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.

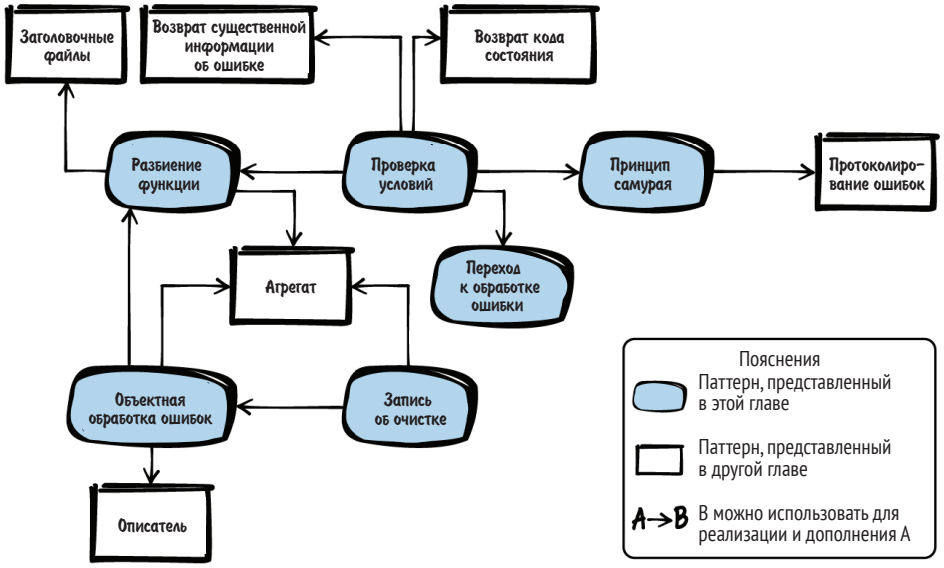

Рис. 1.1. Обзор паттернов для обработки ошибок

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 *Adam* 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.