьезными и чтобы, если ошибка все-таки произойдет, найти ее причину было проще.

Решение

Явно делайте недействительными неинициализированные или освобожденные указатели и всегда проверяйте действительность указателя, прежде чем производить доступ по нему.

В объявлении указательной переменной присвойте ей значение `NULL`. Также сразу после вызова `free` явно присваивайте указателю значение `NULL`. Если используете паттерн «Обертка» выделения и определили макрос, обертывающий функцию `free`, то можете сбросить указатель в `NULL` внутри макроса, чтобы не писать код обнуления указателя после каждого освобождения памяти.

Заведите обертывающую функцию или макрос, который проверяет указатель, и если он равен NULL, то завершает программу и протоколирует ошибку, включая отладочную информацию. Если завершение программы неприемлемо, то в случае нулевого указателя можно выполнить доступ по нему и попытаться обработать ошибку корректно. Это позволит программе продолжить работу с сокращенной функциональностью.

```
void someFunction()
{
 char* pointer = NULL; /* явно сделать недействительным неинициализированный
                          указатель */
 pointer = malloc(1024);
 if (pointer != NULL) /* проверить указатель перед доступом по нему  */
 {
  /* поработать с указателем */
 }
 free(pointer);
 pointer = NULL; /* явно сделать недействительным указатель на освобожденную память */
}
```

Последствия

Ваш код стал немного более защищенным от ошибок, связанных с указателями. Любая такая ошибка может быть идентифицирована и не приводит к неопределенному поведению программы, которое могло бы стоить вам многих часов или даже дней отладки.

Однако за все приходится платить. Код становится длиннее и сложнее. Примененная стратегия аналогична строительному страховочному поясу. Мы проделываем дополнительную работу ради большей безопасности. Для каждого доступа по указателю имеется дополнительная проверка. Такой код труднее читать. Проверка указателя перед доступом по нему требует по меньшей мере одной лишней строки кода. Если вы не завершаете программу, а продолжаете работу с сокращенной функциональностью, то программу становится гораздо труднее читать, сопровождать и тестировать.

Если вы случайно вызовете free несколько раз для одного и того же указателя, то второй вызов не приведет к ошибке, потому что после первого указатель уже недействителен, а вызов free для нулевого указателя безвреден. Тем не менее имеет смысл запротоколировать такую ситуацию, чтобы впоследствии отыскать первопричину ошибки.

Но даже после всего этого вы не застрахованы на 100 % от всех типов ошибок, связанных с указателями. Например, можно забыть об освобождении памяти – и вот вам утечка. Или обратиться по указателю, который не был должным образом инициализирован, но эту ошибку вы, по крайней мере, обнаружите и сможете отреагировать соответственно. Потенциальный недостаток заключается в том, что если вы решите сократить функциональность программы, но продолжить работу, то можете не узнать об ошибке, которую потом будет труднее найти.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В реализации умных указателей на C++ обернутый указатель делается недействительным после освобождения умного указателя.
- Cloudy – программа для физических расчетов (спектрального синтеза). Она содержит код интерполяции данных (фактор Гаунта). Программа проверяет указатели перед доступом по ним и явно сбрасывает указатели в `NULL` после вызова `free`.
- Программа libcpp из коллекции компиляторов GNU (GCC) делает освобожденные указатели недействительными. Например, так делается для указателей в файле *macro.c*.
- Функция `HB_GARBAGE_FUNC` из СУБД MySQL присваивает указателю `ph` значение `NULL`, чтобы предотвратить случайный доступ по нему или повторное освобождение.

Применение к сквозному примеру

Теперь код принял такой вид:

```
/* Получив имя файла 'filename', эта функция читает текст из указанного файла
   и печатает текст, зашифрованный шифром Цезаря. Эта функция отвечает за выделение
   и освобождение буферов, необходимых для хранения содержимого файла. */
void encryptCaesarFile(char* filename)
{
 char* text = NULL; ❶
 int size = getFileLength(filename);
 if(size>0)
 {
  text = safeMalloc(size);
  if(text != NULL) ❷
  {
   readFileContent(filename, text, size);
   caesar(text, strnlen(text, size));
   printf("Зашифрованнй текст: %s\n", text);
  }
  safeFree(text);
  text = NULL; ❶
 }
}
```

❶ В тех местах, где указатель недействителен, мы явно сбрасываем его в `NULL` – для страховки.

❷ Перед доступом по указателю `text` проверяется, что он действителен. Если нет, то указатель не используется (не разыменовывается).

Избыточное выделение памяти в Linux

Имейте в виду, что действительный указатель на память не всегда гарантирует, что к этой памяти можно безопасно обращаться. В современных версиях Linux применяется принцип *избыточного выделения* памяти (overcommit). В результате программе, запрашивающей память, предоставляется виртуальная память, но между ней и физической памятью нет прямого соответствия. Доступна ли требуемая физическая память, проверяется только в момент доступа к ней. Если физической памяти не хватает, то ядро Linux останавливает приложения, потребляющие слишком много памяти (среди них может оказаться и ваше). Избыточное выделение хорошо тем, что становится не так важно проверять, успешно ли выделена память (потому что обычно с этим все хорошо), и вы можете выделить много памяти, чтобы подстраховаться, даже если на самом деле нужно меньше. Но у этого подхода есть и большой недостаток — даже имея действительный указатель, вы не можете быть уверены, что доступ к памяти окажется успешным и программа не «грохнется». Есть и еще один недостаток – вы начинаете лениться, не проверяете возвращенные значения и не вычисляете, сколько памяти вам действительно нужно.

Следующая задача – показать имя файла вместе с зашифрованным текстом. Но вы не хотите выделять память прямо из кучи, поскольку опасаетесь фрагментации памяти в результате многократного выделения небольших блоков (для имен файлов) и больших блоков (для содержимого файлов). Вместо непосредственного выделения динамической памяти вы реализуете «Пул памяти».

Пул памяти

Контекст

Вы часто выделяете и освобождаете динамическую память из кучи блоками примерно одинакового размера. Ни на этапе компиляции, ни в момент инициализации программы неизвестно, когда и где программе понадобятся эти блоки.

Проблема

Частое выделение и освобождение памяти из кучи приводит к фрагментации.

При выделении памяти для объектов, особенно сильно различающихся по размеру, и последующем освобождении некоторых из них куча становится фрагментированной. Даже если размеры блоков, выделяемых вашей программой, примерно одинаковы, сами операции выделения могут чередоваться с

операциями других работающих программ, так что в итоге размеры блоков будут различаться, и возникнет фрагментация.

Функция `malloc` завершается успешно, только если имеется достаточно непрерывной памяти. Это значит, что, даже когда общего объема свободной памяти хватает, `malloc` все равно может вернуть ошибку, если память фрагментирована и непрерывного участка требуемого размера нет. Фрагментация означает, что память используется не очень хорошо.

Фрагментация – серьезная проблема для долго работающих систем, в том числе встраиваемых. Если система работает годами, выделяя и освобождая много мелких блоков, то в конце концов она не сможет выделить блок большого размера. Поэтому обязательно нужно решать проблему фрагментации, если вы не готовы смириться с тем, что систему придется время от времени перезагружать.

Еще одна проблема при использовании динамической памяти, особенно в сочетании со встраиваемыми системами, – то, что выделение памяти из кучи требует времени. Другие процессы могут использовать ту же самую кучу, поэтому операции выделения чередуются, и время выполнения одной операции становится очень трудно предсказать.

Решение

Выделите большой кусок памяти, который будет существовать на всем протяжении работы программы. Во время выполнения выделяйте блоки фиксированного размера из этого пула памяти, а не напрямую из кучи.

Пул памяти можно разместить в статической памяти или выделить из кучи на этапе инициализации программы и освободить в самом конце. Выделение из кучи хорошо тем, что в случае необходимости можно будет выделить дополнительную память и увеличить размер пула.

Напишите функции для выделения и освобождения блоков памяти заранее заданного фиксированного размера из пула. Всюду, где программе нужна память такого размера, она может использовать эти функции (вместо `malloc` и `free`).

```
#define MAX_ELEMENTS 20;
#define ELEMENT_SIZE 255;

typedef struct
{
 bool occupied;
 char memory[ELEMENT_SIZE];
} PoolElement;

static PoolElement memory_pool[MAX_ELEMENTS];

/* Возвращает память размера не меньше 'size' или NULL, если
   в пуле не осталось свободных блоков. */
void* poolTake(size_t size)
{
```

```
 if(size <= ELEMENT_SIZE)
 {
  for(int i=0; i<MAX_ELEMENTS; i++)
  {
   if(memory_pool[i].occupied == false)
   {
     memory_pool[i].occupied = true;
     return &(memory_pool[i].memory);
   }
  }
 }
 return NULL;
}

/* Возвращает блок памяти ('pointer') обратно в пул */
void poolRelease(void* pointer)
{
 for(int i=0; i<MAX_ELEMENTS; i++)
 {
  if(&(memory_pool[i].memory) == pointer)
  {
   memory_pool[i].occupied = false;
   return;
  }
 }
}
```

Выше показана простая реализация пула памяти, которую можно во многих отношениях улучшить. Например, свободные блоки памяти можно было бы хранить в списке, чтобы ускорить нахождение блока. Кроме того, можно было бы использовать мьютексы и семафоры, чтобы пул работал в многопоточной среде.

При работе с пулом памяти необходимо знать, какие данные будут в нем храниться, потому что размер блоков памяти должен быть известен заранее. Можно было бы также использовать эти блоки для хранения меньших данных, но тогда часть памяти будет расходоваться впустую.

В качестве альтернативы блокам фиксированного размера можно было бы даже реализовать пул памяти, допускающий блоки переменного размера. Но при этом, хотя собственная память будет использоваться лучше, возникнет та же проблема фрагментации, что и с кучей.

Последствия

С фрагментацией вы разобрались. Имея пул блоков памяти фиксированной длины, можно быть уверенным, что, как только один блок освобожден, другой можно будет получить. Однако вы должны заранее знать, какие элементы будут храниться в пуле и их размер. Если вы решите хранить также меньшие элементы, то часть памяти будет расходоваться впустую.

При использовании пула переменного размера память впустую не расходуется, зато становится фрагментированной. Эта ситуация чуть лучше, чем в самой куче, потому что вы – единственный пользователь пула (остальные процессы не имеют к нему доступа). Кроме того, вы сами не фрагментируете память, используемую другими процессами. Однако проблема фрагментации никуда не делась.

Неважно, используется пул фиксированного или переменного размера, выигрыш в производительности налицо. Получить память из пула быстрее, чем из кучи, потому что не нужно гарантировать взаимное исключение с другими процессами, нуждающимися в памяти. Кроме того, доступ к памяти, полученной из пула, может оказаться чуть быстрее, потому что все блоки расположены близко друг к другу, что минимизирует накладные расходы на подкачку страниц – механизм, реализуемый операционной системой. Однако первоначальное создание пула занимает некоторое время и замедляет запуск программы.

Память, выделенную из пула, необходимо освобождать, чтобы ее можно было повторно использовать в другом месте программы. Однако вся память, отведенная под пул, остается постоянно занятой и больше никаким процессам недоступна. Если вам столько не нужно, значит, вы неэффективно используете общесистемные ресурсы.

Если начальный размер пула фиксирован, то по прошествии времени в нем может не остаться свободных блоков, пусть даже в куче достаточно доступной памяти. Если размер пула может увеличиваться во время выполнения, то может случиться, что время получения памяти из него неожиданно возрастет, если для получения некоторого блока необходимо расширить пул.

Будьте осторожны при использовании пула памяти в приложениях, критичных с точки зрения информационной или технической безопасности. Пул затрудняет тестирование кода и поиск ошибок работы с памятью инструментами анализа кода. Например, инструментам трудно определить, что вы по ошибке обратились к памяти за границами выделенного из пула блока. Кроме него, ваш процесс владеет другими блоками, расположенными непосредственно до и после того, к которому вы собираетесь обратиться, поэтому инструмент анализа кода не знает, что обращение к памяти за границей блока, взятого из пула, был непреднамеренным. На самом деле анализ кода мог бы предотвратить ошибку Heartbleed в OpenSSL, если бы память не выделялась из пула (см. David A. Wheeler «How to Prevent the Next Heartbleed», July 18, 2020 [дата первой публикации April 29, 2014], https://dwheeler.com/essays/heartbleed.html).

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В системах UNIX используется пул фиксированного размера для объектов процессов.
- В книге David R. Hanson «C Interfaces and Implementations» (Addison-Wesley, 1996) приведен пример реализации пула памяти.
- Паттерн «Пул памяти» обсуждается также в книгах Bruce P. Douglass «Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems»

(Addison-Wesley, 2002) и James Noble and Charles Weir «Small Memory Software: Patterns for Systems With Limited Memory» (Addison-Wesley, 2000).

- В диспетчере памяти ION для Android пулы памяти реализованы в файле *ion_system_heap.c*. При возврате частей памяти в пул у вызывающей стороны есть возможность действительно освободить эту часть, если это критично с точки зрения безопасности.
- В системе моделирования дискретных событий smpl, описанной в книге H. M. MacDougall «Simulating Computer Systems: Techniques and Tools» (MIT Press, 1987), пул памяти используется для событий. Это более эффективно, чем выделять и освобождать память для каждого события, поскольку обработка каждого события занимает совсем немного времени, а моделируемых событий очень много.

Применение к сквозному примеру

Для простоты мы реализуем пул памяти блоком фиксированного размера. К пулу не будет одновременного доступа со стороны нескольких потоков, поэтому можно взять приведенную выше реализацию.

В результате получается следующая финальная версия шифрования файла шифром Цезаря.

```
#define ELEMENT_SIZE 255
#define MAX_ELEMENTS 10

typedef struct
{
 bool occupied;
 char memory[ELEMENT_SIZE];
} PoolElement;

static PoolElement memory_pool[MAX_ELEMENTS];

void* poolTake(size_t size)
{
 if(size <= ELEMENT_SIZE)
 {
  for(int i=0; i<MAX_ELEMENTS; i++)
  {
   if(memory_pool[i].occupied == false)
   {
    memory_pool[i].occupied = true;
    return &(memory_pool[i].memory);
   }
  }
 }
 return NULL;
}
```

```
void poolRelease(void* pointer)
{
 for(int i=0; i<MAX_ELEMENTS; i++)
 {
  if(&(memory_pool[i].memory) == pointer)
  {
   memory_pool[i].occupied = false;
   return;
  }
 }
}

#define MAX_FILENAME_SIZE ELEMENT_SIZE

/* Печатает зашифрованное имя файла 'filename'. Эта функция отвечает
   за выделение и освобождение буферов, требуемых для хранения содержимого файла.
   Примечание: имя файла должно состоять только из заглавных букв,
   и мы готовы к тому, что '.' в имени файла тоже будет сдвинута шифром Цезаря. */
void encryptCaesarFilename(char* filename)
{
 char* buffer = poolTake(MAX_FILENAME_SIZE);
 if(buffer != NULL)
 {
  strlcpy(buffer, filename, MAX_FILENAME_SIZE);
  caesar(buffer, strnlen(buffer, MAX_FILENAME_SIZE));
  printf("\nЗашифрованное имя файла: %s ", buffer);
  poolRelease(buffer);
 }
}

/* Для каждого файла в текущем каталоге эта функция читает текст
   из файла и печатает его зашифрованную версию. */
void encryptDirectoryContent()
{
 struct dirent *directory_entry;
 DIR *directory = opendir(".");
 while((directory_entry = readdir(directory)) != NULL)
 {
  encryptCaesarFilename(directory_entry->d_name);
  encryptCaesarFile(directory_entry->d_name);
 }
 closedir(directory);
}
```

Имея эту версию кода, вы можете выполнить шифрование, не опасаясь типичных подводных камней, встречающихся при работе с динамической памятью в C. Есть уверенность, что все указатели на память действительны, что в

случае отсутствия памяти программа будет принудительно завершена и даже что в предопределенной области памяти отсутствует фрагментация.

Но, глядя на код, вы чувствуете, что он стал чрезмерно сложным. Чтобы просто поработать с динамической памятью, пришлось написать десятки строк кода. Впрочем, большую часть этого кода можно повторно использовать для других операций выделения памяти в вашей кодовой базе. И все же применение одного паттерна за другим дается недаром. Каждый паттерн привносит дополнительную сложность. Однако наша цель не в том, чтобы использовать как можно больше паттернов. Цель в том, чтобы применять только те паттерны, которые помогают решать проблемы. Если, к примеру, фрагментация не составляет для вас большой проблемы, то и не используйте специализированный пул памяти. Если можно что-то упростить, упрощайте: например, выделяйте и освобождайте память с помощью `malloc` и `free`, без посредников. А лучше, если есть такая возможность, не используйте динамическую память вовсе.

Резюме

В этой главе представлены паттерны для работы с памятью в программах на C. Паттерн «Сначала стек» рекомендует размещать переменные в стеке, если это возможно. В паттерне «Вечная память» речь идет об использовании памяти с таким же временем жизни, как у всей программы, с целью избежать сложностей выделения и освобождения динамической памяти. Паттерн «Отложенная очистка» также облегчает освобождение памяти – он рекомендует вообще не заниматься этим. С другой стороны, паттерн «Единоличное владение» рекомендует определять, где и кто должен освобождать память. Паттерн «Обертка выделения» описывает центральное место, где можно обработать ошибки выделения памяти и сделать указатели недействительными. Это, в свою очередь, открывает дорогу к реализации паттерн «Проверка указателя» при разыменовывании переменных. Если фрагментация или выделение памяти на длительное время становятся проблемой, на выручку приходит паттерн «Пул памяти».

Все эти паттерны позволяют снять с плеч программиста груз ответственности за принятие детальных проектных решений о том, какую память использовать и когда ее освобождать. Вместо этого программист может последовать рекомендациям паттернов и легко справиться с управлением памятью на C.

Для дополнительного чтения

По сравнению с другими сложными вопросами программирования на C, на тему управления памятью имеется достаточно литературы. Большая ее часть посвящена базовому синтаксису выделения и освобождения памяти, но в следующих книгах даются и более продвинутые рекомендации.

- В книге James Noble and Charles Weir «Small Memory Software: Patterns for Systems With Limited Memory» (Addison-Wesley, 2000) приведено много хорошо проработанных паттернов управления памятью. Например, есть паттерны, описывающие различные стратегии выделения памяти (в начале или по ходу программы), а также пулы памяти и сборщики

мусора. Для всех паттернов приводятся примеры на разных языках программирования.

- Книга Fedor G. Pikus «Hands-on Design Patterns with C++»[1] (Packt, 2019), как явствует из названия, посвящена не C, но принципы управления памятью в C и C++ схожи, так что можно извлечь информацию о том, как управлять памятью в C. Одна глава специально посвящена владению памятью, в ней объясняется, как механизмы C++ (умные указатели) позволяют недвусмысленно указать, кто владеет памятью.
- В книге Kamran Amini «Extreme C» (Packt, 2019) рассматриваются многочисленные вопросы программирования на C, в том числе процесс компиляции, цепочки инструментов, автономное тестирование, конкурентность, внутрипроцессные коммуникации, а также базовый синтаксис C. Имеется также глава о памяти в куче и стеке, где описываются платформенно-зависимые детали представления этих видов памяти в сегментах кода, данных, стека и кучи.
- В книге Bruce P. Douglass «Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems» (Addison-Wesley, 2002) имеются паттерны для систем реального времени. Некоторые из них касаются выделения и освобождения памяти.

Что дальше

В следующей главе даются рекомендации по передаче информации через границы интерфейсов. Представлены паттерны, раскрывающие механизмы, предоставляемые C для передачи информации между функциями, с рекомендациями о том, какие из них использовать.

[1] *Федор Г. Пикус.* Идиомы и паттерны проектирования в современном C++. Москва, ДМК-Пресс 2020.

Глава 4

Возврат данных из С-функций

Возврат данных из функции – задача, с которой вы сталкиваетесь при написании любого кода длиннее 10 строчек, если хотите, чтобы его впоследствии можно было сопровождать. Вернуть данные просто – нужно лишь передать данные, которые вы собираетесь сделать общими для двух функций, – и в С есть только две возможности: вернуть значение непосредственно или с помощью параметров, «передаваемых по ссылке». Не так уж много вариантов, и говорить особо не о чем – верно? Неверно! Даже простая задача возврата данных из С-функции уже не так проста, как кажется, и есть много способов структурировать программу и параметры функции.

Особенно в С, где выделением и освобождением памяти приходится управлять вручную, передача сложных данных между функциями становится нетривиальной задачей ввиду отсутствия деструктора или сборщика мусора, который помог бы освободить занятую данными память. Вы должны спросить себя: помещать ли данные в стек или выделять память из кучи? Кто должен выделять память: вызывающая или вызываемая сторона?

В этой главе собраны рекомендации, как лучше разделять данные между функциями. Эти паттерны помогут начинающим программистам на С освоить методы возврата данных, применяемые в С, а опытным программистам – лучше понять, почему применяются столь разные методы.

На рис. 4.1 приведен обзор паттернов, рассматриваемых в этой главе, и связей между ними, а в табл. 4.1 – краткое описание этих паттернов.

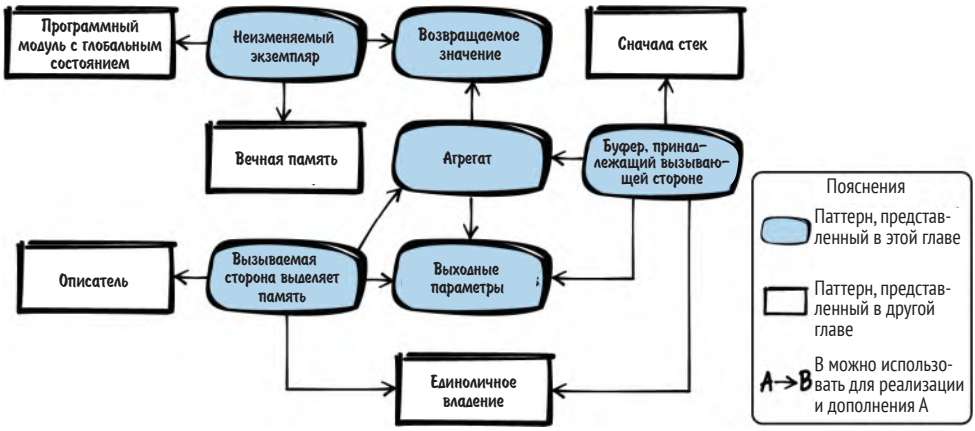


Рис. 4.1. Обзор паттернов для возврата информации

Таблица 4.1. Паттерны для возврата информации

Название паттерна	Краткое описание
Возвращаемое значение	Части, на которые вы хотите разбить функцию, не являются независимыми. Как обычно бывает в процедурном программировании, одна часть производит результат, необходимый другой части. Части разбиваемой функции должны разделять какие-то данные. Поэтому используйте механизм С, предназначенный для получения информации о результате вызова функции: возвращаемое значение. Механизм возврата данных в С копирует результат функции и дает вызывающей стороне доступ к копии
Выходные параметры	Язык С поддерживает возврат только одного значения из функции, поэтому вернуть несколько элементов информации затруднительно. Поэтому возвращайте все данные с помощью единственного вызова функции, эмулируя передачу аргументов по ссылке с помощью указателей
Агрегат	Язык С поддерживает возврат только одного значения из функции, поэтому вернуть несколько элементов информации затруднительно. Поэтому поместите все данные в специально определенный тип. Пусть этот агрегат содержит все связанные данные, которые вы хотите использовать сообща. Определите этот тип в интерфейсе своего компонента, так чтобы вызывающая сторона могла обращаться ко всем данным, хранящимся в экземпляре агрегата

Название паттерна	Краткое описание
Неизменяемый экземпляр	Вы хотите передать информацию, хранящуюся в больших порциях неизменяемых данных, из своего компонента вызывающей стороне. Поэтому разместите экземпляр (например, `struct`), содержащий разделяемые данные, в статической памяти. Предоставляйте эти данные нуждающимся в них пользователям, но организуйте дело так, чтобы они не могли изменить данные
Буфер, принадлежащий вызывающей стороне	Вы хотите передать сложные или большие данные известного размера вызывающей стороне, и эти данные не являются неизменяемыми (т. е. могут быть изменены во время выполнения). Поэтому потребуйте, чтобы вызывающая сторона предоставила буфер и его размер функции, возвращающей данные. Внутри функции скопируйте данные в буфер при условии, что его размер достаточен
Вызываемая сторона выделяет память	Вы хотите передать сложные или большие данные известного размера вызывающей стороне, и эти данные не являются неизменяемыми (т. е. могут быть изменены во время выполнения). Поэтому выделите буфер нужного размера в самой функции, возвращающей данные. Скопируйте данные в буфер и верните указатель на этот буфер

Сквозной пример

Вы хотите представить диагностическую информацию о драйвере Ethernet пользователю. Для начала вы просто помещаете эту функциональность в файл реализации драйвера и напрямую обращаетесь к переменным, содержащим нужную информацию.

```
void ethShow()
{
 printf("получено пакетов: %i\n", driver.internal_data.rec);
 printf("отправлено пакетов: %i\n", driver.internal_data.snd);
}
```

Впоследствии вы приходите к выводу, что эта функциональность, скорее всего, будет разрастаться, поэтому, чтобы код оставался чистым, нужно вынести ее в отдельный файл. Стало быть, необходим простой способ передать информацию из компоненты драйвера Ethernet в диагностический компонент.

Одно из возможных решений: использовать для передачи информации глобальные переменные, но тогда все усилия по разделению файла реализации на части окажутся тщетны. Ведь файл разделяется как раз для того, чтобы показать, что эти части кода слабо связаны друг с другом, а глобальные переменные вернут эту связанность на место.

Гораздо лучшее и очень простое решение состоит в следующем: включить в компонент Ethernet функции-акцессоры, которые предоставят нужную информацию в виде возвращаемого значения.

Возвращаемое значение

Контекст

Вы хотите разбить код на отдельные функции, поскольку помещать все в одну функцию и в один файл реализации – дурная практика, затрудняющая чтение и отладку кода.

Проблема

Части функции, которые вы хотите выделить, не являются независимыми. Как обычно бывает в процедурном программировании, одна часть порождает результат, который нужен другой части. Части функции должны разделять какие-то данные.

Вам нужен простой и понятный механизм разделения данных. Вы хотите сделать явным тот факт, что данные разделяются между функциями, но так, чтобы эти функции не взаимодействовали по каким-то побочным каналам, которые не видны с первого взгляда. Поэтому использование глобальных переменных для возврата информации вызывающей стороне – плохое решение, так как к глобальным переменным можно обратиться и модифицировать их из любой другой части кода. Кроме того, из сигнатуры функции не видно, какие глобальные переменные используются для возврата данных.

У глобальных переменных есть и еще один недостаток – их можно использовать для хранения информации о состоянии, поэтому одинаковые вызовы функций могут вернуть разные результаты. Код становится труднее понять. К тому же, код, в котором для возврата информации применяются глобальные переменные, не реентерабелен, и его небезопасно использовать в многопоточной программе.

Решение

Просто воспользуйтесь тем единственным механизмом, который предусмотрен в С для получения информации о результате выполнения функции: возвращаемым значением. Механизм возврата данных в С копирует результат функции и предоставляет вызывающей стороне доступ к копии.

На рис. 4.2 и в последующем коде показано, как реализовать паттерн «Возвращаемое значение».

Код вызывающей стороны

```
int my_data = getData();
/* использовать my_data */
```

Код вызываемой стороны

```
int getData()
{
 int requested_data;
```

```
  /* .... */
  return requested_data;
}
```

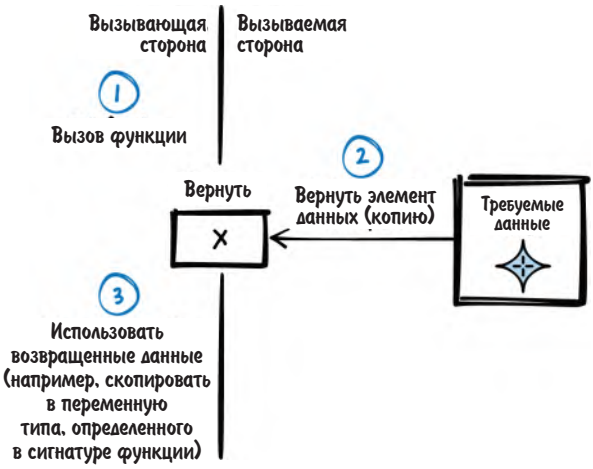


Рис. 4.2. Возвращаемое значение

Последствия

Паттерн «Возвращаемое значение» позволяет вызывающей стороне получить копию результата функции. Никакой другой код, кроме реализации функции, не может модифицировать это значение, а поскольку это копия, оно используется только вызывающей функцией. По сравнению с глобальными переменными лучше видно, какой код влияет на данные, полученные в результате вызова функции.

Кроме того, если вместо глобальных переменных использовать копию результата, то функцию можно сделать реентерабельной и безопасно использовать в многопоточном окружении.

Однако функция может вернуть только один объект встроенного в C типа, указанного в сигнатуре функции. Невозможно определить функцию с несколькими типами возвращаемых значений. Например, нельзя определить функцию, которая возвращает три разных объекта типа `int`. Если вы хотите вернуть больше информации, чем помещается в одном простом скалярном типе C, то должны использовать «Агрегат» или «Выходные параметры».

И если вы хотите вернуть данные массива, то «Возвращаемое значение» не поможет, потому что копируется не содержимое массива, а только указатель на массив. В итоге вызывающая сторона может получить указатель на данные, которые уже вышли из области видимости. Для возврата массивов нужно использовать другие механизмы, например «Буфер, принадлежащий вызывающей стороне», или его аналог, выделяемый вызываемой стороной.

Помните, что если механизма возвращаемого значения достаточно, то всегда следует отдавать предпочтение этому, самому простому, варианту. Не нуж-

но применять более мощные, но и более сложные паттерны типа «Выходные параметры», «Агрегат», «Буфер, принадлежащий вызывающей стороне» или «Вызываемая сторона выделяет память».

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Этот паттерн можно встретить повсюду. Любая функция не типа `void` возвращает данные именно таким способом.
- В любой программе на С имеется функция `main`, которая возвращает значение вызывающей стороне (операционной системе).

Применение к сквозному примеру

Применить паттерн «Возвращаемое значение» было просто. Теперь у вас есть новый диагностический компонент в файле реализации, отличном от самого драйвера Ethernet, и этот компонент получает информацию от драйвера, как показано в следующем коде:

API драйвера Ethernet

```
/* Возвращает общее число полученных пакетов */
int ethernetDriverGetTotalReceivedPackets();

/* Возвращает общее число отправленных пакетов */
int ethernetDriverGetTotalSentPackets();
```

Код вызывающей стороны

```
void ethShow()
{
 int received_packets = ethernetDriverGetTotalReceivedPackets();
 int sent_packets = ethernetDriverGetTotalSentPackets();
 printf("получено пакетов: %i\n", received_packets);
 printf("отправлено пакетов: %i\n", sent_packets);
}
```

Этот код легко читается, а если понадобится добавить какую-то информацию, то нужно будет просто написать дополнительные функции для ее получения. Именно это мы и проделаем ниже. Вы хотите показать больше информации об отправленных пакетах, а именно: сколько пакетов отправлено успешно и сколько не удалось отправить. Первая попытка увенчалась таким кодом:

```
void ethShow()
{
 int received_packets = ethernetDriverGetTotalReceivedPackets();
 int total_sent_packets = ethernetDriverGetTotalSentPackets();
 int successfully_sent_packets = ethernetDriverGetSuccesscullySentPackets();
 int failed_sent_packets = ethernetDriverGetFailedPackets();
```

```
    printf("получено пакетов: %i\n", received_packets);
    printf("отправлено пакетов: %i\n", sent_packets);
    printf("успешно отправлено пакетов: %i\n", successfully_sent_packets);
    printf("не удалось отправить пакетов: %i\n", failed_sent_packets);
}
```

Написав этот код, вы внезапно осознаете, что иногда в противоречии с ожиданиями сумма `successfully_sent_packets` и `failed_sent_packets` оказывается больше, чем `total_sent_packets`. Это объясняется тем, что драйвер выполняется в отдельном потоке и между вызовами функций для получения информации он продолжает работать и обновлять сведения о пакетах. Так, например, если драйвер успешно отправил пакет между обращениями к `ethernetDriverGetTotalSentPackets` и `ethernetDriverGetSuccesscullySentPackets`, то информация, предоставленная пользователю, будет несогласованной.

Одно из возможных решений – гарантировать, что драйвер не работает, пока мы вызываем функции для получения информации о пакетах. Можно, например, использовать с этой целью Мьютекс или Семафор, но при решении такой простой задачи, как получение статистики пакетов, не хотелось бы заморачиваться с этим.

Гораздо меньше усилий требует возврат нескольких элементов информации из одного вызова функции в выходных параметрах.

Выходные параметры

Контекст

Вы хотите вернуть вызывающей стороне из своего компонента данные, представляющие различные элементы информации, которые могут изменяться между отдельными вызовами.

Проблема

С поддерживает возврат только одного значения из функции, что затрудняет возврат нескольких элементов информации.

Использование глобальных переменных для передачи данных, представляющих элементы информации, – плохое решение, потому что такой код был бы нереентерабельным и его было было небезопасно использовать в многопоточной программе. Кроме того, к глобальным переменным можно обратиться и модифицировать их из любой другой части кода, и из сигнатуры функции не видно, какие глобальные переменные используются для возврата данных. Таким образом, код с глобальными переменными было бы трудно понять и сопровождать. Использовать несколько функций для возврата разных значений тоже нехорошо, потому что возвращаемые данные связаны, поэтому разнесение их по разным вызовам функций сделает код менее понятным.

Поскольку элементы данных связаны, вызывающая сторона хочет получить согласованный снимок всех данных. Это проблематично, если несколько возвращаемых значений необходимо получить в многопоточном окружении, потому что данные могут изменяться во время работы. В таком случае нуж-

но гарантировать, что данные не изменяются между несколькими вызовами функций. Но вы не можете знать, закончила вызывающая сторона читать все данные или она захочет получить еще что-то. Поэтому невозможно гарантировать неизменность данных между обращениями вызывающей стороны. При использовании нескольких функций для получения взаимосвязанной информации неизвестно, в течение какого времени данные не должны изменяться. А значит, этот подход не может гарантировать, что вызывающая сторона получит непротиворечивую картину.

Наличие нескольких функций, возвращающих значения, нехорошо еще и потому, что для вычисления связанных элементов данных может потребоваться значительная подготовительная работа. Если, например, вы хотите вернуть домашний и мобильный телефон указанного лица из адресной книги и завели отдельные функции для их получения, то придется искать в адресной книге запись об этом лице при вызове каждой функции. Это непроизводительный расход времени процессора и ресурсов.

Решение

Возвращайте все данные в одном вызове функции, имитировав аргументы, передаваемые по ссылке, с помощью указателей.

Язык С не поддерживает возврат нескольких значений и не имеет встроенного механизма передачи аргументов по ссылке, но передачу по ссылке можно имитировать, как показано на рис. 4.3 и в последующем коде.

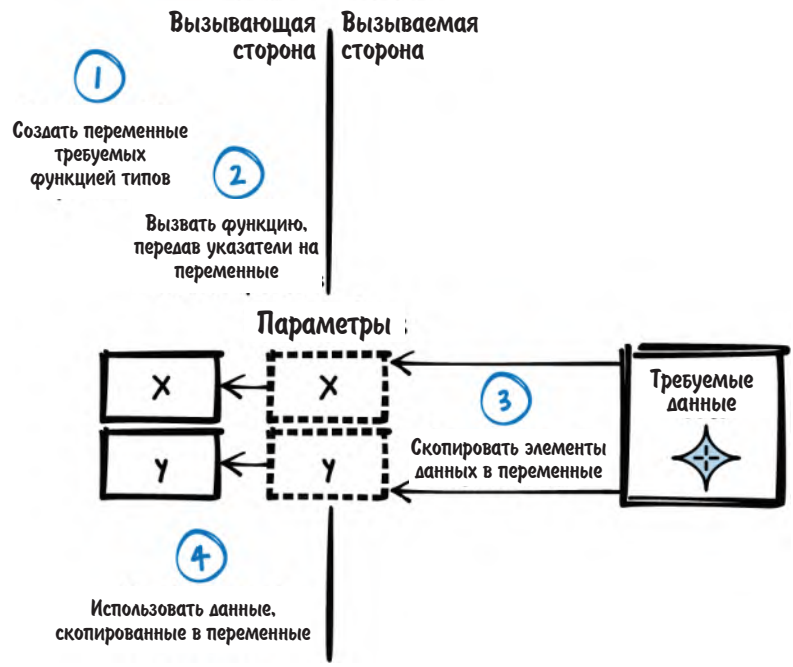


Рис. 4.3. Выходные параметры

Код вызывающей стороны

```
int x,y;
getData(&x,&y);
/* использовать x,y */
```

Код вызываемой стороны

```
void getData(int* x, int* y)
{
 *x = 42;
 *y = 78;
}
```

Заведите одну функцию, принимающую несколько указателей. В ее реализации разыменуйте указатели и скопируйте в переменные, на которые они указывают, данные, которые нужно вернуть вызывающей стороне. Внутри самой функции позаботьтесь о том, чтобы данные не изменялись во время копирования. Для этого можно воспользоваться взаимным исключением.

Многопоточное окружение

В современных системах работа в многопоточном окружении – обычное дело. Чтобы избежать проблем синхронизации в таком окружении лучше всего либо использовать неизменяемые данные, либо не разделять ни данные, ни функции (см. видео Кэвина Хенни «Thinking Outside the Synchronisation Quadrant» по адресу **https://oriel.ly/Sl1ta**). Но это возможно не всегда, и ситуация осложняется, поскольку нужно писать функции таким образом, чтобы их можно было вызывать из нескольких потоков в произвольном порядке и даже одновременно.

Это значит, что функции должны быть реентерабельными, т. е. работать правильно в условиях, когда их выполнение может быть в любой момент прервано и впоследствии возобновлено. Работая с разделяемыми ресурсами, например глобальными переменными, необходимо защищать их от одновременного доступа из других потоков. Это позволяют сделать такие примитивы синхронизации, как Мьютекс или Семафор.

В этой книге не рассматриваются примитивы синхронизации и способы работы с ними, но вы можете обратиться к книге Bruce P. Douglass «Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems» (Addison-Wesley, 2002); в ней же приводятся паттерны конкурентности и управления ресурсами для языка C.

Последствия

Теперь все данные, представляющие взаимосвязанные элементы информации, возвращаются в одном вызове функции, и их согласованность может быть обеспечена (например, путем копирования данных под защитой Мьютекса или Семафора). Функция реентерабельна и может быть использована в многопоточном окружении.

Для каждого дополнительного элемента данных функции передается отдельный указатель. Недостаток очевиден – если нужно вернуть много данных, то список параметров функции растет. Передача большого числа параметров одной функции дурно пахнет, потому что код становится нечитаемым. Именно поэтому несколько выходных параметров используются редко, а вместо этого, чтобы сделать код чище, взаимосвязанные элементы возвращаются с помощью «Агрегата».

Кроме того, для каждого элемента данных вызывающая сторона должна передать функции указатель, т. е. поместить его в стек. Если размер стека сильно ограничен, то это может оказаться проблемой.

Еще один недостаток выходных параметров – то, что, глядя только на сигнатуру функции, невозможно определить, какие параметры являются выходными. Вызывающая сторона может лишь догадываться, что указатель, возможно, является выходным параметром. Но с равным успехом он может быть и входным параметром. Поэтому в документации API должно быть четко описано, какие параметры предназначены для входа, а какие – для выхода.

Для простых скалярных типов C вызывающая сторона может просто передать указатель на переменную в качестве аргумента. Для интерпретации указателя вызванной функцией этого достаточно, потому что известен тип указателя. Чтобы вернуть составные данные, например массивы, нужно использовать один из паттернов – «Буфер, принадлежащий вызывающей стороне» или «Вызываемая сторона выделяет память» – и передавать дополнительную информацию о данных, например размер.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Функция `RegQueryInfoKey` в Windows возвращает информацию о разделе реестра с помощью выходных параметров. Вызывающая сторона предоставляет указатели типа `unsigned long`, а функция записывает в переменные, на которые они указывают, различную информацию, в том числе количество подразделов и размер значения.
- В Apple Cocoa API для программ на C используется дополнительный параметр `NSError` для передачи ошибок, имевших место при вызовах функций.
- Функция `userAuthenticate` в операционной системе реального времени VxWorks применяет возвращаемые значения для возврата информации, в данном случае – что пароль соответствовал имени пользователя. Кроме того, функция принимает выходной параметр, чтобы вернуть

идентификатор пользователя, ассоциированный с переданным именем.

Применение к сквозному примеру

Применив паттерн «Выходные параметры» к нашему примеру, мы получим такой код:

API драйвера Ethernet

```
/* Возвращает информацию о состоянии драйвера в выходных параметрах.
   total_sent_packets --> число отправленных пакетов (успешно и неудачно)
   successfully_sent_packets --> число успешно отправленных пакетов
   failed_sent_packets --> число неудачно отправленных пакетов */
void ethernetDriverGetStatistics(int* total_sent_packets,
     int* successfully_sent_packets, int* failed_sent_packets); ❶
```

❶ Для получения информации об отправленных пакетах нужно только одно обращение к драйверу Ethernet, и сам драйвер может гарантировать, что возвращенные данные согласованы.

Код вызывающей стороны

```
void ethShow()
{
 int total_sent_packets, successfully_sent_packets, failed_sent_packets;
 ethernetDriverGetStatistics(&total_sent_packets, &successfully_sent_packets,
                             &failed_sent_packets);
 printf("отправлено пакетов: %i\n", total_sent_packets);
 printf("успешно отправлено пакетов: %i\n", successfully_sent_packets);
 printf("не удалось отправить пакетов: %i\n", failed_sent_packets);

 int received_packets = ethernetDriverGetTotalReceivedPackets();
 printf("получено пакетов: %i\n", received_packets);
}
```

Вы подумываете о том, чтобы в той же функции получать еще и `received_packets`, но понимаете, что вызов становится все сложнее и сложнее. Одну функцию с тремя выходными параметрами и так уже трудно писать и читать. Да еще и порядок параметров при вызове легко перепутать. Так что добавление четвертого параметра код точно не улучшит.

Чтобы сделать код понятнее, можно использовать паттерн «Агрегат».

Агрегат

Контекст

Вы хотите вернуть несколько взаимосвязанных элементов данных из своего компонента вызывающей стороне, но эти компоненты могут изменяться между вызовами отдельных функций.

Проблема

Язык C поддерживает возврат только одного значения из функции, что затрудняет возврат нескольких элементов информации.

Использование глобальных переменных для передачи данных, представляющих элементы информации, – плохое решение, потому что такой код был бы нереентерабельным и его было было небезопасно использовать в многопоточной программе. Кроме того, к глобальным переменным можно обратиться и модифицировать их из любой другой части кода, и из сигнатуры функции не видно, какие глобальные переменные используются для возврата данных. Таким образом, код с глобальными переменными было бы трудно понять и сопровождать. Использовать несколько функций для возврата разных значений тоже нехорошо, потому что возвращаемые данные связаны, поэтому разнесение их по разным вызовам функций сделает код менее понятным.

Использовать одну функцию с большим числом выходных параметров также плохо, потому что если выходных параметров много, то их легко перепутать, да и код становится нечитаемым. К тому же хотелось бы показать, что параметры тесно связаны, и, быть может, один и тот же набор параметров даже используется для передачи другим функциям или получения информации от них. Если явно делать это с помощью параметров функций, то пришлось бы модифицировать каждую такую функцию при последующем добавлении новых параметров.

Поскольку элементы данных связаны, вызывающая сторона хочет получить согласованный снимок всех данных. Это проблематично, если несколько возвращаемых значений необходимо получить в многопоточном окружении, потому что данные могут изменяться во время работы. В таком случае нужно гарантировать, что данные не изменяются между несколькими вызовами функций. Но вы не можете знать, закончила вызывающая сторона читать все данные или она захочет получить еще что-то. Поэтому невозможно гарантировать неизменность данных между обращениями вызывающей стороны. При использовании нескольких функций для получения взаимосвязанной информации неизвестно, в течение какого времени данные не должны изменяться. А значит, этот подход не может гарантировать, что вызывающая сторона получит непротиворечивую картину.

Наличие нескольких функций, возвращающих значения, нехорошо еще и потому, что для вычисления связанных элементов данных может потребоваться значительная подготовительная работа. Если, например, вы хотите вернуть домашний и мобильный телефон указанного лица из адресной книги и завели отдельные функции для их получения, то придется искать в адресной книге запись об этом лице при вызове каждой функции. Это непроизводительный расход времени процессора и ресурсов.

Решение

Поместите все взаимосвязанные данные в новый тип. Определите тип «Агрегата», содержащий все данные, которые требуется разделить. Определите его в интерфейсе своего компонента, так чтобы вызывающая сторона могла непосредственно обращаться ко всем данным, хранящимся в агрегате.

Для реализации этой идеи определите в своем заголовочном файле структуру `struct`, членами которой являются все типы, возвращаемые из функции. В реализации функции скопируйте возвращаемые данные в члены `struct`, как показано на рис. 4.4. Обеспечьте неизменность данных во время копирования. Для этого можно воспользоваться взаимным исключением в форме Мьютекса или Семафора.

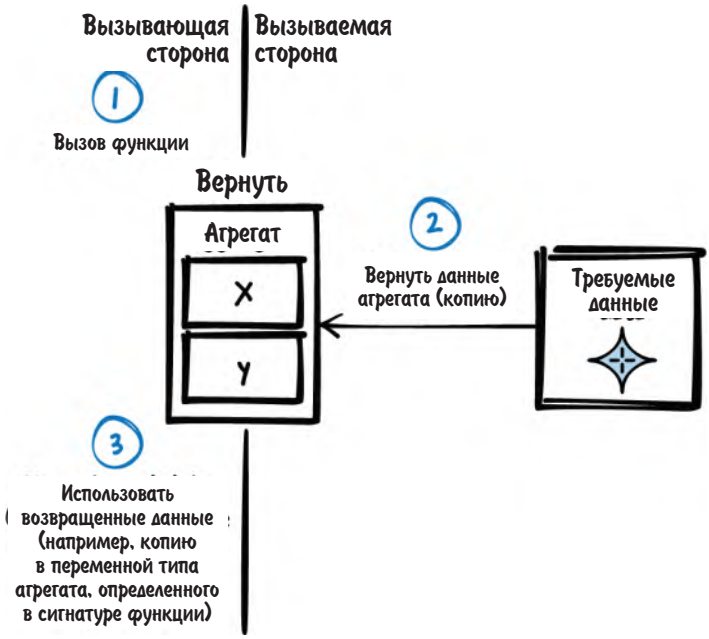


Рис. 4.4. Агрегат

Вернуть `struct` вызывающей стороне можно двумя способами:

- передать всю `struct` в виде возвращаемого значения. С позволяет возвращать значения не только встроенных, но и определенных пользователем типов, в частности структурного типа `struct`;
- передать указатель на `struct` в качестве выходного параметра. Однако при передаче указателя возникает проблема: кто предоставляет и кто владеет памятью, на которую он указывает. Этот вопрос решается паттернами «Буфер, принадлежащий вызываемой стороне» и «Вызываемая сторона выделяет память». Вместо того чтобы передавать указатель и предоставлять вызывающей стороне прямой доступ к «Агрегату», можно скрыть от нее `struct`, воспользовавшись «Описателем».

Ниже показан вариант с передачей всей структуры `struct`.

Код вызывающей стороны

```
struct AggregateInstance my_instance;
my_instance = getData();
```

```
/* использовать my_instance.x
   использовать my_instance.y, ... */
```

Код вызываемой стороны

```
struct AggregateInstance
{
 int x;
 int y;
};

struct AggregateInstance getData()
{
 struct AggregateInstance inst;
 /* заполнить inst.x и inst.y */
 return inst; ❶
}
```

❶ При возврате содержимое `inst` копируется (несмотря на то, что это `struct`), и вызывающая сторона имеет доступ к копии даже после того, как `inst` покидает область видимости.

Последствия

Теперь вызывающая сторона может получить несколько значений, представляющих взаимосвязанные элементы информации, с помощью одного вызова функции с агрегатом. Функция реентерабельна, ее можно безопасно вызывать в многопоточном окружении.

Это дает вызывающей стороне согласованный снимок всех взаимосвязанных элементов и оставляет ее код чистым, потому что нет нужды вызывать несколько функций или одну функцию со многими выходными параметрами.

Когда данные передаются между функциями без использования указателей, а только с помощью возвращаемых значений, все данные находятся в стеке. При передаче одной структуры 10 вложенным функциям эта структура оказывается в стеке в 10 экземплярах. Иногда это не является проблемой, а иногда является – особенно когда структура велика, и вы не хотите впустую растрачивать память, копируя ее в стек целиком каждый раз. Поэтому очень часто вместо передачи или возврата `struct` передается или возвращается указатель на нее.

Если передается указатель на `struct` или если `struct` содержит указатели, следует помнить, что С не будет делать за вас глубокое копирование. С копирует только значения указателей, но не объекты, на которые они указывают. Возможно, это вовсе не то, на что вы рассчитывали, поэтому не забывайте, что, как только в игру вступают указатели, вы должны озаботиться выделением и освобождением памяти. Этот вопрос решается паттернами «Буфер, принадлежащий вызывающей стороне» и «Вызываемая сторона выделяет память».

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В статье Uwe Zdun «Patterns of Argument Passing» (https://oreil.ly/VICgm) этот паттерн с примерами на C++ описывается под названием «Контекстный объект», а в книге Martin Fowler «Refactoring: Improving the Design of Existing Code» (Addison-Wesley, 1999) – под названием «Объект параметров».
- В игре NetHack атрибуты монстров хранятся в агрегатах, и предоставляется функция для получения этой информации.
- В текстовом редакторе sam структуры копируются при передаче их функциям и возврате из функций; это упрощает код.

Применение к сквозному примеру

При использовании «Агрегата» получается такой код.

API драйвера Ethernet

```
struct EthernetDriverStat{
 int received_packets;          /* число полученных пакетов */
 int total_sent_packets;        /* число отправленных пакетов (успешно и неудачно)*/
 int successfully_sent_packets; /* число успешно отправленных пакетов */
 int failed_sent_packets;       /* число неудачно отправленных пакетов  */
};

/* Возвращает статистику работы драйвера Ethernet */
struct EthernetDriverStat ethernetDriverGetStatistics();
```

Код вызывающей стороны

```
void ethShow()
{
 struct EthernetDriverStat eth_stat = ethernetDriverGetStatistics();
 printf("получено пакетов: %i\n", eth_stat.received_packets);
 printf("отправлено пакетов: %i\n", eth_stat.total_sent_packets);
 printf("успешно отправлено пакетов: %i\n", eth_stat.successfully_sent_packets);
 printf("не удалось отправить пакетов: %i\n", eth_stat.failed_sent_packets);
}
```

Теперь осталось всего одно обращение к драйверу Ethernet, и сам драйвер может гарантировать, что возвращенные данные согласованы. Кроме того, код выглядит чище, потому что все взаимосвязанные данные находятся в одной структуре.

Далее вы хотите представить пользователю больше информации о драйвере, а именно какому Ethernet-интерфейсу принадлежит статистика пакетов. Для этого нужно показать имя драйвера и его текстовое описание. То и другое находится в строке, хранящейся в компоненте драйвера. Строка длинная,

и ее точную длину вы не знаете. К счастью, эта строка не изменяется во время выполнения, так что можно обращаться к «Неизменяемому экземпляру».

Неизменяемый экземпляр

Контекст

Ваш компонент содержит много данных, и другой компонент хочет иметь доступ к этим данным.

Проблема

Вы хотите предоставить вызывающей стороне информацию, хранящуюся в большом блоке неизменяемых данных.

Копирование данных при каждом вызове было бы расточительным расходованием памяти, поэтому предоставление всех данных путем возврата агрегата или копирования их в выходные параметры не годится из-за ограничений на размер стека.

Просто возвращать указатель на такие данные обычно не стоит. Проблема в том, что, имея указатель, данные можно модифицировать, а коль скоро несколько вызывающих сторон читают и записывают одни и те же данные, необходимо создавать механизмы, гарантирующие согласованность и актуальность данных. По счастью, в вашей ситуации данные, которые требуется предоставить вызывающей стороне, фиксированы на этапе компиляции или на этапе загрузки программы и не изменяются во время выполнения.

Решение

Заведите объект (например, `struct`), который хранит разделяемые данные в статической памяти. Предоставляйте эти данные пользователям, нуждающимся в доступе к ним, но позаботьтесь о том, чтобы они не могли их модифицировать.

Записывайте данные, которые должны храниться в экземпляре, на этапе компиляции или загрузки и больше не изменяйте их в процессе выполнения программы. Данные можно зашить в программу или инициализировать при запуске программы (см. варианты инициализации в разделе «Программный модуль с глобальным состоянием» ниже, а варианты хранения – в разделе «Вечная память» выше). Как показано на рис. 4.5, даже если несколько вызывающих сторон (и несколько потоков) обращаются к экземпляру одновременно, им не нужно принимать друг друга во внимание, потому что экземпляр не изменяется и, следовательно, всегда находится в согласованном состоянии и содержит требуемую информацию.

Реализуйте функцию, возвращающую указатель на данные. Или можно даже сделать содержащую данные переменную глобальной и считать ее частью API, потому что данные все равно не изменяются во время выполнения. Но тем не менее акцессор чтения лучше, чем глобальные переменные, потому что так проще читать автономные тесты, а если в будущем поведение программы из-

менится (данные перестанут быть неизменяемыми), то не придется изменять интерфейс.

Рис. 4.5. Неизменяемый экземпляр

Чтобы гарантировать, что вызывающая сторона не сможет модифицировать данные, объявите возвращаемый указатель на данные с модификатором `const`, как показано ниже:

Код вызывающей стороны

```
const struct ImmutableInstance* my_instance;
my_instance = getData(); ❶
/* использовать my_instance->x,
   использовать my_instance->y, ... */
```

❶ Вызывающая сторона получает ссылку, но не становится владельцем памяти.

API вызываемой стороны

```
struct ImmutableInstance
{
 int x;
 int y;
};
```

Реализация вызываемой стороны

```
static struct ImmutableInstance inst = {12, 42};
const struct ImmutableInstance* getData()
{
 return &inst;
}
```

Последствия

Вызывающей стороне достаточно вызвать одну простую функцию, чтобы получить доступ даже к сложным и большим данным, и при этом ей не нужно думать, где эти данные хранятся. Вызывающей стороне не нужно предоставлять буферы, в которые будут помещены данные, не нужно очищать память и нет нужды думать о времени жизни данных – они просто существуют.

Вызывающая сторона может читать все данные по возвращенному указателю. Простая функция, получающая этот указатель, реентерабельна, ее безопасно использовать в многопоточном окружении. Сами данные тоже безопасно использовать в многопоточном окружении, потому что они не изменяются во время выполнения, и чтение данных из нескольких потоков не создает проблем.

Однако данные нельзя изменять во время выполнения без принятия дополнительных мер. Если необходимо, чтобы вызывающая сторона могла изменять данные, то можно реализовать что-то вроде копирования при записи. Если данные в принципе могут изменяться во время выполнения, то «Неизменяемый экземпляр» не выход, а для хранения сложных и больших данных следует использовать один из паттернов «Буфер, принадлежащий вызывающей стороне» или «Вызываемая сторона выделяет память».

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В статье Kevlin Henney «Patterns in Java: Patterns of Value» (https://oriel.ly/cVY9N) подробно описан похожий паттерн «Неизменяемый объект» и приведены примеры кода на C++.
- В игре NetHack атрибуты монстров хранятся в «Неизменяемом экземпляре», и для доступа к этой информации предоставляется функция.

Применение к сквозному примеру

Обычно возврат указателя на данные, хранящиеся внутри компонента, – дело непростое. Проблема в том, что если к этим данным обращаются несколько вызывающих сторон (и, быть может, записывают их), то простой указатель не выход, потому что неизвестно, действителен ли этот указатель до сих пор и согласованы ли хранящиеся по нему данные. Но в данном случае все нормально, потому что экземпляр неизменяемый. Имя и описание драйвера – информация, которая задается на этапе компиляции и в дальнейшем не изменяется. Поэтому мы можем просто запросить константный указатель на эти данные.

API драйвера Ethernet

```
struct EthernetDriverInfo{
  char name[64];
  char description[1024];
};
```

```
/* Возвращает имя и описание драйвера */
const struct EthernetDriverInfo* ethernetDriverGetInfo();
```

Код вызывающей стороны

```
void ethShow()
{
 struct EthernetDriverStat eth_stat = ethernetDriverGetStatistics();
 printf("получено пакетов: %i\n", eth_stat.received_packets);
 printf("отправлено пакетов: %i\n", eth_stat.total_sent_packets);
 printf("успешно отправлено пакетов: %i\n", eth_stat.successfully_sent_packets);
 printf("не удалось отправить пакетов: %i\n", eth_stat.failed_sent_packets);

 const struct EthernetDriverInfo* eth_info = ethernetDriverGetInfo();
 printf("Имя драйвера: %s\n", eth_info->name);
 printf("Описание драйвера: %s\n", eth_info->description);
}
```

Следующим шагом, помимо имени и описания Ethernet-интерфейса, мы хотели бы показать пользователю текущий IP-адрес и маску подсети. Они хранятся в виде строки внутри драйвера Ethernet. Оба значения могут изменяться во время выполнения, поэтому не получится вернуть указатель на «Неизменяемый экземпляр».

Можно было бы заставить драйвер упаковать эти строки в «Агрегат» и просто вернуть его (при возврате структуры хранящиеся в ней массивы копируются), но для больших объемов данных такое решение непопулярно, потому что потребляет много памяти в стеке. Обычно вместо этого применяются указатели. Такое решение, которое нас вполне устроило бы, описывается паттерном «Буфер, принадлежащий вызывающей стороне».

Буфер, принадлежащий вызывающей стороне

Контекст

Имеется много данных, которые вы хотите разделить между различными компонентами.

Проблема

Вы хотите предоставить вызывающей стороне сложные или большие данные известного размера, но эти данные не являются неизменяемыми (могут изменяться во время выполнения).

Поскольку данные изменяются во время выполнения (быть может, потому что вы предоставляете вызывающей стороне функции для записи данных), нельзя просто передать вызывающей стороне указатель на статические данные, как в случае «Неизменяемого экземпляра». В этом случае данные, прочитанные одной вызывающей стороной, могли бы оказаться несогласованными (частично перезаписанными), потому что другой поток одновременно записывает эти данные.

Просто скопировать все данные в «Агрегат» и передать его вызывающей стороне в виде возвращенного значения – не выход, потому что большие данные нельзя передать в ограниченном стеке.

Если вместо этого передавать указатель на «Агрегат», то проблемы с ограничением памяти в стеке не возникнет, но надо помнить, что С не занимается глубоким копированием, а возвращает только указатель. Вы должны гарантировать, что данные (хранящиеся в «Агрегате» или в массиве), на которые ведет указатель, остаются действительными после вызова функции. Например, нельзя хранить данные в автоматических переменных внутри функции и передавать на них указатель, потому что после возврата из функции эти переменные выйдут из области видимости.

Возникает вопрос, где же хранить данные. Следует уточнить, кто должен выделить необходимую память – вызывающая или вызываемая сторона – и кто будет отвечать за ее освобождение.

Решение

Потребуйте, чтобы вызывающая сторона предоставила буфер и его размер функции, возвращающей большие или сложные данные. Внутри функции скопируйте данные в буфер, если он достаточно велик.

Обеспечьте неизменность данные во время копирования. Для этого можно применить Мьютекс или Семафор. Тогда у вызывающей стороны будет моментальный снимок данных в буфере, причем она является единственным владельцем этого снимка и, следовательно, может обращаться к нему, не опасаясь несогласованности, даже если исходные данные в это время изменяются.

Вызывающая сторона может предоставить буфер и его размер в отдельных параметрах или упаковать их в «Агрегат» и передать указатель на него.

Поскольку буфер и размер предоставляет вызывающая сторона, она должна знать размер заранее. Чтобы вызывающая сторона могла узнать размер буфера, требуемый размер должен быть частью API. Это можно реализовать, определив размер в виде макроса или определив структуру `struct`, содержащую буфер требуемого размера, в API.

На рис. 4.6 и в последующем коде иллюстрируется идея «Буфера, принадлежащего вызывающей стороне».

Код вызывающей стороны

```
struct Buffer buffer;

getData(&buffer);
/* использовать buffer.data */
```

API вызываемой стороны

```
#define BUFFER_SIZE 256
struct Buffer
{
  char data[BUFFER_SIZE];
```

```
};

void getData(struct Buffer* buffer);
```

Реализация вызываемой стороны

```
void getData(struct Buffer* buffer)
{
 memcpy(buffer->data, some_data, BUFFER_SIZE);
}
```

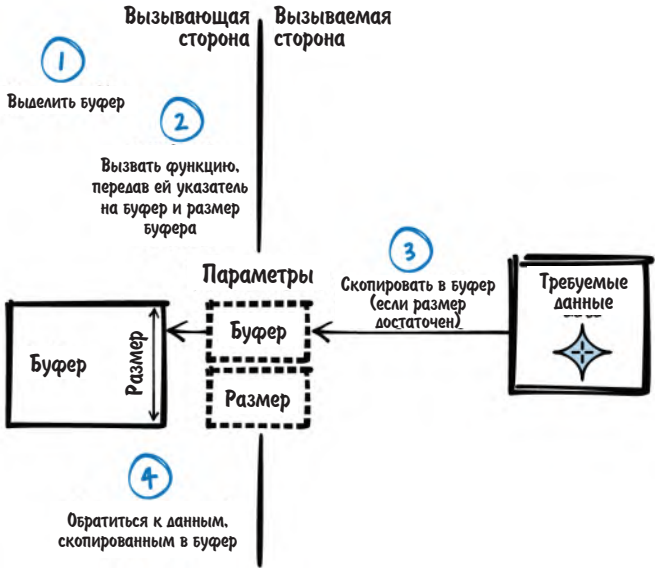


Рис. 4.6. Буфер, принадлежащий вызывающей стороне

Последствия

Большие и сложные данные можно согласованным образом предоставить вызывающей стороне с помощью одного вызова функции. Функция реентерабельна и может безопасно использоваться в многопоточном окружении. Кроме того, вызывающая сторона может безопасно обращаться к полученным данным, потому что является единственным владельцем буфера.

Вызывающая сторона предоставляет буфер ожидаемого размера и даже может решить, в какой памяти этот буфер разместить. Она может разместить буфер в стеке (см. паттерн «Сначала стек») и воспользоваться тем, что стек автоматически очищается после выхода переменной из области видимости. Или же она может выделить буфер в куче, чтобы самостоятельно определять время жизни переменной или не расходовать память в стеке. Кроме того, вызывающая функция могла бы иметь только ссылку на буфер, полученную от функции, которая ее вызвала. В таком случае этот буфер можно просто передать дальше и не выделять несколько буферов.

Занимающая много времени операция выделения и освобождения памяти не выполняется в момент вызова функции. Вызывающая сторона сама решает, когда выполнять эти операции, поэтому вызов функции оказывается более быстрым и детерминированным.

Из API абсолютно ясно, что буфером владеет вызывающая сторона. Именно она должна предоставить буфер, а затем очистить его. Если вызывающая сторона выделяла буфер, то она и отвечает за его последующее освобождение. Вызывающая сторона должна заранее знать размер буфера, а поскольку этот размер известен, то функция может безопасно работать с буфером. Но в некоторых случаях точный размер может быть неизвестен вызывающей стороне, и тогда лучше воспользоваться паттерном «Вызываемая сторона выделяет память».

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В игре NetHack этот паттерн используется для предоставления информации о сохраняемой игре компоненту, который затем фактически записывает текущее состояние игры на диск.
- В операционной системе B&R Automation Runtime этот паттерн используется в функции для получения IP-адреса.
- Функция `fgets` из стандартной библиотеки C читает данные из потока и сохраняет их в предоставленном вызывающей стороной буфере.

Применение к сквозному примеру

Теперь вы предоставляете буфер, принадлежащий вызывающей стороне, функции драйвера Ethernet, а функция копирует в этот буфер свои данные. Вы должны заранее знать размер буфера. Если требуется получить строку IP-адреса, то это не проблема, потому что размер такой строк фиксирован. Поэтому можно просто выделить буфер для IP-адреса в стеке и передать соответствующую автоматическую переменную драйверу. Альтернативно можно было бы выделить буфер в куче, но в данном случае этого не требуется, потому что размер IP-адреса известен и настолько мал, что спокойно поместится в стеке.

API драйвера Ethernet

```
struct IpAddress{
  char address[16];
  char subnet[16];
};

/* Сохраняет IP-адрес и маску подсети в параметре 'ip',
   который должен быть предоставлен вызывающей стороной */
void ethernetDriverGetIp(struct IpAddress* ip);
```

Код вызывающей стороны

```
void ethShow()
{
 struct EthernetDriverStat eth_stat = ethernetDriverGetStatistics();
 printf("получено пакетов: %i\n", eth_stat.received_packets);
 printf("отправлено пакетов: %i\n", eth_stat.total_sent_packets);
 printf("успешно отправлено пакетов: %i\n", eth_stat.successfully_sent_packets);
 printf("не удалось отправить пакетов: %i\n", eth_stat.failed_sent_packets);

 const struct EthernetDriverInfo* eth_info = ethernetDriverGetInfo();
 printf("Имя драйвера: %s\n", eth_info->name);
 printf("Описание драйвера: %s\n", eth_info->description);

 struct IpAddress ip;
 ethernetDriverGetIp(&ip);
 printf("IP-адрес: %s\n", ip.address);
}
```

Далее вы хотите развить диагностический компонент, так чтобы он еще печатал дамп последнего полученного пакета. Этот блок информации слишком велик для размещения в стеке, а поскольку размер Ethernet-пакетов переменный, мы не можем заранее знать, сколько места выделить для буфера. Поэтому «Буфер, принадлежащий вызывающей стороне» не годится.

Конечно, можно было бы просто завести функции `EthernetDriverGetPacketSize()` и `EthernetDriverGetPacket(buffer)`, но тут возникает уже знакомая проблема – нужно вызывать две функции, а между этими вызовами драйвер мог бы получить новый пакет, и данные стали бы несогласованными. К тому же это решение не элегантно, потому что нужно вызывать две функции для достижения одной цели. Гораздо проще воспользоваться паттерном «Вызываемая сторона выделяет память».

Вызываемая сторона выделяет память

Контекст

Имеется много данных, которые вы хотите разделить между различными компонентами.

Проблема

Вы хотите предоставить вызывающей стороне сложные или большие данные неизвестного размера, но эти данные не являются неизменяемыми (могут изменяться во время выполнения).

Поскольку данные изменяются во время выполнения (быть может, потому что вы предоставляете вызывающей стороне функции для записи данных), нельзя просто передать вызывающей стороне указатель на статические данные, как в случае «Неизменяемого экземпляра». В этом случае данные, прочи-

танные одной вызывающей стороной, могли бы оказаться несогласованными (частично перезаписанными), потому что другой поток одновременно записывает эти данные.

Просто скопировать все данные в «Агрегат» и передать его вызывающей стороне в виде возвращенного значения – не выход. С помощью возвращенного значения можно передать только данные известного размера, а поскольку данные велики, их нельзя передать в ограниченном стеке.

Если вместо этого передавать указатель на «Агрегат», то проблемы с ограничением памяти в стеке не возникнет, но надо помнить, что C не занимается глубоким копированием, а возвращает только указатель. Вы должны гарантировать, что данные (хранящиеся в «Агрегате» или в массиве), на которые ведет указатель, остаются действительными после вызова функции. Например, нельзя хранить данные в автоматических переменных внутри функции и передавать на них указатель, потому что после возврата из функции эти переменные выйдут из области видимости и будут стерты.

Возникает вопрос, где же хранить данные. Следует уточнить, кто должен выделить необходимую память – вызывающая или вызываемая сторона – и кто будет отвечать за ее освобождение.

Объем возвращаемых данных не фиксирован на этапе компиляции. Например, может понадобиться вернуть строку заранее неизвестного размера. Поэтому паттерн «Буфер, принадлежащий вызывающий стороне» не подходит, так как вызывающая сторона не знает размер буфера заранее. Вызывающая сторона могла бы предварительно спросить, какой размер буфера необходим (например, с помощью функции `getRequiredBufferSize()`), но это тоже непрактично, потому что для получения одного элемента данных пришлось бы делать несколько вызовов. Кроме того, данные, которые вы собираетесь предоставить, могут измениться между вызовами функций, а тогда вызывающая сторона предоставит буфер неправильного размера.

Решение

Выделите буфер нужного размера внутри функции, предоставляющей большие и сложные данные. Скопируйте данные в буфер и верните указатель на него.

Передайте указатели на буфер и его размер в выходных параметрах. После возврата из функции вызывающая сторона сможет работать с буфером, зная его размер, и будет его единственным владельцем. Вызывающая сторона определяет время жизни буфера и несет ответственность за его освобождение, как показано на рис. 4.7 и в последующем коде.

Код вызывающей стороны

```
char* buffer;
int size;
getData(&buffer, &size);
/* использовать буфер */
free(buffer);
```

Код вызываемой стороны

```
void getData(char** buffer, int* size)
{
 *size = data_size;
 *buffer = malloc(data_size);
 /* записать данные в буфер */ ❶
}
```

❶ Обеспечьте неизменность данных во время копирования в буфер. Это можно сделать, воспользовавшись Мьютексом или Семафором.

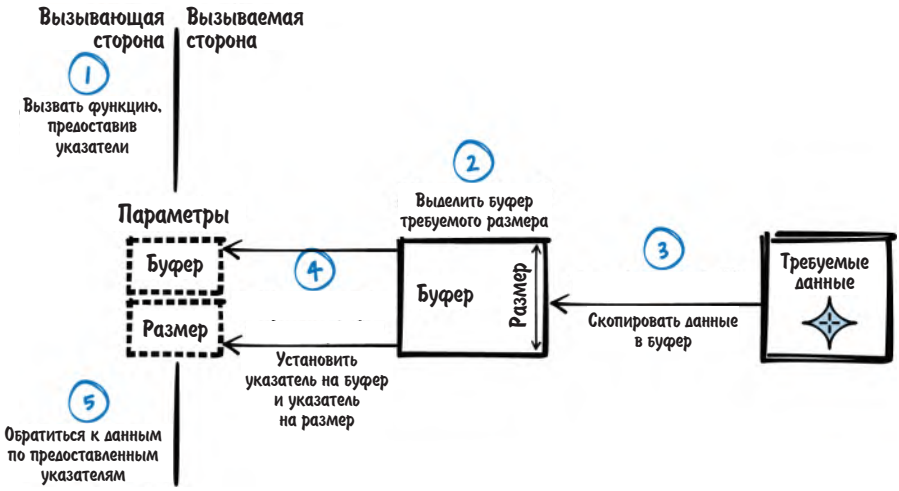


Рис. 4.7. Вызываемая сторона выделяет память

Альтернативно указатели на буфер и его размер можно поместить в «Агрегат» и вернуть его. Чтобы вызывающей стороне было понятно, что в агрегате хранится указатель, можно включить в API дополнительную функцию для его освобождения. Когда предоставляется такая функция, API становится очень похожим на «Описатель», что повышает гибкость, сохраняя при этом совместимость.

Неважно, как именно вызванная функция предоставляет буфер – в агрегате или в выходных параметрах, – вызывающая сторона должна ясно понимать, что именно она владеет буфером и отвечает за его освобождение. Владение должно быть документировано в API.

Последствия

Вызывающая сторона может получить буфер заранее неизвестного размера с помощью одного вызова функции. Эта функция реентерабельна и может безопасно использоваться в многопоточном окружении. Она предоставляет вызывающей стороне согласованную информацию о буфере и его размере. Зная размер, вызывающая сторона может безопасно обращаться к полученным данным. В таких буферах можно даже передавать строки, не завершаемые нулем.

Вызывающая сторона владеет буфером, определяет время его жизни и отвечает за его освобождение (как в случае «Описателя»). Одного взгляда на интерфейс должно быть достаточно, чтобы понять, что это задача именно вызывающей стороны. Один из способов добиться такого понимания – документировать порядок использования в API. Другой способ – предоставить явную функцию очистки, чтобы было очевидно, что имеется нечто, нуждающееся в очистке. У такой функции есть дополнительное преимущество – компонент, выделивший память, и освобождает ее. Это важно, когда два компонента компилируются разными компиляторами или работают на разных платформах – в таком случае функции освобождения и выделения памяти в разных компонентах могут различаться, и тогда просто необходимо, чтобы выделял и освобождал память один и тот же компонент.

Вызывающая сторона не может определить, какого вида память следует использовать для буфера (это было бы возможно при использовании буфера, принадлежащего вызывающей стороне). Теперь вызывающая сторона вынуждена пользоваться той памятью, которая выделена вызванной функцией.

Выделение памяти требует времени, поэтому, по сравнению с буфером, принадлежащим вызывающей стороне, функция становится более медленной и менее детерминированной.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Именно так ведет себя функция `malloc`. Она выделяет память и предоставляет ее вызывающей стороне.
- Функция `strdup` принимает на входе строку выделяет память для ее копии и возвращает копию.
- В Linux функция `getifaddrs` предоставляет информацию о сконфигурированных IP-адресах. Данные помещаются в буфер, выделенный этой функцией.
- В игре NetHack этот паттерн применяется для получения буферов.

Применение к сквозному примеру

Финальный код нашего диагностического компонента копирует данные пакета в буфер, выделенный вызываемой стороной.

API драйвера Ethernet

```
struct Packet
{
 char data[1500]; /* не более 1500 байтов на пакет */
 int size;        /* фактический размер данных в пакета */
};
```

```
/* Возвращается указатель на пакет. Эта память должна быть освобождена вызывающей
   стороной. */
struct Packet* ethernetDriverGetPacket();
```

Код вызывающей стороны

```
void ethShow()
{
 struct EthernetDriverStat eth_stat = ethernetDriverGetStatistics();
 printf("получено пакетов: %i\n", eth_stat.received_packets);
 printf("отправлено пакетов: %i\n", eth_stat.total_sent_packets);
 printf("успешно отправлено пакетов: %i\n", eth_stat.successfully_sent_packets);
 printf("не удалось отправить пакетов: %i\n", eth_stat.failed_sent_packets);

 const struct EthernetDriverInfo* eth_info = ethernetDriverGetInfo();
 printf("Имя драйвера: %s\n", eth_info->name);
 printf("Описание драйвера: %s\n", eth_info->description);

 struct IpAddress ip;
 ethernetDriverGetIp(&ip);
 printf("IP-адрес: %s\n", ip.address);

 struct Packet* packet = ethernetDriverGetPacket();
 printf("Дамп пакета:");
 fwrite(packet->data, 1, packet->size, stdout);
 free(packet);
}
```

В этой окончательной версии диагностического компонента мы видим все описанные способы получить информацию от другой функции. Смешение их всех в одном куске кода – не лучшая идея, потому что немного странно и сбивает с толку, когда один блок данных находится в стеке, а другой – в куче. Не стоит смешивать разные подходы к выделению буферов в одной функции. Лучше выберите какой-нибудь один подход, отвечающий всем вашим потребностям, и придерживайтесь его в рамках одной функции или компонента. Тогда ваш код будет единообразным, и понять его станет проще.

Однако если вам необходимо получить один блок данных от другого компонента и у вас есть выбор (из рассмотренных в этой главе паттернов), то всегда выбирайте самый простой вариант, чтобы сделать код проще. Например, если можно разместить буферы в стеке, так и поступайте, это позволит не освобождать буфер впоследствии.

Резюме

В этой главе были продемонстрированы разные способы возвращать данные из функции и работать с буферами в языке C. Самый простой – использовать паттерн «Возвращаемое значение» для возврата одного элемента данных, но если нужно использовать несколько взаимосвязанных элементов данных, то

следует выбрать паттерн «Выходные параметры» или, еще лучше, «Агрегат». Если подлежащие возврату данные не изменяются во время выполнения, то можно воспользоваться паттерном «Неизменяемый экземпляр». Ели данные возвращаются в буфере, то можно использовать паттерн «Буфер, принадлежащий вызывающей стороне», если размер буфера известен заранее, или паттерн «Вызываемая сторона выделяет память», если размер неизвестен.

Овладев паттернами из этой главы, программист на С будет иметь в своем арсенале средства и рекомендации по передаче данных между функциями и будет знать, какие существуют варианты возврата данных, выделения и освобождения буферов.

Что дальше

В следующей главе мы рассмотрим, как крупные программы организуются в программные модули и как эти модули управляют временем жизни и владением данными. Эти паттерны описывают строительные блоки, из которых конструируются более крупные куски кода на С.

Глава 5

Время жизни и владение данными

В процедурных языках программирования, к числу которых относится и C, нет встроенных объектно ориентированных механизмов. Это несколько усложняет жизнь, потому что рекомендации по проектированию рассчитаны прежде всего на объектно ориентированные программы (как, например, паттерны из книги «банды четырех»).

В этой главе обсуждаются паттерны, описывающие, как структурировать C-программу, включив в нее похожие на объекты элементы. Для таких объектоподобных элементов особое внимание уделяется тому, кто отвечает за их создание и уничтожение, – иными словами, вопросам времени жизни и владения. Эта тема особенно важна для языка C, потому что в нем нет автоматических деструкторов и механизма сборки мусора, поэтому нужно уделять особое внимание очистке ресурсов.

Но что такое «объектоподобный элемент» и в чем его значение для C? Термин *объект* корректно определен в объектно ориентированных языках программирования, но для других непонятно, что под ним понимается. Для C можно дать такое простое определение объекта:

> Объект – это именованная область памяти.
>
> (*Керниган и Ритчи*)

Обычно такой объект описывает набор взаимосвязанных данных, имеющий идентификатор и свойства и используемый для хранения представлений вещей, встречающихся в реальном мире. В объектно ориентированном программировании объект дополнительно обладает способностью к полиморфизму и наследованию. Объектоподобные элементы, рассматриваемые в этой книге, такими способностями не обладают, поэтому в дальнейшем мы не будем употреблять слово *объект*. А будем рассматривать объектоподобный элемент просто как экземпляр структуры данных и называть его *экземпляром*.

Такие экземпляры существуют не сами по себе, а комплектуются кодом, который дает возможность производить над ними операции. Обычно этот код

находится частично в заголовочных файлах, где определяется интерфейс, и частично в файлах реализации. В этой главе совокупность этого взаимосвязанного кода, аналогичного объектно ориентированному классу в том смысле, что он определяет, какие операции можно производить над экземпляром, мы будем называть *программным модулем*.

При программировании на C описанные выше экземпляры данных обычно реализуются как абстрактные типы данных (например, создают структуру `struct` и функции, обращающиеся к членам этой структуры). Примером такого экземпляра является структура `FILE` из стандартной библиотеки C, предназначенная для хранения такой информации, как указатель на файл и текущее положение в файле. Соответствующим стандартным модулем будет API в файле *stdio.h* и реализации функций типа `fopen` и `fclose`, которые предоставляют доступ к экземплярам `FILE`.

На рис. 5.1 приведен обзор паттернов, рассматриваемых в этой главе и связей между ними, а в табл. 5.1 – краткое описание этих паттернов.

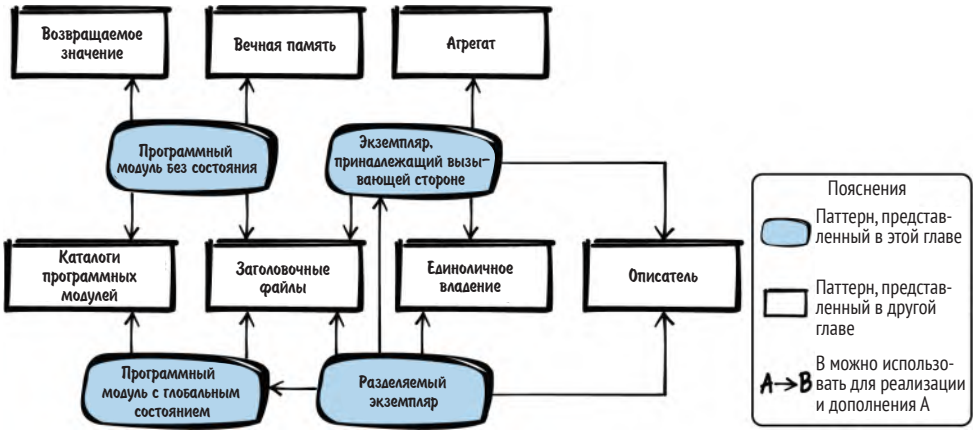


Рис. 5.1. Обзор паттернов, относящихся к времени жизни данных и владению ими

Таблица 5.1. Паттерны, относящиеся к времени жизни данных и владению ими

Название паттерна	Краткое описание
Программный модуль без состояния	Вы хотите предоставить логически связанную функциональность вызывающей стороне и максимально упростить ее использование. Поэтому делайте функции простыми и не храните информацию о состоянии в реализации. Поместите все связанные функции в один заголовочный файл и предоставьте вызывающей стороне этот интерфейс к своему программному модулю

Название паттерна	Краткое описание
Программный модуль с глобальным состоянием	Вы хотите структурировать логически связанный код, нуждающийся в общей информации о состоянии, и максимально упростить его использование. Поэтому заведите один глобальный экземпляр, чтобы все связанные функции могли разделять его. Поместите все функции, работающие с этим экземпляром, в один заголовочный файл и предоставьте вызывающей стороне этот интерфейс к своему программному модулю
Экземпляр, принадлежащий вызывающей стороне	Вы хотите предоставить нескольким вызывающим сторонам или потокам доступ к некоторой функциональности с помощью функций, которые зависят друг от друга. При этом взаимодействие вызывающей стороны с вашими функциями приводит к образованию информации о состоянии. Поэтому потребуйте, чтобы вызывающая сторона передавала вашим функциям экземпляр, используемый для хранения ресурса и информации о состоянии. Предоставьте явные функции для создания и уничтожения таких экземпляров, чтобы вызывающая сторона могла контролировать время их существования
Разделяемый экземпляр	Вы хотите предоставить нескольким вызывающим сторонам или потокам доступ к некоторой функциональности с помощью функций, которые зависят друг от друга. При этом взаимодействие вызывающей стороны с вашими функциями приводит к образованию информации о состоянии, которую все вызывающие стороны хотят разделять. Поэтому потребуйте, чтобы вызывающая сторона передавала вашим функциям экземпляр, используемый для хранения ресурса и информации о состоянии. Используйте один и тот же экземпляр для нескольких вызывающих сторон и оставьте владение этим экземпляром за своим программным модулем

Сквозной пример

В качестве сквозного примера мы в этой главе реализуем драйвер устройства для сетевой карты Ethernet. Сетевая карта устанавливается на машину с операционной системой, предоставляющей описанные в стандарте POSIX функции сокетов для отправки и получения данных по сети. Мы хотим построить некую абстракцию пользователя, потому что нам нужен более простой способ отправлять и получать данные, чем предлагают сокеты, и потому что мы хотим включить дополнительные средства в свой драйвер. Таким образом, мы хотим реализовать нечто, инкапсулирующее детали сокетов. Для этого начнем с простого паттерна «Программный модуль без состояния».

Программный модуль без состояния

Контекст

Вы хотите предоставить вызывающей стороне набор взаимосвязанных функций. Эти функции не применяются к общим разделяемым между ними данным и не требуют предварительной подготовки ресурсов, например памяти, которая должна быть инициализирована до вызова функций.

Проблема

Вы хотите предоставить вызывающей стороне логически связанную функциональность, сделав ее максимально простой в использовании.

Вы хотите, чтобы вызывающей стороне было легко получить доступ к вашей функциональности. Вызывающая сторона не должна заниматься инициализацией и очисткой в интересах предоставляемых функций, и ее не должны интересовать детали реализации.

Необязательно, чтобы функции были очень уж гибкими в части будущих изменений и поддерживали обратную совместимость, но они должны предоставлять удобную абстракцию для доступа к реализованной функциональности.

Существует много способов организовать заголовочные файлы и файлы реализации, и оценивать каждый из них всякий раз, как требуется реализовать какую-то функциональность, слишком долго и накладно.

Решение

Делайте функции простыми и не храните в реализации информацию о состоянии. Поместите все взаимосвязанные функции в один заголовочный файл и предоставьте вызывающей стороне этот интерфейс к своему программному модулю.

Между функциями не должно быть ни обмена информацией, ни разделения информации о внутреннем или внешнем состоянии. Это значит, что функции вычисляют результат или выполняют действие, которое не зависит от вызовов других функций из API (заголовочного файла) или предыдущих вызовов функции. Единственное взаимодействие имеет место между вызывающей и вызываемой функцией (например, в форме возвращаемых значений).

Если функция требует каких-либо ресурсов, например памяти из кучи, то их следует обрабатывать прозрачно для вызывающей стороны. Их необходимо захватить, неявно инициализировать перед использованием и освободить внутри вызова функции. Это позволяет вызывать функции абсолютно независимо друг от друга.

Тем не менее функции остаются взаимосвязанными и потому помещаются в один API. Взаимосвязанность означает, что обычно функции применяются вызывающей стороной совместно (принцип разделения интерфейсов), и если они изменяются, то это происходит по одной и той же причине (принцип общего замыкания). Эти принципы изложены в книге Robert C. Martin «Clean Architecture» (Prentice Hall, 2018)[1].

[1] *Мартин Роберт.* Чистая архитектура. Питер 2022.

Поместите объявления взаимосвязанных функций в один заголовочный файл, а реализации – в один или несколько файлов реализации, находящихся в одном и том же каталоге программного модуля. Функции взаимосвязаны, потому что логически нераздельны, но у них нет общего состояния, и они не влияют на состояние друг друга, поэтому нет необходимости разделять информацию между функциями с помощью глобальных переменных или инкапсулировать эту информацию, передавая экземпляры. Поэтому реализацию каждой функции можно поместить в отдельный файл.

Ниже приведен пример простого программного модуля без состояния.

Код вызывающей стороны

```
int result = sum(10, 20);
```

API (заголовочный файл)

```
/* Возвращает сумму обоих параметров */
int sum(int summand1, int summand2);
```

Реализация

```
int sum(int summand1, int summand2)
{
 /* вычислить результат, опираясь только на параметры и
    не требуя никакой информации о состоянии */
 return summand1 + summand2;
}
```

Вызывающая сторона вызывает `sum` и получает копию результата функции. Если вызвать эту функцию дважды с одними и теми же параметрами, то она вернет одинаковые результаты, потому что в программном модуле не запоминается никакой информации о состоянии. И никакой другой функции, которая хранила бы информацию о состоянии, в этом примере тоже не вызывается.

На рис. 5.2 иллюстрируется паттерн «Программный модуль без состояния».

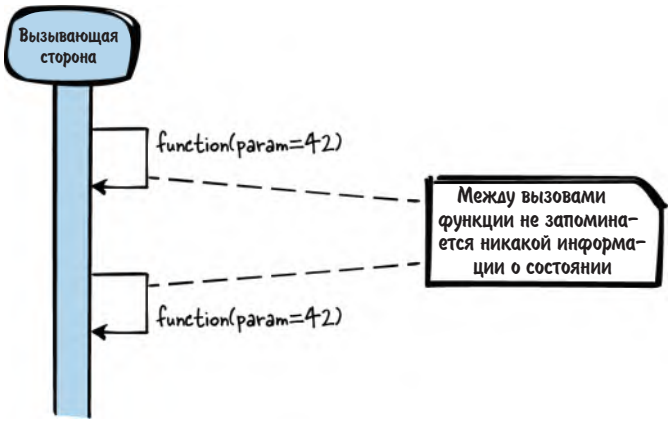


Рис. 5.2. Программный модуль без состояния

Последствия

Интерфейс очень простой, и вызывающей стороне не нужно думать об инициализации или очистке чего-либо в вашем программном модуле. Она просто вызывает функцию независимо от предшествующих вызовов и других частей программы, например других потоков, конкурентно обращающихся к модулю. Благодаря отсутствию информации о состоянии понять, что делает функция, гораздо проще.

Вызывающей стороне не нужно думать о владении, потому что и владеть-то нечем – у функции нет никакого состояния. Ресурсы, необходимые функции, захватываются и освобождаются внутри нее, для вызывающей стороны это прозрачно.

Но не любую функциональность можно предоставить с помощью такого простого интерфейса. Если функции, принадлежащие API, разделяют какую-либо информацию о состоянии или данные (например, одна выделяет ресурсы, необходимые другой), то нужен другой подход, скажем «Программный модуль с глобальным состоянием» или «Экземпляр, принадлежащий вызывающей стороне».

Известные примеры применения

Такой тип взаимосвязанных функций, собранных в один API, встречается всякий раз, как функция, принадлежащая API, не требует разделяемой информации или информации о состоянии. Следующие примеры демонстрируют применение этого паттерна.

- Функции `sin` и `cos` объявлены в одном и том же заголовочном файле *math.h*, и вычисленный результат зависит только от входных данных. Они не хранят никакой информации о состоянии, и вызов с одним и тем же входом порождает один и тот же выход.
- Функции `strcpy` и `strcat`, объявленные в файле *string.h*, не зависят друг от друга. Они не разделяют никакой информации, но взаимосвязаны и потому являются частью одного API.
- Заголовочный файл *VersionHelpers.h* в Windows предоставляет информацию о работающей версии Microsoft Windows. Функции типа `IsWindows7OrGreater` и `IsWindowsServer`, очевидно, взаимосвязаны, но не разделяют никакой информации и не зависят друг от друга.
- В заголовочном файле Linux *parser.h* объявлены функции `match_int` и `match_hex`. Они пытаются разобрать подстроку как целое или шестнадцатеричное значение соответственно. Функции не зависят друг от друга, но тем не менее принадлежат одному и тому же API.
- В исходном коде игры NetHack тоже встречается много примеров этого паттерна. Например, в заголовочный файл *vision.h* включены функции, вычисляющие, видит ли игрок определенные предметы на карте игры. Функции `couldsee(x,y)` и `cansee(x,y)` вычисляют, находится ли предмет на линии прямой видимости игрока и смотрит ли игрок на этот предмет. Функции не зависят друг от друга и не разделяют никакой информации.

- Паттерн «Заголовочные файлы» предлагает вариант этого паттерна с большим упором на гибкость API.
- Паттерн «Экземпляр, связанный с запросом» из книги Markus Voelter et al. «Remoting Patterns» (Wiley, 2007) объясняет, что сервер в распределенном объекте промежуточного слоя ПО должен активировать нового работника при каждом обращении и что, после того как работник обработает запрос, нужно вернуть результат и деактивировать работника. При таких обращениях к серверу не запоминается никакой информации о состоянии, и в этом отношении паттерн напоминает «Программный модуль без состояния» с тем отличием, что последний не имеет дела с удаленными объектами.

Применение к сквозному примеру

Код вашего первого драйвера устройства выглядит так:

API (заголовочный файл)

```
void sendByte(char data, char* destination_ip);
char receiveByte();
```

Реализация

```
void sendByte(char data, char* destination_ip)
{
 /* открыть сокет с ip адресата, отправить данные через этот сокет
    и закрыть его */
}

char receiveByte()
{
 /* открыть сокет для получения данных, подождать некоторое время
    и вернуть полученные данные */
}
```

Пользователь вашего драйвера Ethernet не должен возиться с деталями реализации, например как обращаться к сокетам, а может просто использовать предоставленный API. Обе входящие в состав этого API функции можно вызывать в любой момент независимо друг от друга, и вызывающая сторона может получить от функций данные, не думая о владении и освобождении ресурсов. API действительно простой, но при этом крайне ограниченный.

Далее вы собираетесь расширить функциональность своего драйвера. Вы хотите, чтобы пользователь мог видеть, работает ли связь через Ethernet, а значит, хотите предоставить статистику отправленных и принятых байтов. Простой «Программный модуль без состояния» не позволит это сделать, потому что не выделена память для хранения информации о состоянии между вызовами функций. Чтобы решить задачу, потребуется «Программный модуль с глобальным состоянием».

Программный модуль с глобальным состоянием

Контекст

Вы хотите предоставить вызывающей стороне набор взаимосвязанных функций. Эти функции работают с общими разделяемыми между ними данными и могут требовать предварительной подготовки ресурсов, например памяти, которая должна быть инициализирована до вызова функций. Однако функции не требуют никакой зависящей от вызывающей стороны информации о состоянии.

Проблема

Вы хотите структурировать логически связанный код, требующий общей информации о состоянии, и сделать эту функциональность максимально простой в использовании вызывающей стороной.

Вы хотите, чтобы вызывающей стороне было легко получить доступ к вашей функциональности. Вызывающая сторона не должна заниматься инициализацией и очисткой в интересах предоставляемых функций, и ее не должны интересовать детали реализации. Вызывающей стороне необязательно понимать, что функции обращаются к общим данным.

Необязательно, чтобы функции были очень уж гибкими в части будущих изменений и поддерживали обратную совместимость, но они должны предоставлять удобную абстракцию для доступа к реализованной функциональности.

Решение

Заведите один глобальный экземпляр, чтобы реализации взаимосвязанных функций могли разделять общие ресурсы. Поместите все функции, работающие с этим экземпляром, в один заголовочный файл и предоставьте вызывающей стороне этот интерфейс к своему программному модулю.

Поместите объявление функций своего программного модуля в заголовочный файл, а реализации – в один файл реализации в каталоге программного модуля. В этом файле реализации заведите глобальный экземпляр (статическую на уровне файла структуру `struct` или несколько статических переменных, см. паттерн «Вечная память»). Тогда реализации функций смогут обращаться к этим разделяемым ресурсам; это похоже на механизм работы с закрытыми переменными-членами в объектно ориентированных языках программирования.

Инициализация ресурсов прозрачно производится программным модулем, он же управляет временем их жизни, так что оно и не зависит от времени жизни вызывающих сторон. Если ресурсы вообще нуждаются в инициализации, то это можно сделать как в момент запуска, так и отложенно – непосредственно перед первым использованием.

Из сигнатуры вызова вызывающая сторона не видит, что функции работают с общими ресурсами, так что это нужно явно документировать. Внутри программного модуля доступ к глобальным на уровне файла ресурсам, возможно,

необходимо защищать примитивами синхронизации, например Мьютексом, чтобы к модулю можно было обращаться из разных потоков. Делайте синхронизацию частью реализации своих функций, чтобы вызывающей стороне не нужно было думать об этом.

Ниже приведен пример простого «Программного модуля с глобальным состоянием».

Код вызывающей стороны

```
int result;
result = addNext(10);
result = addNext(20);
```

API (заголовочный файл)

```
/* Прибавляет параметр 'value' к сумме, накопленной
   при выполнении предыдущих вызовов этой функции . */
int addNext(int value);
```

Реализация

```
static int sum = 0;
int addNext(int value)
{
 /* вычисление результата, зависящего от параметра и
    состояния, образовавшегося после предыдущих вызовов функции */
 sum = sum + value;
 return sum;
}
```

Вызывающая сторона вызывает `addNext` и получает копию результата. При вызове функции дважды с одними и тем же параметрами результаты могут различаться, потому что функция хранит информацию о состоянии.

На рис. 5.3 схематически изображена работа «Программного модуля с глобальным состоянием».

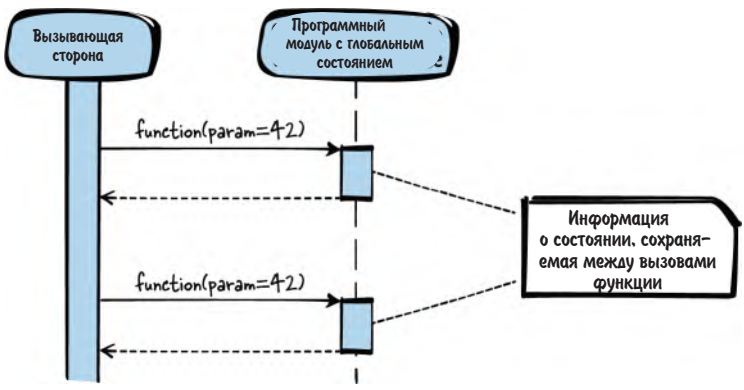


Рис. 5.3. Программный модуль с глобальным состоянием

Последствия

Теперь ваши функции могут разделять информацию или ресурсы, хотя от вызывающей стороны не требуется передавать параметры, содержащие эту информацию, и она не несет ответственности за выделение и освобождение ресурсов. Чтобы добиться такого разделения информации в программном модуле, вы реализовали С-версию паттерна «Одиночка» (Singleton). Но берегитесь этого паттерна – к нему есть много претензий, и иногда его даже называют антипаттерном.

Тем не менее в С такие программные модули с глобальным состоянием широко распространены, поскольку написать ключевое слово `static` перед переменной очень легко, а стоит это сделать, как вы тут же получаете «Одиночку». В некоторых случаях это нормально. Если файлы реализации короткие, то переменные, глобальные на уровне файла, аналогичны закрытым переменным-членам в объектно ориентированном программировании. Если функция не требует информации о состоянии или не работает в многопоточном окружении, то все, возможно, и обойдется. Но если многопоточность или информация о состоянии налицо, а ваш файл реализации становится все длиннее и длиннее, то вы «попали», и «Программный модуль с глобальным состоянием» перестает быть хорошим решением.

Если «Программный модуль с глобальным состоянием» нуждается в инициализации, то это можно сделать на этапе общей инициализации, например при запуске программы, или отложить инициализацию до момента первого использования ресурсов. Однако у отложенного подхода есть недостаток – продолжительность вызовов оказывается переменной, потому что при первом вызове исполняется дополнительный код инициализации. В любом случае захват ресурса прозрачен для вызывающей стороны. Ресурсы принадлежат вашему программному модулю, поэтому вызывающая сторона не обременена владением и не должна явно захватывать и освобождать ресурсы.

Однако не любую функциональность можно предоставить с помощью такого простого интерфейса. Если входящие в состав API функции разделяют информацию, зависящую от вызывающей стороны, то необходим другой подход, например паттерн «Экземпляр, принадлежащий вызывающей стороне».

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Функция `strtok`, объявленная в файле *string.h*, разбивает строку на лексемы. При каждом вызове функции возвращается следующая лексема. Для хранения информации о состоянии, необходимой для решения о том, какую лексему возвращать следующей, функция пользуется статическими переменными.
- Доверенный платформенный модуль (Trusted Platform Module – TPM) позволяет хранить хеш-значения загруженных программ. Соответствующая функция в коде TPM-Emulator v0.7 пользуется статическими переменными для хранения хеш-значений.

- Библиотека `math` использует состояние для генерирования случайных чисел. При каждом вызове функции `rand` вычисляется новое псевдослучайное число, зависящее от предыдущего вызова `rand`. Предварительно нужно вызвать функцию `srand`, чтобы задать начальное значение (статическое) для генератора псевдослучайных чисел, вызываемого `rand`.
- Паттерн «Неизменяемый экземпляр» можно рассматривать как частный случай «Программного модуля с глобальным состоянием», в котором экземпляр не изменяется во время выполнения.
- В игре NetHack информация о предметах (мечах, щитах) хранится в статическом списке, определенном на этапе компиляции, и предоставляются функции для доступа к этой разделяемой информации.
- Паттерн «Статический экземпляр» из книги Markus Voelter et al. «Remoting Patterns» (Wiley, 2007) рекомендует предоставлять удаленные объекты, время жизни которых не зависит от времени жизни вызывающей стороны. Удаленные объекты можно, например, инициализировать на этапе запуска программы, а затем предоставлять вызывающей стороне по запросу. Программный модуль с глобальным состоянием предлагает ту же идею заведения статических данных, но не предполагает наличия нескольких экземпляров для разных вызывающих сторон.

Применение к сквозному примеру

Теперь код нашего драйвера Ethernet принимает описанный ниже вид.

API (заголовочный файл)

```
void sendByte(char data, char* destination_ip);
char receiveByte();
int getNumberOfSentBytes();
int getNumberOfReceivedBytes();
```

Реализация

```
static int number_of_sent_bytes = 0;
static int number_of_received_bytes = 0;

void sendByte(char data, char* destination_ip)
{
 number_of_sent_bytes++;
 /* работа с сокетами */
}

char receiveByte()
{
 number_of_received_bytes++;
 /* работа с сокетами */
}

int getNumberOfSentBytes()
```

```
{
 return number_of_sent_bytes;
}

int getNumberOfReceivedBytes()
{
 return number_of_received_bytes;
}
```

API очень похож на API «Программного модуля без состояния», но теперь за ним кроется возможность сохранять между вызовами функций информацию, необходимую для подсчета отправленных и полученных байтов. Если у этого API есть только один пользователь (один поток), то все хорошо. Но когда потоков несколько, использование статических переменных с неизбежностью ведет к возникновению состояний гонки, если только не реализован какой-то механизм взаимного исключения при доступе к статическим переменным.

Что ж, теперь вы хотите сделать драйвер Ethernet более эффективным и отправлять больше данных. Для этого можно было бы часто вызывать функцию `sendByte`, но сейчас реализация устроена так, что при каждом вызове `sendByte` устанавливается соединение через сокет, отправляются данные и соединение закрывается. На установление и закрытие соединения уходит большая часть времени.

Это очень неэффективно, и вы предпочли бы открыть соединение через сокет один раз, затем отправить все данные, вызывая `sendByte` несколько раз, и в конце закрыть соединение. Однако теперь `sendByte` нуждается в этапе подготовки и этапе очистки. Это состояние нельзя хранить в «Программном модуле с глобальным состоянием», потому что, как только вызывающих сторон становится несколько (т. е. больше одного потока), возникает проблема одновременной отправки данных из нескольких потоков, быть может, даже разным адресатам.

Чтобы решить проблему, нужно предоставить каждой вызывающей стороне свой «Экземпляр, принадлежащий вызывающей стороне».

Экземпляр, принадлежащий вызывающей стороне

Контекст

Вы хотите предоставить вызывающей стороне набор взаимосвязанных функций. Эти функции работают с общими разделяемыми между ними данными и могут требовать предварительной подготовки ресурсов, например памяти, которая должна быть инициализирована до вызова функций. Кроме того, функции разделяют зависящую от вызывающей стороны информацию о состоянии.

Проблема

Вы хотите предоставить нескольким вызывающим сторонам или потокам доступ к функциональности, реализованной взаимозависимыми функциями, и при взаимодействии функций с вызывающей стороной образуется информация о состоянии.

Быть может, одна функция должна быть вызвана раньше другой, потому что она влияет на хранимое в программном модуле состояние, необходимое второй функции. Решить задачу можно с помощью паттерна «Программный модуль с глобальным состоянием», но только в том случае, когда имеется всего одна вызывающая сторона. В многопоточном окружении не существует одного центрального программного модуля, в котором хранилась бы вся зависящая от вызывающей стороны информация о состоянии.

И тем не менее вы хотите скрыть детали реализации от вызывающей стороны, сделав доступ к своей функциональности максимально простым. Следует четко определить, отвечает ли вызывающая сторона за выделение и освобождение ресурсов.

Решение

Потребуйте, чтобы вызывающая сторона передавала всем вашим функциям экземпляр, в котором хранятся ресурсы и информация о состоянии. Предоставьте явные функции для создания и уничтожения таких экземпляров, чтобы вызывающая сторона сама могла определять время их жизни.

Чтобы реализовать такой экземпляр, к которому можно было бы обращаться из нескольких функций, передавайте указатель на `struct` всем функциям, которые нуждаются в разделении ресурсов или информации о состоянии. Функции могут использовать члены `struct`, аналогичные закрытым переменным-членам в объектно ориентированных языках, для сохранения и чтения ресурсов или информации о состоянии.

Структуру `struct` можно объявить в API, чтобы вызывающей стороне было удобно обращаться к ее членам непосредственно. Или же саму `struct` можно объявить в файле реализации, а в API объявить только указатель на нее (как рекомендует паттерн «Описатель»). Тогда вызывающая сторона не будет знать членов `struct` (они аналогичны закрытым переменным-членам) и сможет работать со `struct` только посредством функций.

Поскольку к экземпляру могут обращаться несколько функций, и вы не знаете, когда вызывающая сторона закончит их вызывать, время жизни экземпляра должно определяться вызывающей стороной. Поэтому документируйте, что экземпляром владеет вызывающая сторона, и предоставьте явные функции для создания и уничтожения экземпляра. Между вызывающей стороной и экземпляром существует связь типа агрегирования.

Агрегирование и ассоциация

Если один экземпляр семантически связан с другим, то эти экземпляры ассоциированы. Более сильной разновидностью ассоциации является агрегирование, когда один экземпляр владеет другим.

Ниже приведен пример «Экземпляра, принадлежащего вызывающей стороне».

Код вызывающей стороны

```
struct INSTANCE* inst;
inst = createInstance();
operateOnInstance(inst);
/* доступ к inst->x или inst->y */
destroyInstance(inst);
```

API (заголовочный файл)

```
struct INSTANCE
{
  int x;
  int y;
};

/* Создает экземпляр, необходимый для работы с функцией 'operateOnInstance' */
struct INSTANCE* createInstance();

/* Работает с данными, хранящимися в экземпляре */
void operateOnInstance(struct INSTANCE* inst);

/* Очищает экземпляр, созданный функцией 'createInstance' */
void destroyInstance(struct INSTANCE* inst);
```

Реализация

```
struct INSTANCE* createInstance()
{
 struct INSTANCE* inst;
 inst = malloc(sizeof(struct INSTANCE));
 return inst;
}

void operateOnInstance(struct INSTANCE* inst)
{
 /* работа с inst->x и inst->y */
}

void destroyInstance(struct INSTANCE* inst)
{
 free(inst);
}
```

Функция `operateOnInstance` работает с ресурсами, созданными ранее функцией `createInstance`. Ресурс или информация о состоянии передается между функциями вызывающей стороной, которая должна передавать параметр

`INSTANCE` при каждом вызове функции, а затем освободить ресурсы, вызвав `destroyInstance`.

На рис. 5.4 схематически изображена работа «Экземпляра, принадлежащего вызывающей стороне».

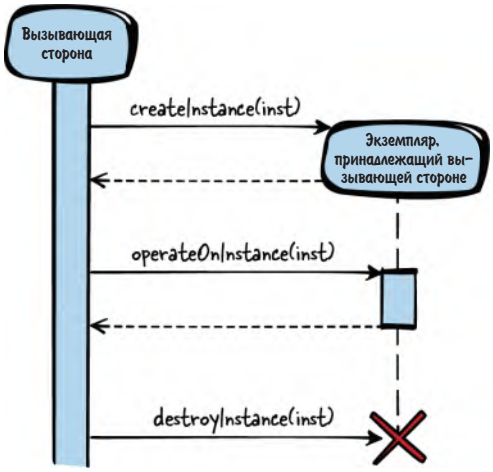


Рис. 5.4. Экземпляр, принадлежащий вызывающей стороне

Последствия

Функции, являющиеся частью вашего API, стали мощнее, потому что теперь могут разделять информацию о состоянии и работать с разделяемыми данными, оставаясь доступными нескольким вызывающим сторонам (т. е. нескольким потокам). У каждого экземпляра, принадлежащего вызывающей стороне, имеются собственные закрытые переменные, и, даже если создается несколько таких экземпляров (например, в нескольких потоках), проблемы не возникает.

Но для достижения такого результата пришлось усложнить API. Понадобилось ввести явные функции `create()` и `destroy()` для управления временем жизни экземпляра, потому что в C нет конструкторов и деструкторов. Это заметно затрудняет работу с экземплярами, потому что вызывающая сторона получает их во владение и отвечает за очистку. Поскольку это необходимо делать вручную путем вызова `destroy()`, а не автоматически с помощью деструктора, как в объектно ориентированных языках программирования, мы получаем распространенный источник утечек памяти. Эта проблема решается паттерном «Объектная обработка ошибок», который рекомендует заводить на вызывающей стороне специальную функцию очистки, чтобы сделать эту задачу более явной.

Кроме того, по сравнению с «Программным модулем без состояния», вызов каждой функции становится чуть более сложным. Каждая функция принимает дополнительный параметр – ссылку на экземпляр, и функции уже нельзя вызывать в произвольном порядке – вызывающая сторона должна знать, какую функцию следует вызывать первой. Это проясняется с помощью сигнатур функций.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Примером использования «Экземпляра, принадлежащего вызывающей стороне», является дважды связанный список в библиотеке `glibc`. Вызывающая сторона создает список, вызывая функцию `g_list_alloc`, после чего может вставлять в него элементы с помощью функции `g_list_insert`. Закончив работу со списком, вызывающая сторона должна очистить его, вызвав `g_list_free`.
- Этот паттерн упоминается в статье Robert Strandh «Modular C» (https://oreil.ly/UVodl), где описано, как писать модульные программы на C. В статье подчеркивается важность выявления абстрактных типов данных, операции над которыми производятся с помощью функций, в приложении.
- Windows API для создания отдельных меню в полосе меню включает функцию создания экземпляра меню (`CreateMenu`), функции для работы с меню (например, `InsertMenuItem`) и функцию уничтожения экземпляра меню (`DestroyMenu`). Все эти функции принимают параметр, в котором передается «Описатель» экземпляра меню.
- В программном модуле Apache для обработки HTTP-запросов имеются функции для создания всей необходимой информации о запросе (`ap_sub_req_lookup_uri`), ее обработки (`ap_run_sub_req`) и уничтожения (`ap_destroy_sub_req`). Эти функции принимают указатель на структуру с экземпляром запроса.
- В игре NetHack экземпляр `struct` используется для представления монстров, и предоставляются функции для создания и уничтожения монстра. NetHack также предоставляет функцию для получения информации о монстрах (`is_starting_pet`, `is_vampshifter`).
- Паттерн «Зависящий от клиента экземпляр» из книги Markus Voelter et al. «Remoting Patterns» (Wiley, 2007) рекомендует в ПО промежуточного уровня для управления распределенными объектами предоставлять удаленные объекты, время жизни которых контролируется клиентами. Сервер создает новые экземпляры для клиентов, а клиент работает с экземпляром, передает его между функциями и уничтожает.

Применение к сквозному примеру

Теперь код драйвера Ethernet принимает следующий вид.

API (заголовочный файл)

```
struct Sender
{
  char destination_ip[16];
  int socket;
};
```

```
struct Sender* createSender(char* destination_ip);
void sendByte(struct Sender* s, char data);
void destroySender(struct Sender* s);
```

Реализация

```
struct Sender* createSender(char* destination_ip)
{
 struct Sender* s = malloc(sizeof(struct Sender));
 /* создать сокет для destination_ip и сохранить его в Sender s */
 return s;
}

void sendByte(struct Sender* s, char data)
{
 number_of_sent_bytes++;
 /* отправить через сокет данные, хранящиеся в Sender s */
}

void destroySender(struct Sender* s)
{
 /* закрыть сокет, хранящийся в Sender s */
 free(s);
}
```

Вызывающая сторона сначала может создать отправителя, затем отправить сразу все данные, после чего уничтожить отправителя. Таким образом, вызывающая сторона может гарантировать, что соединение не придется устанавливать заново в каждом вызове `sendByte()`. Вызывающая сторона владеет созданным отправителем, полностью контролирует время его жизни и отвечает за его очистку.

Код вызывающей стороны

```
struct Sender* s = createSender("192.168.0.1");
char* dataToSend = "Hello World!";
char* pointer = dataToSend;
while(*pointer != '\0')
{
 sendByte(s, *pointer);
 pointer++;
}
destroySender(s);
```

Далее предположим, что вы не единственный пользователь этого API. Его могут использовать несколько потоков. При условии, что один поток создает отправителя для отправки IP-адресу X, а другой – отправителя для отправки Y, все будет хорошо, и драйвер Ethernet создаст два независимых сокета для двух потоков.

Но допустим, что оба потока хотят отправлять данные одному и тому же получателю. Теперь драйвер Ethernet в затруднении, потому что на одном порту можно открыть только один сокет для каждого конечного IP-адреса. Решение проблемы – не позволять двум разным потоком отправлять данные одному и тому же получателю, – второй поток просто получит ошибку при попытке создать отправителя. Однако же можно и разрешить двум потокам отправлять данные одному адресату.

Для этого нужно просто создать «Разделяемый экземпляр».

Разделяемый экземпляр

Контекст

Вы хотите предоставить вызывающей стороне набор взаимосвязанных функций. Эти функции работают с общими разделяемыми между ними данными и могут требовать предварительной подготовки ресурсов, например памяти, которая должна быть инициализирована до вызова функций. Существует несколько контекстов, в которых будет вызываться эта функциональность, и эти контексты разделяются между вызывающими сторонами.

Проблема

Вы хотите предоставить нескольким вызывающим сторонам или потокам доступ к функциональности, реализованной взаимозависимыми функциями, и при взаимодействии функций с вызывающей стороной образуется информация о состоянии, которая должна разделяться между вызывающим сторонами.

Хранить информацию о состоянии в «Программном модуле с глобальным состоянием» – не выход из-за наличия нескольких вызывающих сторон, которым нужна разная информация о состоянии. Хранить информацию о состоянии в «Экземпляре, принадлежащем вызывающей стороне», тоже не годится, потому что некоторые вызывающие стороны могут оперировать с одним и тем же экземпляром или потому что вы не хотите создавать новые экземпляры для каждой вызывающей стороны, стремясь сэкономить ресурсы.

И тем не менее вы хотите скрыть детали реализации от вызывающей стороны, сделав доступ к своей функциональности максимально простым. Следует четко определить, отвечает ли вызывающая сторона за выделение и освобождение ресурсов.

Решение

Потребуйте, чтобы вызывающая сторона передавала всем вашим функциям экземпляр, в котором хранятся ресурсы и информация о состоянии. Используйте один и тот же экземпляр для нескольких вызывающих сторон и оставьте владение этим экземпляром за своим программным модулем.

Как и в случае «Экземпляра, принадлежащего вызывающей стороне», предоставьте вызывающей стороне указатель на `struct` или «Описатель», который

та сможет передавать вашим функциям. Теперь при создании экземпляра вызывающая сторона должна будет также передать идентификатор (например, уникальное имя), определяющий вид создаваемого экземпляра. Зная идентификатор, вы можете проверить, существует ли уже такой экземпляр. Если да, то новый экземпляр не создается, а вместо него возвращается указатель на `struct` или «Описатель» уже созданного экземпляра, который был возвращен другим вызывающим сторонам.

Чтобы узнать, существует ли уже экземпляр, необходимо хранить список уже созданных экземпляров в программном модуле. Для хранения списка можно, например, реализовать паттерн «Программный модуль с глобальным состоянием». Кроме того, вне зависимости от того, был создан экземпляр или нет, можно хранить информацию о том, кто в настоящее время к каким экземплярам обращается или, по крайней мере, сколько вызывающих сторон в настоящее время обращаются к экземпляру. Эта дополнительная информация необходима, потому что в тот момент, когда у экземпляра не останется пользователей, вы должны будете уничтожить его, так как являетесь его единоличным владельцем.

Нужно также проверить, могут ли ваши функции одновременно вызываться разными сторонами для одного и того же экземпляра. В некоторых, более простых, случаях доступ к данным из разных потоков, возможно, и нет нужды синхронизировать, потому что данные только читаются. Тогда можно было бы реализовать паттерн «Неизменяемый экземпляр», который не позволяет вызывающей стороне модифицировать экземпляр. Но в остальных случаях придется реализовать в ваших функциях взаимное исключение при доступе к разделяемым ресурсам.

Ниже приведен пример простого «Разделяемого экземпляра».

Код вызывающей стороны 1

```
struct INSTANCE* inst = openInstance(INSTANCE_TYPE_B);
/* работать с тем же экземпляром, что и вызывающая сторона 2 */
operateOnInstance(inst);
closeInstance(inst);
```

Код вызывающей стороны 2

```
struct INSTANCE* inst = openInstance(INSTANCE_TYPE_B);
/* работать с тем же экземпляром, что и вызывающая сторона 1 */
operateOnInstance(inst);
closeInstance(inst);
```

API (заголовочный файл)

```
struct INSTANCE
{
  int x;
  int y;
};
```

```
/* будут использоваться в роли идентификаторов для функции openInstance */
#define INSTANCE_TYPE_A 1
#define INSTANCE_TYPE_B 2
#define INSTANCE_TYPE_C 3

/* Получить экземпляр с идентификатором 'id'. Создается новый экземпляр, если никакая
   другая вызывающая сторона еще не получала экземпляр с таким же идентификатором. */
struct INSTANCE* openInstance(int id);

/* Работает с данными, хранящимися в экземпляре. */
void operateOnInstance(struct INSTANCE* inst);

/* Освобождает экземпляр, полученный от 'openInstance'.
   Если все вызывающие экземпляры освободили экземпляр, то он уничтожается.
void closeInstance(struct INSTANCE* inst);
```

Реализация

```
#define MAX_INSTANCES 4

struct INSTANCELIST
{
  struct INSTANCE* inst;
  int count;
};

static struct INSTANCELIST list[MAX_INSTANCES];

struct INSTANCE* openInstance(int id)
{
 if(list[id].count == 0)
 {
  list[id].inst = malloc(sizeof(struct INSTANCE));
 }
 list[id].count++;
 return list[id].inst;
}

void operateOnInstance(struct INSTANCE* inst)
{
 /* работа с inst->x и inst->y */
}

static int getInstanceId(struct INSTANCE* inst)
{
 int i;
 for(i=0; i<MAX_INSTANCES; i++)
 {
```

```
    if(inst == list[i].inst)
    {
     break;
    }
   }
   return i;
  }

  void closeInstance(struct INSTANCE* inst)
  {
   int id = getInstanceId(inst);
   list[id].count--;
   if(list[id].count == 0)
   {
    free(inst);
   }
  }
```

Вызывающая сторона получает `INSTANCE`, вызывая `openInstance`. Экземпляр `INSTANCE` мог быть создан в этом вызове функции или ранее при предыдущем вызове, он может использоваться какой-то другой стороной. Затем вызывающая сторона может передать `INSTANCE` функции `operateOnInstance`, чтобы снабдить ее требуемым ресурсом или информацией о состоянии. По завершении работы с экземпляром вызывающая сторона должна вызвать `closeInstance`, чтобы освободить ресурсы, если больше никто с этим экземпляром `INSTANCE` не работает.

На рис. 5.5 схематически изображена работа «Разделяемого экземпляра».

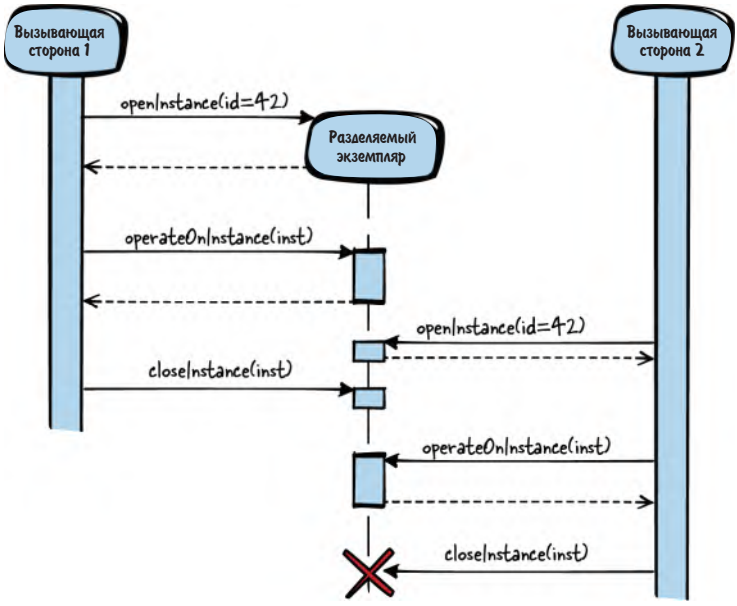


Рис. 5.5. Разделяемый экземпляр

Последствия

Теперь несколько вызывающих сторон могут одновременно обращаться к одному экземпляру. Очень часто это требует организации взаимного исключения, чтобы не нагружать такими заботами пользователя. Это означает, что время работы функции может изменяться, потому что вызывающая сторона не знает, использует ли сейчас те же ресурсы кто-то другой, вследствие чего доступ к ним блокирован.

Именно ваш программный модуль, а не вызывающая сторона владеет экземпляром и отвечает за очистку ресурсов. Вызывающая сторона по-прежнему несет ответственность за освобождение ресурсов, потому что иначе ваш программный модуль не узнает, когда нужно произвести окончательную очистку. Как и в случае «Экземпляра, принадлежащего вызывающей стороне», это частый источник утечек памяти.

Поскольку программный модуль владеет экземплярами, он может очистить их, не дожидаясь, пока вызывающая сторона инициирует очистку. Например, если программный модуль получает сигнал об остановке от операционной системы, он может очистить все экземпляры, потому что является их владельцем.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Пример использования «Разделяемого экземпляра» – функции из заголовочного файла *stdio.h*. Файл может быть открыт несколькими вызывающими сторонами с помощью функции `fopen`. Вызывающая сторона получает «Описатель» файла и может читать или записывать файл (`fread`, `fprintf`). Файл является разделяемым ресурсом. Например, в файле имеется всего одно положение курсора – глобальное для всех вызывающих сторон. Завершив работу с файлом, вызывающая сторона должна закрыть его, вызвав `fclose`.
- Этот паттерн и детали его реализации для объектно ориентированных языков программирования представлен под названием «Считающий описатель» в статье Kevlin Henney «C++ Patterns: Reference Accounting» (https://oreil.ly/inThj). Он описывает, как можно обращаться к разделяемому объекту в куче и как прозрачно управлять временем его жизни.
- К реестру Windows могут одновременно обращаться несколько потоков с помощью функции `RegCreateKey` (которая открывает раздел, если он уже существует). Функция возвращает «Описатель», который другие функции могут использовать для операций с разделом реестра. По завершении операций с реестром все стороны, открывавшие раздел, должны вызвать функцию `RegCloseKey`.
- Средства Windows для работы с Мьютексом (`CreateMutex`) можно использовать для доступа к разделяемому ресурсу (Мьютексу) из нескольких потоков. Мьютекс позволяет организовать межпроцессную синхрони-

зацию. По завершении работы с Мьютексом каждая вызывающая сторона должна закрыть его, вызвав функцию `CloseHandle`.

- Операционная система B&R Automation Runtime позволяет нескольким вызывающим сторонам одновременно обращаться к драйверам устройств. Для выбора одного из доступных устройств служит функция `DmDeviceOpen`. Подсистема драйверов проверяет, доступен ли выбранный драйвер, и если да, то предоставляет «Описатель» вызывающей стороне. Если с одним и тем же драйвером работает несколько вызывающих сторон, то они разделяют один «Описатель». Вызывающие стороны могут одновременно взаимодействовать с драйвером (отправлять или читать данные, управлять его поведением и т. д.), а по завершении работы сообщают об этом подсистеме драйверов, вызывая функцию `DmDeviceClose`.

Применение к сквозному примеру

Теперь драйвер дополнительно реализует следующие функции.

API (заголовочный файл)

```
struct Sender* openSender(char* destination_ip);
void sendByte(struct Sender* s, char data);
void closeSender(struct Sender* s);
```

Реализация

```
struct Sender* openSender(char* destination_ip)
{
 struct Sender* s;
 if(isInSenderList(destination_ip))
 {
  s = getSenderFromList(destination_ip);
 }
 else
 {
  s = createSender(destination_ip);
 }
 increaseNumberOfCallers(s);
 return s;
}

void sendByte(struct Sender* s, char data)
{
 number_of_sent_bytes++;
 /* сохранить данные с помощью сокета, хранящегося в Sender s */
}

void closeSender(struct Sender* s)
```

```
{
 decreaseNumberOfCallers(s);
 if(numberOfCallers(s) == 0)
 {
  /* закрыть сокет, хранящийся в Sender s */
  free(s);
 }
}
```

API сквозного примера изменился не сильно – вместо функций `create`/`destroy` драйвер теперь предоставляет функции `open`/`close`. Вызвав такую функцию, вызывающая сторона получает «Описатель» отправителя и сообщает драйверу, что она теперь с этим отправителем работает. Но вовсе необязательно, что драйвер создает отправителя именно в этот момент. Возможно, это уже было сделано в ответ на предыдущее обращение к драйверу (быть может, из другого потока). Кроме того, вызов `close` необязательно приводит к уничтожению отправителя. Владение этим отправителем остается за драйвером, который сам решает, когда его уничтожить (например, после того как все вызывающие стороны закроют его или при получении какого-то сигнала завершения).

Тот факт, что теперь у нас имеется «Разделяемый экземпляр», а не «Экземпляр, принадлежащий вызывающей стороне», по большей части прозрачен для вызывающей стороны. Но реализация драйвера изменилась – он должен помнить, был ли уже создан конкретный отправитель, и, если да, предоставлять именно его, а не создавать новый. Открывая отправитель, вызывающая сторона не знает, получит она в ответ только что созданный или уже существующий отправитель. В зависимости от этого время работы функции может измениться.

В представленном сквозном примере драйвера продемонстрированы различные виды владения и управления временем жизни. Мы видели, как эволюционировала функциональность простого драйвера Ethernet. Поначалу нам было достаточно «Программного модуля без состояния», потому что драйверу не была нужна никакая информация о состоянии. Затем такая информация о состоянии понадобилась, и это было реализовано с помощью паттерна «Программный модуль с глобальным состоянием». Далее нам потребовалась более производительная функция отправки и поддержка нескольких вызывающих сторон. Сначала мы реализовали эти потребности с помощью «Экземпляра, принадлежащего вызывающей стороне», а затем с помощью «Разделяемого экземпляра».

Резюме

Паттерны, рассмотренные в этой главе, иллюстрируют различные способы структурирования С-программ и времени жизни различных экземпляров. В табл. 5.2 приведено сравнение паттернов с точки зрения последствий.

Таблица 5.2. Сравнение паттернов, относящихся к времени жизни и владению

	Программный модуль без состояния	**Программный модуль с глобальным состоянием**	**Экземпляр, принадлежащий вызывающей стороне**	**Разделяемый экземпляр**
Разделение ресурсов между функциями	Невозможно	Единственный набор ресурсов	Набор ресурсов на каждый экземпляр (= на каждую вызывающую сторону)	Набор ресурсов на каждый экземпляр (разделяемый несколькими вызываемыми сторонами)
Владение ресурсами	Нечем владеть	Программный модуль владеет статическими данными	Вызывающая сторона владеет экземпляром	Программный модуль владеет экземплярами и предоставляет ссылки
Время жизни ресурса	Ресурсы живут не дольше вызова функции	Статические данные живут вечно в программном модуле	Экземпляры живут до тех пор, пока вызывающие стороны их не уничтожат	Экземпляры живут до тех пор, пока программный модуль их не уничтожит
Инициализация ресурса	Нечего инициализировать	На этапе компиляции или в момент запуска	Вызывающей стороной при создании экземпляра	Программным модулем при первом открытии экземпляра вызывающей стороной

Эти паттерны дают программисту на C рекомендации по вариантам разбиения программы на модули, а также по вопросам владения и времени жизни экземпляров.

Для дополнительного чтения

Паттерны, описанные в этой главе, показывают, как предоставить доступ к экземплярам и кто владеет этими экземплярами. Очень похожие вопросы решаются подмножеством паттернов, описанных в книге Markus Voelter et al. «Remoting Patterns» (Wiley, 2007). Там представлены паттерны для построения ПО промежуточного уровня, работающего с распределенными объектами, и три из них посвящены времени жизни и владению объектами, которые создаются удаленными серверами. У паттернов, рассмотренных в этой главе, контекст иной. Это паттерны не для распределенных систем, а для локальных процедурных программ. Они «заточены» под программирование на C, но могут

использоваться и в других процедурных языках. Но некоторые базовые идеи очень похожи на паттерны из книги «Remoting Patterns».

Что дальше

В следующей главе представлены разные виды интерфейсов к программным модулям с акцентом на то, как сделать интерфейс гибким. Рассматривается компромисс между простотой и гибкостью.

Глава 6

Гибкие API

Проектирование интерфейсов с нужными уровнями гибкости и абстракции – один из самых важных аспектов написания программ, поскольку интерфейсы – это контракт, который не может изменяться слишком часто, после того как система введена в эксплуатацию. Из-за этого так важно включать в интерфейс стабильные объявления и абстрагировать детали реализации, которая должна обладать достаточной гибкостью для изменения в будущем.

Для объектно ориентированных языков есть немало наставлений по проектированию интерфейсов (например, в форме паттернов проектирования). Но для процедурных языков типа C их гораздо меньше. Существуют принципы проектирования SOLID (см. врезку ниже), которые содержат общие указания по проектированию хороших программ. Однако конкретно для C найти подробное руководство по проектированию интерфейсов трудно, и тут-то вам помогут паттерны, описанные в этой главе.

SOLID

Принципы SOLID описывают, как реализовать хорошую, гибкую и удобную для сопровождения программу.

- *Принцип единственной обязанности* (**S**ingle-responsibility principle)
  Код имеет только одну обязанность и только одну причину для изменения в будущем.
- *Принцип открытости-закрытости* (**O**pen-closed principle)
  Код должен быть открыт для изменений поведения, не требуя при этом изменений уже существующего кода.
- *Принцип подстановки Лисков* (**L**iskow substitution principle)
  С точки зрения вызывающей стороны фрагменты кода, реализующие один и тот же интерфейс, должны быть взаимозаменяемы.
- *Принцип разделения интерфейсов* (**I**nterface segregation principle)
  Интерфейсы должны быть компактными и рассчитанными на потребности вызывающей стороны.
- *Принцип инверсии зависимости* (**D**ependency inversion principle)
  Модули верхнего уровня должны быть независимы от модулей нижнего уровня.

В статье James Grenning «SOLID Design for Embedded C» имеются дополнительные сведения о том, как реализовать принципы SOLID в C.

На рис. 6.1 показаны все четыре паттерна, рассматриваемых в этой главе, а также связанные с ними, а в табл. 6.1 приведены их краткие описания. Помните, что не все паттерны следует применять во всех возможных контекстах. В общем случае рекомендуется проектировать систему не более сложной, чем она должна быть. Это значит, что некоторые из представленных паттернов имеет смысл применять, только если приобретаемая в результате гибкость уже необходима в вашем API или с большой вероятностью понадобится в будущем. Если же это не так, то паттерн, возможно, не стоит применять, чтобы оставить API максимально простым.

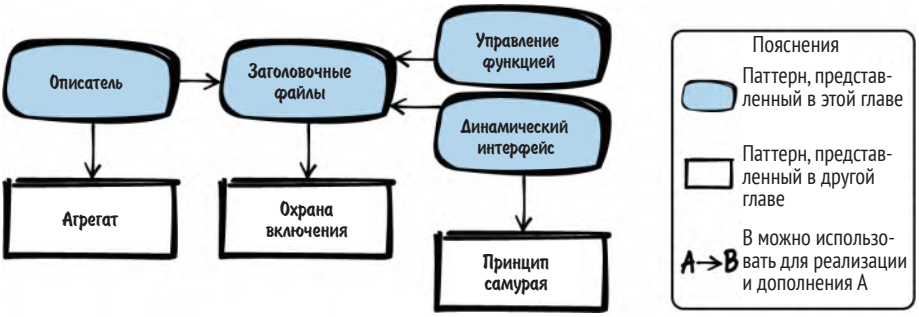


Рис. 6.1. Обзор паттернов для создания гибких API

Таблица 6.1. Паттерны для создания гибких API

Название паттерна	Краткое описание
Заголовочные файлы	Вы хотите, чтобы реализованная вами функциональность была доступна коду, реализованному другими лицами, но при этом хотите скрыть детали реализации от вызывающей стороны. Поэтому включите в свой API объявления функций, реализующих функциональность, предоставляемую пользователям. Скройте все внутренние функции, внутренние данные и определения (реализации) функций в файлах реализации и не передавайте эти файлы пользователям
Описатель	Вам нужно разделить информацию о состоянии или работать с разделяемыми ресурсами в реализациях своих функций, но вы не хотите, чтобы вызывающая сторона видела эту информацию и разделяемые ресурсы. Поэтому заведите функцию, создающую контекст, с которым будет работать вызывающая сторона, и возвращающую абстрактный указатель на внутренние данные в этом контексте. Потребуйте, чтобы вызывающая сторона передавала этот указатель всем вашим функциям, которые смогут тогда воспользоваться внутренними данными для хранения информации о состоянии и ресурсов

Название паттерна	Краткое описание
Динамический интерфейс	Должна быть возможность вызывать реализации с немного различающимся поведением, но при этом не хотелось бы дублировать код, даже код управления логикой реализации и объявления интерфейса. Поэтому определите общий интерфейс для различающейся функциональности в своем API и потребуйте, чтобы вызывающая сторона предоставила функцию обратного вызова для варианта этой функциональности, которую вы сможете вызвать из реализации своей функции
Управление функцией	Вы хотите вызывать реализации с немного различающимся поведением, но не хотите дублировать код, даже код управления логикой реализации и объявления интерфейса. Поэтому добавьте в функцию параметр, в котором функции передается метаинформация об этом вызове, определяющая, какая именно функциональность требуется

Сквозной пример

В качестве сквозного примера мы в этой главе реализуем драйвер сетевой карты Ethernet. Прошивка карты предлагает несколько регистров, с помощью которых можно отправлять и принимать данные, а также конфигурировать саму карту. Вы хотите абстрагировать эти аппаратные детали и сделать так, чтобы на пользователе вашего API не отражались изменения в частях реализации.

Для этого вы создаете API, состоящий из «Заголовочных файлов».

Заголовочные файлы

Контекст

Вы пишете большую программу на C. Вы хотите разбить ее на несколько функций и реализовать эти функции в разных файлах, потому что программа должна быть модульной и удобной для сопровождения.

Проблема

Требуется реализовать функциональность, которая была бы доступна из других файлов реализации, но при этом скрыть детали реализации от вызывающей стороны.

В отличие от многих объектно ориентированных языков программирования, в C нет встроенной поддержки для определения API, абстрагирования функциональности и средств принуждения вызывающей стороны к доступу только посредством этой абстракции. Язык C лишь предоставляет механизм включения одних файлов в другие.

Сторона, вызывающая ваш код, могла бы, используя этот механизм, просто включить ваш файл реализации. Но тогда у нее был бы доступ ко всем внутренним данным в этом файле, в том числе переменным или функциям с файловой областью видимости, которые, по идее, предназначены только для внутренне-

го пользования. Коль скоро вызывающая сторона начинает использовать эту функциональность, изменить ее впоследствии будет непросто, и образуется сильная связанность в тех местах, где вы хотели бы этого избежать. Если вызывающая сторона включает файл реализации, то имена внутренних переменных и функций могут вступать в конфликт с именами, используемыми вызывающей стороной.

Решение

Предоставьте в своем API объявления функций, реализующих ту функциональность, которая должна быть доступна пользователям. Скройте все внутренние функции, внутренние данные и определения (реализации) функций в файле реализации и не предоставляйте этот файл пользователям.

В C принято соглашение – любой пользователь может использовать только те функции, которые объявлены в заголовочном файле (*h*-файле), и не вправе использовать любые другие функции, которые могут быть определены в файлах реализации (*c*-файлах). Иногда эту абстракцию можно сделать обязательной (например, никаким образом нельзя обратиться к статической функции, определенной в другом файле), но в полной мере язык C такого принуждения не поддерживает. Поэтому соглашение не обращаться к чужим файлам реализации важнее механизмов принуждения.

В заголовочный файл не забудьте включить все, что необходимо объявленным в нем функциям. У вызывающей стороны не должно быть необходимости включать другие заголовочные файлы, чтобы воспользоваться функциональностью, объявленной в вашем. Если имеются объявления (например, типы данных или директивы `#define`), необходимые в нескольких заголовочных файлах, то поместите их в отдельный заголовочный файл и включайте его туда, где в этих объявлениях есть нужда. Чтобы один и тот же заголовочный файл не включался несколько раз в одну единицу компиляции, защитите его, применив паттерн «Охрана включения».

В один заголовочный файл помещайте только взаимосвязанные функции. Если функции работают с одним и тем же «Описателем» или выполняют операции, относящиеся к одной предметной области (например, математические вычисления), значит, есть веские основания поместить их объявления в один заголовочный файл. Вообще, если вы можете представить себе ситуацию, в которой понадобятся все функции, то помещайте их в один заголовочный файл.

Четко документируйте поведение своего API в заголовочном файле. У пользователя не должно возникать необходимости заглядывать в файл реализации, чтобы понять, как работают функции, предоставленные с помощью API.

Ниже приведен пример заголовочного файла.

API (h-файл)

```
/* Сортирует числа в массиве 'array' в порядке возрастания.
   'length' – число элементов в 'array'. */
void sort(int* array, int length);
```

Реализация (c-файл)

```
void sort(int* array, int length)
{
 /* здесь должна быть реализация */
}
```

Последствия

Имеется четкое разделение между тем, что важно для вызывающей стороны (*h*-файл), и деталями реализации, до которых вызывающей стороне не должно быть дела (*c*-файл). Таким образом, вы абстрагировали функциональность для вызывающей стороны.

Слишком много заголовочных файлов увеличит время сборки. С одной стороны, это позволяет разбить реализацию на отдельные файлы, и ваш комплект инструментов сможет выполнять инкрементную сборку, перекомпилируя только те файлы, которые изменились. С другой стороны, время полной сборки немного увеличится по сравнению со случаем, когда весь код находится в одном файле, потому что каждый файл необходимо открыть и прочитать.

Если оказывается, что функции взаимодействуют друг с другом больше, чем планировалось, или что их приходится вызывать в разных контекстах, требующих разной информации о внутреннем состоянии, то нужно будет подумать, как реализовать это в своем API. В таких случаях может помочь «Описатель».

Сторона, вызывающая ваши функции, теперь опирается на абстракцию. И может положиться на то, что поведение функций не изменится. Желательно, чтобы API сохранял стабильность. Для расширения функциональности всегда можно добавить в API новые функции. Но иногда приходится расширять существующие функции, и, чтобы не пасовать перед такими будущими изменениями, нужно подумать о том, как сделать функции гибкими, не жертвуя стабильностью. В таких случаях могут оказаться полезны паттерны «Описатель», «Динамический интерфейс» и «Управление функцией».

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Практически любая программа на C, более сложная, чем «Здравствуй, мир», содержит заголовочные файлы.
- Использование заголовочного файла в C аналогично использованию интерфейсов в Java или абстрактных классов в C++.
- Идиома Pimpl рекомендует, как скрыть детали реализации и не помещать их в заголовочный файл. Ее описание можно найти в портлендском репозитории паттернов.

Применение к сквозному примеру

Первый API драйвера устройства выглядит так:

```
void sendByte(char byte);
char receiveByte();
```

```
void setIpAddress(char* ip);
void setMacAddress(char* mac);
```

Пользователь вашего API не должен иметь дело с такими деталями реализации, как доступ к регистрам карты Ethernet, а вы можете изменять эти детали, не затрагивая интересы пользователей.

Но теперь требования к драйверу изменились. В системе появилась вторая, точно такая же сетевая карта Ethernet, и должна быть возможность работать с обеими. Ниже описаны прямолинейные способы добиться этой цели.

- Скопировать код и иметь по одному экземпляру кода для каждой карты. В скопированном коде изменяется только адрес интерфейса, к которому осуществляется доступ. Однако такое дублирование кода никогда не является хорошей идеей, сопровождение программы при этом резко усложнится.
- Добавить в каждую функцию параметр, содержащий адрес сетевой карты (например, строку с именем устройства). Однако высока вероятность, что понадобится больше одного параметра, а если передавать их каждой функции, то API станет громоздким.

Для поддержки несколько карт Ethernet есть идея получше: ввести в API «Описатель».

Описатель

Контекст

Вы хотите предоставить вызывающей стороне набор функций, которые работают с общими ресурсами или разделяют информацию о состоянии.

Проблема

Вы должны разделять информацию о состоянии или работать с общими ресурсами в реализациях своих функций, но не хотите, чтобы вызывающая сторона видела, а тем более могла обращаться к этой информации о состоянии и ресурсам.

Информация о состоянии и общие ресурсы должны оставаться невидимыми вызывающей стороне, потому что в будущем вы, возможно, захотите изменить или расширить их, не требуя изменений в вызывающем коде.

В объектно ориентированных языках программирования данные, с которыми работают функции, реализуются в виде переменных-членов класса. Эти переменные можно сделать закрытыми, если у вызывающей стороны не должно быть к ним доступа. Но C не поддерживает ни классов, ни закрытых переменных-членов.

Просто завести в своем файле реализации «Программный модуль с глобальным состоянием», в котором разделяемые между функциями данные хранятся в статических глобальных переменных, – не выход, потому что должна быть возможность вызывать функции в нескольких контекстах, причем каждая вызывающая сторона будет работать с собственной информацией о состоянии.

И хотя эта информация остается невидимой вызывающим сторонам, у вас должна быть возможность понять, какая информация какой вызывающей стороне принадлежит и как обращаться к этой информации в реализациях своих функций.

Решение

Заведите функцию, создающую контекст, с которым будет работать вызывающая сторона, и возвращающую абстрактный указатель на внутренние данные этого контекста. Потребуйте, чтобы вызывающая сторона передавала этот указатель всем вашим функциям, которые тогда смогут использовать внутренние данные для хранения информации о состоянии и ресурсов.

Ваши функции должны знать, как интерпретировать этот абстрактный указатель, являющийся непрозрачным типом данных, который называется также «Описателем». Однако структура данных, на которую он указывает, не должна быть частью API. API лишь предоставляет функциональность для передачи скрытых данных функциям.

«Описатель» можно реализовать как указатель на «Агрегат», например структуру `struct`. Структура должна содержать всю необходимую информацию о состоянии и прочие переменные – обычно в нее включаются переменные, аналогичные тем, что вы объявили бы как переменные-члены класса в объектно ориентированном языке. Структура должна быть скрыта внутри реализации. API содержит только объявление указателя на `struct`, как показано в коде ниже.

API

```
typedef struct SORT_STRUCT* SORT_HANDLE;

SORT_HANDLE prepareSort(int* array, int length);
void sort(SORT_HANDLE context);
```

Реализация

```
struct SORT_STRUCT
{
  int* array;
  int length;
  /* другие параметры, например порядок сортировки */
};

SORT_HANDLE prepareSort(int* array, int length)
{
 struct SORT_STRUCT* context = malloc(sizeof(struct SORT_STRUCT));
 context->array = array;
 context->length = length;

 /* заполнить context данными или информацией о состоянии */
```

```
 return context;
}

void sort(SORT_HANDLE context)
{
 /* работает с данными из контекста */
}
```

Заведите в своем API одну функцию для создания «Описателя». Эта функция возвращает «Описатель» вызывающей стороне. Затем вызывающая сторона может вызывать другие функции API, нуждающиеся в «Описателе». В большинстве случаев нужна также функция удаления «Описателя» для очистки выделенных ресурсов.

Последствия

Теперь вы можете разделять информацию о состоянии и ресурсы между функциями, не утруждая этой проблемой вызывающую сторону и не давая ей возможности внести в код зависимость от этих внутренних деталей.

Поддерживается несколько экземпляров данных. Функцию создания «Описателя» можно вызывать несколько раз для получения нескольких контекстов, а затем работать с этими контекстами независимо друг от друга.

Если функции, работающие с «Описателем», в будущем изменятся и понадобится разделять другие или дополнительные данные, то члены `struct` можно будет просто изменить, не требуя вносить изменений в вызывающий код.

Из объявлений функций явно видно, что они тесно связаны, потому что все требуют «Описателя». Поэтому, с одной стороны, легко понять, какие функции должны быть помещены в один заголовочный файл, а с другой стороны, вызывающей стороне очень просто найти, какие функции следует применять совместно.

Паттерн «Описатель» требует, чтобы вызывающая сторона передавала один дополнительный параметр каждой функции, а каждый дополнительный параметр делает код сложнее для восприятия.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В стандартной библиотеке C имеется определение типа `FILE` в файле *stdio.h*. В большинстве реализаций этот тип определен как указатель на `struct`, а сама структура не является частью заголовочного файла. Описатель `FILE` создается функцией `fopen`, после чего для открытого файла можно вызывать другие функции (`fwrite`, `fread` и т. д.).
- `struct AES_KEY` в коде OpenSSL используется для передачи контекста между несколькими функциями, относящимися к AES-шифрованию (`AES_set_decrypt_key`, `AES_set_encrypt_key`). Структура и ее члены не скрыты в недрах реализации, а являются частью заголовочного файла, потому что в разных местах кода OpenSSL необходимо знать размер структуры.

- Код протоколирования в проекте Subversion работает с «Описателем». Тип `struct logger_t` определен в файле реализации, а указатель на структуру – в соответствующем заголовочном файле.
- Этот паттерн описан в книге David R. Hanson «C Interfaces and Implementations» (Addison-Wesley, 1996) под названием «Непрозрачный указатель», а в книге Adam Tornhill «Patterns in C» как «Паттерн полноценного абстрактного типа данных».

Применение к сквозному примеру

Теперь вы можете поддержать столько сетевых карт Ethernet, сколько нужно. Каждый созданный экземпляр драйвера порождает собственный контекст данных, который затем передается функциям в виде «Описателя». API драйвера устройства принимает следующий вид.

```
/* INTERNAL_DRIVER_STRUCT содержит данные, разделяемые функциями (например,
   как выбрать сетевую карту, за которую отвечает драйвер) */
typedef struct INTERNAL_DRIVER_STRUCT* DRIVER_HANDLE;

/* 'initArg' содержит информацию, идентифицирующую конкретную
   сетевую карту для данного экземпляра драйвера */
DRIVER_HANDLE driverCreate(void* initArg);
void driverDestroy(DRIVER_HANDLE h);
void sendByte(DRIVER_HANDLE h, char byte);
char receiveByte(DRIVER_HANDLE h);
void setIpAddress(DRIVER_HANDLE h, char* ip);
void setMacAddress(DRIVER_HANDLE h, char* mac);
```

Требования в очередной раз изменились. Теперь вы должны поддерживать несколько разных сетевых карт Ethernet, например от разных производителей. Функциональность всех карт схожа, но детали доступа к регистрам различаются, а следовательно, необходимы разные реализации драйверов. Ниже описаны два прямолинейных способа поддержать это требование.

- Завести два отдельных API драйвера. Недостаток этого подхода в том, что пользователям придется создавать громоздкие механизмы выбора драйвера во время выполнения. Кроме того, наличие двух разных API ведет к дублированию кода, потому что у драйверов как минимум общий поток управления (например, создание и уничтожение драйвера).
- Добавить в API функции типа `sendByteDriverA` и `sendByteDriverB`. Однако обычно вы хотите, чтобы API был минимальным, а включение всех функций драйвера в один API может привести пользователя в замешательство. Кроме того, код пользователя зависит от сигнатур всех функций, включенных в API, а если код от чего-то зависит, то это что-то должно быть как можно меньше (см. принцип разделения интерфейсов).

Гораздо лучше поддержать разные сетевые карты Ethernet с помощью «Динамического интерфейса».

Динамический интерфейс

Контекст

Вы или вызывающая вас сторона хотите реализовать несколько видов функциональности, имеющих схожую логику, но отличающееся в деталях поведение.

Проблема

Должна быть возможность вызывать реализации с несколько различающимся поведением, но при этом нежелательно дублировать код и даже реализацию логики управления и объявление интерфейса.

Вы хотите иметь возможность в будущем добавлять новые поведения в реализацию объявленного интерфейса, не требуя от пользователей, работающих с существующими поведениями, что-либо изменять в своем коде.

Быть может, вы хотите предоставлять пользователям не только различные поведения без дублирования кода, но и механизм, позволяющий им создавать собственные поведения.

Решение

Определите в своем API общий интерфейс для различающейся в деталях функциональности и потребуйте от вызывающей стороны предоставлять функцию обратного вызова, которую вы сможете вызывать из своей реализации функции.

Для реализации такого интерфейса на C определите в своем API сигнатуры функций. Вызывающая сторона сможет реализовать функции с такими сигнатурами и присоединить их с помощью указателей на функции. Их можно присоединить и хранить в вашем программном модуле на постоянной основе или указывать при каждом вызове функции, как показано в коде ниже.

API

```
/* Функция compare должна возвращать true, если x меньше y,
   и false в противном случае */
typedef bool (*COMPARE_FP)(int x, int y);

void sort(COMPARE_FP compare, int* array, int length);
```

Реализация

```
void sort(COMPARE_FP compare, int* array, int length)
{
 int i, j;for(i=0; i<length; i++)
 {
  for(j=i; j<length; j++)
  {
   /* вызвать предоставленную пользователем функцию */
```

```
    if(compare(array[i], array[j]))
    {
     swap(&array[i], &array[j]);
    }
   }
  }
 }
```

Вызывающая сторона

```
#define ARRAY_SIZE 4

bool compareFunction(int x, int y)
{
 return x<y;
}

void sortData()
{
 int array[ARRAY_SIZE] = {3, 5, 6, 1};
 sort(compareFunction, array, ARRAY_SIZE);
}
```

Обязательно документируйте рядом с определением сигнатуры функции поведение, ожидаемое от ее реализаций. Кроме того, документируйте, что должно происходить, когда к вызову функции не присоединена никакая реализация. Возможно, в этом случае программа должна быть завершена («Принцип самурая») или же предоставляется какая-то реализация по умолчанию.

Последствия

Вызывающая сторона может использовать разные реализации без какого-либо дублирования кода. Не дублируется ни управляющая логика, ни интерфейс, ни документация интерфейса.

Реализации могут быть добавлены вызывающей стороной в будущем без изменения API. Это означает, что роли проектировщика API и поставщика реализаций можно полностью разделить.

Теперь в своем коде вы исполняете код, предоставленный вызывающей стороной. Следовательно, вы должны быть уверены, что вызывающая сторона знает, что должна делать функция. При наличии ошибок в коде вызывающей стороны первым подозреваемым становится ваш код, потому что именно в его контексте имело место некорректное поведение.

Применение указателей на функции означает, что интерфейс стал зависеть от платформы и от языка программирования. Этот паттерн можно использовать, только если код вызывающей стороны написан на C. Невозможно добавить в интерфейс функциональность маршалинга и предложить его вызывающей стороне, написанной, к примеру, на Java.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В статье James Grenning «SOLID Design for Embedded C» (https://oreil.ly/kGZVG) этот паттерн и его вариант названы «Динамический интерфейс» и «Потиповой динамический интерфейс».
- Представленное решение – это C-версия паттерна проектирования «Стратегия». Альтернативные реализации этого паттерна на C можно найти в книгах Adam Tornhill «Patterns in C» (Leanpub, 2014) и David R. Hanson «C Interfaces and Implementations» (Addison-Wesley, 1996).
- В подсистемах драйверов часто используются указатели на функции, которые драйвер инициализирует своей функцией на этапе загрузки системы. Драйверы устройств в ядре Linux обычно устроены именно так.
- Функция `svn_sort__hash` в проекте Subversion сортирует список по значению некоторого ключа. Она принимает в качестве параметра указатель на функцию `comparison_func`, которая должна вернуть информацию о том, какой из двух ключей больше.
- Функция `OPENSSL_LH_new` из библиотеки OpenSSL создает хеш-таблицу. Вызывающая сторона должна предоставить указатель на функцию хеширования, которая используется как функция обратного вызова при работе с хеш-таблицей.
- В коде Wireshark указатель на функцию `proto_tree_foreach_func` предоставляется в качестве параметра функции обхода древовидных структур. Функция, на которую он указывает, решает, какие действия произвести над элементами дерева.

Применение к сквозному примеру

Теперь API вашего драйвера поддерживает несколько сетевых карт Ethernet. Конкретные драйверы этих карт должны реализовать функции отправки и приема данных и предоставить их в отдельном заголовочном файле. Пользователь API может включить этот файл и присоединить конкретные функции отправки и приема.

У вас появилось дополнительное преимущество – пользователи вашего API могут предложить собственную реализацию драйвера. Таким образом, вы, как проектировщик API, не зависите от поставщика реализации драйвера. Интеграция новых драйверов не требует внесения изменений в API, а значит, не требует никакой работы от вас как проектировщика API. Все это становится возможным при таком определении API:

```
typedef struct INTERNAL_DRIVER_STRUCT* DRIVER_HANDLE;
typedef void (*DriverSend_FP)(char byte); /* это */
typedef char (*DriverReceive_FP)();       /* определение интерфейса */

struct DriverFunctions
```

```
{
  DriverSend_FP fpSend;
  DriverReceive_FP fpReceive;
};

DRIVER_HANDLE driverCreate(void* initArg, struct DriverFunctions f);
void driverDestroy(DRIVER_HANDLE h);
void sendByte(DRIVER_HANDLE h, char byte); /* внутри себя вызывает fpSend */
char receiveByte(DRIVER_HANDLE h);         /* внутри себя вызывает fpReceive */
void setIpAddress(DRIVER_HANDLE h, char* ip);
void setMacAddress(DRIVER_HANDLE h, char* mac);
```

Требования снова изменились. Теперь надо поддерживать не только сетевые карты Ethernet, но и другие интерфейсные карты (например, карты USB). С точки зрения API часть функциональности этих интерфейсов похожа (функции отправки и приема данных), а часть совершенно различна (например, в интерфейсе USB нет IP-адреса, но могут быть другие конфигурационные параметры).

Прямолинейный подход – предоставить разные API для драйверов разных типов. Но тогда будет дублироваться код функций отправки/приема и создания/уничтожения.

Более правильное решение для поддержки разных видов драйверов устройств в одном абстрактном API дает паттерн «Управление функцией».

Управление функцией

Контекст

Вы хотите реализовать несколько видов функциональности, имеющих схожую логику, но отличающееся в деталях поведение.

Проблема

Вы хотите вызывать реализации с несколько различающимся поведением, но при этом нежелательно дублировать код и даже реализацию логики управления и объявление интерфейса.

У вызывающей стороны должна быть возможность использовать существующие поведения, реализованные вами. Но у вас также должна быть возможность в будущем добавлять новые поведения, не затрагивая существующие и не требуя от пользователей что-либо изменять в своем коде.

Динамический интерфейс не выход, потому что вы не хотите, чтобы пользователи могли присоединять собственные реализации. Причиной может быть, например, стремление сделать интерфейс проще для вызывающей стороны. А быть может, присоединить реализации трудно, например, потому что вызывающая сторона написана на другом языке.

Решение

Добавьте в функцию параметр, в котором передается метаинформация о вызове функции и указывается, что именно должно быть сделано.

По сравнению с «Динамическим интерфейсом» вы не требуете от вызывающей стороны предоставления реализации, а позволяете ей выбрать одну из существующих реализаций.

Чтобы реализовать этот паттерн, вы применяет основанную на данных абстракцию, включив дополнительный параметр (например, целое значение, определенное с помощью `enum` или `#define`), который определяет поведение функции. Функция проверяет этот параметр и вызывает соответствующую реализацию.

API

```
#define QUICK_SORT 1
#define MERGE_SORT 2
#define RADIX_SORT 3

void sort(int algo, int* array, int length);
```

Реализация

```
void sort(int algo, int* array, int length)
{
 switch(algo)
 {
  case QUICK_SORT: ❶
   quicksort(array, length);
  break;
  case MERGE_SORT:
   mergesort(array, length);
  break;
  case RADIX_SORT:
   radixsort(array, length);
  break;
 }
}
```

❶ При добавлении новой функциональности в будущем нужно будет просто добавить еще один элемент `enum` или директиву `#define` и выбрать соответствующую ему реализацию.

Последствия

Вызывающая сторона может использовать разные реализации без какого-либо дублирования кода. Не дублируется ни управляющая логика, ни интерфейс, ни документация интерфейса.

Новую функциональность легко добавить впоследствии. Для этого не придется как-то затрагивать существующие реализации и уже написанный код вызывающей стороны.

По сравнению с динамическим интерфейсом это паттерн проще, когда нужно выбирать функциональность, реализованную в разных программах или на разных платформах (например, вызовы удаленных процедур), потому что API не предусматривает передачи зависящих от программы указателей.

Предлагая на выбор разные поведения в одной функции, легко поддаться искушению и включить разнородную функциональность, которой в одной функции не место. Это нарушает принцип единственной обязанности.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В драйверах устройств управление функцией часто используется для реализации функциональности, которая не укладывается в стандартный набор: инициализация-чтение-запись. В контексте драйверов этот паттерн обычно называется «Управление вводом-выводом». Эта концепция описана в книге Elecia White «Making Embedded Systems: Design Patterns for Great Software» (O'Reilly, 2011).
- В некоторых системных вызовах Linux имеются флаги, добавляющие новую функциональность в зависимости от значения флага, без нарушения работоспособности старого кода.
- Концепция API, управляемых данными, в общем виде описана в книге Martin Reddy «API Design for C++» (Morgan Kaufmann, 2011).
- В коде OpenSSL функция `CTerr` используется для протоколирования ошибок. Она принимает параметр типа `enum`, который описывает, как и где протоколировать ошибку.
- Функция POSIX для работы с сокетами `ioctl` принимает числовой параметр `cmd`, определяющий, какое действие следует произвести с сокетом. Допустимые значения этого параметра определены и документированы в заголовочном файле, и с момента выхода первой версии этого файла в него было добавлено много значений, а значит, и новых видов поведения.
- Функция `svn_fs_ioctl` из проекта Subversion выполняет различные операции ввода-вывода, зависящие от файловой системы. Она принимает параметр типа `struct svn_fs_ioctl_code_t`. Эта структура содержит числовое значение, определяющее, какую операцию следует выполнить.

Применение к сквозному примеру

Ниже показана финальная версия API вашего драйвера.

Driver.h

```
typedef struct INTERNAL_DRIVER_STRUCT* DRIVER_HANDLE;
typedef void (*DriverSend_FP)(char byte);
typedef char (*DriverReceive_FP)();
```

```
typedef void (*DriverIOCTL_FP)(int ioctl, void* context);

struct DriverFunctions
{
  DriverSend_FP fpSend;
  DriverReceive_FP fpReceive;
  DriverIOCTL_FP fpIOCTL;
};

DRIVER_HANDLE driverCreate(void* initArg, struct DriverFunctions f);
void driverDestroy(DRIVER_HANDLE h);
void sendByte(DRIVER_HANDLE h, char byte);
char receiveByte(DRIVER_HANDLE h);
void driverIOCTL(DRIVER_HANDLE h, int ioctl, void* context);
/* параметр "context" служит для передачи такой информации, как
   IP-адрес, нужной для конфигурирования реализации */
```

EthIOCTL.h

```
#define SET_IP_ADDRESS 1
#define SET_MAC_ADDRESS 2
```

UsbIOCTL.h

```
#define SET_USB_PROTOCOL_TYPE 3
```

Пользователи, желающие использовать функции, специфичные для Ethernet или USB (например, приложение, которое фактически отправляет или принимает данные через интерфейс), должны знать, с драйвером какого типа они работают, чтобы вызывать правильные операции управления вводом-выводом, а также включить нужный файл: *EthIOCTL.h* или *UsbIOCTL.h*.

На рис. 6.2 показаны отношения включения файлов при такой версии API драйвера. Отметим, что файл *EthApplication.c* не зависит от заголовочных файлов, специфичных для USB. Если, например, будет добавлен USB-IOCTL, то файл *EthApplication.c* даже не придется перекомпилировать, потому что ни один из файлов, от которых он зависит, не изменился.

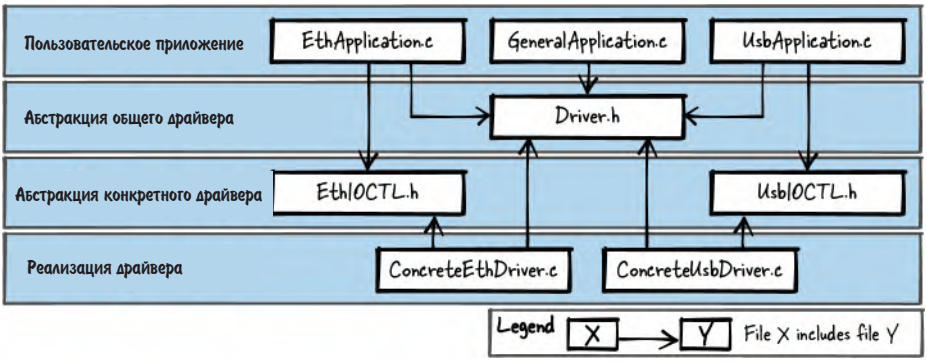


Рис. 6.2. Связи между файлами для управления функцией

Имейте в виду, что из всех представленных в этой главе фрагментов кода этот последний, самый гибкий вариант драйвера устройства не всегда является тем, что вы ищете. За гибкость приходится расплачиваться сложностью интерфейса, а, пытаясь сделать свой код максимально гибким, вы тем не менее должны стремиться к максимальной простоте.

Резюме

В этой главе рассмотрены четыре паттерна API для программ на C и продемонстрировано их применение в сквозном примере проектирования драйвера устройства. Паттерн «Заголовочные файлы» рекомендует базовую идею сокрытия деталей реализация в *c*-файлах и предоставления хорошо определенного интерфейса в *h*-файлах. Паттерн «Описатель» формулирует хорошо известную концепцию передачи непрозрачных типов данных между функциями с целью разделения информации о состоянии. Паттерн «Динамический интерфейс» позволяет не дублировать логику посредством включения в программу определенного вызывающей стороной кода с помощью функции обратного вызова. В паттерне «Управление функцией» используется дополнительный параметр, определяющий, какое конкретное действие должна выполнить вызываемая функция. Эти паттерны представляют собой основные варианты проектирования гибких интерфейсов на C с помощью абстракций.

Для дополнительного чтения

Для тех, кто хочет знать больше, ниже приведены ресурсы, которые помогут вам расширить знания о проектировании API.

- В статье James Grenning «SOLID Design for Embedded C» (https://oreil.ly/07SUX) рассматриваются все пять принципов проектирования SOLID в общем виде, а также описывается, как реализовать гибкие интерфейсы на C. Уникальной эту статью делает то, что она включает не только вопрос об интерфейсах конкретно на C, но и фрагменты кода со всеми деталями.
- В книге Adam Tornhill «Patterns in C» (Leanpub, 2014) представлено несколько паттернов с фрагментами кода на C. Включены C-версии паттернов «банды четырех», в том числе «Стратегия» и «Наблюдатель», а также специфические для C паттерны и идиомы. В книге не разбираются интерфейсы как отдельная тема, но в некоторых паттернах взаимодействия описаны на уровне интерфейса.
- В книге Martin Reddy «API Design for C++» (Morgan Kaufmann, 2011) рассматриваются принципы проектирования интерфейсов, приводятся примеры паттернов объектно ориентированных интерфейсов и обсуждаются вопросы качества интерфейсов, в частности тестирование и документация. Книга посвящена проектированию API на C++, но некоторые ее части равным образом применимы и к C.

- В книге David R. Hanson «C Interfaces and Implementations» (Addison-Wesley, 1996) описывается проектирование интерфейсов, включая код конкретных компонентов на C.

Что дальше

В следующей главе мы подробно обсудим, как найти правильный уровень абстракции и правильный интерфейс для одного очень конкретного вида приложений: проектирование и реализация итераторов.

Глава 7

Гибкие интерфейсы итераторов

Обход множества элементов – типичная операция в любой программе. В некоторых языках программирования имеются даже встроенные конструкции для этой цели, а для объектно ориентированных языков существуют наставления в форме паттернов проектирования по реализации итерирования в общем виде. Однако для процедурных языков типа C таких наставлений очень мало.

Глагол «итерировать» означает повторять одно и то же несколько раз. В программировании это обычно означает выполнение одного и того же программного кода для нескольких элементов данных. Такие операции часто необходимы, и потому в C встроена их поддержка для массивов, как показано в примере ниже:

```
for (i=0; i<MAX_ARRAY_SIZE; i++)
{
 doSomethingWith(my_array[i]);
}
```

Если вы хотите обойти другую структуру данных, например красно-черное дерево, то должны будете реализовать функцию итерирования самостоятельно. Возможно, вы захотите снабдить ее параметрами, специфичными для конкретной структуры, например обходить ли дерево в глубину или в ширину. В литературе можно найти рекомендации по реализации конкретных структур данных и интерфейсов итерирования для них. Если вы воспользуетесь таким «заточенным» под структуру данных интерфейсом итерирования, а структура изменится, то придется адаптировать как саму функцию итерирования, так и весь вызывающий ее код. Иногда это нормально и даже необходимо, потому что структура нуждается в специализированном итераторе, например с целью оптимизации производительности.

А иногда, если требуется предоставить интерфейс итерирования, пересекающий границы компонентов, абстракция, допускающая утечку деталей реализации, неприемлема, так как может потребовать изменения интерфейса в будущем. Например, если вы продаете клиентам компонент, пре-

доставляющий функции итерирования, и клиент пишет свой код, пользующийся этими функциями, то, вероятно, он ожидает, что его код продолжит работать без изменений после выхода новой версии компонента, в которой, быть может, используется другая структура данных. В таком случае вам придется приложить дополнительные усилия для обеспечения совместимости интерфейса, чтобы клиенту не пришлось изменять (или даже перекомпилировать) свой код.

Отсюда мы и начнем эту главу. Я представлю три паттерна, которые позволят вам, разработчику, предоставить стабильный интерфейс итератора пользователям (клиентам). Эти паттерны не описывают конкретные виды итераторов для конкретных структур данных. Вместо этого предполагается, что где-то внутри вашей реализации уже имеются функции для извлечения элементов из структуры данных. Паттерны предлагают, как абстрагировать эти функции, чтобы предоставить стабильный интерфейс итерирования.

На рис. 7.1 показаны паттерны, рассматриваемые в этой главе, и связи между ними, а в табл. 7.1 приведены их краткие описания.

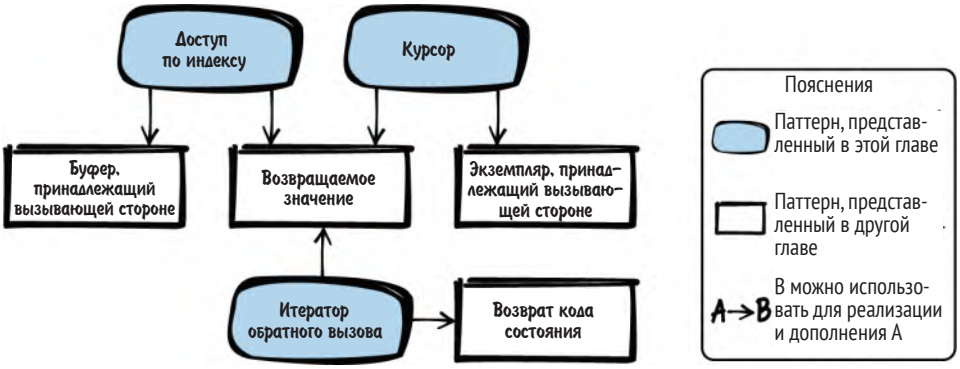


Рис. 7.1. Обзор паттернов для создания интерфейсов итератора

Таблица 7.1. Паттерны для создания интерфейсов итератора

Название паттерна	Краткое описание
Доступ по индексу	Вы хотите, чтобы пользователь мог удобно перебирать элементы вашей структуры данных, при этом нужно сохранить возможность изменения внутреннего устройства этой структуры, не изменяя пользовательский код. Поэтому предоставьте функцию, которая принимает индекс для адресации элемента в вашей структуре данных и возвращает его содержимое. Пользователь вызывает эту функцию в цикле для обхода всех элементов

Название паттерна	Краткое описание
Курсор	Вы хотите предоставить пользователю такой интерфейс итератора, который был бы устойчив относительно изменения элементов в процессе итерирования и который позволил бы изменять впоследствии структуру данных, не внося изменений в пользовательский код. Поэтому создайте экземпляр курсора, указывающий на элемент структуры данных. Функция итерирования принимает этот экземпляр итератора в качестве аргумента, получает элемент, на который указывает итератор, и модифицирует экземпляр итератора, так чтоб он указывал на следующий элемент. Пользователь в цикле вызывает эту функцию, чтоб получать элементы по одному
Итератор обратного вызова	Вы хотите предоставить устойчивый интерфейс итерирования, который не требовал бы от пользователя реализации цикла для обхода всех элементов и позволял бы в будущем вносить изменения в структуру данных, не изменяя пользовательский код. Поэтому воспользуйтесь своей уже существующей структурой данных и реализованными вами операциями для обхода всех элементов в ней и вызывайте предоставленную пользователем функцию для каждого элемента в процессе этого обхода. Эта пользовательская функция принимает содержимое элемента в качестве параметра и выполняет операции над этим элементом. Чтобы начать итерирование, пользователь вызывает всего одну функцию, а весь обход производится внутри вашей реализации

Сквозной пример

В своем приложении вы реализовали компонент управления доступом, в котором имеется структура данных и функция для произвольного доступа к любому из хранящихся в ней элементов. Конкретно в следующем коде имеется массив структур, в которых хранится информация об учетных записях, в том числе логины и пароли:

```
struct ACCOUNT
{
  char loginname[MAX_NAME_LENGTH];
  char password[MAX_PWD_LENGTH];
};
struct ACCOUNT accountData[MAX_USERS];
```

В коде ниже показано, как пользователь может обратиться к этой структуре, чтобы прочитать, например, логины:

```
void accessData()
{
 char* loginname;
```

```
  loginname = accountData[0].loginname;
  /* сделать что-то с loginname */

  loginname = accountData[1].loginname;
  /* сделать что-то с loginname */
}
```

Разумеется, можно было бы вообще не «заморачиваться» абстрагированием доступа к структуре данных, а позволить другим программистам получить указатель на эту структуру, обойти ее элементы в цикле и получить доступ к хранящейся в них информации. Но это неудачная мысль, потому что в структуре данных может находиться информация, которую вы не хотели бы предоставлять клиенту. Если интерфейс с клиентом должен оставаться стабильным, то, однажды предоставив клиенту доступ к некоторой информации, вы уже не сможете ее удалить – если вдруг клиент решил воспользоваться ей, то после удаления его код перестанет работать.

Чтобы избежать этой проблемы, гораздо лучше дать клиенту доступ только к необходимой информации. Простое решение – предоставить «Доступ по индексу».

Доступ по индексу

Контекст

Имеется множество элементов, хранящихся в структуре данных, к которой возможен произвольный доступ. Например, имеется массив или база данных с функциями для произвольной выборки одного элемента. Пользователь хочет обойти эти элементы.

Проблема

Вы хотите дать пользователю удобную возможность обходить элементы в вашей структуре данных, так чтобы ее внутреннее устройство можно было изменять, не требуя внесения изменений в пользовательский код.

Пользователем может быть любой, кто пишет код, который не выпускается синхронно с вашей кодовой базой, поэтому вы должны гарантировать, что будущие версии вашей реализации смогут работать с пользовательским кодом, написанным для текущей версии. Таким образом, у пользователя не должно быть доступа к внутренним деталям реализации, например прямого доступа к структуре данных, в которой вы храните элементы, потому что эти детали могут измениться в будущем.

Решение

Предоставьте функцию, которая принимает индекс, адресующий элемент в вашей структуре данных, и возвращает содержимое этого элемента. Пользователь вызывает эту функцию в цикле, чтобы обойти все элементы, как показано на рис. 7.2.

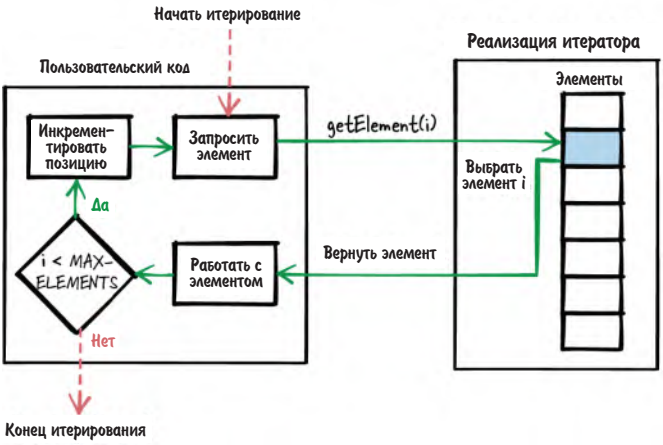


Рис. 7.2. Итерирование с доступом по индексу

В случае массива пользователь может напрямую использовать индекс, чтобы получить значение одного элемента или обойти все элементы. Но если такой индекс передается функции, то возможен обход более сложных структур данных без раскрытия информации об их внутреннем строении пользователю.

Чтобы достичь этой цели, предоставляйте пользователям только те данные, которые им нужны, и не раскрывайте все содержимое структуры данных. Например, возвращайте указатель не на всю структуру, а только на ее член, интересный пользователю:

Код вызывающей стороны

```
void* element;

element = getElement(1);
/* работать с элементом 1 */

element = getElement(2);
/* работать с элементом 2 */
```

API итератора

```
#define MAX_ELEMENTS 42

/* Извлечь только элемент, идентифицируемый значением 'index' */
void* getElement(int index);
```

Последствия

Используя индекс, пользователь может удобно обходить элементы в цикле. Ему не нужно иметь дело с внутренними деталями структуры данных, в которой хранятся данные. Если в ее реализации что-то изменится (например, извлекаемый элемент будет переименован), то пользователю не придется перекомпилировать свой код.

Другие изменения внутренней структуры данных могут вызвать более серьезные трудности. Например, если вместо массива (структуры с произвольным доступом) вы решите использовать связанный список (структуру с последовательным доступом), то придется каждый раз проходить по списку до позиции с указанным индексом. Это абсолютно неэффективно, и, чтобы сделать возможными такие изменения структуры данных, лучше воспользоваться паттерном «Курсор» или «Итератор обратного вызова».

Если пользователь запрашивает только данные примитивных типов, которые С-функция может вернуть в качестве значения, то в ответ он получает копию элемента. Если сам элемент структуры в это время изменится, то пользователь этого не заметит. Но если пользователь получает данные более сложного типа (например, строку), то, по сравнению с прямым доступом к структуре данных, «Доступ по индексу» дает дополнительное преимущество – скопировать текущий элемент данных можно потокобезопасным образом и предоставить его, например, в «Буфере, принадлежащем вызывающей стороне». Если же программа работает не в многопоточном окружении, то можно просто вернуть указатель на данные сложного типа.

Обращаясь к множеству элементов, пользователь часто хочет обойти их все. Если тем временем кто-то добавит элемент в структуру или удалит из нее элемент, то представление пользователя об индексе для доступа к элементам становится недействительным, и он может, например, получить один и тот же элемент дважды. Прямолинейное решение этой проблемы – просто скопировать все интересующие пользователя элементы в массив и предоставить к нему монопольный доступ пользователю, который затем сможет беспрепятственно обойти его. Пользователь получит эту копию в «Единоличное владение» и сможет даже модифицировать элементы. Но если явно такое требование не выдвигается, то копировать все элементы, пожалуй, излишне. Гораздо удобнее – предоставить «Итератор обратного вызова», тогда пользователю не нужно будет думать о возможном изменении данных.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В статье *Ja*mes Noble «Iterators and Encapsulation» (https://oreil.ly/fganK) описывается паттерн «Внешний итератор». Это объектно ориентированная версия описанной здесь идеи.
- В книге Mark Allen Weiss «Data Structures and Problem Solving Using Java» (Addison-Wesley, 2006) описан этот подход, который назван «доступом с массивоподобным интерфейсом».
- Функция `service_response_time_get_column_name` в коде Wireshark возвращает имя столбца в статистической таблице. Имя адресуется индексом, предоставленным пользователем. Имена столбцов не могут изменяться во время выполнения, поэтому даже в многопоточном окружении такой способ итерирования по столбцам и доступа к данным безопасен.
- В проекте Subversion есть код, служащий для построения таблицы строк. К строкам можно обращаться с помощью функции `svn_fs_x__string_`

`table_get`, которая принимает индекс, адресующий нужную строку. Полученная строка копируется в предоставленный буфер.

- Функция `TXT_DB_get_by_index` в библиотеке OpenSSL возвращает строку, выбранную по индексу из текстовой базы данных, сохраняя ее в предоставленном буфере.

Применение к сквозному примеру

Теперь у вас есть чистая абстракция для чтения логинов, и вы не раскрываете пользователям никаких внутренних деталей реализации.

```
char* getLoginName(int index)
{
 return accountData[index].loginname;
}
```

Пользователям не нужно возиться с доступом к массиву структур. А дополнительное преимущество в том, что доступ к нужным данным стал проще, и пользователь не сможет воспользоваться не предназначенной для него информацией. Например, он не сможет обратиться к другим элементам структуры, которые могут измениться в будущем, но только в том случае, когда их никто не использует, потому что вы не хотите «поломать» пользовательский код.

Пользователь этого интерфейса, например желающий написать функцию, которая проверяет, начинается ли логин с буквы X, поступает следующим образом:

```
bool anyoneWithX()
{
 int i;
 for(i=0; i<MAX_USERS; i++)
 {
  char* loginName = getLoginName(i);
  if(loginName[0] == 'X')
  {
   return true;
  }
 }
 return false;
}
```

Эта реализация вас устраивала до тех пор, пока структура данных, в которой хранятся логины, не изменилась, поскольку вам понадобился более удобный способ вставлять и удалять данные об учетных записях – операции, которые для массивов крайне неэффективны. Теперь логины хранятся не в простом массиве, а в структуре данных, которая предоставляет операцию перехода к следующему элементу, но не позволяет произвольный доступ к элементам. Точнее говоря, это связанный список, обращения к которому производятся, как показано в коде ниже.

```
struct ACCOUNT_NODE
{
  char loginname[MAX_NAME_LENGTH];
  char password[MAX_PWD_LENGTH];
  struct ACCOUNT_NODE* next;
};

struct ACCOUNT_NODE* accountList;

struct ACCOUNT_NODE* getFirst()
{
 return accountList;
}

struct ACCOUNT_NODE* getNext(struct ACCOUNT_NODE* current)
{
 return current->next;
}

void accessData()
{
 struct ACCOUNT_NODE* account = getFirst();
 char* loginname = account->loginname;
 account = getNext(account);
 loginname = account->loginname;
 ...
}
```

При нашем текущем интерфейсе сделать это сложно, потому что он допускает только произвольный доступ к логину по индексу. Чтобы поддержать новое требование, нужно эмулировать индекс путем вызова функции `getNext` столько раз, сколько необходимо, чтобы добраться до элемента с указанным индексом. Но это очень неэффективно. И вся эта неурядица случилась потому, что вы спроектировали интерфейс, оказавшийся недостаточно гибким. Чтобы решить проблему, предоставьте для доступа к логинам «Курсор».

Курсор

Контекст

Имеется множество элементов в структуре данных, к которой можно обращаться произвольно или последовательно. Например, это может быть массив, связанный список, хеш-таблица или дерево. Пользователь хочет обойти элементы.

Проблема

Вы хотите предоставить пользователю интерфейс итерирования, который обеспечил бы стабильную работу даже в случае, если элементы изменяются во время обхода,

и который позволил бы вам изменять внутреннее устройство структуры данных, не требуя внесения изменений в пользовательский код.

Пользователем может быть любой, кто пишет код, который не выпускается синхронно с вашей кодовой базой, поэтому вы должны гарантировать, что будущие версии вашей реализации смогут работать с пользовательским кодом, написанным для текущей версии. Таким образом, у пользователя не должно быть доступа к внутренним деталям реализации, например прямого доступа к структуре данных, в которой вы храните элементы, потому что эти детали могут измениться в будущем.

Кроме того, при работе в многопоточном окружении вы хотите, чтобы интерфейс обеспечивал стабильное и четко определенное поведение в случае, когда содержимое элемента изменяется в процессе итерирования. Даже имея дело со сложными данными, например строками, пользователь не должен думать о том, что другие потоки могут изменять данные, пока он их читает.

Вас не пугают дополнительные усилия, которые придется приложить, чтобы все это реализовать, потому что у вашего кода много пользователей, и если удастся переместить трудности из кода пользователя в ваш код, то общий объем усилий уменьшится.

Решение

Создайте экземпляр итератора, который указывает на элемент в вашей структуре данных. Функция итерирования принимает этот экземпляр, извлекает элемент, на который он указывает, и модифицирует экземпляр, так чтобы он указывал на следующий элемент. Пользователь итеративно вызывает эту функцию для получения одного элемента за другим, как показано на рис. 7.3.

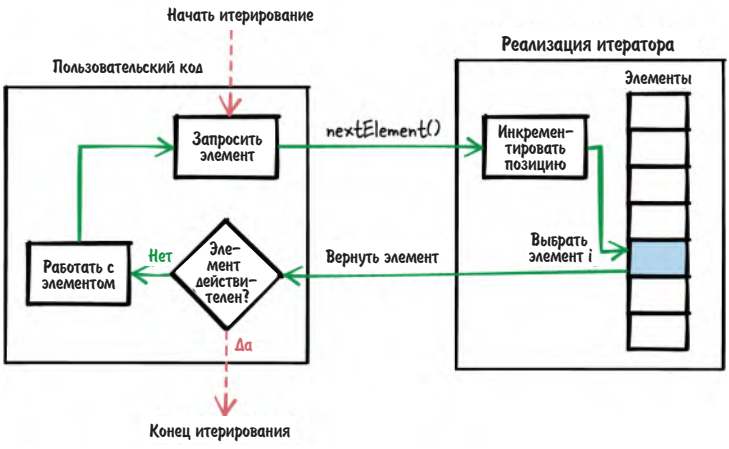


Рис. 7.3. Итерирование с помощью курсора

В интерфейсе итератора должно быть две функции для создания и уничтожения экземпляра итератора и еще одна для собственно итерирования и получения текущего элемента. Наличие явных функций создания и уничтоже-

ния позволяет иметь экземпляр, в котором хранится внутреннее состояние итерирования (позиция, данные текущего элемента). Пользователь должен передавать этот экземпляр всем функциям итерирования, как показано в коде ниже.

Код вызывающей стороны

```
void* element;
ITERATOR* it = createIterator();

while(element = getNext(it))
{
  /* работать с элементом */
}

destroyIterator(it);
```

API итератора

```
/* Создает итератор и устанавливает его на первый элемент */
ITERATOR* createIterator();

/* Возвращает элемент, на который указывает итератор, и устанавливает
   итератор на следующий элемент. Возвращает NULL, если элемента
   не существует. */
void* getNext(ITERATOR* iterator);

/* Уничтожает итератор, созданный функцией createIterator() */
void destroyIterator(ITERATOR* iterator),
```

Если вы не хотите, чтобы пользователь мог получить доступ к внутренним данным итератора, то можете скрыть их и предоставить пользователю «Описатель». Тогда, даже если эти внутренние данные изменятся, на пользователе это не отразится.

При получении текущего элемента данные примитивных типов можно вернуть непосредственно в виде «Возвращаемого значения». Сложные данные можно возвращать по ссылке или копировать в экземпляр итератора. Последний способ хорош тем, что данные остаются согласованными, даже если в процессе итерирования структура данных изменяется (например, потому что ее модифицировали из другого потока).

Последствия

Пользователь может обойти данные, просто вызывая функцию `getNext`, пока остаются элементы. Ему не нужно иметь дела с внутренней структурой, из которой берутся данные, и не нужно думать об индексах элементов и о том, сколько их всего. Но отсутствие возможности индексировать элементы означает также, что нельзя обратиться к произвольному элементу (как в случае доступа по индексу).

Даже если структура данных изменится, например при переходе от связанного списка к структуре с произвольным доступом, например массиву, это изменение можно будет скрыть в реализации итератора, и пользователю не придется ни изменять, ни перекомпилировать свой код.

Неважно, к каким данным обращается пользователь – простым или сложным, – он может не опасаться, что полученный элемент станет недействительным, если в процессе доступа исходный элемент изменится или будет удален. Но для этого пользователю теперь приходится явно вызывать функции создания и уничтожения итератора. По сравнению с «Доступом по индексу» нужно больше вызовов.

При работе с множеством элементов пользователю часто нужно обходить их. Если тем временем кто-то другой добавит элемент в структуру, то пользователь может пропустить этот элемент в процессе обхода. Если вы считаете это проблемой и хотите гарантировать, что элементы вообще не изменяются во время обхода, то проще использовать паттерн «Итератор обратного вызова».

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В статье James Noble «Iterators and Encapsulation» (https://oreil.ly/fganK) описывается объектно ориентированная версия этого итератора под названием «Волшебный кук» (Magic Cookie).
- В статье Jed Liu et al. «Interruptible Iterators» (https://oreil.ly/BxFJJ) описанная здесь идея называется *объектом курсора*.
- Такой вид итерирования применяется для доступа к файлам. Например, функция `getline` обходит строки в файле, а позиция итератора хранится в структуре `FILE`.
- В библиотеке OpenSSL имеются функции `ENGINE_get_first` и `ENGINE_get_next` для обхода списка движков шифрования. Оба принимают указатель на структуру `ENGINE`, в которой хранится текущая позиция итерирования.
- В коде Wireshark имеются функции `proto_get_first_protocol` и `proto_get_next_protocol` для обхода списка сетевых протоколов. Они принимают указатель на `void` в качестве выходного параметра, в котором сохраняется информация о состоянии.
- В проекте Subversion для генерирования дельты между файлами применяется функция `datasource_get_next_token`. Ее нужно вызывать в цикле, чтобы получить следующую дельту от переданного объекта источника данных, который хранит позицию итерирования.

Применение к сквозному примеру

Теперь у вас есть следующая функция для получения логинов:

```
struct ITERATOR
{
  char buffer[MAX_NAME_LENGTH];
```

```
  struct ACCOUNT_NODE* element;
};

struct ITERATOR* createIterator()
{
 struct ITERATOR* iterator = malloc(sizeof(struct ITERATOR));
 iterator->element = getFirst();
 return iterator;
}

char* getNextLoginName(struct ITERATOR* iterator)
{
 if(iterator->element != NULL)
 {
  strcpy(iterator->buffer, iterator->element->loginname);
  iterator->element = getNext(iterator->element);
  return iterator->buffer;
 }
 else
 {
  return NULL;
 }
}

void destroyIterator(struct ITERATOR* iterator)
{
 free(iterator);
}
```

Ниже показано, как этот интерфейс используется:

```
bool anyoneWithX()
{
 char* loginName;
 struct ITERATOR* iterator = createIterator();
 while(loginName = getNextLoginName(iterator)) ❶
 {
  if(loginName[0] == 'X')
  {
   destroyIterator(iterator); ❷
   return true;
  }
 }
 destroyIterator(iterator);
 return false;
}
```

❶ Приложению больше не приходится иметь дело с индексом и максимальным числом элементов.
❺ В данном случае необходимость уничтожить итератор приводит к дублированию кода.

Следующее требование – вы хотите реализовать не только функцию `anyoneWithX`, но еще и функцию, которая, к примеру, сообщает, сколько логинов начинается с буквы Y. Можно было бы просто скопировать код, модифицировать тело цикла `while` и подсчитать количество вхождений Y, но такой подход ведет к дублированию кода, потому что обе функции содержат один и тот же код создания и уничтожения итератора и организации цикла. Чтобы избавиться от дублирования, можно использовать паттерн «Итератор обратного вызова».

Итератор обратного вызова

Контекст

Имеется множество элементов в структуре данных, к которой можно обращаться произвольно или последовательно. Например, это может быть массив, связанный список, хеш-таблица или дерево. Пользователь хочет обойти элементы.

Проблема

Вы хотите предоставить стабильный интерфейс итерирования, который не требовал бы от пользователя реализации цикла для обхода всех элементов и который позволил бы вам менять внутреннюю структуру данных в будущем, не требуя внесения изменения в пользовательский код.

Пользователем может быть любой, кто пишет код, который не выпускается синхронно с вашей кодовой базой, поэтому вы должны гарантировать, что будущие версии вашей реализации смогут работать с пользовательским кодом, написанным для текущей версии. Таким образом, у пользователя не должно быть доступа к внутренним деталям реализации, например прямого доступа к структуре данных, в которой вы храните элементы, потому что эти детали могут измениться в будущем.

Кроме того, при работе в многопоточном окружении вы хотите, чтобы интерфейс обеспечивал стабильное и четко определенное поведение в случае, когда содержимое элемента изменяется в процессе итерирования. Даже имея дело со сложными данными, например строками, пользователь не должен думать о том, что другие потоки могут изменять данные, пока он их читает. Также необходимо гарантировать, что пользователь видит каждый элемент ровно один раз. И это должно быть так, даже если другие потоки пытаются создавать новые или удалять существующие элементы во время обхода.

Вас не пугают дополнительные усилия, которые придется приложить, чтобы все это реализовать, потому что у вашего кода много пользователей, и если удастся переместить трудности из кода пользователя в ваш код, то общий объем усилий уменьшится.

Вы хотите сделать доступ к элементам как можно проще. В частности, пользователя не должны волновать такие детали, как отображение между индексом

и элементом или количество элементов в структуре. И еще у пользователя не должно быть необходимости реализовать цикл обхода в своем коде, потому что это ведет к дублированию кода. Таким образом, ни доступ по индексу, ни курсор вас не устраивают.

Решение

Используйте уже имеющиеся в вашей структуре операции для обхода всех элементов и вызывайте предоставленную пользователем функцию для каждого элемента в процессе обхода. Эта функция получает в качестве параметра содержимое элемента и может выполнять над ним операции. Пользователь вызывает только одну функцию, чтобы запустить обход, после чего все итерирование происходит внутри вашей реализации, как показано на рис. 7.4.

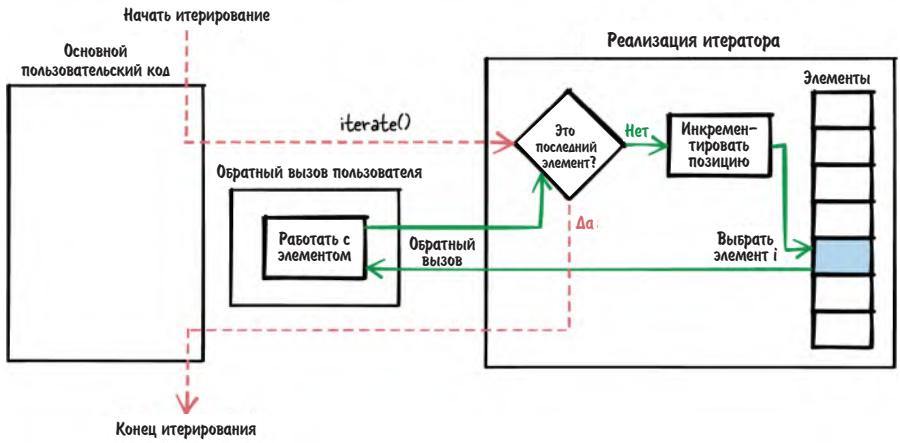


Рис. 7.4. Итерирование с помощью итератора обратного вызова

Для реализации этой идеи вы должны объявить в своем интерфейсе указатель на функцию. Эта функция принимает элемент, который следует обработать в процессе обхода. Реализует ее пользователь и передает вашей функции итерирования. Внутри своей реализации вы обходите все элементы и для каждого вызываете пользовательскую функцию, передавая ей текущий элемент в качестве параметра.

Можно еще добавить в функцию итерирования параметр типа `void*`, который она будет просто передавать пользовательской функции. Это позволит пользователю передать вашей функции некоторую информацию о контексте.

Код вызывающей стороны

```
void myCallback(void* element, void* arg)
{
 /* работать с элементом */
}
```

```
void doIteration()
{
 iterate(myCallback, NULL);
}
```

API итератора

```
/* Обратный вызов для итерирования, реализуется вызывающей стороной. */
typedef void (*FP_CALLBACK)(void* element, void* arg);

/* Обходит все элементы и вызывает callback(element, arg) для каждого элемента. */
void iterate(FP_CALLBACK callback, void* arg);
```

Иногда пользователь не хочет обходить все элементы, ему нужно просто найти один конкретный элемент. Чтобы обработать этот случай более эффективно, вы можете добавить `break` в свою функцию итерирования. Например, вы можете объявить указатель на функцию, которая возвращает значение типа `bool`, и если она вернет `true`, то прекращать обход. Тогда пользователь сможет сигнализировать о том, что искомый элемент найден, и сэкономить на обходе остальных элементов.

При реализации функции итерирования, которая может работать в многопоточном окружении, не забудьте учесть ситуацию, когда в процессе итерирования другие потоки изменяют текущий элемент, добавляют новые или удаляют существующие. В таком случае вы можете вернуть пользователю, запустившему обход, код состояния или вообще предотвратить такие изменения, заблокировав на время обхода доступ к элементам для записи.

Поскольку реализация может гарантировать, что данные не изменяются во время обхода, нет необходимости копировать элементы, с которыми работает пользователь. Пользователь просто получает указатель на сами данные и работает с ними непосредственно.

Последствия

Теперь пользовательский код обхода всех элементов состоит всего из одной строки. Все детали, например индекс элемента и максимальное число элементов, скрыты в реализации итератора. Пользователю даже не нужно реализовывать цикл для обхода элементов. Не нужно ни создавать, ни уничтожать итератор, а также иметь дело с внутренней структурой данных, в которой хранятся элементы. Даже если тип этой структуры изменится, пользователю не придется перекомпилировать свой код.

Если элементы структуры изменяются во время обхода, то реализация итератора может отреагировать соответственно, а это значит, что пользователь видит согласованные данные, не прибегая к блокировкам в своем коде. Все это возможно, потому что поток управления не переходит от пользовательского кода к коду итератора и обратно, а остается целиком внутри реализации итератора, которая может заметить, что элементы были изменены во время обхода, и предпринять нужные действия.

Пользователь может обойти все элементы, но цикл обхода реализован внутри итератора, так что пользователь не может обращаться к произвольному элементу, как в случае «Доступа по индексу».

В обратном вызове ваша реализация выполняет пользовательский код для каждого элемента. Это означает, что вы должны доверять пользовательскому коду. Например, если реализация итератора блокирует все элементы во время обхода, то вы ожидаете, что пользовательский код обработает элемент быстро и не станет выполнять длительные операции, потому что, пока обход не закончится, все остальные попытки обратиться к данным, будут блокированы.

Использование обратных вызовов означает, что интерфейс зависит от платформы и языка программирования, потому что вы вызываете код, находящийся на вызывающей стороне, а это можно сделать, только если применяются одинаковые соглашения о вызове (т. е. способы передачи параметров и возврата значений). Следовательно, при реализации итератора на C этот паттерн пригоден, только если пользовательский код тоже написан на C. Невозможно предоставить написанный на C итератор обратного вызова пользователю, пишущему на Java (другие паттерны итераторов позволяют это сделать, хотя и с трудом).

Читать код программы с обратными вызовами труднее. Так, по сравнению с простым циклом `while` прямо в коде, сложнее понять, что программа обходит элементы, видя лишь одну строку коду с параметром, содержащим указатель на функцию обратного вызова. Поэтому очень важно называть функцию так, чтобы сразу становилось ясно, что она выполняет итерирование.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В статье James Noble «Iterators and Encapsulation» (https://oreil.ly/fganK) описывается объектно ориентированная версия этого итератора под названием «Внутренний итератор».
- Функция `svn_iter_apr_hash` в проекте Subversion обходит все элементы хеш-таблицы, которая передана ей в качестве параметра. Для каждого элемента таблицы вызывается функция по указателю, переданному вызывающей стороной, и если она возвращает `SVN_ERR_ITER_BREAK`, то итерирование прекращается.
- Функция `ossl_provider_forall_loaded` из библиотеки OpenSSL обходит множество объектов-поставщиков. Она принимает указатель на функцию и вызывает эту функцию для каждого объекта. При запуске итерирования можно передать параметр типа `void*`, который затем будет передаваться при каждом обратном вызове; таким образом, пользователь может передать свой контекст процессу итерирования.
- В Wireshark функция `conversation_table_iterate_tables` обходит список объектов «диалога». В каждом таком объекте хранится информация о собранных из сети данных. Функция принимает указатель на функцию обратного вызова и параметр типа `void*`. Для каждого объекта диалога вызывается функция обратного вызова, которой этот параметр передается в качестве контекста.

Применение к сквозному примеру

Теперь вы предоставляете следующую функцию для доступа к логинам:

```
typedef void (*FP_CALLBACK)(char* loginName, void* arg);

void iterateLoginNames(FP_CALLBACK callback, void* arg)
{
 struct ACCOUNT_NODE* account = getFirst(accountList);
 while(account != NULL)
 {
  callback(account->loginname, arg);
  account = getNext(account);
 }
}
```

Ниже показано, как использовать этот интерфейс:

```
void findX(char* loginName, void* arg)
{
 bool* found = (bool*) arg;
 if(loginName[0] == 'X')
 {
  *found = true;
 }
}

void countY(char* loginName, void* arg)
{
 int* count = (int*) arg;
 if(loginName[0] == 'Y')
 {
  (*count)++;
 }
}

bool anyoneWithX()
{
 bool found=false;
 iterateLoginNames(findX, &found); ❶
 return found;
}

int numberOfUsersWithY()
{
 int count=0;
 iterateLoginNames(countY, &count);
```

```
  return count;
}
```

❶ Приложение больше не содержит явного цикла.

Функцию обратного вызова можно было бы улучшить, поручив ей возвращать значение, показывающее, продолжать или прекратить итерации. В таком случае итерации можно было бы прекратить, например, после того как findX найдет первый логин, начинающийся буквой X.

Резюме

В этой главе продемонстрировано три способа реализации интерфейса итерирования. В табл. 7.2 сравниваются последствия описанных паттернов.

Таблица 7.2. Сравнение паттернов итераторов

	Доступ по индексу	**Курсор**	**Итератор обратного вызова**
Доступ к элементам	Допускается произвольный доступ	Только последовательный доступ	Только последовательный доступ
Изменения структуры данных	Внутренняя структура данных может быть легко заменена только на другую структуру с произвольным доступом	Внутренняя структура данных может быть легко изменена	Внутренняя структура данных может быть легко изменена
Утечка информации через интерфейс	Количество элементов; факт использования структуры данных с произвольным доступом	Позиция итератора (пользователь может остановить итерирование и продолжить его позднее)	–
Дублирование кода	Цикл в пользовательском коде; инкрементирование индекса в пользовательском коде	Цикл в пользовательском коде	–

	Доступ по индексу	Курсор	Итератор обратного вызова
Стабильность	Трудно реализовать стабильное поведение итератора	Трудно реализовать стабильное поведение итератора	Реализовать стабильное поведение итератора легко, потому что поток управления остается в коде итератора, а операции вставки, удаления и изменения можно просто заблокировать на все время обхода (но при этом будут заблокированы другие обходы)
Платформы	Интерфейс можно использовать на разных языках и платформах	Интерфейс можно использовать на разных языках и платформах	Можно использовать только в пределах одного языка и платформы (с одинаковыми соглашениями о вызове)

Для дополнительного чтения

Для интересующихся читателей ниже перечислены ресурсы, которые помогут расширить знания о проектировании интерфейса итераторов.

- Наиболее близким к итераторам в C источником является онлайновая версия записок к лекциям Джеймса Эспнеса (https://oreil.ly/2fuPK). В них описываются различные способы проектирования итераторов в C, обсуждаются их преимущества и недостатки и приводятся примеры исходного кода.
- Есть наставления по итераторам для других языков программирования, но многие изложенные в них идеи применимы и к C. Например, в статье James Noble «Iterators and Encapsulation» (https://oreil.ly/GWR0F) описано восемь паттернов проектирования объектно ориентированных итераторов, в книге Mark Allen Weiss «Data Structures and Problem Solving Using Java» (Addison-Wesley, 2006) – способы проектирования итераторов в Java, а в книге Mark Jason Dominus «Higher-Order Perl» (Morgan Kaufmann, 2005) – способы проектирования итераторов в Perl.
- В статье Owen Astrachan and Eugene Wallingford «Loop Patterns» (https://oreil.ly/JsEKb) описываются паттерны реализации циклов, включены примеры кода на C++ и Java. Многие идеи применимы также к C.

- В книге David R. Hanson «C Interfaces and Implementations» (Addison-Wesley, 1996) описаны реализации на C и интерфейсы нескольких широко распространенных структур данных, в том числе связанных списков и хеш-таблиц. Разумеется, в состав интерфейсов входят функции для обхода этих структур данных.

Что дальше

Следующая глава посвящена организации файлов с кодом в больших программах. В результате применения описанных в предыдущих главах паттернов определения интерфейсов и программирования их реализаций образуется много файлов. Их необходимо организовать, чтобы обеспечить модульность крупной программы.

Глава 8

Организация файлов в модульных программах

Всякий программист, который пишет большую программу и хочет, чтобы ее было удобно сопровождать, сталкивается с вопросом: как сделать программу модульной? На самую важную часть этого вопроса, связанную с зависимостями между программными модулями, ответ дают, например, принципы проектирования SOLID, описанные в книге Robert C. Martin «Clean Code: A Handbook of Agile Software Craftsmanship» (Prentice Hall, 2008), или паттерны проектирования, описанные в книге «банды четырех» «Design Patterns: Elements of Reusable Object-Oriented Software» (Prentice Hall, 1997).

Однако стремление сделать программу модульной поднимает также вопрос о том, как организовать исходные файлы. На этот вопрос еще нет четкого ответа – отсюда и плохо структурированные кодовые базы. Добавить модульность позже трудно, потому что вы не знаете, какие файлы следовало бы разнести по разным программным модулям или разным кодовым базам. Кроме того, программисту трудно найти файлы, содержащие API, которые вы собираетесь использовать, и потому можно привнести в API зависимости, которые вам вовсе не нужны. Это проблема особенно остро стоит в языке C, потому что в нем нет никакого механизма, который позволил бы пометить, что API предназначен только для внутреннего использования, и ограничить доступ к нему.

В других языках программирования такие механизмы есть, равно как и рекомендации по структурированию файлов. Например, в языке Java имеется концепция *пакетов*. Java предлагает разработчику подразумеваемый по умолчанию способ организации классов в пакеты, а стало быть, и файлов внутри пакета. В таких же языках, как C, разработчик вынужден придумывать свой собственный подход к структурированию заголовочных файлов, содержащих объявления функций, и файлов реализации, содержащих определения этих функций.

В этой главе показано, как решать эту проблему: программистам на C предлагаются рекомендации по организации файлов, в частности заголовочных (API), так чтобы можно было разрабатывать большие модульные программы.

На рис. 8.1 показаны паттерны, рассматриваемые в этой главе, а в табл. 8.1 приведены их краткие описания.

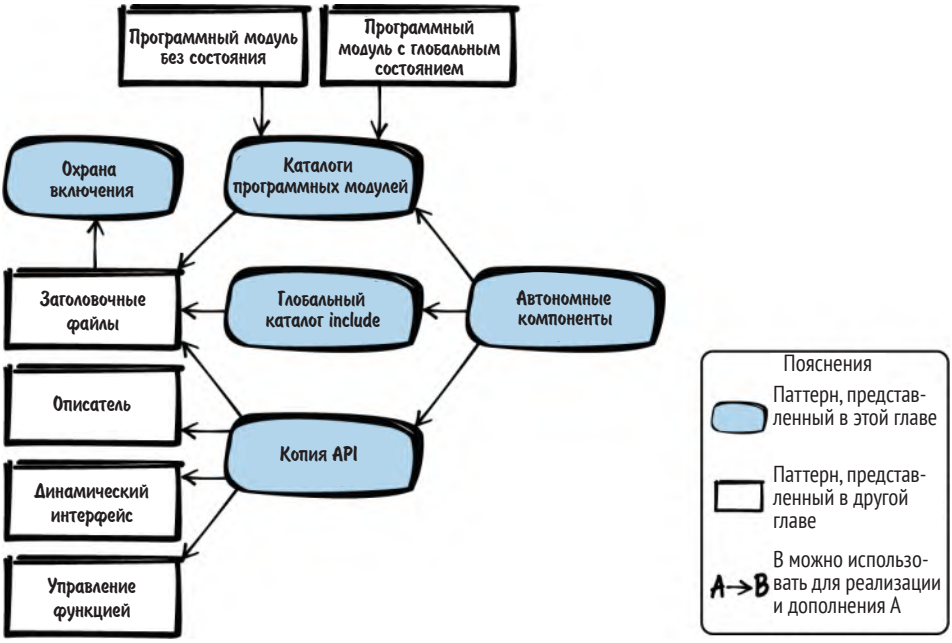


Рис. 8.1. Обзор паттернов для организации файлов с кодом

Таблица 8.1. Паттерны для организации файлов в модульных программах

Название паттерна	Краткое описание
Охрана включения	Включить один и тот же заголовочный файл легко, но это приведет к ошибкам компиляции, если в файле имеются определения типов или макросы определенного вида, поскольку в процессе компиляции они будут определены повторно. Поэтому защищайте содержимое заголовочных файлов от повторного включения, чтобы разработчику, пользующемуся ими, не нужно было думать, сколько раз включен файл. Для этого можно использовать директиву `#ifdef` или `#pragma once`
Каталоги программных модулей	Разнесение кода по нескольким файлам увеличивает количество файлов в кодовой базе. Если хранить все файлы в одном каталоге, то становится трудно обозревать их, особенно когда кодовая база велика. Поэтому помещайте тесно связанные заголовочные файлы и файлы реализации в один каталог. Назовите этот каталог, так чтобы имя отражало функциональность, предоставляемую заголовочными файлами

Название паттерна	Краткое описание
Глобальный каталог include	Чтобы включить файлы из других программных модулей, приходится использовать относительные пути вида `../othersoftwaremodule/file.h`. Вы должны знать точное местоположение других заголовочных файлов. Поэтому заведите в кодовой базе один глобальный каталог, содержащий API всех программных модулей. Добавьте этот каталог в глобальный список путей к включаемым файлам, поддерживаемый вашим инструментарием
Автономные компоненты	Структура каталогов не позволяет увидеть зависимости в коде. Любой программный модуль может просто включить заголовочные файлы из любого другого программного модуля, поэтому проверить зависимости с помощью компилятора невозможно. Поэтому идентифицируйте программные модули, которые содержат схожую функциональность и должны развертываться вместе. Поместите эти модули в общий каталог и заведите подкаталог для тех заголовочных файлов, которые нужны вызывающей стороне
Копия API	Вы хотите разрабатывать, присваивать номера версий и развертывать части кодовой базы независимо друг от друга. Однако для этого необходимо иметь четко определенные интерфейсы между частями кода и возможность хранить этот код в различных репозиториях. Поэтому, чтобы использовать функциональность другого компонента, скопируйте его API. Отдельно соберите этот компонент и скопируйте артефакты сборки и его публичные заголовочные файлы. Поместите эти файлы в каталог внутри своего компонента, сконфигурируйте этот каталог как путь к глобальному include

Сквозной пример

Вы реализуете часть программы, которая печатает хеш-значение содержимого файла. Начнем со следующего кода простой хеш-функции:

main.c

```
#include <stdio.h>

static unsigned int adler32hash(const char* buffer, int length)
{
 unsigned int s1=1;
 unsigned int s2=0;
 int i=0;

 for(i=0; i<length; i++)
 {
  s1=(s1+buffer[i]) % 65521;
```

```
    s2=(s1+s2) % 65521;
  }
  return (s2<<16) | s1;
}

int main(int argc, char* argv[])
{
  char* buffer = "Some Text";
  unsigned int hash = adler32hash(buffer, 100);
  printf("Хеш-значение: %u", hash);
  return 0;
}
```

Эта программа просто печатает хеш-значение фиксированной строки на консоль. Далее вы хотите развить этот код: читать содержимое файла и печатать его хеш-значение. Можно было бы просто добавить весь этот код в файл *main.c*, но тогда файл получился бы слишком длинным, и чем длиннее становился бы файл, тем труднее было бы его сопровождать.

Гораздо лучше завести отдельные файлы реализация, а для доступа к их функциональности – заголовочные файлы. Тогда получится следующий код для чтения содержимого файла и печати его хеш-значения. Чтобы было проще понять, какие части кода изменились, код, оставшийся без изменения, опущен:

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "hash.h"
#include "filereader.h"

int main(int argc, char* argv[])
{
  char* buffer = malloc(100);
  getFileContent(buffer, 100);
  unsigned int hash = adler32hash(buffer, 100);
  printf("Hash value: %u", hash);
  return 0;
}
```

hash.h

```
/* Возвращает хеш-значение буфера "buffer" размера "length".
   Для вычисления хеша применяется алгоритм Adler32. */
unsigned int adler32hash(const char* buffer, int length);
```

hash.c

```
#include "hash.h"
```

```
unsigned int adler32hash(const char* buffer, int length)
{
  /* этот код не изменялся */
}
```

filereader.h

```
/* Читает содержимое файла и сохраняет его в буфере "buffer",
   если его длина "length" достаточна. */
void getFileContent(char* buffer, int length);
```

filereader.c

```
#include <stdio.h>
#include "filereader.h"

void getFileContent(char* buffer, int length)
{
 FILE* file = fopen("SomeFile", "rb");
 fread(buffer, length, 1, file);
 fclose(file);
}
```

Разнесение кода по нескольким файлам сделало его более модульным, потому что теперь, когда вся связанная функциональность находится в одном файле, зависимости можно сделать явными. В настоящий момент вся кодовая база размещена в одном каталоге, как показано на рис. 8.2.

Рис. 8.2. Организация файлов

После того как выделены заголовочные файлы, вы можете включить их в свои файлы реализации. Однако очень скоро вы столкнетесь с проблемой – если заголовочный файл включается несколько раз, то возникает ошибка компиляции. Чтобы решить ее, можно организовать «Охрану включения».

Охрана включения

Контекст

В реализации выделено несколько файлов. В файл реализации включаются заголовочные файлы, чтобы иметь опережающие объявления кода, который вы хотите вызывать или использовать.

Проблема

Заголовочный файл легко случайно включить несколько раз, но это приводит к ошибкам компиляции, если в него входят определения типов или макросы, потому что в процессе компиляции они переопределяются.

В языке C директива `#include` заставляет препроцессор полностью скопировать включаемый файл в текущую единицу компиляции. Если, например, в заголовочном файле определена структура `struct` и этот файл включен несколько раз, то эта структура будет несколько раз скопирована в единицу компиляции, что приведет к ошибке.

Чтобы избежать этого, нужно постараться не включать файлы более одного раза. Однако при включении заголовочного файла вы обычно не знаете, включены ли в него еще какие-то заголовочные файлы. И потому непреднамеренно включить один и тот же файл дважды очень просто.

Решение

Защитите содержимое заголовочных файлов от повторного включения, так чтобы разработчику, использующему заголовочные файлы, не нужно было думать о том, сколько раз он включен. Для этого служат директивы `#ifdef` и `#pragma once`.

В примере ниже показано, как применяется паттерн «Охрана включения».

somecode.h

```
#ifndef SOMECODE_H
#define SOMECODE_H
 /* здесь должно быть содержимое заголовочного файла */
#endif
```

othercode.h

```
#pragma once
 /* здесь должно быть содержимое заголовочного файла */
```

В процессе сборки директивы `#ifdef` и `#pragma once` защищают содержимое заголовочного файла от повторного включения в одну единицу компиляции.

Директива `#pragma once` не определена в стандарте C, но поддерживается большинством препроцессоров. Тем не менее имейте в виду, что при переходе на другой комплект инструментов с ней могут возникнуть проблемы.

Директиву `#ifdef` поддерживают все препроцессоры C, но возникает другая трудность: необходимо использовать уникальное имя макроса. Обычно применяется схема именования, при которой имя макроса связано с именем заголовочного файла, но тогда могут возникать несогласованности, если вы переименуете файл, но забудете переименовать охранный макрос. Кроме того, возможны конфликты имен при использовании стороннего кода. Всех этих проблем можно избежать, если вместо имени заголовочного файла исполь-

зовать какое-нибудь другое уникальное имя, например текущую временную метку[1] или UUID.

Последствия

Будучи разработчиком, который включает заголовочные файлы, вы не хотите думать о том, может ли какой-то файл быть включен несколько раз. Это намного упростило бы жизнь, особенно когда имеются вложенные директивы `#include`, потому что трудно узнать точно, какие именно файлы уже включены.

Приходится использовать либо нестандартную директиву `#pragma once`, либо придумывать схему образования уникальных имен макросов в директивах `#ifdef`. Хотя в большинстве случаев имена, основанные на именах файлов, годятся, возможны проблемы, когда точно такие же имена используются в стороннем коде. Кроме того, возможны несогласованности, когда вы переименовываете свои собственные файлы, но забываете переименовать макрос. Тут, впрочем, выручают некоторые IDE, которые сами присваивают имя макросу, охраняющему включение, при создании нового заголовочного файла и модифицируют его при переименовании файла.

Директивы `#ifdef` предотвращают ошибки компиляции при многократном включении заголовочного файла, но не предотвращают многократное открытие и копирование включаемого файла в единицу компиляции. Это ненужное действие, увеличивающее время компиляции, можно оптимизировать. Один из возможных подходов – окружить каждое предложение `#include` дополнительной охраной включения, но тогда включение файлов становится более громоздким. Кроме того, при работе с большинством современных компиляторов это вообще не нужно, потому что они самостоятельно оптимизируют процесс компиляции (например, кешируют содержимое заголовочных файлов или запоминают, какие файлы уже были включены).

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Этот паттерн используется практически в любой программе на C, содержащей более одного файла.
- В книге John Lakos «Large-Scale C++ Software Design» (Addison-Wesley, 1996) описано, как оптимизировать производительность охраны включения путем включения дополнительных директив вокруг каждой директивы `#include`.
- В портлендском репозитории паттернов описан паттерн «Охрана включения», а также паттерн для оптимизации времени компиляции путем дополнительной охраны вокруг каждой директивы `#include`.

Применение к сквозному примеру

В следующем примере паттерн «Охрана включения» гарантирует, что, даже если заголовочный файл включен дважды, ошибки компиляции не возникнет.

[1] Вряд ли текущая временная метка может надежно защитить от конфликтов имен со сторонним кодом. – *Прим. перев.*

hash.h

```
#ifndef HASH_H
#define HASH_H
/* Возвращает хеш-значение буфера "buffer" размера "length".
   Для вычисления хеша применяется алгоритм Adler32. */
#endif
```

filereader.h

```
#ifndef FILEREADER_H
#define FILEREADER_H
/* Читает содержимое файла и сохраняет его в буфере "buffer",
   если его длина "length" достаточна. */
void getFileContent(char* buffer, int length);
#endif
```

На следующем шаге совершенствования кода вы хотите также печатать хеш-значение, вычисленное другой хеш-функцией. Просто добавить еще один файл `hash.c` с другой хеш-функцией не получится, потому что имена файлов должны быть уникальны. Можно было бы, конечно, назвать новый файл по-другому. Но тогда вас вряд ли обрадовала бы ситуация, при которой в одном каталоге скапливается все больше и больше файлов, из-за чего трудно понять, что с чем связано. Чтобы исправить дело, можно воспользоваться паттерном «Каталоги программных модулей».

Каталоги программных модулей

Контекст

Вы разбиваете исходный код на несколько файлов реализации и описываете их функциональность с помощью заголовочных файлов. В кодовой базе становится все больше и больше файлов.

Проблема

Разбиение кода на файлы увеличивает количество файлов в кодовой базе. Если все файлы находятся в одном каталоге, то в них трудно разобраться, особенно когда кодовая база велика.

Но если помещать файлы в разные каталоги, то возникает вопрос, какой куда. Должно быть легко найти файлы, которые логически связаны друг с другом, и должно быть легко решить, когда помещать дополнительные файлы, если потребуется.

Решение

Помещайте заголовочные файлы и файлы реализации, относящиеся к одной и той же тесно связанной функциональности, в один каталог. Называйте этот каталог, ориентируясь на функциональность, описанную в заголовочных файлах.

Далее каталог и его содержимое мы будем называть *программным модулем*. Часто программный модуль содержит весь код, предоставляющий операции над экземплярами, представленными «Описателями». В таком случае программный модуль является не объектно ориентированным аналогом объектно ориентированного класса. Хранение всех файлов программного модуля в одном каталоге эквивалентно хранению всех файлов, относящихся к классу, в одном каталоге.

Теоретически программный модуль может содержать один заголовочный файл и один файл реализации. Главный критерий помещения файлов в один каталог – высокая сцепленность файлов в каталоге и низкая связанность с другими каталогами программных модулей.

Когда имеются заголовочные файлы, используемые только внутри программного модуля, и заголовочные файлы, используемые вне него, называйте файлы так, чтобы было понятно, какие из них не должны использоваться вне модуля (например, используйте суффикс *internal*, как показано на рис. 8.3 и в коде ниже).

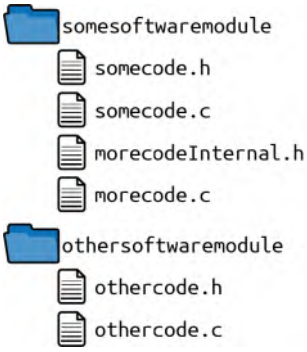


Рис. 8.3. Организация файлов

somecode.c

```
#include "somecode.h"
#include "morecode.h"
#include "../othersoftwaremodule/othercode.h"
...
```

morecode.c

```
#include "morecodeInternal.h"
...
```

othercode.c

```
#include "othercode.h"
...
```

В примере выше показано, как файлы включаются, но не показана их реализация. Отметим, что файлы из того же программного модуля включить легко. А чтобы включить заголовочные файлы из других программных модулей, нужно знать пути к этим модулям.

Когда файлы распределены по нескольким каталогам, необходимо настроить инструменты, так чтобы все эти файлы компилировались. Быть может, ваша IDE автоматически компилирует все файлы в подкаталогах кодовой базы, а быть может, необходимо вручную задать параметры или изменить файлы Makefile, чтобы файлы из новых каталогов компилировались.

Конфигурирование каталогов включаемых файлов и самих файлов для успешной компиляции

Современные IDE для программирования на C обычно предоставляют среду, в которой программист может сосредоточиться только на программировании и не думать о процедуре сборки. В таких IDE имеются параметры сборки, которые позволяют легко настроить, какие каталоги содержат включаемые файлы. А программисту остается писать код, а не файлы Makefile и команды компилятора. В этой главе предполагается, что у вас именно такая IDE, поэтому на синтаксисе Makefile мы не останавливаемся.

Последствия

Разнесение файлов с кодом по разным каталогам позволяет иметь одноименные файлы. Это удобно, когда используется сторонний код, потому что в нем могут оказаться файлы с такими же именами, как в вашей кодовой базе.

Однако называть файлы одинаково, даже когда они находятся в разных каталогах, все равно не рекомендуется. Особенно это относится к заголовочным файлам: уникальные имена нужны, чтобы сам факт включения файла не зависел от порядка поиска в каталогах включаемых файлов. Чтобы сделать имена файлов уникальными, снабжайте все файлы одного программного модуля коротким уникальным префиксом.

Когда все файлы, относящиеся к программному модулю, находятся в одном каталоге, становится проще искать логически связанные файлы, коль скоро известно имя модуля. Количество файлов в одном программном модуле обычно невелико, так что пробежать глазами все файлы в каталоге можно быстро.

Большинство зависимостей в коде сосредоточены внутри программного модуля, так что теперь в одном каталоге находятся тесно связанные файлы. Поэтому программистам, пытающимся разобраться в какой-то части кода, гораздо проще понять, какие еще файлы имеют к ней отношение. Все файлы реализации, находящиеся вне каталога программного модуля, обычно несущественны для понимания функциональности этого модуля.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В исходном коде Git часть кода находится в каталогах, а остальной код включает соответствующие заголовочные файлы по относительным путям. Например, файл *kwset.c* включает заголовок *compat/obstack.h*.
- В системе мониторинга и визуализации сетевых данных в реальном времени Netdata файлы с кодом распределены по каталогам типа *database* или *registry*, в каждом из которых находится несколько файлов. Для включения файлов из других каталогов используются относительные пути.
- Программа анализа сети Nmap организует программные модули в виде каталогов, например *ncat* и *ndiff*. Для включения заголовочных файлов из других модулей используются относительные пути.

Применение к сквозному примеру

Код почти не изменился. Добавлен новый заголовочный файл и новый файл реализации, содержащие еще одну хеш-функцию. Положение файлов изменилось, о чем можно судить по путям к включаемым файлам. Помимо разнесения файлов по каталогам, их имена также изменены, чтобы обеспечить уникальность.

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "adler/adlerhash.h"
#include "bernstein/bernsteinhash.h"

#include "filereader/filereader.h"
int main(int argc, char* argv[])
{
 char* buffer = malloc(100);
 getFileContent(buffer, 100);

 unsigned int hash = adler32hash(buffer, 100);
 printf("Хеш-значение Adler32: %u", hash);

 unsigned int hash = bernsteinHash(buffer, 100);
 printf("Хеш-значение Bernstein: %u", hash);

 return 0;
}
```

bernstein/bernsteinhash.h

```
#ifndef BERNSTEINHASH_H
```

```
#define BERNSTEINHASH_H
/* Возвращает хеш-значение буфера "buffer" размера "length".
   Для вычисления хеша применяется алгоритм Д. Дж. Бернштейна. */
unsigned int bernsteinHash(const char* buffer, int length);
#endif
```

bernstein/bernsteinhash.c

```
#include "bernsteinhash.h"

unsigned int bernsteinHash(const char* buffer, int length)
{
 unsigned int hash = 5381;
 int i;
 for(i=0; i<length; i++)
 {
  hash = 33 * hash ^ buffer[i];
 }
 return hash;
}
```

Разнесение файлов с кодом по разным каталогам – широко распространенная практика. Это упрощает поиск нужного файла и позволяет иметь файлы с одинаковыми именами. Но все равно лучше присваивать файлам уникальные имена, например, добавляя уникальный префикс, зависящий от программного модуля. Тогда структура каталогов и файлов получается такой, как на рис. 8.4.

Рис. 8.4. Организация файлов

Все файлы, логически связанные друг с другом, теперь находятся в одном каталоге. Каталоги способствуют хорошей организации файлов, а к заголовочным файлам из других каталогов можно обращаться по относительным путям.

Однако относительные пути порождают проблему: если вы захотите переименовать какой-нибудь каталог, то должны будете модифицировать и другие файлы – исправить пути в них. Такая зависимость нежелательна, но от нее можно избавиться, прибегнув к паттерну «Глобальный каталог include».

Глобальный каталог include

Контекст

Имеются заголовочные файлы, код структурирован в соответствии с паттерном «Каталоги программных модулей».

Проблема

Чтобы включить файлы из других программных модулей, вам необходимо использовать относительные пути вида *../othersoftwaremodule/file.h*. Вы должны знать точное положение включаемого заголовочного файла.

Если путь к включаемому заголовочному файлу изменится, то придется изменить код, включающий этот файл. Например, если другой программный модуль переименован, то вы должны будете модифицировать свой код. Поэтому возникает зависимость от имени и местоположения другого программного модуля.

Будучи разработчиком, вы хотите ясно понимать, какие заголовочные файлы принадлежат API программного модуля, который вы намереваетесь использовать, а какие являются внутренними, не предназначенным для использования вне программного модуля.

Синтаксис #include

Для всех включаемых файлов допустимо также заключать имена в двойные кавычки (`#include "stdio.h"`). Такие имена большинство препроцессоров C ищут сначала по относительному пути, а если не найдут там, то в глобальных каталогах, сконфигурированных в системе и используемых комплектом инструментов. При включении файлов, не принадлежащих вашей кодовой базе, обычно употребляется синтаксис с угловыми скобками (`#include <stdio.h>`); тогда файл ищется только в глобальных каталогах. Но тот же синтаксис можно использовать и для файлов из вашей кодовой базы, если они включаются не по относительному пути.

Решение

Заведите в кодовой базе один глобальный каталог и поместите в него API всех программных модулей. Добавьте этот каталог в список путей к включаемым файлам в своем комплекте инструментов.

Оставьте все файлы реализации и все заголовочные файлы, используемые только каким-то одним программным модулем, в каталоге этого модуля. Если заголовочный файл используется еще в каком-то коде, поместите его в глобальный каталог, который обычно называется */include*, как показано на рис. 8.5 и в нижеследующем коде.

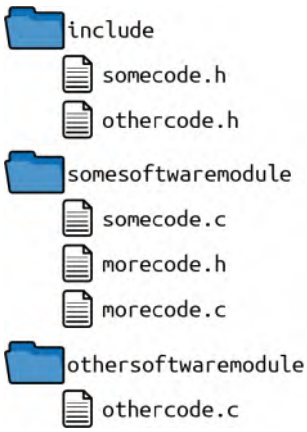


Рис. 8.5. Организация файлов

В качестве пути к глобальному каталогу включаемых файлов сконфигурирован */include*.

somecode.c

```
#include <somecode.h>
#include <othercode.h>
#include "morecode.h"
...
```

morecode.c

```
#include "morecode.h"
...
```

othercode.c

```
#include <othercode.h>
...
```

Во фрагментах кода выше показано, как включаются файлы. Отметим, что никаких относительных путей больше нет. Чтобы было понятно, какие вклю-

чаемые файлы находятся в глобальном каталоге include, их имена в директиве `#include` заключены в угловые скобки.

Глобальный каталог *include* следует конфигурировать в настройках комплекта инструментов, или если вы вручную пишете Makefile'ы и команды компилятора, то прямо в них.

Если количество заголовочных файлов в этом каталоге становится слишком велико или если имеются очень специфичные заголовочные файлы, которые используются немногими программными модулями, то следует подумать о разбиении кодовой базы на «Автономные компоненты».

Последствия

Совершенно ясно, какие заголовочные файлы предназначены для использования другими программными модулями, а какие являются внутренними для одного модуля.

Больше нет необходимости использовать относительные пути, чтобы включить файлы из других программных модулей. Но код, принадлежащий этим модулям, теперь находится не в одном каталоге, а в разных местах кодовой базы.

Когда все API помещаются в один каталог, количество файлов в этом каталоге может стать слишком большим, поэтому будет трудно найти логически связанные файлы. Будьте осторожнее, чтобы не оказалось, что заголовочные файлы из всей кодовой базы скопились в единственном каталоге include. Это сведет на нет все преимущества каталогов программных модулей. А что вы станете делать, если программный модуль A единственный, кому нужны интерфейсы программного модуля B? В случае только что описанного решения нужно будет поместить интерфейсы программного модуля B в глобальный каталог include. Однако если больше никому эти интерфейсы не нужны, то и не стоит делать их доступными для всех. Чтобы решить эту проблему, воспользуйтесь паттерном «Автономные компоненты».

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В коде OpenSSL имеется каталог */include*, содержащий все заголовочные файлы, используемые в нескольких программных модулях.
- В игре NetHack все заголовочные файлы находятся в каталоге */include*. Файлы реализации не разнесены по программным модулям, а находятся в одном каталоге */src*.
- В коде OpenZFS для Linux имеется один глобальный каталог */include*, содержащий все заголовочные файлы. Этот каталог сконфигурирован как путь к включаемым файлам в Makefile'ах, находящихся в каталогах с файлами реализации.

Применение к сквозному примеру

Местоположение заголовочных файлов в вашей кодовой базе изменилось. Вы перенесли их в глобальный каталог include, который сконфигурировали в

комплекте инструментов. Теперь можно просто включать файлы, не указывая относительного пути. Поэтому в директивах `#include` теперь используются не двойные кавычки, а угловые скобки.

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <adlerhash.h>
#include <bernsteinhash.h>
#include <filereader.h>

int main(int argc, char* argv[])
{
 char* buffer = malloc(100);
 getFileContent(buffer, 100);
 unsigned int hash = adler32hash(buffer, 100);
 printf("Хеш-значение Adler32: %u", hash);
 hash = bernsteinHash(buffer, 100);
 printf("Хеш-значение Bernstein: %u", hash);
 return 0;
}
```

Новая организация файлов и глобальный каталог */include* показаны на рис. 8.6.

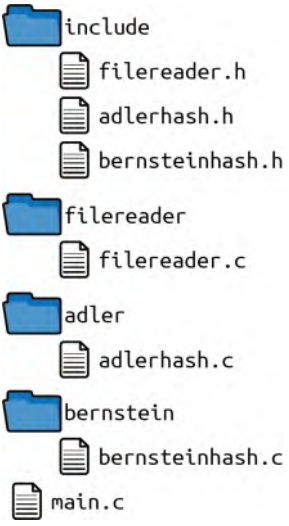


Рис. 8.6. Организация файлов

Теперь, даже если вы переименуете какой-то каталог, трогать файлы реализации не придется. То есть вы еще немного ослабили связи в реализациях.

Вы снова хотите обобщить код – использовать хеш-функции не только для хеширования содержимого файлов, но и в другом контексте, для вычисления

псевдослучайного числа на основе хеш-функции. Вы хотите, чтобы можно было разработать два приложения, в которых используются хеш-функции, причем независимо друг от друга, быть может, даже силами разных команд.

Разделять общий глобальный каталог include с другой командой разработчиков не годится, поскольку вы не хотите смешивать файлы, написанные разными командами. Вы хотите разделить приложения, насколько это возможно. Для этого организуйте их как «Автономные компоненты».

Автономный компонент

Контекст

Имеются каталоги программных модулей и, возможно, глобальный каталог include. Количество программных модулей растет, а ваш код становится больше.

Проблема

Структура каталогов не позволяет разглядеть зависимости в коде. Любой программный модуль может включить заголовочные файлы из любого другого модуля, поэтому компилятор не может автоматически проверить зависимости.

Включать заголовочные файлы можно по относительным путям, следовательно, любой программный модуль может включить заголовочные файлы из любого другого модуля.

Обозревать программные модули становится труднее с увеличением их количества. Если на предыдущем шаге в одном каталоге скопилось слишком много файлов, поэтому вы прибегли к каталогам программных модулей, то теперь слишком много стало самих каталогов

Как и случае зависимостей, не всегда возможно понять, какой код за что отвечает, глядя только на структуру кода. Если над кодом трудится несколько команд, то иногда необходимо определить, кто за какой программный модуль несет ответственность.

Решение

Определите программные модули, содержащие сходную функциональность, которые следует развертывать вместе. Поместите эти модули в общий каталог и заведите специальный подкаталог для их заголовочных файлов, представляющих интерес для вызывающей стороны.

Такую группу программных модулей вместе со всеми их заголовочными файлами будем называть *компонентом*. По сравнению с программными модулями компонент обычно больше и может быть развернут независимо от остальной кодовой базы.

Группируя программные модули, проверьте, какую часть кода можно развернуть независимо от остальной кодовой базы. Посмотрите, какие части кода разрабатываются разными командами и, следовательно, могут быть развернуты, сохраняя лишь слабую связь с остальной кодовой базой. Такие группы программных модулей будут кандидатами в компоненты.

Если имеется единственный глобальный каталог include, переместите все заголовочные файлы, принадлежащие компоненту, из этого каталога в специальный каталог внутри компонента (например, *myComponent/include*). Разработчики, которым этот компонент нужен, могут добавить этот путь в свои глобальные пути к включаемым файлам или соответственно модифицировать Makefile и команду компилятора.

Чтобы проверить, использует ли код какого-то компонента только ту функциональность, которая ему разрешена, можно воспользоваться комплектом инструментов. Например, если имеется компонент, абстрагирующий операционную систему, то может быть желательно, чтобы весь остальной код пользовался этой абстракцией и не вызывал функции конкретной операционной системы. Комплект инструментов можно настроить так, что пути к включаемым файлам, где объявлены функции операционной системы, будут доступны только абстрагирующему ОС компоненту. Всему остальному коду будет доступен только каталог с интерфейсом вашей абстракции. Тогда неопытный разработчик, который не знает о сущствовании абстракции операционной системы и попытается использовать функции ОС напрямую, вынужден будет прибегнуть к относительным путям к объявлению этих функций, иначе код не откомпилируется (и, хочется надеяться, это подскажет разработчику, что так делать не надо).

На рис. 8.7 и в следующем за ним коде показана файловая структура и пути к каталогам включаемых файлов.

Рис. 8.7. Организация файлов

Сконфигурированы следующие глобальные пути к включаемым файлам:

- */somecomponent/include*;
- */nextcomponent/include*.

somecode.c

```
#include <somecode.h>
#include <othercode.h>
#include "morecode.h"
...
```

morecode.c

```
#include "morecode.h"
...
```

othercode.c

```
#include <othercode.h>
...
```

nextcode.c

```
#include <nextcode.h>
#include <othercode.h> // использовать API другого компонента
...
```

Последствия

Программные модули хорошо организованы, и найти логически связанные модули стало проще. Если код разбит на компоненты правильно, то должно быть понятно, в какой компонент следует добавить новый код.

Благодаря размещению логически связанных файлов в одном каталоге становится проще конфигурировать все относящееся к этому компоненту в комплекте инструментов. Например, можно задать более строгий режим предупреждений для новых компонентов, создаваемых в кодовой базе, и автоматически проверять зависимости между кодом разных компонентов.

Когда код разрабатывается несколькими командами, каталоги компонентов упрощают задачу распределения обязанностей между командами, потому что компоненты обычно очень слабо связаны друг с другом. Даже функциональность продукта в целом может не зависеть от компонентов. Распределять обязанности проще на уровне компонентов, а не программных модулей.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В коде GCC имеются отдельные компоненты со своими каталогами, где собраны заголовочные файлы, например */libffi/include* или *libcpp/include*.

- В операционной системе RIOT каждый драйвер находится в своем каталоге. Например, в каталогах */drivers/xbee* и */drivers/soft_spi* имеются подкаталоги *include*, которые содержат все интерфейсы данного компонента.
- В системе обратного проектирования Radare имеются четко определенные компоненты, каждый со своим каталогом *include*, где находятся все его интерфейсы.

Применение к сквозному примеру

Вы добавили реализацию псевдослучайных чисел на основе одной из хеш-функций. Кроме того, вы выделили три разные части кода:

- хеш-функции;
- вычисление хеш-значения файла;
- вычисление псевдослучайного числа.

Все три части кода хорошо отделены друг от друга, их могли бы разрабатывать разные команды, и даже развертывать их можно было бы независимо.

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <adlerhash.h>
#include <bernsteinhash.h>
#include <filereader.h>
#include <pseudorandom.h>

int main(int argc, char* argv[])
{
 char* buffer = malloc(100);
 getFileContent(buffer, 100);

 unsigned int hash = adler32hash(buffer, 100);
 printf("Хеш-значение Adler32 hash value: %u", hash);

 hash = bernsteinHash(buffer, 100);
 printf("Хеш-значение hash value: %u", hash);

 unsigned int random = getRandomNumber(50);
 printf("Случайное значение: %u", random);

 return 0;
}
```

randrandomapplication/include/pseudorandom.h

```
#ifndef PSEUDORANDOM_H
#define PSEUDORANDOM_H
```

```
/* Возвращает псевдослучайное число, меньшее 'max' */
unsigned int getRandomNumber(int max);
#endif
```

randomapplication/pseudorandom/pseudorandom.c

```
#include <pseudorandom.h>
#include <adlerhash.h>

unsigned int getRandomNumber(int max)
{
 char* seed = "seed-text";
 unsigned int random = adler32hash(seed, 10);
 return random % max;
}
```

Новая структура каталогов в вашем коде показана ниже. Обратите внимание, как хорошо каждая часть кода отделена от других. Например, весь код, относящийся к хеш-функциям, находится в одном каталоге. Разработчику, использующему эти функции, будет легко найти, где находится их API – в подкаталогах *include*, как показано на рис. 8.8.

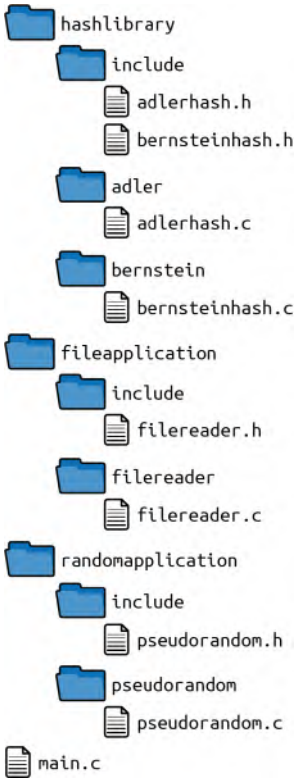


Рис. 8.8. Организация файлов

Для этого кода в комплекте инструментов сконфигурированы следующие глобальные каталоги include:

- */hashlibrary/include*;
- */fileapplication/include*;
- */randomapplication/include*.

Теперь код разнесен по трем разным каталогам, но еще остались зависимости, которые можно было бы устранить. Взгляните на пути к включаемым файлам. Имеется одна кодовая база, и все каталоги include принадлежат ей. Однако коду хеш-функций нет никакой необходимости знать о пути к включаемым файлам кода обработки файлов.

Кроме того, вы компилируете весь код и просто компонуете все объектные файлы в один исполняемый. Однако может возникнуть желание разбить этот код на части и развернуть его независимо. Возможно, вам нужно одно приложение, которое печатает вычисленный хеш, и другое, печатающее псевдослучайное число. Эти два приложения следует развертывать независимо, но в обоих должен использоваться один и тот же код хеш-функции, который вы не хотите дублировать.

Чтобы разорвать связь между приложениями и определить способ доступа к функциональности из других частей, не разделяя внутреннюю информацию, например пути к включаемым файлам этих частей, следует завести «Копию API».

Копия API

Контекст

Имеется большая кодовая база, над которой работают разные команды. Функциональность в ней абстрагируется с помощью заголовочных файлов, организованных в каталоги программных модулей. Лучше всего, когда имеются хорошо организованные «Автономные компоненты», и их интерфейсы существуют уже некоторое время, так что вы уверены в их стабильности.

Проблема

Вы хотите разрабатывать, версионировать и развертывать части кодовой базы независимо друг от друга. Однако для этого необходимы четко определенные интерфейсы между частями кода и возможность разнести их по разным репозиториям.

Если у вас уже есть автономные компоненты, то большая часть пути пройдена. Для компонентов определены интерфейсы, и весь их код уже находится в отдельных каталогах, так что его нетрудно поместить в разные репозитории.

Но между компонентами по-прежнему существует зависимость, выражающаяся в структуре каталогов: сконфигурированный путь к включаемым файлам. Он содержит полный путь к коду другого компонента, и если, например, имя этого компонента изменится, то придется изменить сконфигурированный путь. Вам такая зависимость ни к чему.

Решение

Чтобы воспользоваться функциональностью из другого компонента, скопируйте его API. Отдельно соберите этот другой компонент и скопируйте артефакты сборки и открытые заголовочные файлы. Поместите эти файлы в каталог внутри вашего компонента и сконфигурируйте этот каталог как глобальный путь к включаемым файлам.

Копирование кода представляется плохой идеей. В общем случае так оно и есть, но здесь вы копируете только интерфейс другого компонента – объявления функций из заголовочных файлов; так что нескольких копий реализации нет. Задумайтесь, что вы делаете, когда устанавливаете стороннюю библиотеку, – вы точно так же копируете ее интерфейсы для доступа к функциональности.

Помимо копий заголовочных файлов, вам нужно использовать артефакты сборки другого компонента при сборке своего собственного. Вы могли бы версионировать и развернуть другой компонент как отдельную библиотеку, с которой компонуется ваш компонент. На рис. 8.9 и в следующем за ним файле показана организация файлов.

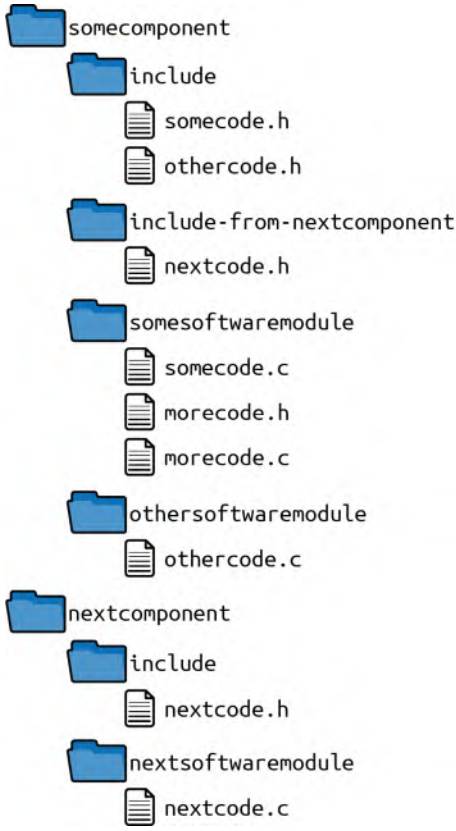


Рис. 8.9. Организация файлов

Совместимость интерфейсов

Интерфейс прикладного программирования (API) остается совместимым, если нет необходимости что-то изменять в коде вызывающей стороны. Совместимость API будет нарушена, если, например, добавить еще один параметр в существующую функцию или изменить тип возвращаемого значения или параметров.

Двоичный интерфейс приложения (ABI) остается совместимым, если нет необходимости перекомпилировать код вызывающей стороны. Совместимость ABI будет нарушена, если, например, изменить платформу, для которой откомпилирован код, или перейти на новый компилятор с другим соглашением о вызове функций.

Вот как сконфигурированы глобальные пути к включаемым файлам для `somecomponent`:

- */include*;
- */include-from-nextcomponent*.

somecode.c

```
#include <somecode.h>
#include <othercode.h>
#include "morecode.h"
...
```

morecode.c

```
#include "morecode.h"
...
```

othercode.c

```
#include <othercode.h>
...
```

А так сконфигурированы глобальные пути к включаемым файлам для `nextcomponent`:

- */include*.

nextcode.c

```
#include <nextcode.h>
...
```

Заметим, что прежний код теперь разбит на два блока. Стало возможно поместить их в разные репозитории или, иными словами, организовать две отдельных кодовых базы. Больше не осталось зависимостей между структурами каталогов двух компонентов. Однако теперь мы оказались в ситуации, когда разные версии компонентов должны обеспечить совместимость интерфейсов даже после изменения реализаций. В зависимости от стратегии развертывания вы должны определить, какой вид совместимости интерфейсов (по API или по ABI) собираетесь обеспечить. Чтобы сохранить гибкость интерфейсов, не жертвуя совместимостью, можно воспользоваться паттернами «Описатель», «Динамический интерфейс» и «Управление функцией».

Последствия

Больше не осталось структурных зависимостей между компонентами. Можно переименовать один компонент, не изменяя директив `#include` в коде других компонентов (или, как их теперь можно назвать, других кодовых баз).

Теперь код можно поместить в разные репозитории, и нет никакой необходимости знать пути к другим компонентам, чтобы включить их заголовочные файлы. Чтобы получить заголовочные файлы другого компонента, вы их копируете. То есть первоначально вы должны знать, откуда взять заголовочные файлы и артефакты сборки. Быть может, другой компонент предоставляет какой-то установщик или просто версионированный список всех необходимых файлов.

Вам необходимо соглашение о последующей совместимости интерфейсов, чтобы воспользоваться главным преимуществом разделения кодовых баз: независимой разработкой и версионированием. Требование совместимости интерфейсов ограничивает свободу разработки компонентов, предоставляющих такие интерфейсы, потому что, коль скоро функция начинает кем-то использоваться, ее уже нельзя произвольно изменять. Даже совместимые изменения, например добавление новой функции в существующий заголовочный файл, могут вызывать трудности. Дело в том, что в этом случае вы стали бы предоставлять разный набор функциональности в разных версиях заголовочного файла, и вызывающим сторонам было бы сложнее определить, какую версию они должны использовать. Кроме того, стало бы труднее писать код, работающий с любой версией вашего заголовочного файла.

Номера версий

Способ присваивания номеров версий интерфейсам должен отражать наличие несовместимых изменений. Общепринятое семантическое версионирование указывает в самом номере версии, были ли внесены какие-то крупные изменения. Семантический номер версии состоит из трех чисел (например, 1.0.7), о наличии несовместимого изменения говорит изменение первого числа.

За гибкость отдельных кодовых баз вы расплачиваетесь дополнительной сложностью: необходимостью думать о совместимости API и усложненной процедуры сборки (копирование заголовочных файлов, их синхронизация, связывание с другим компонентом, версионирование интерфейсов).

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В Wireshark копируется API независимо развертываемого компонента Kazlib, который эмулирует исключения.
- Библиотека B&R Visual Components обращается к функциональности операционной системы Automation Runtime, в которой работает. Она развертывается и версионируется независимо от Automation Runtime. Для доступа к функциональности Automation Runtime открытые заголовочные файлы последней копируются в кодовую базу Visual Components.
- Компания Education First разрабатывается цифровые обучающие программы. При сборке программы включаемые файлы копируются в глобальный каталог include, чтобы разорвать связи между компонентами в кодовой базе.

Применение к сквозному примеру

Теперь различные части кода отделены друг от друга. Реализация хеш-функций раскрывает четко определенный интерфейс к коду печати хеш-значений файлов и коду генерирования псевдослучайных чисел. Дополнительно эти части кода помещены в разные каталоги. Даже API других компонентов копируется, чтобы весь код, необходимый одному из компонентов, находился в его собственном каталоге. Код каждого компонента можно было бы хранить в отдельном репозитории и версионировать и развертывать независимо от других компонентов.

Реализации вообще не изменились. Только были скопированы API других компонентов и изменены пути к их кодовым базам, где находятся включаемые файлы. Код хеширования отделен от главного приложения. Он рассматривается как независимо развертываемый компонент и только связывается с остальным приложением. В примере 8.1 приведен код главного приложения, которое теперь отдельно от библиотеки хеширования.

Пример 8.1. Код главного приложения

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <adlerhash.h>
#include <bernsteinhash.h>
```

```
#include <filereader.h>
#include <pseudorandom.h>

int main(int argc, char* argv[])
{
 char* buffer = malloc(100);
 getFileContent(buffer, 100);

 unsigned int hash = adler32hash(buffer, 100);
 printf("Хеш-значение Adler32: %u\n", hash);

 hash = bernsteinHash(buffer, 100);
 printf("Хеш-значение hash value: %u\n", hash);

 unsigned int random = getRandomNumber(50);
 printf("Случайное значение: %u\n", random);

 return 0;
}
```

randomapplication/include/pseudorandom.h

```
#ifndef PSEUDORANDOM_H
#define PSEUDORANDOM_H
/* Возвращает псевдослучайное число, меньшее 'max' */
unsigned int getRandomNumber(int max);
#endif
```

randomapplication/pseudorandom/pseudorandom.c

```
#include <pseudorandom.h>
#include <adlerhash.h>

unsigned int getRandomNumber(int max)
{
 char* seed = "seed-text";
 unsigned int random = adler32hash(seed, 10);
 return random % max;
}
```

fileapplication/include/filereader.h

```
#ifndef FILEREADER_H
#define FILEREADER_H
/* Читает содержимое файла и сохраняет его в буфере "buffer",
   если его длина "length" достаточна. */
void getFileContent(char* buffer, int length);
#endif
```

*_fileapplication/filereader/filereader.c_*

```
#include <stdio.h>
#include "filereader.h"
void getFileContent(char* buffer, int length)
{
 FILE* file = fopen("SomeFile", "rb");
 fread(buffer, length, 1, file);
 fclose(file);
}
```

Структура каталогов и пути к включаемым файлам для этого кода показаны на рис. 8.10 и в следующем за ним примере. Отметим, что исходный код, относящийся к реализации хеш-функций, больше не является частью этой кодовой базы. Чтобы обратиться к этой функциональности, нужно включить скопированные заголовочные файлы, а затем в процессе сборки прикомпоновать файл с расширением *.a*.

Рис. 8.10. Организация файлов

Сконфигурированные пути к включаемым файлам:

- */hashlibrary*;
- */fileapplication/include*;
- */randomapplication/include*.

В примере 8.2 реализация хеш-функций теперь находится в отдельном репозитории. При любом изменении кода можно поставить новую версию библиотеки. Это значит, что объектный файл, являющийся результатом компиляции библиотеки, следует скопировать в другой код, и при условии, что API библиотеки хеширования не изменился, больше ничего делать не нужно.

Пример 8.2. Код библиотеки хеширования

inc/adlerhash.h

```
#ifndef ADLERHASH_H
#define ADLERHASH_H
/* Возвращает хеш-значение буфера "buffer" размера "length".
   Для вычисления хеша применяется алгоритм Adler32. */
unsigned int adler32hash(const char* buffer, int length);
#endif
```

adler/adlerhash.c

```
#include "adlerhash.h"
unsigned int adler32hash(const char* buffer, int length)
{
 unsigned int s1=1;
 unsigned int s2=0;
 int i=0;

 for(i=0; i<length; i++)
 {
  s1=(s1+buffer[i]) % 65521;
  s2=(s1+s2) % 65521;
 }
 return (s2<<16) | s1;
}
```

inc/bernsteinhash.h

```
#ifndef BERSTEINHASH_H
#define BERNSTEINHASH_H
/* Возвращает хеш-значение буфера "buffer" размера "length".
   Для вычисления хеша применяется алгоритм Д. Дж. Бернштейна. */
unsigned int bernsteinHash(const char* buffer, int length);
#endif
```

bernstein/bernsteinhash.c

```
#include "bernsteinhash.h"

unsigned int bernsteinHash(const char* buffer, int length)
{
```

```
    unsigned int hash = 5381;
    int i;
    for(i=0; i<length; i++)
    {
     hash = 33 * hash ^ buffer[i];
    }
    return hash;
}
```

Структура каталогов и пути к включаемым файлам для этого кода показаны на рис. 8.11 и в следующем за ним примере. Отметим, что исходный код, относящийся к обработке файлов и генерированию псевдослучайных чисел, больше не является частью этой кодовой базы. Кодовая база стала общей и может быть применена в других контекстах.

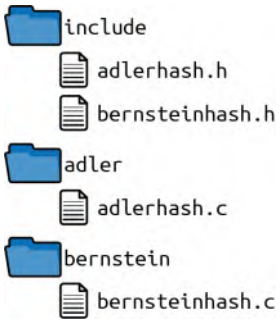


Рис. 8.11. Организация файлов

Сконфигурированные пути к включаемым файлам:

- */include*.

Начав с простого приложения хеширования, мы пришли к коду, который позволяет разрабатывать и развертывать код хеширования независимо от его приложений. Сделав еще один шаг, мы даже смогли разбить оба приложения на части, которые можно развертывать отдельно.

Организация структуры каталогов так, как предлагается в этом примере, – вовсе не самая важная проблема при создании модульного кода. Есть много более важных вопросов, которые явно не затрагивались в этой главе и в сквозном примере, например зависимости в коде – для решения этого вопроса служат принципы SOLID. Однако, коль скоро вопрос с зависимостями решен и код является модульным, показанная в этом примере структура каталогов упрощает разделение владения кодом, а заодно позволяет версионировать и развертывать его независимо от других частей кодовой базы.

Резюме

В этой главе представлены паттерны, описывающие как структурировать исходные и заголовочные файлы для построения крупных модульных программ на C.

Паттерн «Охрана включения» гарантирует, что заголовочный файл не будет включен несколько раз. Паттерн «Каталоги программных модулей» рекомендует помещать все файлы, относящиеся к программному модулю, в один каталог. Паттерн «Глобальный каталог include» рекомендует помещать все заголовочные файлы, используемые несколькими программными модулями, в один глобальный каталог. Для более крупных программ паттерн «Автономный компонент» рекомендует вместо этого заводить один глобальный каталог заголовочных файлов для каждого компонента. Чтобы разорвать связи между этими компонентами, паттерн «Копия API» рекомендует копировать заголовочные файлы и артефакты сборки, используемые в других компонентах.

Представленные паттерны в какой-то степени надстроены один над другим. Последующие паттерны легче применить, если уже применены предыдущие. После применения всех паттернов к кодовой базе она достигнет высокого уровня гибкости, позволяющего разрабатывать и развертывать ее части независимо. Однако такая гибкость не всегда необходима и обходится недаром: каждый паттерн увеличивает сложность кодовой базы. В частности, для совсем небольших кодовых баз необязательно стремиться к раздельному развертыванию, поэтому, скорее всего, не нужно копировать API. Быть может, будет достаточно ограничиться применением паттернов «Заголовочные файлы» и «Охрана включения». Не нужно слепо применять все паттерны. Применяйте их только тогда, когда сталкиваетесь с описанной в паттерне проблемой, и для решения этой проблемы увеличение сложности оправдано.

Включив эти паттерны в свой арсенал, программист получает инструментарий и пошаговое наставление по построению модульных программ на C и организации их файлов.

Что дальше

В следующей главе рассматривается аспект, общий для многих крупномасштабных программ: создание многоплатформенного кода. Представлены паттерны, рекомендующие, как организовать код таким образом, чтобы была единственная кодовая база для нескольких процессорных архитектур или операционных систем.

Глава 9

Бегство из ада #ifdef

Язык C широко распространен, особенно в системах, где требуется высокая производительность или нужно программировать на уровне оборудования. В последнем случае возникает необходимость работать с разными вариантами оборудования. Помимо разнородного оборудования, некоторые программы поддерживают несколько операционных систем или версий какого-то продукта. Для решения таких проблем чаще всего применяют директивы препроцессора `#ifdef`, чтобы различать возможные варианты. Препроцессор C обладает большой силой, но эта сила должна применяться ответственно, структурированным способом.

Однако именно директивы `#ifdef` – главная слабость препроцессора. Он не поддерживает никаких механизмов, способных как-то ограничить их использование. Это печально, потому что употребить их во вред ничего не стоит. Очень просто добавить в код еще один вариант оборудования или еще одну необязательную возможность – нужно лишь включить лишний `#ifdef`. Кроме того, директивами `#ifdef` легко злоупотребить для быстрого исправления ошибок, затрагивающих только один вариант. Это делает код с несколькими вариантами более разветвленным, и в итоге получается, что вносить исправления для каждого варианта нужно по отдельности.

Такое неструктурированное и ситуативное использование директив `#ifdef` – прямая дорога в ад. Код становится нечитаемым и непригодным для сопровождения, а это как раз то, чего разработчики должны всеми силами избегать. В этой главе описываются подходы к выходу из такой ситуации или ее предотвращению.

Приводятся подробные наставления по реализации вариантов, будь то варианты операционной системы или оборудования, в коде на C. Обсуждается пять паттернов, рекомендующих, как организовать или даже вовсе избавиться от директив `#ifdef`. Эти паттерны можно рассматривать как введение в организацию подобного кода или как руководство по рефакторингу неструктурированного кода, содержащего `#ifdef`.

На рис. 9.1 показано, как выбраться из кошмара `#ifdef`, а в табл. 9.1 приведен краткий обзор паттернов, обсуждаемых в этой главе.

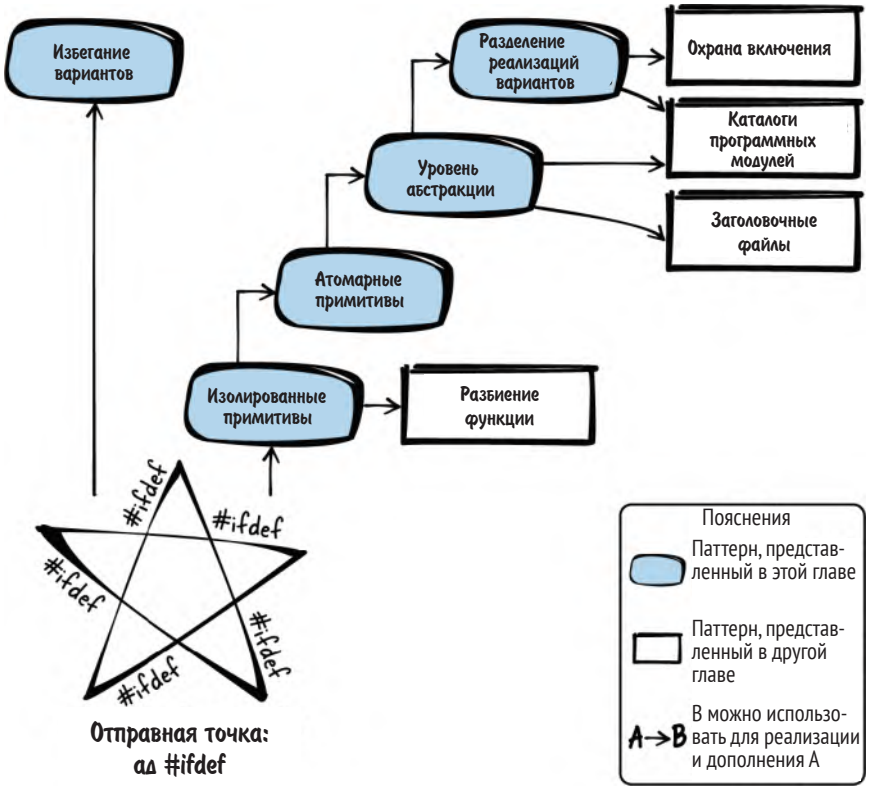


Рис. 9.1. Выход из ада `#ifdef`

Таблица 9.1. Паттерны, позволяющие избежать ада `#ifdef`

Название паттерна	Краткое описание
Избегание вариантов	Использование разных функций для каждой платформы затрудняет чтение и написание кода. От программиста требуется понимать, правильно использовать и тестировать эти многочисленные функции, чтобы обеспечить одинаковую функциональность на нескольких платформах. Поэтому пользуйтесь стандартизованными функциями, имеющимися на всех платформах. Если стандартизованных функций нет, то подумайте, не стоит ли отказаться от соответствующей функциональности
Изолированные примитивы	Варианты кода, организованные с помощью директив `#ifdef`, делают код нечитаемым. Очень трудно следить за потоком программы, который реализован несколько раз для разных платформ. Поэтому изолируйте свои варианты кода. В файле реализации поместите многовариантный код в отдельные функции и вызывайте эти функции из основной программы, которая таким образом будет содержать только платформенно независимый код

Название паттерна	Краткое описание
Атомарные примитивы	Функцию, которая содержит варианты и вызывается из основной программы, все равно трудно понять, потому что код, содержащий многочисленные `#ifdef`, был помещен туда только для того, чтобы избавиться от него в основной программе, но проще он от этого не стал. Поэтому делайте примитивы атомарными. В каждой функции обрабатывайте только один вариант. Если, например, нужно обработать несколько вариантов операционных систем и оборудования, то заведите для каждого свою функцию
Уровень абстракции	Вы хотите использовать функциональность, которая отвечает за платформенные варианты в нескольких местах кодовой базы, но не хотите дублировать код этой функциональности. Поэтому предоставьте API для каждого вида функциональности, требующего платформенно зависимого кода. В заголовочном файле объявляйте только платформенно независимые функции, а весь платформенно зависимый код, заключенный внутри `#ifdef`, помещайте в файл реализации. Вызывающая сторона должна будет включить только ваш заголовочный файл, но не платформенно зависимые файлы
Разделение реализаций вариантов	Платформенно зависимые реализации по-прежнему содержат директивы `#ifdef`, чтобы различать варианты кода. Из-за этого трудно понять, какую часть кода следует собирать для какой платформы. Поэтому помещайте реализацию каждого варианта в отдельный файл и пофайлово выбирайте, что хотите компилировать для какой платформы

Сквозной пример

Пусть требуется реализовать запись текста в файл, который должен храниться во вновь созданном каталоге, находящемся в зависимости от конфигурационного флага либо в текущем, либо в домашнем каталоге. Усложним задачу – код должен работать как в Windows, так и в Linux.

Первая попытка – написать один файл реализации, содержащий весь код для всех конфигураций и всех операционных систем. При этом файл будет содержать много директив `#ifdef` для различения вариантов кода.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef __unix__
  #include <sys/stat.h>
  #include <fcntl.h>
   #include <unistd.h>
#elif defined _WIN32
  #include <windows.h>
```

```
#endif

int main()
{
 char dirname[50];
 char filename[60];
 char* my_data = "Write this data to the file";
 #ifdef __unix__
   #ifdef STORE_IN_HOME_DIR
     sprintf(dirname, "%s%s", getenv("HOME"), "/newdir/");
     sprintf(filename, "%s%s", dirname, "newfile");
   #elif defined STORE_IN_CWD
     strcpy(dirname, "newdir");
     strcpy(filename, "newdir/newfile");
   #endif
   mkdir(dirname,S_IRWXU);
   int fd = open (filename, O_RDWR | O_CREAT, 0666);
   write(fd, my_data, strlen(my_data));
   close(fd);
 #elif defined _WIN32
   #ifdef STORE_IN_HOME_DIR
     sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"), getenv("HOMEPATH"),
             "\\newdir\\");
     sprintf(filename, "%s%s", dirname, "newfile");
   #elif defined STORE_IN_CWD
     strcpy(dirname, "newdir");
     strcpy(filename, "newdir\\newfile");
   #endif
   CreateDirectory (dirname, NULL);
   HANDLE hFile = CreateFile(filename, GENERIC_WRITE, 0, NULL,
                             CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
   WriteFile(hFile, my_data, strlen(my_data), NULL, NULL);
   CloseHandle(hFile);
 #endif
 return 0;
}
```

Это не код, а хаос в чистом виде. Логика программы полностью дублируется. Это нельзя назвать кодом, не зависящим от операционной системы, на самом деле это две системно-зависимые реализации, помещенные в один файл. Особенно уродливо выглядят ортогональные варианты кода для разных операционных систем и разных мест создания каталога – из-за вложенных `#ifdef`, в которых очень трудно разобраться. При чтении кода приходится постоянно перескакивать из одного места в другое. Чтобы проследить логику программы, нужно пропустить код из других ветвей `#ifdef`. Из-за такого дублирования логики у программиста непроизвольно возникает желание исправлять ошибки или добавлять новые возможности только в тот вариант кода, с которым он

сейчас работает. В итоге поведение разных вариантов все дальше расходится, и код становится трудно сопровождать.

С чего начать? Как разгрести этот хаос? Первый шаг – всюду, где возможно, пользоваться стандартизованными функциями, чтобы избежать вариантов.

Избегание вариантов

Контекст

Вы пишете переносимый код, который должен работать на нескольких программных или аппаратных платформах. Некоторые используемые функции доступны на одной платформе, но на другой их нет вообще, или синтаксис либо семантика отличаются. Поэтому вы реализуете варианты кода – по одному для каждой платформы. А для разграничения вариантов пользуетесь директивами `#ifdef`.

Проблема

Если использовать разные функции для каждой платформы, то код становится труднее читать и писать. Программисту необходимо сначала понять, научиться правильно использовать и протестировать несколько функций, чтобы в конечном итоге обеспечить единую функциональность на нескольких платформах.

Очень часто ставится цель реализовать одинаковое поведение на всех платформах, но если используются платформенно зависимые функции, то достичь этой цели труднее, и может понадобиться писать дополнительный код. Дело в том, что на разных платформах может различаться не только синтаксис, но и – в меньшей мере – семантика.

Использование нескольких функций для нескольких платформ затрудняет написание, чтение и понимание кода. Необходимость различать функции с помощью директив `#ifdef` делает код длиннее и требует от читателя перескакивать взглядом через строки, чтобы понять, что делает код, принадлежащий одной ветви `#ifdef`.

Для любого кода, который вам предстоит написать, задайте себе вопрос, а стоит ли овчинка выделки. Если функциональность не особенно важна, а из-за использования платформенно зависимых функций реализация и поддержка этой функциональности очень трудна, то, быть может, от нее лучше вообще отказаться.

Решение

Используйте стандартизованные функции, доступные на всех платформах. Если таковых не существует, подумайте, нужно ли вообще реализовывать функциональность.

Примерами стандартизованных функций могут служить функции из стандартной библиотеки C и функции POSIX. Проверьте, доступны ли такие функции на всех платформах, которые вы собираетесь поддерживать. Если есть такая возможность, используйте стандартизованные функции, а не платформенно зависимые, как показано в примере ниже.

Код вызывающей стороны

```
#include <standardizedApi.h>

int main()
{
 /* вызывается всего одна функция вместо нескольких разграниченных ifdef */
 somePosixFunction();
 return 0;
}
```

Стандартизованный API

```
/* эта функция доступна во всех операционных системах,
   отвечающих стандарту POSIX */
somePosixFunction();
```

Повторим: если нужной вам стандартизованной функции не существует, то, возможно, вообще не стоит реализовывать запрашиваемую функциональность. Если ее можно реализовать только с помощью платформенно зависимых функций, то, быть может, не стоит тратиться на реализацию, тестирование и сопровождение.

Но иногда функциональность необходимо предоставить, даже если подходящей стандартизованной функции нет. Это значит, что нужно использовать разные функции на разных платформах или даже дописать функциональность, которая имеется на одной платформе, но отсутствует на другой. Чтобы сделать это структурированным образом, напишите «Изолированные примитивы» для разных вариантов кода и скройте их за «Уровнем абстракции».

Например, чтобы избежать вариантов, пользуйтесь функциями из стандартной библиотеки C, например `fopen`, вместо системно-зависимых функций типа `open` в Linux или `CreateFile` в Windows. Другой пример – стандартные функции работы со временем. Избегайте таких системно-зависимых функций, как `GetLocalTime` в Windows и `localtime_r` в Linux; пользуйтесь стандартизованной функцией `localtime`, объявленной в заголовочном файле *time.h*.

Последствия

Код проще писать и читать, потому что один блок кода используется на разных платформах. Программисту не нужно разбираться в разных функциях для разных платформ и перескакивать между ветвями `#ifdef` при чтении кода.

Поскольку на всех платформах работает один и тот же блок кода, функциональность не различается. Но стандартизованная функция, возможно, не самый эффективный способ реализовать нужную функциональность на всех платформах. На некоторых платформах могут предлагаться специализированные функции, например использующие специальное оборудование, доступное на этой платформе, для достижения более высокой производительности. Не исключено, что стандартизованные функции этими возможностями не пользуются.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В коде текстового редактора VIM используются системно-независимые функции `fopen`, `fwrite`, `fread` и `fclose` для доступа к файлам.
- В библиотеке OpenSSL каждое сообщение в журнале снабжается текущей временной меткой. Для этого текущее время UTC преобразуется в местное системно-независимой функцией `localtime`.
- Функция `BIO_lookup_ex` из библиотеки OpenSSL ищет узел и службу для подключения. Эта функция компилируется в Windows и в Linux и пользуется системно-независимой функцией `htons` для преобразования значения в сетевой порядок байтов.

Применение к сквозному примеру

Что касается доступа к файлам, тут вам повезло: для этой цели существуют системно-независимые функции. Теперь код принимает вид:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef __unix__
  #include <sys/stat.h>
#elif defined _WIN32
  #include <windows.h>
#endif

int main()
{
 char dirname[50];
 char filename[60];
 char* my_data = "Write this data to the file";
 #ifdef __unix__
   #ifdef STORE_IN_HOME_DIR
     sprintf(dirname, "%s%s", getenv("HOME"), "/newdir/");
     sprintf(filename, "%s%s", dirname, "newfile");
   #elif defined STORE_IN_CWD
     strcpy(dirname, "newdir");
     strcpy(filename, "newdir/newfile");
   #endif
   mkdir(dirname,S_IRWXU);
 #elif defined _WIN32
   #ifdef STORE_IN_HOME_DIR
     sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"), getenv("HOMEPATH"),
             "\\newdir\\");
     sprintf(filename, "%s%s", dirname, "newfile");
   #elif defined STORE_IN_CWD
```

```
    strcpy(dirname, "newdir");
    strcpy(filename, "newdir\\newfile");
  #endif
  CreateDirectory(dirname, NULL);
#endif
FILE* f = fopen(filename, "w+"); ❶
fwrite(my_data, 1, strlen(my_data), f);
fclose(f);

return 0;
}
```

❶ Функции `fopen`, `fwrite` и `fclose` – часть стандартной библиотеки C, они доступны и в Windows, и в Linux.

Вызов стандартизированных функций для работы с файлами уже сделал код значительно проще. Вместо отдельных вызовов для Windows и Linux мы теперь имеем единый код. Это гарантирует, что мы получим одинаковую функциональность в обеих операционных системах, и можно не бояться, что разные реализации разойдутся после исправления ошибок или добавления новых возможностей.

Однако код все еще изобилует директивами `#ifdef`, поэтому читать его очень трудно. Следовательно, нужно позаботиться о том, чтобы основную логику программы не замутняли варианты кода. Заведите «Изолированные примитивы», отделяющие варианты кода от основной логики.

Изолированные примитивы

Контекст

В вашем коде вызываются платформенно зависимые функции. Для разных платформ код содержит разные варианты, разграниченные директивами `#ifdef`. Паттерн «Избегание вариантов» неприменим, потому что для нужной вам возможности нет стандартизованной функции, пригодной для всех платформ.

Проблема

Организация вариантов кода с помощью `#ifdef` делает код нечитаемым. Очень трудно проследить поток выполнения программы, поскольку он реализован несколько раз для разных платформ.

Пытаясь понять код, вы обычно фокусируете внимание на какой-то одной платформе, но `#ifdef` заставляют вас перескакивать между строками, чтобы найти интересующий вариант.

Из-за директив `#ifdef` код еще и труднее сопровождать. Они приглашают программиста исправить код только для интересующей его в данный момент платформы и не трогать прочий – вдруг что-то сломается. Но исправление ошибки или добавление новой возможности только для одной платформы оз-

начает, что поведение на разных платформах начинает расходиться. Альтернатива – исправлять ошибку на всех платформах по-разному – требует тестирования на всех платформах.

Тестировать код с большим числом вариантов трудно. Каждый новый набор `#ifdef` удваивает объем тестирования, потому что нужно протестировать все возможные комбинации. Хуже того, каждая такая директива удваивает количество собираемых двоичных файлов, подлежащих тестированию. Это приводит к логистической проблеме, так как увеличивается время сборки и количество двоичных файлов, передаваемых в отдел тестирования и заказчику.

Решение

Изолируйте варианты кода. В файле реализации поместите код с вариантами в отдельные функции и вызывайте эти функции из главной программы, которая будет содержать только платформенно независимый код.

Каждая ваша функция должна либо содержать программную логику, либо заниматься обработкой вариантов. Никакая функция не должна делать то и другое. Поэтому либо в функции нет ни одной директивы `#ifdef`, либо имеются директивы `#ifdef` с единственным зависящим от варианта вызовом функции в каждой ветви. Вариантом может быть как программная возможность, которая включается или выключается конфигурационным параметром, так и выбор платформы:

```
void handlePlatformVariants()
{
 #ifdef PLATFORM_A
   /* вызов функции на платформе A */
 #elif defined PLATFORM_B ❶
   /* вызов функции на платформе B */
#endif
}

int main()
{
 /* здесь должна быть логика программы */
 handlePlatformVariants();
 /* продолжение логики программы */
}
```

❶ Как и в предложениях `else if`, взаимно исключающие варианты можно выразить с помощью директивы `#elif`.

Наличие всего одного вызова функции в каждой ветви `#ifdef` должно обеспечить приемлемый уровень абстракции для функций, обрабатывающих варианты. Обычно абстрагируется ровно одна зависящая от платформы или конфигурации функция, подлежащая обертыванию.

Если функции обработки вариантов все еще сложны и содержат каскады вложенных `#ifdef`, то постарайтесь использовать только «Атомарные варианты».

Последствия

Основную логику программы легко проследить, потому что варианты кода отделены от нее. Читая основной код, больше нет необходимости перескакивать между строками, чтобы выяснить, что делает код на конкретной платформе.

Чтобы определить, что делается на одной платформе, нужно посмотреть на вызываемую функцию, реализующую данный вариант. Вынесение этого кода в отдельную функцию хорошо тем, что ее можно вызывать из других мест файла программы, избежав тем самым дублирования кода. Если эта функциональность необходима и в других файлах реализации, то следует прибегнуть к паттерну «Уровень абстракции».

В функции обработки вариантов не следует помещать никакую программную логику, тогда будет проще найти ошибки, встречающиеся не на всех платформах, поскольку легко выявить те места кода, где поведение зависит от платформы.

Проблема дублирования кода теряет остроту, потому что основная логика программы отделена от реализации вариантов. Больше нет соблазна продублировать логику программы, а значит, нет и опасности случайно исправить ошибки только в одном варианте.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В коде текстового редактора VIM изолирована функция `htonl2`, которая преобразует данные в сетевой порядок байтов. В VIM `htonl2` определена как макрос в файле реализации. Этот макрос компилируется по-разному в зависимости от порядка байтов на платформе.
- Функция `BIO_ADDR_make` в библиотеке OpenSSL копирует информацию о сокете во внутреннюю структуру `struct`. В функции используются директивы `#ifdef` для обработки системных и функциональных различий между Linux/Windows и IPv4/IPv6. Все эти варианты отделены от основной логики программы.
- Функция `load_rcfile` в программе GNUplot читает данные из файла инициализации и изолирует системно-зависимые операции доступа к файлу от остального кода.

Применение к сквозному примеру

После выделения «Изолированных примитивов» основную логику программы стало гораздо проще читать, и не нужно перескакивать между строками, чтобы различить варианты:

```
void getDirectoryName(char* dirname)
{
 #ifdef __unix__
   #ifdef STORE_IN_HOME_DIR
     sprintf(dirname, "%s%s", getenv("HOME"), "/newdir/");
```

```
    #elif defined STORE_IN_CWD
      strcpy(dirname, "newdir/");
    #endif
  #elif defined _WIN32
    #ifdef STORE_IN_HOME_DIR
      sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"), getenv("HOMEPATH"),
              "\\newdir\\");
    #elif defined STORE_IN_CWD
      strcpy(dirname, "newdir\\");
    #endif
  #endif
}

void createNewDirectory(char* dirname)
{
  #ifdef __unix__
    mkdir(dirname,S_IRWXU);
  #elif defined _WIN32
    CreateDirectory (dirname, NULL);
  #endif
}

int main()
{
  char dirname[50];
  char filename[60];
  char* my_data = "Write this data to the file";
  getDirectoryName(dirname);
  createNewDirectory(dirname);
  sprintf(filename, "%s%s", dirname, "newfile");
  FILE* f = fopen(filename, "w+");
  fwrite(my_data, 1, strlen(my_data), f);
  fclose(f);
  return 0;
}
```

Теперь варианты кода хорошо изолированы. Без вариантов функция `main` легко читается, и ее логика понятна. Однако новая функция `getDirectoryName` по-прежнему изобилует директивами `#ifdef`, и понять ее нелегко. На помощь придут «Атомарные примитивы».

Атомарные примитивы

Контекст

Вы реализовали в коде варианты с помощью директив `#ifdef` и поместили варианты в отдельные функции, где их обрабатывают «Изолированные примитивы». Примитивы отделяют функции от основного потока программы,

благодаря чему основная программа оказывается хорошо структурированной и понятной.

Проблема

Функцию, которая содержит варианты и вызывается из основной программы, все еще трудно понять, потому что в нее помещен весь сложный код, пестрящий `#ifdef`, лишь бы вынести его из основной программы.

Обработать все варианты в одной функции трудно, если вариантов много. Например, если в одной функции директивы `#ifdef` используются для различения типов оборудования и операционных систем, то добавить еще одну операционную систему сложно, потому что это нужно сделать для всех вариантов оборудования. Каждый вариант больше не получается обработать в одном месте; усилия умножаются на количество вариантов. Это проблема. Нужно придумать, как добавлять новые варианты только в одном месте кода.

Решение

Сделайте примитивы атомарными. В каждой функции обрабатывайте только один вид вариантов. Если требуется обрабатывать несколько видов вариантов – например, операционной системы и оборудования, – то заведите для каждого отдельную функцию.

Пусть одна из этих функций вызывает другую, которая уже абстрагирует один вид варианта. Если абстрагируется зависимость от платформы и от функциональности, то пусть функция, зависящая от функциональности, вызывает платформенно зависимую, потому что обычно функциональность предоставляется на всех платформах. Следовательно, платформенно зависимые функции должны быть самыми атомарными, как показано в коде ниже.

```
void handleHardwareOfFeatureX()
{
 #ifdef HARDWARE_A
   /* вызвать функцию, реализующую возможность X на оборудовании A */
 #elif defined HARDWARE_B || defined HARDWARE_C
   /* вызвать функцию, реализующую возможность X на оборудовании B и C */
 #endif
}

void handleHardwareOfFeatureY()
{
 #ifdef HARDWARE_A
   /* вызвать функцию, реализующую возможность Y на оборудовании A */
 #elif defined HARDWARE_B
   /* вызвать функцию, реализующую возможность Y на оборудовании B */
 #elif defined HARDWARE_C
   /* вызвать функцию, реализующую возможность Y на оборудовании C */
 #endif
```

```
}

void callFeature()
{
 #ifdef FEATURE_X
   handleHardwareOfFeatureX();
 #elif defined FEATURE_Y
   handleHardwareOfFeatureY();
 #endif
}
```

Если существует функция, которая должна предоставлять функциональность для нескольких видов вариантов, а также обрабатывать эти виды, то, возможно, сфера ответственности функции выбрана неудачно. Быть может, функция слишком общая или делает больше одной вещи. Разделите функцию на части, как рекомендует паттерн «Разбиение функции».

Вызывайте «Атомарные примитивы» из основного кода, содержащего программную логику. Если хотите использовать «Атомарные примитивы» в других файлах реализации с четко определенным интерфейсом, то пользуйтесь паттерном «Уровень абстракции».

Последствия

Каждая функция теперь обрабатывает только один вид варианта. Поэтому функции легко понять, так как больше нет каскадов `#ifdef`. Каждая функция теперь абстрагирует только один вид варианта и делает не более одной вещи. То есть функции устроены в соответствии с принципом единственной обязанности.

Из-за отсутствия каскадов `#ifdef` у программиста меньше соблазна просто обработать дополнительный вид варианта в одной функции, потому что начинать новый каскад `#ifdef` не так естественно, как продолжать существующий.

При наличии отдельных функций каждый вид варианта легко расширить, добавив дополнительный вариант. Для этого нужно добавить только одну ветвь `#ifdef` в одну функцию, а функции, обрабатывающие другие виды вариантов, трогать не придется.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В библиотеке OpenSSL файл *threads_pthread.c* содержит функции для работы с потоками. Существуют отдельные функции для абстрагирования операционных систем и функции, абстрагирующие отсутствие механизма pthreads.
- В коде SQLite имеются функции для абстрагирования системно-зависимого доступа к файлам (например, `fileStat`) на этапе компиляции.
- Функция Linux `boot_jump_linux` вызывает другую функцию, которая выполняет различные действия на этапе загрузки в зависимости от ар-

хитектуры процессора, определяемой директивами `#ifdef`. Затем `boot_jump_linux` вызывает еще одну функцию, которая использует `#ifdef` для выбора сконфигурированных ресурсов (USB, сеть и т. д.), подлежащих инициализации.

Применение к сквозному примеру

После реализации «Атомарных примитивов» получается следующий код функций определения пути к каталогу:

```
void getHomeDirectory(char* dirname)
{
 #ifdef __unix__
   sprintf(dirname, "%s%s", getenv("HOME"), "/newdir/");
 #elif defined _WIN32
   sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"), getenv("HOMEPATH"),
           "\\newdir\\");
 #endif
}

void getWorkingDirectory(char* dirname)
{
 #ifdef __unix__
   strcpy(dirname, "newdir/");
 #elif defined _WIN32
   strcpy(dirname, "newdir\\");
 #endif
}

void getDirectoryName(char* dirname)
{
 #ifdef STORE_IN_HOME_DIR
   getHomeDirectory(dirname);
 #elif defined STORE_IN_CWD
   getWorkingDirectory(dirname);
 #endif
}
```

Теперь варианты кода очень хорошо изолированы. Для получения имени каталога у нас имеется не одна запутанная функция со множеством `#ifdef`, а несколько функций, по одной `#ifdef` в каждой. Так понять код гораздо проще, потому что каждая функция выполняет что-то одно, а не занимается различением разных видов вариантов с помощью каскадов `#ifdef`.

Сами функции очень простые и легко читаются, но файл реализации все равно остался слишком длинным. Кроме того, один файл реализации содержит как основную логику программы, так и код различения вариантов. Из-за этого параллельная разработка и раздельное тестирование вариантов почти невозможны.

Чтобы улучшить ситуацию, разделите файл реализации на два: зависящий и не зависящий от вариантов. Для этого нужно создать «Уровень абстракции».

Уровень абстракции

Контекст

Имеются платформенные варианты, различаемые в коде директивами `#ifdef`. Возможно, вы уже реализовали «Изолированные примитивы», чтобы отделить варианты от логики программы, и даже сделали эти примитивы атомарными.

Проблема

Вы хотите использовать функциональность обработки платформенных вариантов в нескольких местах кодовой базы, но не хотите дублировать код этой функциональности.

Раньше вызывающие стороны, возможно, использовались для работы с платформенно зависимыми функциями напрямую, но вы решили от этого отказаться, потому что каждой вызывающей стороне приходится реализовывать платформенные варианты самостоятельно. В общем случае вызывающая сторона не должна иметь дела с платформенными вариантами. Она не должна знать о деталях реализации для разных платформ и не должна использовать директивы `#ifdef` или включать платформенно зависимые заголовочные файлы.

Вы даже рассматриваете возможность привлечения новых программистов (не тех, что отвечают за платформенно независимый код) к разработке и тестированию платформенно зависимого кода.

Вы хотите в будущем иметь возможность изменять платформенно зависимый код, не ставя вызывающую сторону в известность об этом изменении. Если разработчики платформенно зависимого кода исправят ошибку на одной платформе или добавят новую платформу, то в код вызывающей стороны не нужно будет вносить изменений.

Решение

Предоставьте API для любой функциональности, нуждающейся в платформенно зависимом коде. Объявите в заголовочном файле только платформенно независимые функции, а все платформенно зависимые `#ifdef` помещайте в файл реализации. Вызывающая сторона будет включать только ваш заголовочный файл и никаких платформенно зависимых.

Постарайтесь спроектировать стабильный API для уровня абстракции, потому что изменение этого API в будущем потребует внесения изменений в код вызывающей стороны, что не всегда возможно. Однако спроектировать стабильный API очень трудно. В поисках платформенных абстракций попробуйте изучить различные платформы, даже те, которые пока не поддерживаете. Поняв, как они работают и чем отличаются, вы сможете создать API для абстраги-

рования особенностей этих платформ. Тогда вам не нужно будет изменять API впоследствии, даже при добавлении новых платформ.

Тщательно документируйте API. Каждую функцию снабдите комментариями, разъясняющими, что она делает. Опишите, на каких платформах поддерживаются функции, если это не определено четко для всей кодовой базы.

В примере ниже показан простой «Уровень абстракции».

caller.c

```
#include "someFeature.h"

int main()
{
 someFeature();
 return 0;
}
```

someFeature.h

```
/* Предоставляет доступ общего вида к someFeature.
   Поддерживается на платформах A и B. */
void someFeature();
```

someFeature.c

```
void someFeature()
{
 #ifdef PLATFORM_A
   performFeaturePlatformA();
 #elif defined PLATFORM_B
   performFeaturePlatformB();
 #endif
}
```

Последствия

Абстрагированные возможности можно использовать в любом месте кода, а не только в одном файле реализации. Иными словами, теперь роли вызывающей и вызываемой стороны обособились. Вызываемая сторона обязана иметь дело с платформенными вариантами, а вызывающая может стать платформенно независимой.

Преимущество такой организации в том, что вызывающая сторона не обязана содержать платформенно зависимый код. Она может просто включить предоставленный ей заголовочный файл и не включать никаких платформенно зависимых. А недостаток в том, что вызывающая сторона больше не вправе напрямую использовать платформенно зависимые функции. Если ее автор привык к этим функциям, то может быть недоволен абстрагированной функциональностью, считая, что ее трудно использовать или что она недостаточно эффективна.

Платформенно зависимый код теперь можно разрабатывать и даже тестировать отдельно от прочего кода. Тестирование становится практически осу-

ществимым даже при наличии многих платформ, потому что вы можете подставить имитацию аппаратно зависимого кода ради написания простых тестов платформенно независимого кода.

Сумма всех таких платформенно зависимых функций и их API образует уровень платформенной абстракции в кодовой базе. Имея этот уровень, очень легко понять, какой код зависит от платформы, а какой не зависит и, следовательно, в какие части кода вносить изменения при добавлении новой платформы.

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- Большинство крупных программ, работающих на нескольких платформах, имеют уровень аппаратных абстракций. Например, такой уровень есть на платформе Nokia Maemo, где он используется для того, чтобы абстрагировать, какие фактические драйверы устройств загружены.
- Функция `sock_addr_inet_pton` в веб-сервере lighttpd преобразует IP-адрес из текстового представления в двоичное. Внутри реализации используются директивы `#ifdef`, чтобы различить варианты кода для IPv4 и IPv6. Стороны, вызывающие этот API, различия не видят.
- В программе сжатия данных gzip функция `getprogname` возвращает имя вызывающей программы. Способ его получения зависит от операционной системы, для определения которой используются директивы `#ifdef`. Вызывающей стороне безразлично, в какой ОС вызывается функция.
- Аппаратная абстракция используется в протоколе Ethernet с временным разделением каналов (Time-Triggered Ethernet), описанном в работе Флемминга Бунзеля на получение степени бакалавра «Hardware-Abstraction of an Open Source Real-Time Ethernet Stack – Design, Realisation and Evaluation» (https://oreil.ly/hs0Jh). Уровень аппаратных абстракций содержит функции для доступа к прерываниям и таймерам. Чтобы не потерять в производительности, функции сделаны встраиваемыми (`inline`).

Применение к сквозному примеру

Теперь код стал гораздо более линейным. Каждая функция выполняет всего одно действие, а детали реализации вариантов скрыты за фасадом API.

directoryNames.h

```
/* Копирует путь к новому каталогу "newdir", находящемуся в
   домашнем каталоге пользователя, в параметр "dirname".
   Работает в Linux и в Windows. */
void getHomeDirectory(char* dirname);
/* Копирует путь к новому каталогу "newdir", находящемуся в
   текущем рабочем каталоге, в параметр "dirname".
   Работает в Linux и в Windows. */

void getWorkingDirectory(char* dirname);
```

directoryNames.c

```
#include "directoryNames.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void getHomeDirectory(char* dirname)
{
 #ifdef __unix__
   sprintf(dirname, "%s%s", getenv("HOME"), "/newdir/");
 #elif defined _WIN32
   sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"), getenv("HOMEPATH"),
           "\\newdir\\");
 #endif
}

void getWorkingDirectory(char* dirname)
{
 #ifdef __unix__
   strcpy(dirname, "newdir/");
 #elif defined _WIN32
   strcpy(dirname, "newdir\\");
 #endif
}
```

directorySelection.h

```
/* Копирует путь к новому каталогу "newdir" в "dirname".
   Каталог находится в домашнем каталоге пользователя, если
   определен макрос STORE_IN_HOME_DIR, или в текущем рабочем
   каталоге, если определен макрос STORE_IN_CWD. */
void getDirectoryName(char* dirname);
```

directorySelection.c

```
#include "directorySelection.h"
#include "directoryNames.h"

void getDirectoryName(char* dirname)
{
 #ifdef STORE_IN_HOME_DIR
   getHomeDirectory(dirname);
 #elif defined STORE_IN_CWD
   getWorkingDirectory(dirname);
 #endif
}
```

directoryHandling.h

```
/* Создает новый каталог с указанным именем ("dirname").
   Работает в Linux и в Windows. */
void createNewDirectory(char* dirname);
```

directoryHandling.c

```
#include "directoryHandling.h"
#ifdef __unix__
  #include <sys/stat.h>
#elif defined _WIN32
  #include <windows.h>
#endif

void createNewDirectory(char* dirname)
{
 #ifdef __unix__
   mkdir(dirname,S_IRWXU);
 #elif defined _WIN32
   CreateDirectory (dirname, NULL);
 #endif
}
```

main.c

```
#include <stdio.h>
#include <string.h>
#include "directorySelection.h"
#include "directoryHandling.h"

int main()
{
 char dirname[50];
 char filename[60];
 char* my_data = "Write this data to the file";
 getDirectoryName(dirname);
 createNewDirectory(dirname);
 sprintf(filename, "%s%s", dirname, "newfile");
 FILE* f = fopen(filename, "w+");
 fwrite(my_data, 1, strlen(my_data), f);
 fclose(f);
 return 0;
}
```

Файл, содержащий основную логику программы, наконец-то стал полностью независимым от операционной системы; в него даже не включены системно-зависимые заголовочные файлы.

Благодаря отделению файлов реализации с помощью паттерна «Уровень абстракции» становится проще понять файлы и повторно использовать функции в других частях кода. Кроме того, можно разделить разработку, тестирование и сопровождение платформенно зависимого и платформенно независимого кода.

Если за фасадом «Уровня абстракции» находятся «Изолированные примитивы» и вы организовали их в соответствии с видом абстрагируемого варианта, то в итоге получите уровень аппаратных абстракций или уровень абстракций операционной системы. Количество файлов с кодом теперь резко увеличилось – особенно тех, в которых обрабатываются различные варианты, – и, возможно, стоит подумать о размещении их в «Каталогах программных модулей».

Код, в котором используется API уровня абстракции, стал очень чистым, но в реализациях ниже уровня API по-прежнему используются `#ifdef`. Недостаток такой организации в том, что эти реализации придется модифицировать и они будут расти, если, например, будет добавлена поддержка новых операционных систем. Чтобы избежать модификации файлов реализации при добавлении нового варианта, можно воспользоваться паттерном «Разделение реализаций вариантов».

Разделение реализаций вариантов

Контекст

Имеются платформенные варианты, скрытые за фасадом «Уровня абстракции». В реализациях платформенно зависимого кода варианты различаются с помощью директив `#ifdef`.

Проблема

Реализации платформенно зависимого кода содержат директивы `#ifdef` для различения вариантов. Из-за этого трудно понять, какая часть кода для какой платформы собирается.

Поскольку код для разных платформ помещен в один файл, невозможно отобрать платформенно зависимый код на основе файлов. Однако именно такой подход принят в инструментах типа Make, обычно отвечающих (посредством файлов Makefile) за выбор файлов, которые следует откомпилировать для конкретной платформы.

Глядя на код извне, невозможно понять, какие части зависят от платформы, а какие нет, но при переносе кода на другую платформу это было бы весьма желательно – чтобы быстро определить, какие части нуждаются в модификации.

Принцип открытости-закрытости гласит, что для добавления новых возможностей (или переноса на новую платформу) не должно возникать необходимости в модификации существующего кода. Код должен быть открыт для таких модификаций. Однако если варианты различаются с помощью `#ifdef`, то без изменения существующих реализаций при добавлении платформы не обойтись, потому что в уже имеющуюся функцию необходимо добавить еще одну ветвь `#ifdef`.

Решение

Поместите реализацию каждого варианта в отдельный файл реализации и выберите, какие файлы нужно компилировать для каждой платформы.

Логически связанные функции для одной и той же платформы по-прежнему можно помещать в один файл. Например, можно собрать в одном файле все функции работы с сокетами в Windows, а в другом – аналогичные функции для Linux.

После разделения файлов по платформам ничто не мешает использовать директивы `#ifdef` для определения того, какой код компилируется на конкретной платформе. Например, в файле *someFeatureWindows.c* может присутствовать директива `#ifdef _WIN32`, окружающая весь код, – по аналогии с «Охраной включения»:

someFeature.h

```
/* Предоставляет доступ общего вида к someFeature.
   Поддерживается на платформах A и B. */
someFeature();
```

someFeatureWindows.c

```
#ifdef _WIN32
  someFeature()
  {
   performWindowsFeature();
  }
#endif
```

someFeatureLinux.c

```
#ifdef __unix__
  someFeature()
  {
   performLinuxFeature();
  }
#endif
```

Вместо применения `#ifdef` для условной компиляции всего содержимого файла можно использовать другие платформенно независимые механизмы типа Make, которые решают, какие файлы следует компилировать для конкретной платформы. Если ваша IDE помогает генерировать Makefile'ы, то эта альтернатива может оказаться более удобной, но помните, что при смене IDE, возможно, придется заново настроить механизм выбора подлежащих компиляции файлов.

Разделение файлов по платформам поднимает вопрос о том, куда эти файлы помещать и как их называть.

- Можно собрать платформенно зависимые файлы для каждого программного модуля в одном месте и назвать их так, чтобы было понятно, к какой платформе они относятся (например, *fileHandlingWindows.c*). Преимущество таких «Каталогов программных модулей» в том, что реализации каждого программного модуля находятся в одном месте.
- Можно вместо этого поместить все платформенно зависимые файлы кодовой базы в один каталог, создав в нем подкаталоги для каждой платформы. Преимущество этого подхода в том, что все файлы для одной платформы находятся в одном месте, так что в IDE проще указать, какие файлы для какой платформы компилируются.

Последствия

Теперь можно вообще отказаться от `#ifdef` в коде, а вместо этого различать варианты на основе файлов с помощью таких инструментов, как Make.

В каждом файле реализации теперь имеется только один вариант кода, поэтому нет необходимости перескакивать между строками при чтении кода, чтобы прочитать ту единственную ветвь `#ifdef`, которая вас интересует. Код стал гораздо проще для чтения и понимания.

Когда исправляется ошибка на одной платформе, не нужно трогать файлы для других платформ. При переносе кода на новую платформу следует только добавить новые файлы, а существующие можно не модифицировать.

Легко понять, какая часть кода зависит от платформы и какой код следует добавить при переносе на новую платформу. Либо все платформенно зависимые файлы находятся в одном каталоге, либо файлы именуются так, чтобы было ясно, какие из них платформенно зависимы.

Однако если каждый вариант помещается в отдельный файл, то файлов становится много. Чем больше файлов, тем сложнее процедура сборки и тем дольше код компилируется. Подумайте об организации файлов, например с помощью «Каталогов программных модулей».

Известные примеры применения

Следующие примеры демонстрируют применение этого паттерна.

- В библиотеке Simple Audio Library, описанной в книге Brian Hook «Write Portable Code: An Introduction to Developing Software for Multiple Platforms» (No Starch Press, 2005), отдельные файлы реализации используются для предоставления доступа к потокам и Мьютексам в Linux и OS X. В файлах реализации директивы `#ifdef` используются для того, чтобы на каждой платформе компилировался только относящийся к ней код.
- Модуль мультипроцессирования в веб-сервере Apache, отвечающий за обработку доступа к веб-серверу, реализован в отдельных файлах для Windows и Linux. В файлах реализации директивы используются для того, чтобы на каждой платформе компилировался только относящийся к ней код.
- В начальном загрузчике U-Boot исходный код для каждой поддерживаемой аппаратной платформы находится в отдельном каталоге. Каждый

такой каталог содержит среди прочего файл *cpu.c*, в котором находится функция сброса процессора. Makefile решает, какой каталог (и файл *cpu.c*) компилировать, – в этих файлах нет директив `#ifdef`. Основная логика U-Boot вызывает функцию сброса процессора, не заботясь об аппаратных деталях платформы.

Применение к сквозному примеру

После применения паттерна «Разделение реализаций вариантов» получается следующий окончательный код создания каталога и записи данных в файл.

directoryNames.h

```
/* Копирует путь к новому каталогу "newdir", находящемуся
   в домашнем каталоге пользователя,  в "dirname".
   Работает в Linux и в Windows. */
void getHomeDirectory(char* dirname);
/* Копирует путь к новому каталогу "newdir", находящемуся
   в текущем рабочем каталоге,  в "dirname".
   Работает в Linux и в Windows. */
void getWorkingDirectory(char* dirname);
```

directoryNamesLinux.c

```
#ifdef __unix__
  #include "directoryNames.h"
  #include <string.h>
  #include <stdio.h>
  #include <stdlib.h>

  void getHomeDirectory(char* dirname)
  {
   sprintf(dirname, "%s%s", getenv("HOME"), "/newdir/");
  }

  void getWorkingDirectory(char* dirname)
  {
   strcpy(dirname, "newdir/");
  }
#endif
```

directoryNamesWindows.c

```
#ifdef _WIN32
  #include "directoryNames.h"
  #include <string.h>
  #include <stdio.h>
  #include <windows.h>

  void getHomeDirectory(char* dirname)
```

```
  {
   sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"), getenv("HOMEPATH"),
           "\\newdir\\");
  }

  void getWorkingDirectory(char* dirname)
  {
   strcpy(dirname, "newdir\\");
  }
#endif
```

directorySelection.h

```
/* Копирует путь к новому каталогу "newdir" в "dirname".
   Каталог находится в домашнем каталоге пользователя,
   если определен макрос STORE_IN_HOME_DIR, или в текущем
   рабочем каталоге, если определен макрос STORE_IN_CWD. */
void getDirectoryName(char* dirname);
```

directorySelectionHomeDir.c

```
#ifdef STORE_IN_HOME_DIR
  #include "directorySelection.h"
  #include "directoryNames.h"
  void getDirectoryName(char* dirname)
  {
   getHomeDirectory(dirname);
  }
#endif
```

directorySelectionWorkingDir.c

```
#ifdef STORE_IN_CWD
  #include "directorySelection.h"
  #include "directoryNames.h"
  void getDirectoryName(char* dirname)
  {
   return getWorkingDirectory(dirname);
  }
#endif
```

directoryHandling.h

```
/* Создает новый каталог с указанным именем ("dirname").
   Работает в Linux и в Windows. */
void createNewDirectory(char* dirname);
```

directoryHandlingLinux.c

```
#ifdef __unix__
```

```
  #include <sys/stat.h>
  void createNewDirectory(char* dirname)
  {
   mkdir(dirname,S_IRWXU);
  }
#endif
```

directoryHandlingWindows.c

```
#ifdef _WIN32
  #include <windows.h>
  void createNewDirectory(char* dirname)
  {
   CreateDirectory(dirname, NULL);
  }
#endif
```

main.c

```
#include "directorySelection.h"
#include "directoryHandling.h"
#include <string.h>
#include <stdio.h>

int main()
{
 char dirname[50];
 char filename[60];
 char* my_data = "Write this data to the file";
 getDirectoryName(dirname);
 createNewDirectory(dirname);
 sprintf(filename, "%s%s", dirname, "newfile");
 FILE* f = fopen(filename, "w+");
 fwrite(my_data, 1, strlen(my_data), f);
 fclose(f);
 return 0;
}
```

В этом коде по-прежнему остались директивы `#ifdef`. В каждом файле реализации есть одна большая `#ifdef`, гарантирующая, что код компилируется для нужной платформы и варианта. Альтернативно решение о том, какие файлы компилировать, можно перенести в Makefile. Тогда от `#ifdef` мы избавимся, но заменим их другим механизмом выбора вариантов. Какой механизм использовать – не так важно. Гораздо важнее изолировать и абстрагировать варианты – этому и была посвящена вся эта глава.

Хотя файлы кода выглядели бы гораздо чище при использовании других механизмов обработки вариантов, сложность никуда не делась. Перемещение сложности в Makefile'ы может оказаться хорошей идеей, потому что цель

Makefile'ов в том и заключается, чтобы решить, какие файлы собирать. В других ситуациях лучше пользоваться директивами `#ifdef`. Например, если вы собираете системно-зависимый код, то, быть может, для принятия решений о сборке используется одна коммерческая IDE для Windows и другая для Linux. При таких условиях применение `#ifdef` в коде гораздо чище; настройка того, какие файлы компилировать для каждой операционной системы, выполняется всего один раз с помощью `#ifdef`, и при переходе на другую IDE изменять что-то нет необходимости.

Финальный код сквозного примера очень четко показал, как шаг за шагом улучшать код, содержащий системно-зависимые или другие варианты. По сравнению с начальным кодом финальный прекрасно читается и может быть легко расширен путем включения дополнительных возможностей или перенесен на другие операционные системы – изменять уже написанный код при этом не придется.

Резюме

В этой главе описаны паттерны, касающиеся обработки вариантов, например зависимостей от оборудования или операционной системы, в коде на C. Особое внимание уделено организации кода и избавлению от директив `#ifdef`.

Паттерн «Избегание вариантов» рекомендует использовать стандартизованные функции вместо «самописных». Этот паттерн следует применять при любой возможности, поскольку он решает все проблемы вариантов одним ударом. Однако стандартизованная функция имеется не всегда, и в таких случаях программисту приходится реализовывать собственную функцию для абстрагирования варианта. В качестве отправной точки паттерн «Изолированные примитивы» рекомендует помещать варианты в отдельные функции, а паттерн «Атомарные примитивы» – обрабатывать в каждой такой функции только один вариант. Паттерн «Уровень абстракции» делает следующий шаг, чтобы скрыть реализации примитивов за фасадом API. А паттерн «Разделение реализаций вариантов» предлагает помещать каждый вариант в отдельный файл реализации.

Включив эти паттерны в свой арсенал, программист получает инструментарий и пошаговое наставление по работе с вариантами кода в программах на C, что позволит ему лучше структурировать код и сбежать из ада `#ifdef`.

Опытным программистам некоторые паттерны могут показаться очевидными – и это хорошо. Одна из задач паттернов – научить людей работать правильно; когда человек знает, как сделать правильно, паттерны уже не нужны, потому что человек интуитивно делает именно то, что они рекомендуют.

Для дополнительного чтения

Для интересующихся читателей ниже перечислены ресурсы, которые помогут расширить знания об абстрагировании платформы и вариантов.

- В книге Brian Hook «Write Portable Code: An Introduction to Developing Software for Multiple Platforms» (No Starch Press, 2005) рассказано, как

писать переносимый код на C. Рассматриваются зависимости от операционной системы и оборудования с примерами действий в конкретных ситуациях, например как обработать различия в порядке байтов, размерах типов данных или символах разделения строк.

- Статья Henry Spencer and Geoff Collyer «#ifdef Considered Harmful» (https://oreil.ly/eZ2CW) – первая, в которой скептически оценивается использование директив `#ifdef`. В ней подробно рассматриваются проблемы, возникающие при их бессистемном применении, и предлагаются альтернативы.
- В статье Didier Malenfant «Writing Portable Code» (https://oreil.ly/XkTbj) описывается, как структурировать переносимый код и какую функциональность следует поместить за фасад уровня абстракции.

Что дальше

Итак, теперь в вашем арсенале стало больше паттернов. Далее мы научимся применять эти и другие паттерны, описанные в предыдущих главах. В следующих главах рассмотрены объемные примеры кода, демонстрирующие применение всех паттернов.

Часть II

Истории о паттернах

Рассказывание историй – привычный и естественный способ передачи информации. Что касается паттернов, то иногда бывает трудно понять, как описанные паттерны можно применить в реальном контексте. Чтобы продемонстрировать это, во второй части книги рассказаны истории о применении паттернов программирования на C из первой части к разработке больших программ. Вы узнаете, как такие программы строятся постепенно, и убедитесь, что паттерны упрощают жизнь, подсказывая хорошие проектные решения.

Глава 10

Реализация протоколирования

Выбор правильного паттерна, подходящего к ситуации, очень помогает при проектировании программного обеспечения. Но иногда трудно найти правильный паттерн и решить, когда его применять. Указания на этот счет можно найти в разделах «Контекст» и «Проблема» для паттернов, описанных в первой части книги. Но обычно понять, как что-то сделать, гораздо проще на конкретном примере.

В этой главе рассказана история о применении паттернов из первой части к сквозному примеру, который был выделен из реализации системы протоколирования в одной производственной системе. Чтобы воспринимать код было не очень тяжело, некоторые аспекты исходной системы опущены. Например, при проектировании кода не уделялось внимания производительности и тестопригодности. Тем не менее пример убедительно показывает, как, применяя паттерны, шаг за шагом построить систему протоколирования.

История о паттернах

Представьте себе, что вам поручено сопровождать реально эксплуатируемую программу на C. Если возникает ошибка, вы садитесь в машину, едете к заказчику и приступаете к отладке. И все было нормально, пока заказчик не переехал в другой город. На машине туда ехать несколько часов, что вас совсем не радует.

Вы предпочли бы решать проблему, не отходя от своего стола, чтобы сэкономить время и нервы. Иногда можно воспользоваться удаленной отладкой. Но иногда нужны подробные и точные сведения о том, в каком состоянии находилась программа в момент ошибки, а получить такие сведения удаленно очень трудно – особенно когда ошибка спорадическая.

Наверное, вы уже догадались, какое решение позволит избежать долгих поездок. Нужно реализовать протоколирование и в случае ошибки попросить заказчика прислать вам файлы журналов с отладочной информацией. Иными словами, вы хотите реализовать паттерн «Протоколирование ошибок», что позволит анализировать ошибки постфактум и исправлять их, не настаивая на

воспроизведении. Звучит просто, но для реализации протоколирования предстоит принять много важных проектных решений.

Организация файлов

Для начала организуйте заголовочные файлы и файлы реализации, которые вам предположительно понадобятся. У вас уже имеется большая кодовая база, поэтому вы хотите четко отделить эти файлы от остального кода. Как их организовать? Поместить все относящиеся к протоколированию файлы в один каталог? Или поместить все заголовочные файлы в один каталог?

Для ответа на эти вопросы вы просматриваете паттерны, касающиеся организации файлов, они описаны в главах 6 и 8. Вы внимательно читаете раздел «Проблема» и доверяете рекомендациям, приведенным в разделе «Решение». В итоге остается три паттерна, имеющих прямое отношение к вашим проблемам.

Название паттерна	Краткое описание
Каталоги программных модулей	Помещайте тесно связанные заголовочные файлы и файлы реализации в один каталог. Назовите этот каталог, так чтобы имя отражало функциональность, предоставляемую заголовочными файлами
Заголовочные файлы	Включите в свой API объявления функций, реализующих функциональность, предоставляемую пользователям. Скройте все внутренние функции, внутренние данные и определения (реализации) функций в файлах реализации и не передавайте эти файлы пользователям
Глобальный каталог include	Заведите в кодовой базе один глобальный каталог, содержащий API всех программных модулей. Добавьте этот каталог в глобальный список путей к включаемым файлам, поддерживаемый вашим инструментарием

Создайте каталог программного модуля для своих файлов реализации и поместите заголовочный файл своего модуля протоколирования в уже имеющийся глобальный каталог include. Размещение этого заголовочного файла в глобальном каталоге include имеет то преимущество, что стороны, вызывающие ваш код, будут точно знать, какой заголовочный файл им нужно использовать.

Организация файлов должна быть такой, как на рис. 10.1.

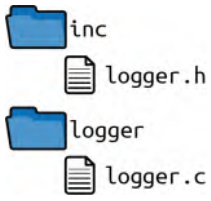


Рис. 10.1. Организация файлов

При такой организации все файлы реализации, которые относятся только к вашему модулю протоколирования, можно поместить в каталог *logger*. А интерфейс, используемый в других частях программы, можно поместить в каталог *inc*.

Центральная функция протоколирования

Для начала реализуем центральную функцию протоколирования ошибок, которая принимает текст сообщения об ошибке, добавляет к нему текущую временную метку и отправляет на стандартный вывод. Наличие временной метки упростит анализ сообщений об ошибках впоследствии.

Поместите объявление функции в файл *logger.h*. Чтобы защититься от повторного включения заголовочного файла, добавьте «Охрану включения». В этом коде нет необходимости хранить информацию о состоянии или выполнять какую-то инициализацию; просто реализуйте «Программный модуль без состояния». Отсутствие состояния несет много преимуществ: код протоколирования остается простым, и вы обходите многие сложности при работе в многопоточном окружении.

Название паттерна	Краткое описание
Охрана включения	Защищайте содержимое заголовочных файлов от повторного включения, чтобы разработчику, пользующемуся ими, не нужно было думать, сколько раз включен файл. Для этого можно использовать директиву `#ifdef` или `#pragma once`
Программный модуль без состояния	Делайте функции простыми и не храните информацию о состоянии в реализации. Поместите все связанные функции в один заголовочный файл и предоставьте вызывающей стороне этот интерфейс к своему программному модулю

logger.h

```
#ifndef LOGGER_H
#define LOGGER_H
void logging(const char* text);
#endif
```

Код вызывающей стороны

```
logging("Some text to log");
```

Для реализации функции, объявленной в файле *logger.h*, вызовите `printf`, которая добавит временную метку и выведет сообщение на `stdout`. Но что, если вызывающая сторона передаст недопустимые параметры, например нулевой указатель? Следует ли проверять входные данные и сообщать вызывающей стороне об ошибке? Придерживайтесь «Принципа самурая», который гласит, что возвращать информацию об ошибках программирования не нужно.

Название паттерна	Краткое описание
Принцип самурая	Либо возвращайте из функции управление, если все хорошо, либо не возвращайте вовсе. Если ошибку невозможно обработать, то аварийно завершайте программу

Передайте полученное от вызывающей стороны сообщение функции `printf`, и в случае недопустимых параметров программа просто «упадет», что облегчит вызывающей стороне поиск ошибок программирования, приведших к недопустимому входу.

logger.c

```
void logging(const char* text)
{
 time_t mytime = time(NULL);
 printf("%s %s\n", ctime(&mytime), text);
}
```

А что, если функция вызывается в многопоточной программе? Может ли переданная функции строке быть изменена другими потоками, или она должна оставаться неизменной, пока функция протоколирования не закончит работу? В примере выше вызывающая сторона должна передать `text` функции `logging` и отвечает за то, чтобы строка оставалась действительной, до возврата управления. Поэтому налицо паттерн «Буфер, принадлежащий вызывающей стороне». Это поведение должно быть документировано в интерфейсе функции.

Название паттерна	Краткое описание
Буфер, принадлежащий вызывающей стороне	Потребуйте, чтобы вызывающая сторона предоставила буфер и его размер функции, возвращающей данные. Внутри функции скопируйте данные в буфер при условии, что его размер достаточен

logger.h

```
/* Печатает текущую временную метку и переданную строку на stdout.
   Строка должна оставаться действительной, пока функция не вернет управление. */
void logging(const char* text);
```

Фильтрация источника сообщений

Теперь представьте, что все программные модули вызывают функцию протоколирования, чтобы записать что-то в журнал. Вывод может оказаться хаотическим, особенно если программа многопоточная.

Чтобы было проще найти нужную информацию, вы решаете добавить возможность конфигурирования, так чтобы печаталась информация только от сконфигурированных модулей. Для этого добавьте еще один параметр, кото-

рый будет идентифицировать текущий программный модуль. Добавьте функцию, разрешающую печатать вывод для заданного модуля.

logger.h

```
/* Печатает текущую временную метку и переданную строку на stdout.
   Строка должна оставаться действительной, пока функция не
   вернет управление. Параметр module идентифицирует программный
   модуль, вызвавший эту функцию. */
void logging(const char* module, const char* text);

/* Разрешает печатать вывод от указанного модуля. */
bool enableModule(const char* module);
```

Код вызывающей стороны

```
logging("MY-SOFTWARE-MODULE", "Some text to log");
```

Как отслеживать, сообщения от каких модулей следует печатать? Нужно ли хранить информацию о состоянии в глобальной переменной, или любая глобальная переменная дурно пахнет? Быть может, чтобы избежать глобальных переменных, нужно передавать всем функциям дополнительный параметр, в котором хранится информация о состоянии? Следует ли выделять необходимую память на все время работы программы? Ответ на эти вопросы ведет к реализации «Программного модуля с глобальным состоянием» с использованием «Вечной памяти».

Название паттерна	Краткое описание
Программный модуль с глобальным состоянием	Заведите один глобальный экземпляр, чтобы все связанные функции могли разделять его. Поместите все функции, работающие с этим экземпляром, в один заголовочный файл и предоставьте вызывающей стороне этот интерфейс к своему программному модулю
Вечная память	Размещайте данные в памяти, которая остается доступной в течение всего времени работы программы

logger.c

```
#define MODULE_SIZE 20
#define LIST_SIZE 10
typedef struct
{
  char module[MODULE_SIZE];
} LIST;
static LIST list[LIST_SIZE];
```

Этот список заполняется, когда вызывается функция, разрешающая печать сообщений от модулей:

logger.c

```
bool enableModule(const char* module)
{
 for(int i=0; i<LIST_SIZE; i++)
 {
  if(strcmp(list[i].module, "") == 0)
  {
   strcpy(list[i].module, module);
   return true;
  }
  if(strcmp(list[i].module, module) == 0)
  {
   return false;
  }
 }
 return false;
}
```

Этот код добавляет имя программного модуля в список, если в списке имеется свободная позиция и этого имени еще нет в списке. Видя возвращенное значение, вызывающая сторона может сказать, произошла ли ошибка, но не может сказать, какая. Вы не возвращаете код состояния; вы возвращаете только существенную информацию об ошибке, но нет никакого сценария, в котором вызывающая сторона могла бы по-разному отреагировать на описанные ситуации. Это поведение следует документировать в объявлении функции.

Название паттерна	Краткое описание
Возвращаемое значение	Используйте механизм C, предназначенный для получения информации о результате вызова функции: возвращаемое значение. Механизм возврата данных в C копирует результат функции и дает вызывающей стороне доступ к копии.
Возврат существенной информации об ошибке	Возвращайте только ту информацию об ошибке, которая существенна для вызывающей стороны. Информация существенна, если вызывающая сторона может на нее отреагировать.

logger.h

```
/* Разрешает печать сообщений для указанного модуля. Возвращает true
   в случае успеха и false в случае ошибки (больше нельзя разрешить
   ни одного модуля или для указанного модуля печать уже разрешена). */
bool enableModule(const char* module);
```

Условное протоколирование

Теперь, когда все активированные программные модули находятся в списке, можно реализовать условное протоколирование в зависимости от того, активен модуль или нет.

logger.c

```
void logging(const char* module, const char* text)
{
 time_t mytime = time(NULL);
 if(isInList(module))
 {
  printf("%s %s\n", ctime(&mytime), text);
 }
}
```

Но как реализовать функцию `isInList`? Существует несколько способов обхода списка. Можно было бы завести «Курсор», который предоставляет метод `getNext`, абстрагирующий внутреннюю структуру данных. Но так ли это здесь необходимо? В конце концов, нужно только пройтись по массиву в своем собственном программном модуле. Поскольку данные, подлежащие обходу, не пересекают границы API, для которого нужно поддерживать совместимость, можно обойтись гораздо более простым решением. Паттерн «Доступ по индексу» рекомендует напрямую использовать индекс для доступа к элементам.

Название паттерна	Краткое описание
Доступ по индексу	Предоставьте функцию, которая принимает индекс для адресации элемента в вашей структуре данных и возвращает его содержимое. Пользователь вызывает эту функцию в цикле для обхода всех элементов

logger.c

```
bool isInList(const char* module)
{
 for(int i=0; i<LIST_SIZE; i++)
 {
  if(strcmp(list[i].module, module) == 0)
  {
   return true;
  }
 }
 return false;
}
```

Теперь весь код протоколирования в зависимости от программного модуля написан. Он просто обходит структуру данных, увеличивая индекс. Такой же вид итерирования уже использовался в функции `enableModule`.

Несколько мест протоколирования

Далее вы решили помещать сообщения в несколько мест. До сих пор весь вывод направлялся на `stdout`, но вы хотите, чтобы вызывающая сторона могла

сконфигурировать ваш код, так чтобы он мог писать напрямую в файл. Обычно такое конфигурирование производится еще до первого действия. Начнем с функции, которая позволяет сконфигурировать, куда будут помещаться все последующие сообщения.

logger.h

```
/* Все будущие сообщения будут направляться на stdout */
void logToStdout();

/* Все будущие сообщения будут записываться в файл */
void logToFile();
```

Чтобы реализовать выбор места назначения, можно было бы просто добавить предложение `if` или `switch` и вызывать в нем нужную функцию в зависимости от сконфигурированного ранее места назначения. Однако после каждого добавления нового места назначения пришлось бы модифицировать эту часть кода. Принцип открытости-закрытости такие решения не поощряет. Гораздо лучше реализовать «Динамический интерфейс».

Название паттерна	Краткое описание
Динамический интерфейс	Определите общий интерфейс для различающейся функциональности в своем API и потребуйте, чтобы вызывающая сторона предоставила функцию обратного вызова для варианта этой функциональности, которую вы сможете вызвать из реализации своей функции.

logger.c

```
typedef void (*logDestination)(const char*);
static logDestination fp = stdoutLogging;

void stdoutLogging(const char* buffer)
{
 printf("%s", buffer);
}

void fileLogging(const char* buffer)
{
 /* еще не реализовано */
}

void logToStdout()
{
 fp = stdoutLogging;
}

void logToFile()
```

```
{
 fp = fileLogging;
}

#define BUFFER_SIZE 100
void logging(const char* module, const char* text)
{
 char buffer[BUFFER_SIZE];
 time_t mytime = time(NULL);
 if(isInList(module))
 {
  sprintf(buffer, "%s %s\n", ctime(&mytime), text);
  fp(buffer);
 }
}
```

Написанный ранее код сильно изменился, но теперь места назначения можно добавлять, не внося изменений в функцию `logging`. Функция `stdoutLogging` уже реализована, а `fileLogging` – пока нет.

Протоколирование в файл

Чтобы протоколировать в файл, можно было бы просто открывать и закрывать файл при каждой записи сообщения. Но это не очень эффективно, и если вы собираетесь протоколировать много информации, то этот подход будет отнимать много времени. А какие есть альтернативы? Можно было бы открыть файл один раз и оставить открытым. Но откуда вы знаете, когда открывать файл? И когда его закрывать?

Среди паттернов, описанных в этой книге, нет решающего эту проблему. Однако поиск в Google быстро находит подходящий паттерн: «Отложенный захват» (Lazy Acquisition). При первом вызове функции `fileLogging` откройте файл и оставьте его открытым. Сохранить дескриптор файла можно в «Вечной памяти».

Название паттерна	Краткое описание
Отложенный захват	Неявно инициализирует объект или данные при первом использовании (см. Michael Kirchner and Prashant Jain «Pattern-Oriented Software Architecture: Volume 3: Patterns for Resource Management» [Wiley, 2004]).
Вечная память	Размещайте данные в памяти, которая остается доступной в течение всего времени работы программы.

logger.c

```
void fileLogging(const char* buffer)
{
 static int fd = 0;  ❶
```

```
 if(fd == 0)
 {
  fd = open("log.txt", O_RDWR | O_CREAT, 0666);
 }
 write(fd, buffer, strlen(buffer));
}
```

❶ Такие статические переменные инициализируются один раз, а не при каждом вызове функции.

Чтобы не усложнять код, мы не обеспечили потокобезопасность. Потокобезопасный код должен был бы защитить «Отложенный захват» с помощью Мьютекса, гарантирующего, что захват производится только один раз.

А что с закрытием? Для некоторых приложений, в том числе рассматриваемого в этой главе, файл можно и не закрывать. Представьте, что вам нужно протоколировать ошибки, пока приложение работает, а после его закрытия вы полагаетесь на операционную систему, которая закроет все файлы, оставшиеся открытыми. Если вы боитесь, что информация не сохранится в случая сбоя системы, то можете время от времени сбрасывать файл на диск.

Кросс-платформенная обработка файлов

До сих пор протоколирование в файл велось в системах Linux, но вы хотите, чтобы код работал и на платформе Windows. Пока это не так.

Для поддержки нескольких платформ вы прежде всего рассматриваете «Избегание вариантов», стремясь к тому, чтобы код был одинаков на всех платформах. В случае записи файлов это возможно при использовании функций `fopen`, `fwrite` и `fclose`, доступных как в Linux, так и в Windows.

Название паттерна	Краткое описание
Избегание вариантов	Пользуйтесь стандартизованными функциями, имеющимися на всех платформах. Если стандартизованных функций нет, то подумайте, не стоит ли отказаться от соответствующей функциональности

Однако вы хотите, чтобы код протоколирования в файл был максимально эффективным, и готовы использовать для этого платформенно зависимые функции доступа к файлам. Но как реализовать платформенно зависимый код? Полное дублирование кодовой базы с одной версией для Windows и другой для Linux не годится, потому что внесение изменений и сопровождение дублированного кода превратило бы вашу жизнь в кошмар.

Вы решаете использовать в коде директивы `#ifdef` для различения платформ. Но не является ли и такой подход дублированием кода? Ведь если в коде присутствуют огромные блоки `#ifdef`, то вся программная логика в этих блоках дублируется. Как избежать дублирования кода, но поддержать несколько платформ?

И снова путь показывают паттерны. Сначала определите платформенно независимые интерфейсы для функциональности, требующей платформенно зависимых функций. Иными словами, определите «Уровень абстракции».

Название паттерна	Краткое описание
Уровень абстракции	Предоставьте API для каждого вида функциональности, требующего платформенно зависимого кода. В заголовочном файле объявляйте только платформенно независимые функции, а весь платформенно зависимый код, заключенный внутри `#ifdef`, помещайте в файл реализации. Вызывающая сторона должна будет включить только ваш заголовочный файл, но не платформенно зависимые файлы

logger.c

```
void fileLogging(const char* buffer)
{
 void* fileDescriptor = initiallyOpenLogFile();
 writeLogFile(fileDescriptor, buffer);
}

/* Открывает файл журнала при первом вызове.
   Работает в Linux и в Windows. */
void* initiallyOpenLogFile()
{
 ...
}

/* Записывает содержимое буфера в файл журнала.
   Работает в Linux и в Windows. */
void writeLogFile(void* fileDescriptor, const char* buffer)
{
 ...
}
```

За фасадом «Уровня абстракции» находятся «Изолированные примитивы» ваших вариантов кода. Это значит, что вы не заключаете в `#ifdef` несколько функций, а заводите только одну `#ifdef` в каждой функции. Следует ли заключать в `#ifdef` всю реализацию функции или только платформенно зависимую часть?

Нужно и то и другое. У вас должны быть «Атомарные примитивы». Хорошо, когда функция содержит только платформенно зависимый код. Если это не так, то нужно разбить функцию на более мелкие. Это лучший способ сделать платформенно зависимый код управляемым.

Название паттерна	Краткое описание
Изолированные примитивы	Изолируйте свои варианты кода. В файле реализации поместите многовариантный код в отдельные функции и вызывайте эти функции из основной программы, которая таким образом будет содержать только платформенно независимый код

Название паттерна	Краткое описание
Атомарные примитивы	Делайте примитивы атомарными. В каждой функции обрабатывайте только один вариант. Если, например, нужно обработать несколько вариантов операционных систем и оборудования, то заведите для каждого свою функцию

Ниже показаны реализации «Атомарных примитивов».

logger.c

```
void* initiallyOpenLogFile()
{
#ifdef __unix__
  static int fd = 0;
  if(fd == 0)
  {
   fd = open("log.txt", O_RDWR | O_CREAT, 0666);
  }
  return fd;
#elif defined _WIN32
  static HANDLE hFile = NULL;
  if(hFile == NULL)
  {
   hFile = CreateFile("log.txt", GENERIC_WRITE, 0, NULL,
                      CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
  }
  return hFile;
#endif
}

void writeLogFile(void* fileDescriptor, const char* buffer)
{
#ifdef __unix__
  write((int)fileDescriptor, buffer, strlen(buffer));
#elif defined _WIN32
  WriteFile((HANDLE)fileDescriptor, buffer, strlen(buffer), NULL, NULL);
#endif
}
```

Этот код шедевром не выглядит. Но, вообще-то, платформенно зависимый код редко выглядит красиво. Можно ли еще что-то сделать, чтобы код было удобнее читать и сопровождать? Один из возможных подходов – воспользоваться паттерном «Разделение реализаций вариантов», чтобы разнести реализации по отдельным файлам.

Название паттерна	Краткое описание
Разделение реализаций вариантов	Помещайте реализацию каждого варианта в отдельный файл и пофайлово выбирайте, что хотите компилировать для какой платформы

fileLinux.c

```
#ifdef __unix__
void* initiallyOpenLogFile()
{
 static int fd = 0;
 if(fd == 0)
 {
  fd = open("log.txt", O_RDWR | O_CREAT, 0666);
 }
 return fd;
}

void writeLogFile(void* fileDescriptor, const char* buffer)
{
 write((int)fileDescriptor, buffer, strlen(buffer));
}
#endif
```

fileWindows.c

```
#ifdef _WIN32
void* initiallyOpenLogFile()
{
 static HANDLE hFile = NULL;
 if(hFile == NULL)
 {
  hFile = CreateFile("log.txt", GENERIC_WRITE, 0, NULL,
                     CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
 }
 return hFile;
}

void writeLogFile(void* fileDescriptor, const char* buffer)
{
 WriteFile((HANDLE)fileDescriptor, buffer, strlen(buffer), NULL, NULL);
}
#endif
```

Эти файлы гораздо проще читать по сравнению со случаем, когда код для Linux и Windows находится вперемежку в одной функции. Кроме того, вместо условной компиляции платформенного кода с помощью директив `#ifdef`, те-

перь можно вообще избавиться от `#ifdef` и перенести выбор подлежащих компилированию файлов в Makefile.

Использование средства протоколирования

После внесения финальных изменений в функциональность протоколирования ваш код может отправлять сообщения от сконфигурированных программных модулей на `stdout` или в файл платформенно независимым способом. Ниже показано, как использовать протоколирование в программе:

```
enableModule("MYMODULE");
logging("MYMODULE", "Log to stdout");
logToFile();
logging("MYMODULE", "Log to file");
logging("MYMODULE", "Log to file some more");
```

Приняв все проектные решения и успешно реализовав их, вы чувствуете большое облегчение. Вы отрываете руки от клавиатуры и в восхищении смотрите на свой код. Вы поражены тем, с какой легкостью вопросы, поначалу казавшиеся такими трудными, были разрешены с помощью паттернов. Использование паттернов хорошо тем, что снимает бремя принятия сотен решений с ваших плеч.

Долгие поездки на машине ради исправления ошибок ушли в прошлое. Теперь вы просто получаете нужную отладочную информацию в файлах журналов. Заказчик доволен, потому что ошибки исправляются быстрее. Но важнее то, что ваша жизнь стала лучше. Вы можете создавать более качественные программы, и еще остается время, чтобы приходить домой пораньше.

Резюме

Вы написали код протоколирования шаг за шагом, применяя описанные в части I паттерны для решения проблем в порядке поступления. Поначалу у вас было много вопросов: как организовать файлы и как обрабатывать ошибки. Паттерны показали путь. Они руководили вашими действиями и упростили создание этого кода. Благодаря им вы стали лучше понимать, почему код выглядит и ведет себя именно так. На рис. 10.2 приведен обзор решений, которые помогли принять паттерны.

Разумеется, код еще можно улучшить в разных направлениях. Например, мы не ограничиваем размер журналов и не производим их ротацию. Мы не поддерживаем конфигурирование уровня протоколирования, чтобы не печатать лишнее, когда это не нужно. Чтобы не усложнять код, эти возможности опущены, но их можно добавить.

В следующей главе мы расскажем другую историю о том, как применить паттерны к еще одной программе промышленного уровня.

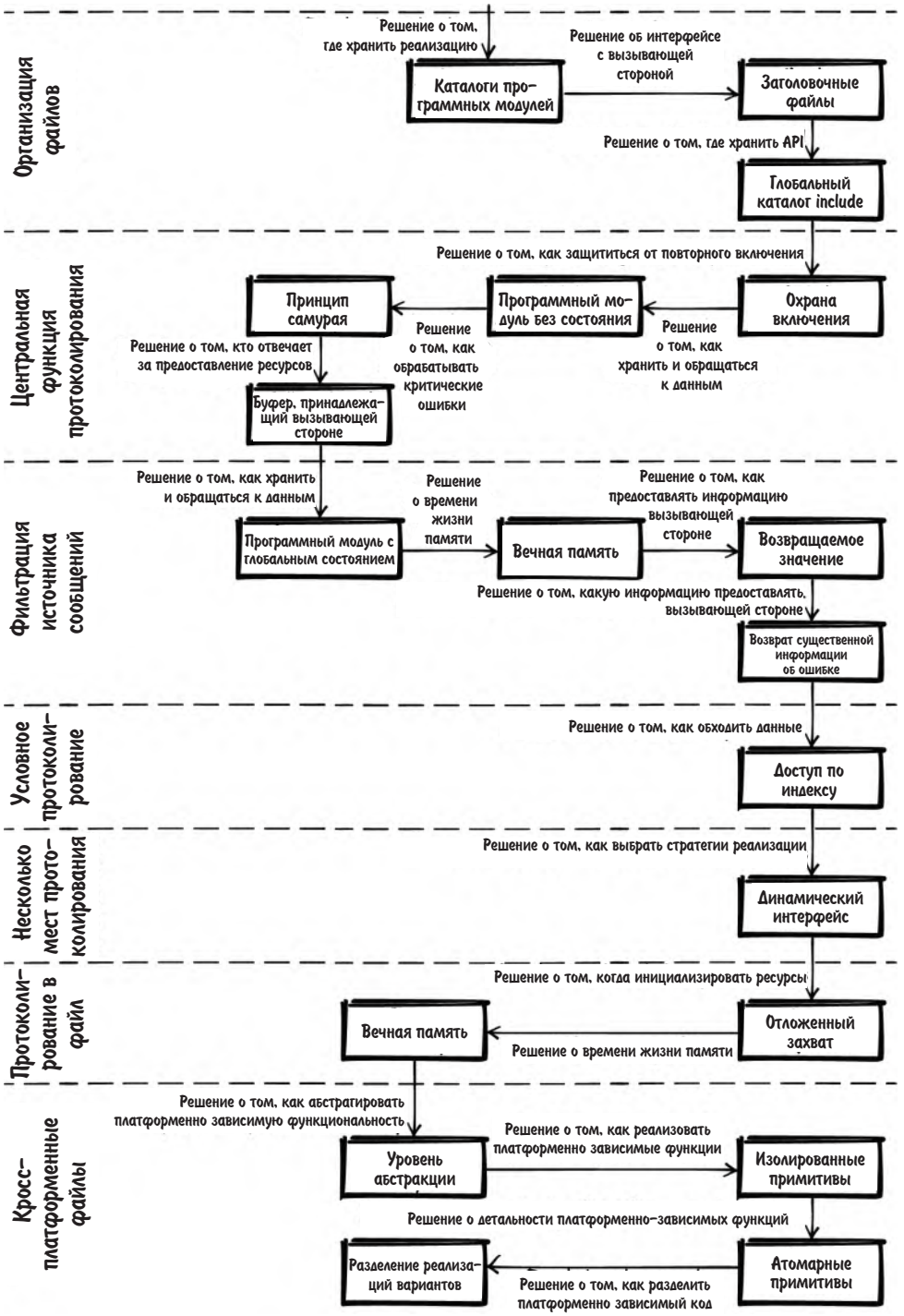


Рис. 10.2. Паттерны, примененные в этой истории

Глава 11

Построение системы управления пользователями

В этой главе рассказывается история о применении паттернов из первой части книги к сквозному примеру. Пример показывает, как проектные решения, принятые с помощью паттернов, приносят пользу программистам. Основа примера взята из реальной системы управления пользователями.

История о паттернах

Представьте, что вы только что окончили университет и начали работать в компании, разрабатывающей программное обеспечение. Начальник вручает вам спецификацию на часть программы для хранения имен и паролей пользователей и поручает реализовать ее. Программа должна проверять, что введенный пароль правилен, а также предоставлять функциональность для создания, удаления и просмотра пользователей.

Вы горите желанием доказать начальнику, какой вы хороший программист, но в самом начале в голове крутится множество вопросов. Следует ли поместить весь код в один файл? В университете вам говорили, что это дурной тон, но тогда сколько должно быть файлов? Какие части кода помещать в один файл? Нужно ли проверять параметры каждой функции? Должны ли функции возвращать подробную информацию об ошибке? Вас учили, как писать программы, которые работают, но не рассказывали, как писать хороший код, который было бы удобно сопровождать. Так что же делать? С чего начать?

Организация данных

Для ответа на эти вопросы начните с обзора описанных в этой книге паттернов, чтобы получить наставление о том, как писать хорошие программы на C. Начните с той части системы, которая отвечает за хранение имен и паролей пользователей. Ваши вопросы теперь должны быть сфокусированы на том, как хранить данные в программе. Следует ли хранить их в глобальных переменных? Или в локальных переменных внутри функции? Нужно ли выделять динамическую память?

Для начала рассмотрим, какую же задачу вы хотите решить в своем приложении: вы не уверены, как хранить данные об именах пользователей. Пока что не видно никакой необходимости делать эти данные постоянными; вы просто хотите иметь возможность создавать их и обращаться к ним во время выполнения. Кроме того, вы не хотите, чтобы сторона, вызывающая ваши функции, должна была возиться с выделением памяти и инициализацией этих данных.

Далее поищите паттерны, относящиеся к вашей конкретной проблеме. Еще раз просмотрите паттерны, касающиеся времени жизни данных и владения ими (глава 5). Внимательно прочитайте разделы «Проблема» во всех этих паттернах и найдите тот, который как можно точнее соответствует вашей проблеме и последствия которого для вас приемлемы. Это будет паттерн «Программный модуль с глобальным состоянием», который рекомендует использовать «Вечную память» в форме глобальных переменных с файловой областью видимости, чтобы данные были доступны только внутри этого файла.

Название паттерна	Краткое описание
Программный модуль с глобальным состоянием	Заведите один глобальный экземпляр, чтобы все связанные функции могли разделять его. Поместите все функции, работающие с этим экземпляром, в один заголовочный файл и предоставьте вызывающей стороне этот интерфейс к своему программному модулю
Вечная память	Размещайте данные в памяти, которая остается доступной в течение всего времени работы программы

```
#define MAX_SIZE 50
#define MAX_USERS 50

typedef struct
{
  char name[MAX_SIZE];
  char pwd[MAX_SIZE];
} USER;

static USER userList[MAX_USERS]; ❶
```

❶ Массив `userList` содержит данные о пользователях. Он доступен только из файла реализации. Поскольку он хранится в статической памяти, нет необходимости выделять для него память вручную (что сделало бы код более гибким, но и более сложным).

Хранение паролей

В этом упрощенном примере мы храним пароли в открытом виде. Никогда, ни при каких обстоятельствах не делайте этого в реальном приложении. Пароли нужно хранить в виде подсоленного хеша, вычисленного по открытому значению.

Организация файлов

Далее определим интерфейс с вызывающей стороной. Нужно, чтобы реализацию можно было изменить в будущем, не требуя внесения изменений в код вызывающей стороны. Теперь нужно решить, какую часть программы следует определить в интерфейсе, а какую – в файле реализации.

Эту проблему мы решим с помощью «Заголовочных файлов». Поместите в интерфейс (*h*-файл) как можно меньше кода (только то, что необходимо вызывающей стороне). Все остальное должно находиться в файлах реализации (*c*-файлах). Чтобы защититься от повторного включения заголовочного файла, воспользуйтесь паттерном «Охрана включения».

Название паттерна	Краткое описание
Заголовочные файлы	Включите в свой API объявления функций, реализующих функциональность, предоставляемую пользователям. Скройте все внутренние функции, внутренние данные и определения (реализации) функций в файлах реализации и не передавайте эти файлы пользователям
Охрана включения	Защищайте содержимое заголовочных файлов от повторного включения, чтобы разработчику, пользующемуся ими, не нужно было думать, сколько раз включен файл. Для этого можно использовать директиву `#ifdef` или `#pragma once`

user.h

```
#ifndef USER_H
#define USER_H

#define MAX_SIZE 50
#endif
```

user.c

```
#include "user.h"
#define MAX_USERS 50
typedef struct
{
  char name[MAX_SIZE];
  char pwd[MAX_SIZE];
} USER;
static USER userList[MAX_USERS];
```

Теперь вызывающая сторона может воспользоваться константой `MAX_SIZE` и узнать максимальную длину строк в программном модуле. По соглашению вызывающая сторона понимает, что все находящееся в *h*-файле использовать можно, а ничего из находящегося в *c*-файле – нельзя.

На следующем шаге предстоит убедиться, что ваши файлы с кодом хорошо отделены от кода вызывающей стороны, так что конфликтов по именам не возникнет. Нужно ли помещать все свои файлы в один каталог, или лучше

поместить *h*-файлы для всей кодовой базы в одно место, чтобы их было проще включать?

Создайте каталог программного модуля и поместите в него все файлы своего модуля – интерфейсы и реализации.

Название паттерна	Краткое описание
Каталоги программных модулей	Помещайте тесно связанные заголовочные файлы и файлы реализации в один каталог. Назовите этот каталог, так чтобы имя отражало функциональность, предоставляемую заголовочными файлами

При такой структуре каталога, как показано на рис. 11.1, легко найти все файлы, относящиеся к вашему коду. Теперь не нужно бояться, что имена ваших файлов реализации совпадут с именами других файлов.

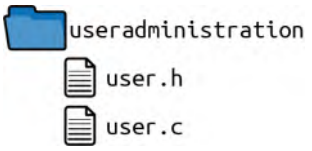


Рис. 11.1. Организация файлов

Аутентификация: обработка ошибок

Вот теперь можно приступить к реализации функциональности доступа к данным. Начнем с функции, которая проверяет, что введенный пользователем пароль совпадает с ранее сохраненным. Определите поведение функции, объявив ее в заголовочном файле и документировав его в комментариях, расположенных в непосредственной близости от объявления.

Функция должна информировать вызывающую сторону о том, правилен введенный пароль указанного пользователя или нет. Это можно сделать с помощью возвращаемого значения. Но какую информацию возвращать? Нужно ли подробно сообщать вызывающей стороне о любой ошибке?

Возвращайте только существенную информацию, потому что при реализации любой относящейся к безопасности функциональности принято сообщать только то, что абсолютно необходимо, и не больше. Вызывающая сторона не должна знать, что произошло: указанного пользователя не существует или указанный пароль неверен. Просто сообщите, прошла аутентификация или нет.

Название паттерна	Краткое описание
Возвращаемое значение	Используйте механизм C, предназначенный для получения информации о результате вызова функции: возвращаемое значение. Механизм возврата данных в C копирует результат функции и дает вызывающей стороне доступ к копии
Возврат существенной информации об ошибке	Возвращайте только ту информацию об ошибке, которая существенна для вызывающей стороны. Информация существенна, если вызывающая сторона может на нее отреагировать

user.h

```
/* Возвращает true, если пользователь с указанным именем существует
   и указанный пароль правилен. */
bool authenticateUser(char* username, char* pwd);
```

Этот комментарий точно определяет, какое значение должна возвращать функция, но ничего не говорит о том, что должно быть, когда входные данные недопустимы. Что делать, если, например, передан нулевой указатель? Следует ли проверять, что указатель не равен NULL, или просто игнорировать недопустимый вход?

Потребуйте, чтобы пользователь передавал допустимые данные, потому что противное означает ошибку программирования на вызывающей стороне, а такие ошибки не должны оставаться незамеченными. Согласно «Принципу самурая» программу следует завершать в случае недопустимых входных данных, и это поведение должно быть документировано в заголовочном файле.

Название паттерна	Краткое описание
Принцип самурая	Либо возвращайте управление, если все хорошо, либо не возвращайте вовсе. Если ошибку невозможно обработать, то аварийно завершайте программу

user.h

```
/* Возвращает true, если пользователь с указанным именем существует
   и указанный пароль правилен, в противном случае false.
   Завершает программу в случае недопустимых входных данных (строка NULL). */
bool authenticateUser(char* username, char* pwd);
```

user.c

```
bool authenticateUser(char* username, char* pwd)
{
 assert(username);
 assert(pwd);
 for(int i=0; i<MAX_USERS; i++)
 {
  if(strcmp(username, userList[i].name) == 0 &&
     strcmp(pwd, userList[i].pwd) == 0)
  {
   return true;
  }
 }
 return false;
}
```

Применяя «Принцип самурая», вы снимаете с вызывающей стороны бремя ответственности за проверку возвращенных значений, означающих недопус-

тимый вход, поскольку в таком случае программа просто «падает». Вы выбрали макрос `assert` вместо неконтролируемого аварийного останова (например, в результате передачи недопустимых параметров функции `strcmp`). Когда речь идет о вопросах безопасности, поведение программы должно быть определено даже в случае ошибок.

На первый взгляд, завершение программы – слишком радикальное решение, но при таком подходе вызов с неверными параметрами не останется незамеченным. В конечном итоге эта стратегия повышает надежность кода. Она препятствует появлению тонких ошибок (каковыми являются в том числе недопустимые параметры), которые могли бы проявиться позже в коде вызывающей стороны.

Аутентификация: протоколирование ошибок

Далее нужно регистрировать случаи вызова с недопустимыми параметрами. Протоколируйте ошибки, обнаруженные функцией `authenticateUser`; эта информация должна быть доступна при последующем аудите безопасности. Для протоколирования возьмите код из главы 10 или напишите более простой типа показанного ниже.

Название паттерна	Краткое описание
Протоколирование ошибок	Используйте разные каналы: один для предоставления информации об ошибке, существенной для вызывающей стороны, а другой для предоставления информации, существенной для разработчика. Например, записывайте отладочную информацию об ошибке в файл журнала и не возвращайте ее вызывающей стороне

Этот механизм протоколирования трудно предоставить на разных платформах, например в Linux и в Windows, потому что разные операционные системы предлагают разные функции для доступа к файлам. Кроме того, кросс-платформенный код трудно правильно реализовать и поддерживать. Как же сделать реализацию протоколирования максимально простой? Обратите внимание на паттерн «Избегание вариантов» и пользуйтесь стандартизованными функциями, имеющимися на любой платформе.

Название паттерна	Краткое описание
Избегание вариантов	Пользуйтесь стандартизованными функциями, имеющимися на всех платформах. Если стандартизованных функций нет, то подумайте, не стоит ли отказаться от соответствующей функциональности

К счастью, в стандарте C определены функции доступа к файлам, которые можно использовать в Windows и Linux. Хотя системно-зависимые функции могут быть более производительными и предоставлять дополнительные системно-зависимые возможности, здесь это несущественно. Просто пользуйтесь функциями, определенными в стандарте C.

Для реализации протоколирования вызывайте следующую функцию, если был обнаружен неверный пароль:

user.c

```
static void logError(char* username)
{
 char logString[200];
 sprintf(logString, "Ошибка при входе в систему. Пользователь:%s\n", username);
 FILE* f = fopen("logfile", "a+");  ❶
 fwrite(logString, 1, strlen(logString), f);
 fclose(f);
}
```

❶ Используйте платформенно независимые функции `fopen`, `fwrite` и `fclose`. Этот код работает в Windows и Linux, и нет никаких докучливых `#ifdef` для обработки платформенных вариантов.

Для хранения журнальной информации в коде используется паттерн «Сначала стек», потому что сообщение небольшое и в стек поместится. К тому же это самый простой способ, потому что не нужно возиться с освобождением памяти.

Название паттерна	Краткое описание
Сначала стек	По умолчанию размещайте переменные в стеке, это даст возможность воспользоваться механизмом автоматической очистки памяти

Добавление пользователей: обработка ошибок

Итак, сейчас у вас есть функция, которая проверяет правильность пароля для пользователя, хранящегося в списке, только вот список пользователей пока еще пуст. Чтобы его заполнить, напишите функцию, позволяющую вызывающей стороне добавить пользователя.

Проверяйте, что имена пользователей уникальны, и сообщайте вызывающей стороне, был ли пользователь добавлен успешно или произошла ошибка, например такой пользователь уже существует или в списке не осталось места.

Теперь вы должны решить, как информировать вызывающую сторону об ошибках. Следует ли использовать возвращаемое значение, или лучше установить переменную `errno`? И какую информацию предоставить, и какой тип данных использовать для ее возврата?

В данном случае лучше вернуть код состояния, потому что имеются разные ошибки, и вы хотите, чтобы вызывающая сторона могла их различать. А в случае недопустимых параметров просто завершайте программу («Принцип самурая»). Определите коды ошибок в своем интерфейсе, чтобы у вас и у вызывающей стороны было общее понимание и вызывающая сторона могла правильно отреагировать на ошибку.

Название паттерна	Краткое описание
Возврат кода состояния	Используйте возвращаемое значение функции для возврата информации о состоянии. Возвращайте значение, представляющее конкретное состояние. Вызывающая и вызываемая сторона должны одинаково интерпретировать возвращаемые значения

user.h

```
typedef enum{
  USER_SUCCESSFULLY_ADDED,
  USER_ALREADY_EXISTS,
  USER_ADMINISTRATION_FULL
} USER_ERROR_CODE;

/* Добавляет нового пользователя с именем 'username' и паролем 'pwd' (аварийно
   завершает программу в случае NULL). Возвращает USER_SUCCESSFULLY_ADDED в
   случае успеха, USER_ALREADY_EXISTS, если пользователь с таким именем уже
   существует, и USER_ADMINISTRATION_FULL, если больше нельзя добавить ни
   одного пользователя. */
USER_ERROR_CODE addUser(char* username, char* pwd);
```

Далее реализуйте функцию `addUser`. Проверьте, существует ли пользователь с таким именем, и, если нет, добавьте пользователя. Чтобы разделить эти задачи, воспользуйтесь паттерном «Разбиение функции», чтобы разнести обязанности по разным функциям. Сначала напишите функцию, проверяющую существование пользователя.

Название паттерна	Краткое описание
Разбиение функции	Разбейте ее на части. Выделите часть функции, которая кажется полезной сама по себе, создайте из нее новую функцию и вызовите ее

user.c

```
static bool userExists(char* username)
{
 for(int i=0; i<MAX_USERS; i++)
 {
  if(strcmp(username, userList[i].name) == 0)
  {
   return true;
  }
 }
 return false;
}
```

Теперь эту функцию можно вызвать из функции добавления пользователей, чтобы добавлять только тех пользователей, которые еще не существуют. Надо ли проверять существование в самом начале функции или непосредственно перед добавлением пользователя в список? Какое решение сделает код проще для чтения и сопровождения?

Разместите «Проверку условий» в начале функции и возвращайте управление немедленно, если действие невозможно выполнить, потому что пользователь уже существует. Если условия проверяются в начале функции, то следить за потоком программы проще.

Название паттерна	Краткое описание
Проверка условий	Сначала проверьте выполнение обязательных предусловий и сразу же верните управление, если они не выполняются

user.c

```
USER_ERROR_CODE addUser(char* username, char* pwd)
{
 assert(username);
 assert(pwd);
 if(userExists(username))
 {
  return USER_ALREADY_EXISTS;
 }
 for(int i=0; i<MAX_USERS; i++)
 {
  if(strcmp(userList[i].name, "") == 0)
  {
   strcpy(userList[i].name, username);
   strcpy(userList[i].pwd, pwd);
   return USER_SUCCESSFULLY_ADDED;
  }
 }
 return USER_ADMINISTRATION_FULL;
}
```

Написанный до сих пор код позволяет добавлять новых пользователей и проверять для них правильность пароля.

Итерирование

Далее нужно добавить возможность прочитать имена всех пользователей путем реализации итератора. Можно, конечно, просто предоставить интерфейс для доступа к массиву `userList` по индексу, но тогда возникнут проблемы, если вы захотите изменить внутреннюю структуру данных (например, перейти на связанный список) или если одна вызывающая сторона захочет обратиться к массиву, когда другая его модифицирует.

Чтобы предоставить вызывающей стороне интерфейс итератора, решающий вышеупомянутые проблемы, реализуйте паттерн «Курсор», который использует «Описатель», чтобы скрыть детали внутренней структуры.

Название паттерна	Краткое описание
Курсор	Создайте экземпляр курсора, указывающий на элемент структуры данных. Функция итерирования принимает этот экземпляр итератора в качестве аргумента, получает элемент, на который указывает итератор, и модифицирует экземпляр итератора, так чтобы он указывал на следующий элемент. Пользователь в цикле вызывает эту функцию, чтоб получать элементы по одному
Описатель	Заведите функцию, создающую контекст, с которым будет работать вызывающая сторона, и возвращающую абстрактный указатель на внутренние данные в этом контексте. Потребуйте, чтобы вызывающая сторона передавала этот указатель всем вашим функциям, которые смогут тогда воспользоваться внутренними данными для хранения информации о состоянии и ресурсов

user.h

```
typedef struct ITERATOR* ITERATOR;

/* Создает экземпляр итератора. Возвращает NULL в случае ошибки. */
ITERATOR createIterator();

/* Возвращает следующий элемент, полученный от итератора. */
char* getNextElement(ITERATOR iterator);

/* Уничтожает экземпляр итератора. */
void destroyIterator(ITERATOR iterator);
```

Вызывающая сторона полностью контролирует создание и уничтожение итератора. Таким образом, мы имеем сочетание паттернов «Единоличное владение» и «Экземпляр, принадлежащий вызывающей стороне». Вызывающая сторона может просто создать «Описатель» итератора и использовать его для доступа к списку имен пользователей. Если создание завершилось неудачно, то об этом скажет «Специальное возвращаемое значение» `NULL`. Такое использование специального возвращаемого значения вместо явного кода ошибки упрощает работу с функцией, потому что не нужны дополнительные параметры для возврата информации об ошибке. Закончив работу с итератором, вызывающая сторона может уничтожить «Описатель».

Название паттерна	Краткое описание
Единоличное владение	Уже в момент выделения памяти ясно и недвусмысленно определите, где она должна быть освобождена и кто за это отвечает
Экземпляр, принадлежащий вызывающей стороне	Потребуйте, чтобы вызывающая сторона передавала вашим функциям экземпляр, используемый для хранения ресурса и информации о состоянии. Предоставьте явные функции для создания и уничтожения таких экземпляров, чтобы вызывающая сторона могла контролировать время их существования
Специальное возвращаемое значение	Используйте возвращаемое значение для возврата вычисленных функцией данных, но зарезервируйте одно или несколько специальных значений на случай ошибки

Поскольку интерфейс предоставляет вызывающей стороне явные функции для создания и уничтожения итератора, это естественным образом ведет к отдельным функциям для инициализации и очистки ресурсов, связанных с итератором. «Объектная обработка ошибок» хороша тем, что привносит четко разграниченные обязанности в ваши функции, что упростит их расширение впоследствии, если возникнет такая необходимость. Это разграничение хорошо видно в следующем коде, где весь код инициализации находится в одной функции, а весь код очистки – в другой.

Название паттерна	Краткое описание
Объектная обработка ошибок	Поместите инициализацию и очистку в разные функции по аналогии с идеей конструкторов и деструкторов в объектно ориентированном программировании

user.c

```
struct ITERATOR
{
  int currentPosition;
  char currentElement[MAX_SIZE];
};

ITERATOR createIterator()
{
 ITERATOR iterator = (ITERATOR) calloc(sizeof(struct ITERATOR),1);
  return iterator;
}

char* getNextElement(ITERATOR iterator)
{
 if(iterator->currentPosition < MAX_USERS)
 {
  strcpy(iterator->currentElement,userList[iterator->currentPosition].name);
```

```
  iterator->currentPosition++;
 }
 else
 {
  strcpy(iterator->currentElement, "");
 }
 return iterator->currentElement;
}

void destroyIterator(ITERATOR iterator)
{
 free(iterator);
}
```

Как в этом коде передать вызывающей стороне имя пользователя? Следует ли просто предоставить ей указатель на данные? А если вы копируете данные в буфер, то кто должен его выделить?

В этой ситуации буфер для строки выделяет вызываемая сторона. Тогда вызывающая сторона имеет полный доступ к строке, но не имеет возможности изменить ее в `userList`. Кроме того, она может не опасаться, что другой поток изменит данные, пока она будет их читать.

Название паттерна	Краткое описание
Вызываемая сторона выделяет память	Выделите буфер нужного размера в самой функции, возвращающей данные. Скопируйте данные в буфер и верните указатель на этот буфер

Применение системы управления пользователями

Итак, вы написали код управления пользователями. Ниже показано, как им можно воспользоваться.

```
char* element;
addUser("A", "pass");
addUser("B", "pass");
addUser("C", "pass");

ITERATOR it = createIterator();

while(true)
{
 element = getNextElement(it);
 if(strcmp(element, "") == 0)
 {
  break;
 }
```

```
    printf("Пользователь: %s ", element);
    printf("Аутентификация успешна? %d\n", authenticateUser(element, "pass"));
  }

  destroyIterator(it);
```

В этой главе паттерны помогли вам спроектировать окончательный код. Теперь вы можете сказать начальнику, что справились с задачей реализации требуемой системы хранения имен и паролей пользователей. Применив основанный на паттернах подход к проектированию, вы воспользовались документированными решениями, проверенными на практике.

Резюме

В этой главе вы писали код шаг за шагом, применяя описанные в части I паттерны для решения проблем по мере возникновения. В начале было много вопросов о том, как организовать файлы и обрабатывать ошибки. Паттерны показали путь. Они упростили пошаговое построение кода. Благодаря им вы стали лучше понимать, почему код выглядит и ведет себя именно так. На рис. 11.2 показано, какие паттерны вы применяли. Вы видите, сколько решений пришлось принять и сколько из них направлялось паттернами.

Построенная система управления пользователями содержит базовые средства для добавления, поиска и аутентификации пользователей. Конечно, в нее можно добавить много других функций, например: изменение пароля, хранение паролей в зашифрованном виде или проверка того, что пароль отвечает определенным критериям безопасности. Эта дополнительная функциональность не рассматривалась, чтобы было проще понять ту сторону разработки, которая связана с паттернами.

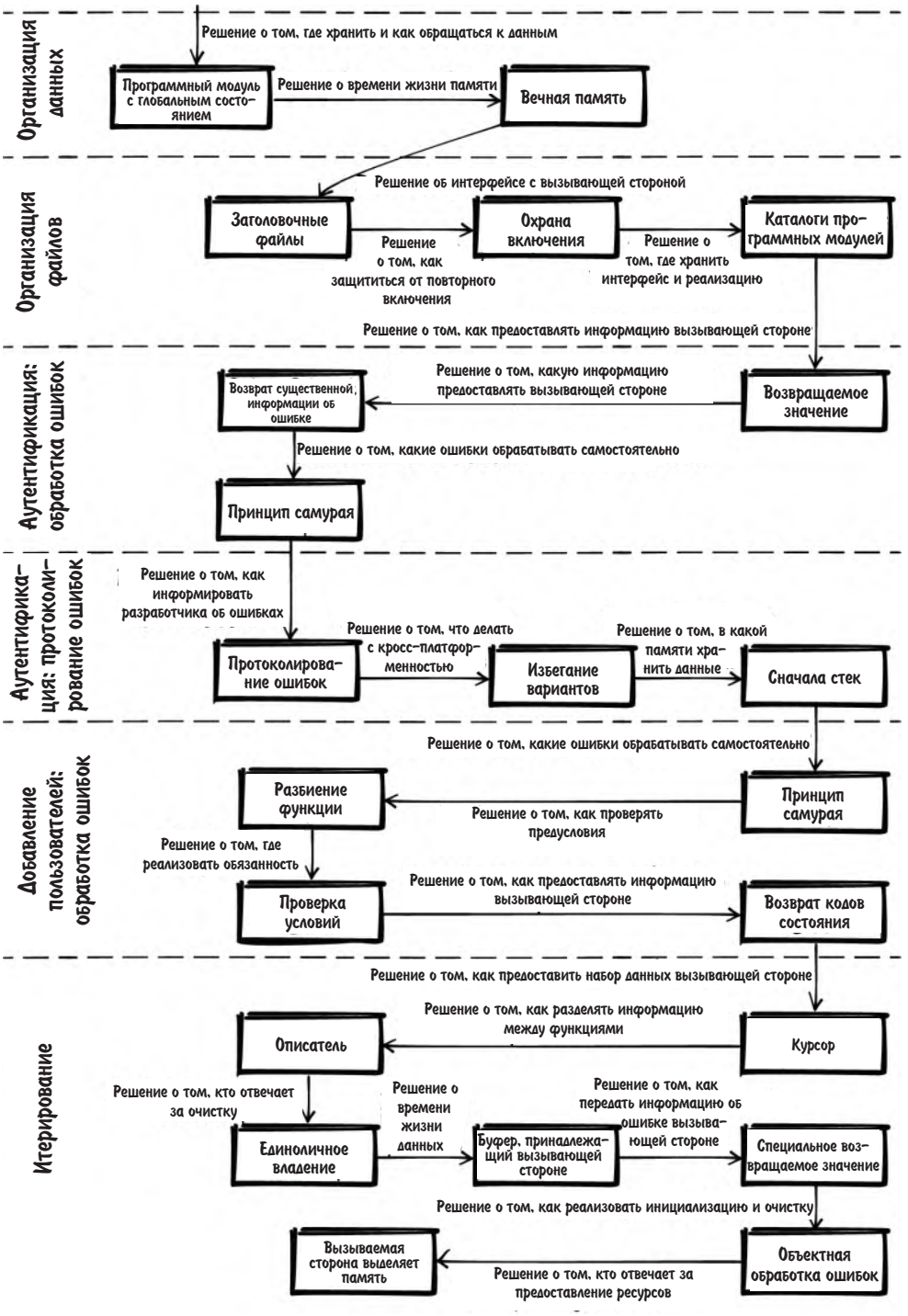


Рис. 11.2. Паттерны, примененные в этой истории

Глава 12

Заключение

Чему вы научились

Прочитав эту книгу, вы познакомились с некоторыми продвинутыми идеями программирования на C. Глядя на примеры больших программ, вы теперь будете понимать, почему код выглядит так, а не иначе. Вы знаете, какие рассуждения стоят за выбором проектных решений. Например, вы теперь понимаете, почему в примере драйвера Ethernet, приведенном в предисловии к этой книге, имеется явная функция `driverCreate` и структура данных `DRIVER_HANDLE` для хранения информации о состоянии. Паттерны из первой части направляли решения, принятые в этом примере, равно как и во многих других, встречающихся в книге.

Истории о паттернах во второй части продемонстрировали применение паттернов и постепенное построение кода. Когда вы столкнетесь с очередной проблемой при программировании на C, просмотрите разделы «Проблема» в описаниях паттернов и решите, какой из них отвечает вашей проблеме. Если такой обнаружится, то вам повезло, потому что вы сможете следовать наставлениям паттерна.

Для дополнительного чтения

Эта книга поможет вам вырасти от начинающего до опытного программиста на C. Вот еще несколько книг, которые помогли лично мне отточить навыки программирования на C.

- В книге Robert C. Martin «Clean Code: A Handbook of Agile Software Craftsmanship» (Prentice Hall, 2008) обсуждаются базовые принципы реализации высококачественного кода, который должен работать длительное время. Это полезное чтение для любого программиста; рассматриваются такие темы, как тестирование, документация, стиль кодирования и многое другое.
- В книге James W. Grenning «Test-Driven Development for Embedded C» (Pragmatic Bookshelf, 2011) используется сквозной пример для объяснения того, как писать автономные тесты для программ на C, работающих на уровне, близком к оборудованию.

- Книга Peter van der Linden «Expert C Programming» (Prentice Hall, 1994) – одна из первых, посвященных продвинутому программированию на C. В ней подробно описывается синтаксис C и типичные подводные камни. Обсуждаются также вопросы управления памятью в C и принципы работы компоновщика.
- К моей книге близко примыкает книга Adam Tornhill «Patterns in C» (Leanpub, 2014). В ней также описываются паттерны с упором на том, как реализовать на C паттерны из книги «банды четырех».

Заключительные замечания

По сравнению с новичком, только приступающим к изучению C, вы теперь обладаете знаниями о методах, применяемых при построении крупномасштабных производственных программ на C. Вы знаете, как:

- обрабатывать ошибки, даже в отсутствие механизма исключений;
- управлять памятью, хотя в вашем распоряжении нет сборщика мусора и деструкторов, автоматически освобождающих память;
- реализовывать гибкие интерфейсы, даже когда нет встроенных механизмов абстрагирования;
- организовывать файлы и код в отсутствие классов и пакетов.

Теперь вы можете писать на C, несмотря на отсутствие в нем некоторых удобств, присущих современным языкам программирования.

Об авторе

Кристофер Прешерн организует конференции по паттернам проектирования и другие мероприятия с целью улучшить владение паттернами. Работая программистом в компании ABB, он собрал и документировал практические знания о том, как писать код производственного качества. Он читал лекции по кодированию и контролю качества в Технологическом университете Граца. Имеет степень доктора философии по компьютерным наукам.

Об иллюстрации на обложке

На обложке этой книги изображен какаду инка (*Lophochroa leadbeateri*), или какаду Лидбитера. Этот какаду среднего размера был отрыт майором Томасом Митчеллом, исследователем и описателем Юго-Восточной Австралии. Обитает в засушливых и полузасушливых частях Австралии, предпочитает лесистые местности, где питается семенами растений. Оперение в основном белое и бледно-розовое, более насыщенного розового цвета под крыльями. Хохолок окрашен в ярко-красный, желтый и белый цвета. Самцы и самки отличаются мало; самцы немного крупнее, с коричневыми глазами, тогда как у самок красноватые глаза и более широкая желтая полоса на хохолке.

Какаду инка популярны в качестве домашних питомцев, хотя это очень общительные птицы, требующие от хозяина немалого внимания. В природе гнездятся парами и нуждаются в большой территории, поэтому их среда обитания уязвима к фрагментации. Хотя этот вид не считается таксоном минимального риска, их количество уменьшилось в связи с вырубкой лесов. Им также угрожает браконьерское отлавливание для содержания в неволе. Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой исчезновения, все они важны для нашего мира.

Предметный указатель

Р

С

У

Ф

Х

Ц

Э

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
Тел.: **+7(499) 782-38-89**. Электронная почта: **books@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:
www.galaktika-dmk.com.

Прешерн К.

Язык C. Мастерство программирования

Принципы, практики и паттерны

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Слинкин А. Н.*
Корректор *Абросимова Л. А.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Печать цифровая. Усл. печ. л. 24,29.
Тираж 100 экз.

Веб-сайт издательства: **www.dmkpress.com**

O'REILLY

Авторитетные рекомендации по программированию на C найти трудно. Для объектно-ориентированных языков в них нет недостатка, но для C их на удивление мало.

В этой книге начинающие и опытные программисты на C найдут наставления по принятию проектных решений, включая пошаговое применение паттернов к сквозным примерам.

Кристофер Прешерн, один из ведущих членов сообщества паттернов проектирования, рассказывает, как организовать программу на C, как обрабатывать ошибки и проектировать гибкие интерфейсы. Ищете ли вы конкретный паттерн или вам нужен обзор проектных решений, относящихся к определенной теме, эта книга будет в помощь.

В части I вы научитесь реализовывать проверенные практикой подходы к программированию на языке C; часть II покажет, как паттерны программирования на C применяются к реализации более крупных программ.

Приведены следующие группы паттернов:

- обработка ошибок;
- возврат информации об ошибках;
- управление памятью;
- возврат данных из C-функций;
- время жизни данных и владение данными;
- гибкие интерфейсы итераторов;
- организация файлов в модульных программах;
- бегство из ада #ifdef.

«Изобилующая реальными примерами, эта книга — отличный источник, который поможет вам сделать свой код более чистым и удобным для сопровождения».

Дэвид Гриффитс, автор книги «Head First C»

«Книга наставит начинающих на путь, ведущий к программам на C корпоративного качества».

Роберт Ханмер, программный архитектор

Кристофер Прешерн работает программистом в компании ABB, собрал и задокументировал практические знания о том, как писать код производственного качества. Читает лекции по программированию и контролю качества в Технологическом университете Граца.

ISBN 978-6-01810-340-7