Проектирование классов и наследование

Наиболее важный аспект разработки программного обеспечения — ясно понимать, что именно вы пытаетесь построить.

— Бьярн Страуструп (Bjarne Stroustrup)

Какого вида классы предпочитает разрабатывать и строить ваша команда? Почему?

Интересно, что большинство рекомендаций данного раздела вызваны в первую очередь вопросами зависимостей. Например, наследование — вторая по силе взаимосвязь, которую можно выразить в C++ (первая — отношение дружбы), и такую сильную связь надо использовать очень осторожно и продуманно.

В этом разделе мы сконцентрируем внимание на ключевых вопросах проектирования классов — как сделать это правильно, как не допустить ошибку, избежать ловушек, и в особенности — как управлять зависимостями.

В следующем разделе мы обратимся к Большой Четверке специальных функций — конструктору по умолчанию, копирующему конструктору, копирующему присваиванию и деструктору.

В этом разделе мы считаем самой важной рекомендацию 33 — “Предпочитайте минимальные классы монолитным”.

32. Ясно представляйте, какой вид класса вы создаете

Резюме

Существует большое количество различных видов классов, и следует знать, какой именно класс вы создаете.

Обсуждение

Различные виды классов служат для различных целей и, таким образом, следуют различным правилам.

Классы-значения (например, `std::pair`, `std::vector`) моделируют встроенные типы. Эти классы обладают следующими свойствами.

- Имеют открытые деструктор, копирующий конструктор и присваивание с семантикой значения.
- Не имеют виртуальных функций (включая деструктор).
- Предназначены для использования в качестве конкретных классов, но не в качестве базовых (см. рекомендацию 35).
- Обычно размещаются в стеке или являются непосредственными членами другого класса.

Базовые классы представляют собой строительные блоки иерархии классов. Базовый класс обладает следующими свойствами.

- Имеет деструктор, который является либо открытым и виртуальным, либо защищенным и невиртуальным (см. рекомендацию 50), а также копирующий конструктор и оператор присваивания, не являющиеся открытыми (см. рекомендацию 53).
- Определяет интерфейс посредством виртуальных функций.
- Обычно объекты такого класса создаются динамически в куче как часть объекта производного класса и используются посредством (интеллектуальных) указателей.

Говоря упрощенно, классы свойств представляют собой шаблоны, которые несут информацию о типах. Класс свойств обладает следующими характеристиками.

- Содержит только операторы `typedef` и статические функции. Класс не имеет модифицируемого состояния или виртуальных функций.
- Обычно объекты данного класса не создаются (конструкторы могут быть заблокированы).

Классы стратегий (обычно шаблоны) являются фрагментами сменного поведения. Классы стратегий обладают следующими свойствами.

- Могут иметь состояния и виртуальные функции, но могут и не иметь их.
- Обычно объекты данного класса не создаются, и он выступает в качестве базового класса или члена другого класса.

Классы исключений представляют собой необычную смесь семантики значений и ссылок. При генерации исключений они передаются по значению, но должны перехватываться по ссылке (см. рекомендацию 73). Классы исключений обладают следующими свойствами.

- Имеют открытый деструктор и конструкторы, не генерирующие исключений (в особенности копирующий конструктор, генерация исключения в котором приводит к завершению работы программы).
- Имеют виртуальные функции и часто реализуют клонирование (см. рекомендацию 54).
- Предпочтительно делать их производными от `std::exception`.

Вспомогательные классы обычно поддерживают отдельные идиомы (например, RAII — см. рекомендацию 13). Важно, чтобы их корректное использование не было сопряжено с какими-либо трудностями и наоборот — чтобы применять их некорректно было очень трудно (например, см. рекомендацию 53).

Ссылки

[Abrahams01b] • [Alexandrescu00a] • [Alexandrescu00b] • [Alexandrescu01] §3 • [Meyers96] §13 • [Stroustrup00] §8.3.2, §10.3, §14.4.6, §25.1 • [Vandevoorde03] §15

33. Предпочитайте минимальные классы монолитным

Резюме

Разделяй и властвуй: небольшие классы легче писать, тестировать и использовать. Они также применимы в большем количестве ситуаций. Предпочитайте такие небольшие классы, которые воплощают простые концепции, классам, пытающимся реализовать как несколько концепций, так и сложные концепции (см. рекомендации 5 и 6).

Обсуждение

Разработка больших причудливых классов — типичная ошибка новичка в объектно-ориентированном проектировании. Перспектива иметь класс, который предоставляет полную и сложную функциональность "в одном флаконе", может оказаться очень привлекательной. Однако подход, состоящий в разработке небольших, минимальных классов, которые легко комбинировать, на практике по ряду причин оказывается более успешен для систем любого размера и сложности.

- Минимальный класс воплощает одну концепцию на соответствующем уровне детализации. Монолитный класс обычно включает несколько отдельных концепций, и использование только одной из них влечет за собой излишние накладные расходы (см. рекомендации 5 и 11).
- Минимальный класс легче понять и проще использовать (в том числе повторно).
- Минимальный класс проще в употреблении. Монолитный класс часто должен использоваться как большое неделимое целое. Например, монолитный класс `Matrix` может попытаться реализовать и использовать экзотическую функциональность — такую как вычисление собственных значений матрицы — даже если большинству пользователей этого класса требуются всего лишь азы линейной алгебры. Лучшим вариантом будет реализация различных функциональных областей в виде функций, не являющихся членами, которые работают с минимальным типом `Matrix`. Тогда эти функциональные области могут быть протестированы и использованы отдельно только теми пользователями, кто в них нуждается (см. рекомендацию 44).
- Монолитные классы снижают инкапсуляцию. Если класс имеет много функций-членов, которые не обязаны быть членами, но тем не менее являются таковыми (таким образом обеспечивается излишняя видимость закрытой реализации), то закрытые члены-данные класса становятся почти столь же плохими с точки зрения дизайна, как и открытые переменные.
- Монолитные классы обычно являются результатом попыток предсказать и предоставить "полное" решение некоторой проблемы; на практике же такие действия почти никогда не приводят к успешному результату.
- Монолитные классы сложнее сделать корректными и безопасными в связи с тем, что при их разработке зачастую нарушается принцип "Один объект — одна задача" (см. рекомендации 5 и 44).

Ссылки

[Cargill92] pp. 85-86, 152, 174-177 • [Lakos96] §0.2.1-2, §1.8, §8.1-2 • [Meyers97] §18 • [Stroustrup00] §16.2.2, §23.4.3.2, §24.4.3 • [Sutter04] §37-40

34. Предпочитайте композицию наследованию

Резюме

Избегайте “налога на наследство”: наследование — вторая по силе после отношения дружбы взаимосвязь, которую можно выразить в C++. Сильные связи нежелательны, и их следует избегать везде, где только можно. Таким образом, следует предпочитать композицию наследованию, кроме случаев, когда вы точно знаете, что делаете и какие преимущества дает наследование в вашем проекте.

Обсуждение

Наследованием часто злоупотребляют даже опытные разработчики. Главное правило в разработке программного обеспечения — снижение связности. Если взаимоотношение можно выразить несколькими способами, используйте самую слабую из возможных взаимосвязей.

Известно, что наследование — практически самое сильное взаимоотношение, которое можно выразить средствами C++; сильнее его только отношение дружбы, и пользоваться им следует только при отсутствии функционально эквивалентной более слабой альтернативы. Если вы можете выразить отношения классов с использованием только лишь композиции, следует использовать этот способ.

В данном контексте “композиция” означает простое использование некоторого типа в виде переменной-члена в другом типе. В этом случае вы можете хранить и использовать объект таким образом, который обеспечивает вам контроль над степенью взаимосвязи.

Композиция имеет важные преимущества над наследованием.

- *Большая гибкость без влияния на вызывающий код*: закрытые члены-данные находятся под полным вашим контролем. Вы можете хранить их по значению, посредством (интеллектуального) указателя или с использованием идиомы Pimpl (см. рекомендацию 43), при этом переход от одного способа хранения к другому никак не влияет на код вызывающей функции: все, что при этом меняется, — это реализация функций-членов класса, использующих упомянутые члены-данные. Если вы решите, что вам требуется иная функциональность, вы можете легко изменить тип или способ хранения члена при полной сохранности открытого интерфейса. Если же вы начнете с открытого наследования, то скорее всего вы не сможете легко и просто изменить ваш базовый класс в случае необходимости (см. рекомендацию 37).
- *Большая обособленность в процессе компиляции, уменьшение времени компиляции*. Хранение объекта посредством указателя (предпочтительно — интеллектуального указателя), а не в виде непосредственного члена или базового класса позволяет также снизить зависимости заголовочных файлов, поскольку объявление указателя на объект не требует полного определения класса этого объекта. Наследование, напротив, всегда требует видимости полного определения базового класса. Распространенная методика состоит в том, чтобы собрать все закрытые члены воедино посредством одного непрозрачного указателя (идиома Pimpl, см. рекомендацию 43).
- *Меньше странностей*. Наследование от некоторого типа может вызвать проведение поиска имен среди функций и шаблонов функций, определенных в том же пространстве имен, что и упомянутый тип. Этот тонкий момент с трудом поддается отладке (см. также рекомендацию 58).

- *Большая применимость*. Не все классы проектируются с учетом того, что они будут выступать в роли базовых (см. рекомендацию 35). Однако большинство классов вполне могут справиться с ролью члена.
- *Большая надежность и безопасность*. Более сильное связывание путем наследования затрудняет написание безопасного в смысле ошибок кода (см. [Sutter02] §23).
- *Меньшая сложность и хрупкость*. Наследование приводит к дополнительным усложнениям, таким как сокрытие имен и другим, возникающим при внесении изменений в базовый класс.

Конечно, это все не аргументы против наследования как такового. Наследование предоставляет программисту большие возможности, включая заменимость и/или возможность перекрытия виртуальных функций (см. рекомендации с 36 по 39 и подраздел исключений данной рекомендации). Но не платите за то, что вам не нужно; если вы можете обойтись без наследования, вам незачем мириться с его недостатками.

Исключения

Используйте открытое наследование для моделирования заменимости (см. рекомендацию 37).

Даже если от вас не требуется предоставление отношения заменимости вызывающим функциям, вам может понадобиться закрытое или защищенное наследование в перечисленных далее ситуациях (мы постарались хотя бы грубо отсортировать их в порядке уменьшения распространенности).

- Если вам требуется перекрытие виртуальной функции.
- Если вам нужен доступ к защищенному члену.
- Если вам надо создавать объект до используемого, а уничтожать — после, сделайте его базовым классом.
- Если вам приходится заботиться о виртуальных базовых классах.
- Если вы знаете, что получите выгоду от оптимизации пустого базового класса и что в вашем случае она будет выполнена используемым вами компилятором (см. рекомендацию 8).
- Если вам требуется управляемый полиморфизм, т.е. отношение заменимости, которое должно быть видимо только определенному коду (посредством дружбы).

Ссылки

[Cargill92] pp. 49-65, 101-105 • [Cline99] §5.9-10, 8.11-12, 37.04 • [Dewhurst03] §95 • [Lakos96] §1.7, §6.3.1 • [McConnell93] §5 • [Meyers97] §40 • [Stroustrup00] §24.2-3 • [Sutter00] §22-24, §26-30 • [Sutter02] §23

35. Избегайте наследования от классов, которые не спроектированы для этой цели

Резюме

Классы, предназначенные для автономного использования, подчиняются правилам проектирования, отличным от правил для базовых классов (см. рекомендацию 32). Использование автономных классов в качестве базовых является серьезной ошибкой проектирования и его следует избегать. Для добавления специфического поведения предпочтительно вместо функций-членов добавлять обычные функции (см. рекомендацию 44). Для того чтобы добавить состояние, вместо наследования следует использовать композицию (см. рекомендацию 34). Избегайте наследования от конкретных базовых классов.

Обсуждение

Использование наследования там, где оно не требуется, подрывает доверие к мощи объектно-ориентированного программирования. В C++ при определении базового класса следует выполнить некоторые специфические действия (см. также рекомендации 32, 50 и 54), которые весьма сильно отличаются (а зачастую просто противоположны) от действий при разработке автономного класса. Наследование от автономного класса открывает ваш код для массы проблем, причем ваш компилятор в состоянии заметить только их малую часть.

Начинающие программисты зачастую выполняют наследование от классов-значений, таких как класс `string` (стандартный или иной) просто чтобы "добавить больше функциональности". Однако определение свободной функции (не являющейся членом) существенно превосходит создание класса `super_string` по следующим причинам.

- Свободные функции хорошо вписываются в существующий код, который работает с объектами `string`. Если же вместо этого вы предоставляете класс `super_string`, вам придется вносить изменения в ваш код, заменяя типы и сигнатуры функций.
- Функции интерфейса, которые получают параметры типа `string`, при использовании наследования должны сделать одно из трех: а) отказаться от дополнительной функциональности `super_string` (бесполезно), б) копировать свои аргументы в объекты `super_string` (расточительно) или в) преобразовать ссылки на `string` в ссылки на `super_string` (затруднительно и потенциально некорректно).
- Функции-члены `super_string` не должны получить больший доступ к внутреннему устройству класса `string`, чем свободные функции, поскольку класс `string`, вероятно, не имеет защищенных (`protected`) членов (вспомните — этот класс не предназначался для работы в качестве базового).
- Если класс `super_string` скрывает некоторые из функций класса `string` (а переопределение невиртуальных функций в производном классе не является перекрытием — это просто сокрытие), это может вызвать неразбериху в коде, работающем с объектами `string`, которые создаются автоматическим преобразованием из класса `super_string`.

Словом, лучше добавлять новую функциональность посредством новых свободных (не являющихся членами) функций (см. рекомендацию 44). Чтобы избежать проблем поиска имен, убедитесь, что вы поместили функции в то же пространство имен, что и тип, для расширения

функциональности которого они предназначены (см. рекомендацию 57). Некоторые программисты не любят свободные функции из-за их синтаксиса `Fun(str)` вместо `str.Fun()`, но это не более чем вопрос привычки.

Но что если класс `super_string` наследуется из класса `string` для добавления состояний, таких как кодировка или кэшированное значение количества слов? Открытое наследование не рекомендуется и в этом случае, поскольку класс `string` не защищен от срезки (см. рекомендацию 54), и любое копирование `super_string` в `string` молча уберет все старательно хранимые дополнительные состояния.

И наконец, наследование класса с открытым невиртуальным деструктором рискует получить эффект неопределенного поведения при удалении указателя на объект типа `string`, который на самом деле указывает на объект типа `super_string` (см. рекомендацию 50). Это неопределенное поведение может даже оказаться вполне допустимым при использовании вашего компилятора и распределителя памяти, но оно все равно рано или поздно выявится в виде затаившихся ошибок, утечек памяти, разрушенной кучи и кошмаров переноса на другую платформу.

Примеры

Пример 1. Композиция вместо открытого или закрытого наследования. Что делать, если вам нужен тип `localized_string`, который "почти такой же, как и `string`, но с дополнительными данными и функциями и небольшими переделками имеющихся функций-членов `string`", и при этом реализация многих функций остается неизменной? В этом случае реализуйте ее с помощью класса `string`, но не наследованием, а комбинированием (что предупредит срезку и неопределенное полиморфное удаление), и добавьте транзитные функции для того, чтобы сделать видимыми функции класса `string`, оставшиеся неизменными:

```
class localized_string {
public:
    // ... Обеспечьте транзитные функции для тех
    // функций-членов string, которые остаются неизменными
    // (например, определите функцию insert, которая
    // вызывает impl_.insert) ...

    void clear(); // Маскирует/переопределяет clear()

    bool is_in_klingon() const; // Добавляет функциональность

private:
    std::string impl_;
    // ... Дополнительные данные-члены ...
};
```

Конечно, писать транзитные функции для всех функций-членов, которые вы хотите сохранить, — занятие утомительное, но зато такая реализация существенно лучше и безопаснее, чем использование открытого или закрытого наследования.

Пример 2. `std::unary_function`. Хотя класс `std::unary_function` не имеет виртуальных функций, на самом деле он создан для использования в качестве базового класса и не противоречит рассматриваемой рекомендации. (Однако класс `unary_function` может быть усовершенствован добавлением защищенного деструктора — см. рекомендацию 50.)

Ссылки

[Dewhurst03] §70, §93 • [Meyers97] §33 • [Stroustrup00] §24.2-3, §25.2

36. Предпочитайте предоставление абстрактных интерфейсов

Резюме

Вы любите абстракционизм? Абстрактные интерфейсы помогают вам сосредоточиться на проблемах правильного абстрагирования, не вдаваясь в детали реализации или управления состояниями. Предпочтительно проектировать иерархии, реализующие абстрактные интерфейсы, которые моделируют абстрактные концепции.

Обсуждение

Предпочитайте определять абстрактные интерфейсы и выполнять наследование от них. Абстрактный интерфейс представляет собой абстрактный класс, составленный полностью из (чисто) виртуальных функций и не обладающий состояниями (членами-данными), причем обычно без реализации функций-членов. Заметим, что отсутствие состояний в абстрактном интерфейсе упрощает дизайн всей иерархии (см. соответствующие примеры в [Meyers96]).

Желательно следовать принципу инверсии зависимостей (Dependency Inversion Principle, DIP; см. [Martin96a] и [Martin00]). Данный принцип состоит в следующем.

- Высокоуровневые модули не должны зависеть от низкоуровневых. И те, и другие должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей; вместо этого детали должны зависеть от абстракций.

Из DIP следует, что корнями иерархий должны быть абстрактные классы, в то время как конкретные классы в этой роли выступать не должны (см. рекомендацию 35). Абстрактные базовые классы должны беспокоиться об определении функциональности, но не о ее реализации.

Принцип инверсии зависимостей имеет три фундаментальных преимущества при проектировании.

- *Повышение надежности*. Менее стабильные части системы (реализации) зависят от более стабильных частей (абстракций). Надежный дизайн — тот, в котором воздействие изменений ограничено. При плохом проектировании небольшие изменения в одном месте расходятся кругами по всему проекту и оказывают влияние на самые неожиданные части системы. Именно это происходит, когда проект строится на конкретных базовых классах.
- *Повышение гибкости*. Дизайн, основанный на абстрактных интерфейсах, в общем случае более гибок. Если абстракции корректно смоделированы, то при появлении новых требований легко разработать новые реализации. И напротив, дизайн, зависящий от многих конкретных деталей, оказывается более жестким в том смысле, что новые требования приводят к существенным изменениям в ядре системы.
- *Хорошая модульность*. Дизайн, опирающийся на абстракции, обладает хорошей модульностью благодаря простоте зависимостей: высокоизменяемые части зависят от стабильных частей, но не наоборот. Дизайн же, в котором интерфейсы перемешаны с деталями реализации, применить в качестве отдельного модуля в другой системе оказывается очень сложно.

Закон Второго Шанса гласит: “Самая важная вещь — интерфейс. Все остальное можно подправить и позже, но если интерфейс разработан неверно, то может оказаться так, что у вас уже не будет второго шанса его исправить” [Sutter04].

Обычно для того, чтобы обеспечить полиморфное удаление, при проектировании отдается предпочтение открытому виртуальному деструктору (см. рекомендацию 50), если только вы не используете брокер объектов (наподобие COM или CORBA), который использует альтернативный механизм управления памятью.

Будьте осторожны при использовании множественного наследования классов, которые не являются абстрактными интерфейсами. Дизайн с использованием множественного наследования может быть очень выразительным, но он труднее в разработке и более подвержен ошибкам. В частности, особенно сложным при использовании множественного наследования становится управление состояниями.

Как указывалось в рекомендации 34, наследование от некоторого типа может привести и к проблемам поиска имен, поскольку в таком случае в поиске участвуют функции из пространства имен наследуемого типа (см. также рекомендацию 58).

Примеры

Пример. Программа резервного копирования. При прямом, наивном проектировании высокоуровневые компоненты зависят от низкоуровневых деталей. например, плохо спроектированная программа резервного копирования может иметь архивирующий компонент, который непосредственно зависит от типов или подпрограмм для чтения структуры каталогов, и других, которые записывают данные на ленту. Адаптация такой программы к новой файловой системе и аппаратному обеспечению для резервного копирования потребует существенной переработки проекта.

Если же логика системы резервного копирования построена на основе тщательно спроектированных абстракций файловой системы и устройства резервного копирования, то переработка не потребуется — достаточно будет добавить в систему новую реализацию абстрактного интерфейса. Естественное решение подобной проблемы состоит в том, что новые требования должны удовлетворяться новым кодом, но старый код при этом изменяться не должен.

Исключения

Оптимизация пустого базового класса представляет собой один из примеров, когда наследование (предпочтительно не открытое) используется для сугубо оптимизационных целей (но см. рекомендацию 8).

Может показаться, что в дизайне на основе стратегий высокоуровневые компоненты зависят от деталей реализации (стратегий). Однако это всего лишь использование статического полиморфизма. Здесь имеются абстрактные интерфейсы, просто они не указаны явно посредством чисто виртуальных функций.

Ссылки

[Alexandrescu01] • [Cargill92] pp. 12-15, 215-218 • [Cline99] §5.18-20, 21.13 • [Lakos96] §6.4.1 • [Martin96a] • [Martin00] • [Meyers96] §33 • [Stroustrup00] §12.3-4, §23.4.3.2, §23.4.3.5, §24.2-3, §25.3, §25.6 • [Sutter04] §17

37. Открытое наследование означает заменимость. Наследовать надо не для повторного использования, а чтобы быть повторно использованным

Резюме

Открытое наследование позволяет указателю или ссылке на базовый класс в действительности обращаться к объекту некоторого производного класса без изменения существующего кода и нарушения его корректности.

Не применяйте открытое наследование для того, чтобы повторно использовать код (находящийся в базовом классе); открытое наследование необходимо для того, чтобы быть повторно использованным (существующим кодом, который полиморфно использует объекты базового класса).

Обсуждение

Несмотря на двадцатилетнюю историю объектно-ориентированного проектирования, цель и практика открытого наследования часто понимается неверно, и многие применения наследования оказываются некорректными.

Открытое наследование в соответствии с принципом подстановки Лисков (Liskov Substitution Principle [Liskov88]) всегда должно моделировать отношение "является" ("работает как"): все контракты базового класса должны быть выполнены, для чего все перекрытия виртуальных функций-членов не должны требовать большего или обещать меньше, чем их базовые версии. Код, использующий указатель или ссылку на `Base`, должен корректно вести себя в случае, когда указатель или ссылка указывают на объект `Derived`.

Неверное использование наследования нарушает корректность. Обычно некорректно реализованное наследование не подчиняется явным или неявным контрактам, установленным базовым классом. Такие контракты могут оказаться очень специфичными и хитроумными, и программист должен быть особенно осторожен, когда их нельзя выразить непосредственно в коде (некоторые шаблоны проектирования помогают указать в коде его предназначение — см. рекомендацию 39).

Наиболее часто в этой связи упоминается следующий пример. Рассмотрим два класса — `Square` (квадрат) и `Rectangle` (прямоугольник), каждый из которых имеет виртуальные функции для установки их высоты и ширины. Тогда `Square` не может быть корректно унаследован от `Rectangle`, поскольку код, использующий видоизменяемый `Rectangle`, будет полагать, что функция `SetWidth` не изменяет его высоту (независимо от того, документирован ли данный контракт классом `Rectangle` явно или нет), в то время как функция `Square::SetWidth` не может одновременно выполнить этот контракт и свой инвариант "квадратности". Но и класс `Rectangle` не может корректно наследовать классу `Square`, если его клиенты Square полагают, например, что для вычисления его площади надо возвести в квадрат ширину, либо используют какое-то иное свойство, которое выполняется для квадрата и не выполняется для прямоугольника.

Описание "является" для открытого наследования оказывается неверно понятым при использовании аналогий из реального мира: квадрат "является" прямоугольником в математическом смысле , но с точки зрения поведения `Square` не является `Rectangle`. Вот почему вместо

“является” мы предпочитаем говорить “работает как” (или, если вам это больше нравится, “используется как”) для того, чтобы такое описание воспринималось максимально правильно.

Открытое наследование действительно связано с повторным использованием, но опять же не так, как привыкло думать множество программистов. Как уже указывалось, цель открытого наследования — в реализации заменимости (см. [Liskov88]). Цель отрытого наследования не в том, чтобы производный класс мог повторно использовать код базового класса для того, чтобы с его помощью реализовать свою функциональность. Такое отношение “реализован посредством” вполне корректно, но должно быть смоделировано при помощи композиции или, только в отдельных случаях, при помощи закрытого или защищенного наследования (см. рекомендацию 34).

Вот еще одно соображение по поводу рассматриваемой темы. При корректности и целесообразности динамического полиморфизма композиция “эгоистична” в отличие от “щедрого” наследования. Поясним эту мысль.

Новый производный класс представляет собой частный случай существующей более общей абстракции. Существующий (динамический) полиморфный код, который использует `Base&` или `Base*` путем вызова виртуальных функций `Base`, должен быть способен прозрачно использовать объекты класса `MyNewDerivedType`, производного от `Base`. Новый производный тип добавляет новую функциональность к существующему коду, который при этом не должен изменяться. Однако несмотря на неизменность имеющегося кода, при использовании им новых производных объектов его функциональность возрастает.

Новые требования, естественно, должны удовлетворяться новым кодом, но они не должны требовать внесения изменений в существующий код (см. рекомендацию 36).

До объектно-ориентированного программирования было легко решить вопрос вызова старого кода новым. Открытое наследование упростило прозрачный и безопасный вызов нового кода старым (так что применяйте шаблоны, предоставляющие возможности статического полиморфизма, который хорошо совместим с полиморфизмом динамическим — см. рекомендацию 64).

Исключения

Классы стратегий добавляют новое поведение путем открытого наследования, но это не является неверным употреблением открытого наследования для моделирования отношения “реализован посредством”.

Ссылки

[Cargill92] pp. 19-20 • [Cline99] §5.13, §7.01-8.15 • [Dewhurst03] §92 • [Liskov88] • [Meyers97] §35 • [Stroustrup00] §12.4.1, §23.4.3.1, §24.3.4 • [Sutter00] §22-24

38. Практикуйте безопасное перекрытие

Резюме

Ответственно подходите в перекрытию функций. Когда вы перекрываете виртуальную функцию, сохраняйте заменимость; в частности, обратите внимание на пред- и постусловия в базовом классе. Не изменяйте аргументы по умолчанию виртуальных функций. Предпочтительно явно указывать перекрываемые функции как виртуальные. Не забывайте о сокрытии перегруженных функций в базовом классе.

Обсуждение

Хотя обычно производные классы добавляют новые состояния (т.е. члены-данные), они моделируют *подмножества* (а не надмножества) своих базовых классов. При корректном наследовании производный класс моделирует частный случай более общей базовой концепции (см. рекомендацию 37).

Это имеет прямые следствия для корректного перекрытия функций. Соблюдение отношения включения влечет за собой заменимость — операции, которые применимы ко всему множеству, применимы и к любому его подмножеству. Если базовый класс гарантирует выполнение определенных пред- и постусловий некоторой операции, то и любой производный класс должен обеспечить их выполнение. Перекрытие может предъявлять меньше требований и предоставлять б*о*льшую функциональность, но никогда не должно предъявлять б*о*льшие требования или меньше обещать — поскольку это приведет к нарушению контракта с вызывающим кодом.

Определение в производном классе перекрытия, которое может быть неуспешным (например, генерировать исключения; см. рекомендацию 70), будет корректно только в том случае, когда в базовом классе не объявлено, что данная операция всегда успешна. Например, скажем, класс `Employee` содержит виртуальную функцию-член `GetBuilding`, предназначение которой — вернуть код здания, в котором работает объект `Employee`. Но что если мы захотим написать производный класс `RemoteContractor`, который перекрывает функцию `GetBuilding`, в результате чего она может генерировать исключения или возвращать нулевой код здания? Такое поведение корректно только в том случае, если в документации класса `Employee` указано, что функция `GetBuilding` может завершаться неуспешно, и в классе `RemoteContractor` сообщение о неудаче выполняется документированным в классе `Employee` способом.

Никогда не изменяйте аргумент по умолчанию при перекрытии. Он не является частью сигнатуры функции, и клиентский код будет невольно передавать различные аргументы в функцию, в зависимости от того, какой узел иерархии обращается к ней. Рассмотрим следующий пример:

```
class Base {
    // ...
    virtual void Foo(int x = 0);
};

class Derived : public Base {
    // ...
    virtual void Foo(int x = 1); // Лучше так не делать...
};

Derived *pD = new Derived;
pD->Foo();                       // Вызов pD->Foo(1)

Base *pB = pD;
pB->Foo();                       // Вызов pB->Foo(0)
```

У некоторых может вызвать удивление, что одна и та же функция-член одного и того же объекта получает разные аргументы в зависимости от статического типа, посредством которого к ней выполняется обращение.

Желательно добавлять ключевое слово `virtual` при перекрытии функций, несмотря на его избыточность — это сделает код более удобным для чтения и понимания.

Не забывайте о том, что перекрытие может скрывать перегруженные функции из базового класса, например:

```
class Base{                     // ...
    virtual void Foo( int );
    virtual void Foo( int, int );
    void Foo( int, int, int );
};

class Derived : public Base { // ...
    virtual void Foo( int );  // Перекрывает Base::Foo(int),
                              // скрывая остальные функции
};

Derived d;
d.Foo( 1 );                   // Все в порядке
d.Foo( 1, 2 );                // Ошибка
d.Foo( 1, 2, 3 );             // Ошибка
```

Если перегруженные функции из базового класса должны быть видимы, воспользуйтесь объявлением `using` для того, чтобы повторно объявить их в производном классе:

```
class Derived : public Base { // ...
    virtual void Foo( int );  // Перекрытие Base::Foo(int)
    using Base::Foo;          // Вносит все прочие перегрузки
                              // Base::Foo в область видимости
};
```

Примеры

Пример: `Ostrich` (Страус). Если класс `Bird` (Птица) определяет виртуальную функцию `Fly` и вы порождаете новый класс `Ostrich` (известный как птица, которая не летает) из класса `Bird`, то как вы реализуете `Ostrich::Fly`? Ответ стандартный — "по обстоятельствам". Если `Bird::Fly` гарантирует успешность (т.е. обеспечивает гарантию бессбойности; см. рекомендацию 71), поскольку способность летать есть неотъемлемой частью модели `Bird`, то класс `Ostrich` оказывается неадекватной реализацией такой модели.

Ссылки

[Dewhurst03] §73-74, §78-79 • [Sutter00] §21 • [Keffer95] p. 18

39. Виртуальные функции стоит делать неоткрытыми, а открытые — невиртуальными

Резюме

В базовых классах с высокой стоимостью изменений (в частности, в библиотеках) лучше делать открытые функции невиртуальными. Виртуальные функции лучше делать закрытыми, или защищенными — если производный класс должен иметь возможность вызывать их базовые версии (этот совет не относится к деструкторам; см. рекомендацию 50).

Обсуждение

Большинство из нас на собственном горьком опыте выучило правило, что члены класса должны быть закрытыми, если только мы не хотим специально обеспечить доступ к ним со стороны внешнего кода. Это просто правило хорошего тона обычной инкапсуляции. В основном это правило применимо к членам-данным (см. рекомендацию 41), но его можно с тем же успехом использовать для любых членов класса, включая виртуальные функции.

В частности, в объектно-ориентированных иерархиях, внесение изменений в которые обходится достаточно дорого, предпочтительна полная абстракция: лучше делать открытые функции невиртуальными, а виртуальные функции — закрытыми (или защищенными, если производные классы должны иметь возможность вызывать базовые версии). Это — шаблон проектирования Nonvirtual Interface (NVI). (Этот шаблон похож на другие, в особенности на Template Method [Gamma95], но имеет другую мотивацию и предназначение.)

Открытая виртуальная функция по своей природе решает две различные параллельные задачи.

- *Она определяет интерфейс*. Будучи открытой, такая функция является непосредственной частью интерфейса класса, предоставленного внешнему миру.
- *Она определяет детали реализации*. Будучи виртуальной, функция предоставляет производному классу возможность заменить базовую реализацию этой функции (если таковая имеется), в чем и состоит цель настройки.

В связи с существенным различием целей этих двух задач, они могут конфликтовать друг с другом (и зачастую так и происходит), так что одна функция не в состоянии в полной мере решить одновременно две задачи. То, что перед открытой виртуальной функцией ставятся две существенно различные задачи, является признаком недостаточно хорошего разделения зон ответственности и по сути нарушения рекомендаций 5 и 11, так что нам следует рассмотреть иной подход к проектированию.

Путем разделения открытых функций от виртуальных мы достигаем следующих значительных преимуществ.

- *Каждый интерфейс может приобрести свой естественный вид*. Когда мы разделяем открытый интерфейс от интерфейса настройки, каждый из них может легко приобрести тот вид, который для него наиболее естественен, не пытаясь найти компромисс, который заставит их выглядеть идентично. Зачастую эти два интерфейса требуют различного количества функций и/или различных параметров; например, внешняя вызывающая функция может выполнить вызов одной открытой функции `Process`, которая выполняет логическую единицу работы, в то время как разработчик данного класса может предпочесть перекрыть только некоторые части этой работы, что естественным

образом моделируется путем независимо перекрываемых виртуальных функций (например, `DoProcessPhase1`, `DoProcessPhase2`), так что производному классу нет необходимости перекрывать их все (точнее говоря, данный пример можно рассматривать как применение шаблона проектирования Template Method).

- *Управление базовым классом*. Теперь базовый класс находится под полным контролем своего интерфейса и стратегии и может обеспечить пост- и предусловия интерфейса (см. рекомендации 14 и 68), причем выполнить всю эту работу в одном удобном повторно используемом месте — невиртуальной функции интерфейса. Такое "предварительное разложение" обеспечивает лучший дизайн класса.
- *Базовый класс более устойчив к изменениям*. Мы можем позже изменить наше мнение и добавить некоторую проверку пост- или предусловий, или разделить выполнение работы на большее количество шагов или переделать ее, реализовать более полное разделение интерфейса и реализации с использованием идиомы Pimpl (см. рекомендацию 43), или внести иные изменения в базовый класс, и все это никак не повлияет на код, использующий данный класс или наследующий его. Заметим, что гораздо проще начать работу с использования NVI (даже если открытые функции представляют собой однострочные вызовы соответствующих виртуальных функций), а уже позже добавлять все проверки и инструментальные средства, поскольку эта работа никак не повлияет на код, использующий или наследующий данный класс. Ситуация окажется существенно сложнее, если начать с открытых виртуальных функций и позже изменять их, что неизбежно приведет к изменениям либо в коде, который использует данный класс, либо в наследующем его.

См. также рекомендацию 54.

Исключения

NVI не применим к деструкторам в связи со специальным порядком их выполнения (см. рекомендацию 50).

NVI непосредственно не поддерживает ковариантные возвращаемые типы. Если вам требуется ковариантность, видимая вызывающему коду без использования `dynamic_cast` (см. также рекомендацию 93), проще сделать виртуальную функцию открытой.

Ссылки

[Allison98] §10 • [Dewhurst03] §72 • [Gamma95] • [Keffer95 pp. 6-7] • [Koenig97] §11 • [Sutter00] §19, §23 • [Sutter04] §18

40. Избегайте возможностей неявного преобразования типов

Резюме

Не все изменения прогрессивны: неявные преобразования зачастую приносят больше вреда, чем пользы. Дважды подумайте перед тем, как предоставить возможность неявного преобразования к типу и из типа, который вы определяете, и предпочитайте полагаться на явные преобразования (используйте конструкторы, объявленные как `explicit`, и именованные функции преобразования типов).

Обсуждение

Неявные преобразования типов имеют две основные проблемы.

- Они могут проявиться в самых неожиданных местах.
- Они не всегда хорошо согласуются с остальными частями языка программирования.

Неявно преобразующие конструкторы (конструкторы, которые могут быть вызваны с одним аргументом и не объявлены как `explicit`) плохо взаимодействуют с перегрузкой и приводят к созданию невидимых временных объектов. Преобразования типов, определенные как функции-члены вида `operator T` (где `T` — тип), ничуть не лучше — они плохо взаимодействуют с неявными конструкторами и позволяют без ошибок скомпилировать разнообразные бессмысленные фрагменты кода (примеров чего несть числа — см. приведенные в конце рекомендации ссылки; мы приведем здесь только пару из них).

В C++ последовательность преобразований типов может включать не более одного пользовательского преобразования. Однако когда в эту последовательность добавляются встроенные преобразования, ситуация может оказаться предельно запутанной. Решение здесь простое и состоит в следующем.

- По умолчанию используйте `explicit` в конструкторах с одним аргументом (см. рекомендацию 54):

```
class Widget { // ...
  explicit Widget(unsigned int widgetizationFactor);
  explicit Widget(const char* name, const Widget* other = 0);
};
```

- Используйте для преобразований типов именованные функции, а не соответствующие операторы:

```
class String { // ...
  const char* as_char_pointer() const; // В традициях c_str
};
```

См. также обсуждение копирующих конструкторов, объявленных как `explicit`, в рекомендации 54.

Примеры

Пример 1. Перегрузка. Пусть у нас есть, например, `Widget::Widget(unsigned int)`, который может быть вызван неявно, и функция `Display`, перегруженная для `Widget` и `double`. Рассмотрим следующий сюрприз при разрешении перегрузки:

```
void Display(double);        // Вывод double
void Display(const Widget&); // Вывод Widget

Display(5);                  // Гм! Создание и вывод Widget
```

Пример 2. Работающие ошибки. Допустим, вы снабдили класс `String` оператором `operator const char*`:

```
class String {
// ...
public:
    operator const char*(); // Грустное решение...
};
```

В результате этого становятся компилируемыми масса глупостей и опечаток. Пусть `s1` и `s2` — объекты типа `String`. Все приведенные ниже строки компилируются:

```
int x = s1 - s2;        // Неопределенное поведение
const char* p = s1 - 5; // Неопределенное поведение
p = s1 + '0';           // Делает не то, что вы ожидаете
if( s1 == "0" ) { ... } // Делает не то, что вы ожидаете
```

Именно по этой причине в стандартном классе `string` отсутствует `operator const char*`.

Исключения

При нечастом и осторожном использовании неявные преобразования типов могут сделать код более коротким и интуитивно более понятным. Стандартный класс `std::string` определяет неявный конструктор, который получает один аргумент типа `const char*`. Такое решение отлично работает, поскольку проектировщики класса приняли определенные меры предосторожности.

- Не имеется автоматического преобразования `std::string` в `const char*`; такое преобразование типов выполняются при помощи двух именованных функций — `c_str` и `data`.
- Все операторы сравнений, определенные для `std::string` (например, `==`, `!=`, `<`), перегружены для сравнения `const char*` и `std::string` в любом порядке (см. рекомендацию 29). Это позволяет избежать создания скрытых временных переменных.

Но и при этом возникают определенные неприятности, связанные с перегрузкой функций.

```
void Display( int );
void Display( std::string );

Display( NULL ); // Вызов Display(int)
```

Этот результат для некоторых может оказаться сюрпризом. (Кстати, если бы выполнялся вызов `Display(std::string)`, код бы обладал неопределенным поведением, поскольку создание `std::string` из нулевого указателя некорректно, но конструктор этого класса не обязан проверять передаваемое ему значение на равенство нулю.)

Ссылки

[Dewhurst03] §36-37 • [Lakos96] §9.3.1 • [Meyers96] §5 • [Murray93] §2.4 • [Sutter00] §6, §20, §39

41. Делайте данные-члены закрытыми (кроме случая агрегатов в стиле структур C)

Резюме

Данные-члены должны быть закрыты. Только в случае простейших типов в стиле структур языка C, объединяющих в единое целое набор значений, не претендующих на инкапсуляцию и не обеспечивающих поведение, делайте все данные-члены открытыми. Избегайте смешивания открытых и закрытых данных, что практически всегда говорит о бестолковом дизайне.

Обсуждение

Сокрытие информации является ключом к качественной разработке программного обеспечения (см. рекомендацию 11). Желательно делать все данные-члены закрытыми; закрытые данные — лучшее средство для сохранения инварианта класса, в том числе при возможных вносимых изменениях.

Открытые данные — плохая идея, если класс моделирует некоторую абстракцию и, следовательно, должен поддерживать инварианты. Наличие открытых данных означает, что часть состояния вашего класса может изменяться неконтролируемо, непредсказуемо и асинхронно с остальной частью состояния. Это означает, что абстракция разделяет ответственность за поддержание одного или нескольких инвариантов с неограниченным множеством кода, который использует эту абстракцию, и совершенно очевидно, что такое положение дел просто недопустимо с точки зрения корректного проектирования.

Защищенные данные обладают всеми недостатками открытых данных, поскольку наличие защищенных данных означает, что абстракция разделяет ответственность за поддержание одного или нескольких инвариантов с неограниченным множеством кода — теперь это код существующих и будущих производных классов. Более того, любой код может читать и модифицировать защищенные данные так же легко, как и открытые — просто создав производный класс и используя его для доступа к данным.

Смешивание открытых и закрытых данных-членов в одном и том же классе является непоследовательным и попросту запутывает пользователей. Закрытые данные демонстрируют, что у вас есть некоторые инварианты и нечто, предназначенное для их поддержания. Смешивание их с открытыми данными-членами означает, что при проектировании так окончательно и не решено, должен ли класс представлять некоторую абстракцию или нет.

Не закрытые данные-члены почти всегда хуже даже простейших функций для получения и установки значений, поскольку последние обеспечивают устойчивость кода к возможным внесениям изменений.

Подумайте о сокрытии закрытых членов класса с использованием идиомы Pimpl (см. рекомендацию 43).

Примеры

Пример 1. Корректная инкапсуляция. Большинство классов (например, `Matrix`, `File`, `Date`, `BankAccount`, `Security`) должны закрывать все данные-члены и открывать соответствующие интерфейсы. Позволение вызывающему коду непосредственно работать с внутренними данными класса работает против представленной им абстракции и поддерживаемых им инвариантов.

Агрегат `Node`, широко используемый в реализации класса `List`, обычно содержит некоторые данные и два указателя на `Node`: `next_` и `prev_`. Данные-члены `Node` не должны быть скрыты от `List`. Однако не забудьте рассмотреть еще пример 3.

Пример 2. `TreeNode`. Рассмотрим контейнер `Tree<T>`, реализованный с использованием `TreeNode<T>`, агрегата, используемого в `Tree`, который хранит указатели на предыдущий, следующий и родительский узлы и сам объект `T`. Все члены `TreeNode` могут быть открытыми, поскольку их не надо скрывать от класса `Tree`, который непосредственно манипулирует ими. Однако класс `Tree` должен полностью скрывать класс `TreeNode` (например, как вложенный закрытый класс или как определенный только в файле реализации класса `Tree`), поскольку это — детали внутренне реализации класса `Tree`, от которых не должен зависеть и с которыми не должен иметь дела вызывающий код. И наконец, `Tree` не скрывает содержащиеся в контейнере объекты `T`, поскольку за них отвечает вызывающий код; контейнеры используют абстракцию итераторов для предоставления доступа к содержащимся объектам, в то время как внутренняя структура контейнера остается скрытой.

Пример 3. Функции получения и установки значений. Если не имеется лучшей предметной абстракции, открытые и защищенные данные-члены (например, `color`) могут, как минимум, быть сделаны закрытыми и скрыты за функциями получения и установки значений (например, `GetColor`, `SetColor`); Тем самым обеспечивается минимальный уровень абстракции и устойчивость к изменениям.

Использование функций повышает уровень общения по поводу “цвета” от конкретного состояния до абстрактного, которое мы можем реализовать тем способом, который сочтем наиболее приемлемым. Мы можем изменить внутреннее представление цвета, добавить код для обновления дисплея при изменении цвета, добавить какие-то инструментальные средства или внести еще какие-то изменения — и все это без каких-либо изменений в вызывающем коде. В худшем случае вызывающий код потребуется перекомпилировать (т.е. мы сохраняем совместимость на уровне исходных текстов); в лучшем — не потребуется ни перекомпиляция, ни даже перекомпоновка (если изменения сохраняют бинарную совместимость). Ни совместимость на уровне исходных текстов, ни бинарная совместимость при внесении таких изменений невозможны, если исходный дизайн содержит открытый член `color`, с которым тесно связан вызывающий код.

Исключения

Функции получения и установки значений полезны, но дизайн класса, состоящего практически из одних таких функций, оставляет желать лучшего. Подумайте над тем, требуется ли в таком случае обеспечение абстракции или достаточно ограничиться простой структурой.

Агрегаты значений (известные как структуры в стиле C) просто хранят вместе набор различных данных, но при этом не обеспечивают ни их поведение, ни делают попыток моделирования абстракций или поддержания инвариантов. Такие агрегаты не предназначены для того, чтобы быть абстракциями. Все их данные-члены должны быть открытыми, поскольку эти данные-члены и представляют собой интерфейс. Например, шаблон класса `std::pair<T,U>` используется стандартными контейнерами для объединения двух несвязанных элементов типов `T` и `U`, и при этом `pair` сам по себе не привносит ни поведения, ни каких-либо инвариантов.

Ссылки

[Dewhurst03] §80 • [Henricson97] pg. 105 • [Koenig97] §4 • [Lakos96] §2.2 • [Meyers97] §20 • [Murray93] §2.3 • [Stroustrup00] §10.2.8, §15.3.1.1, §24.4.2-3 • [SuttHysl04a]

42. Не допускайте вмешательства во внутренние дела

Резюме

Избегайте возврата дескрипторов внутренних данных, управляемых вашим классом, чтобы клиенты не могли неконтролируемо изменять состояние вашего объекта, как своего собственного.

Обсуждение

Рассмотрим следующий код:

```
class Socket {
public:
    // ... Конструктор, который открывает handle_,
    // деструктор, который закрывает handle_, и т.д. ...
    int GetHandle() const { return handle_; } // Плохо!
private:
    int handle_; // Дескриптор операционной системы
};
```

Сокрытие данных — мощный инструмент абстракции и модульности (см. рекомендации 11 и 41). Однако сокрытие данных при одновременном обеспечении доступа к их дескрипторам обречено на провал, потому что это то же, что и закрыть свою квартиру на замок и положить ключ под коврик у входа или просто оставить его в замке. Вот почему это так.

- *В этом случае клиент имеет две возможности реализации функциональности*. Он может воспользоваться абстракцией вашего класса (`Socket`) либо непосредственно работать с реализацией, на которой основан ваш класс (дескриптор сокета в стиле C). В последнем случае объект оказывается не осведомлен об изменениях, происходящих с ресурсом, которым он, как ему кажется, владеет. Теперь класс не в состоянии надежно обогатить или усовершенствовать функциональность (например, обеспечить прокси, журнализацию, сбор статистики и т.п.), поскольку клиенты могут просто обойти эти возможности реализации, как и любые другие инварианты, которые вы, как вы полагаете, добавили в ваш класс. Это делает невозможной, в частности, корректную обработку возникающих ошибок (см. рекомендацию 70).
- *Класс не может изменять внутреннюю реализацию своей абстракции, поскольку от нее зависят клиенты*. Если в будущем класс `Socket` будет обновлен для поддержки другого протокола с использованием других низкоуровневых примитивов, вызывающий код, который будет по-прежнему получать доступ к дескриптору `handle_` и работать с ним, окажется некорректным.
- *Класс не в состоянии обеспечить выполнение его инвариантов, поскольку вызывающий код может изменить состояние без ведома класса*. Например, кто-то может закрыть дескриптор, используемый объектом `Socket`, минуя вызов функции-члена `Socket`, а это приведет к тому, что объект станет недействительным.
- Код клиента может хранить дескрипторы, возвращаемые вашим классом, и пытаться использовать их после того, как код вашего класса сделает их недействительными.

Распространенная ошибка заключается в том, что действие `const` на самом деле неглубокое и не распространяется посредством указателей (см. рекомендацию 15). Например, `Socket::GetHandle` — константный член; пока мы рассматриваем ситуацию с точки зрения

компилятора, возврат handle_ сохраняет константность объекта. Однако непосредственный вызов функций операционной системы с использованием значения handle_ вполне может изменять данные, к которым косвенно обращается handle_.

Приведенный далее пример очень прост, хотя в данном случае ситуация несколько лучше — мы можем снизить вероятность случайного неверного употребления возвращаемого значения, описав его тип как const:

```
class String {
    char* buffer_;
public:
    char* GetBuffer() const { return buffer_; }
    // Плохо: следует возвращать const char*

    // ...
};
```

Хотя функция GetBuffer константная, технически этот код вполне корректен. Понятно, что клиент может использовать эту функцию GetBuffer для того, чтобы изменить объект String множеством разных способов, не прибегая к явному преобразованию типов. Например, strcpy(s.GetBuffer(),"Very Long String...") — вполне законный код; любой компилятор пропустит его без каких бы то ни было замечаний. Если бы мы объявили возвращаемый тип как const char*, то представленный код вызвал бы, по крайней мере, ошибку времени компиляции, так что случайно поступить столь опасно было бы просто невозможно — вызывающий код должен был бы использовать явное преобразование типов (см. рекомендации 92 и 95).

Но даже возврат указателей на const не устраняет возможности случайного некорректного использования, поскольку имеется еще одна проблема, связанная с корректностью внутренних данных класса. В приведенном выше примере с классом String, вызывающий код может сохранить значение, возвращаемое функцией GetBuffer, а затем выполнить операции, которые приведут к росту (и перемещению) буфера String, что в результате может привести к использованию сохраненного, но более недействительного указателя на несуществующий в данный момент буфер. Таким образом, если вы считаете, что у вас есть причины для обеспечения такого доступа ко внутреннему состоянию, вы должны детально документировать, как долго возвращаемое значение остается корректным и какие операции делают его недействительным (сравните с гарантиями корректности явных итераторов стандартной библиотеки; см. [C++03]).

Исключения

Иногда классы обязаны предоставить доступ ко внутренним дескрипторам по причинам, связанным с совместимостью, например, для интерфейса со старым кодом или при использовании других систем. Например, std::basic_string предоставляет доступ к своему внутреннему дескриптору посредством функций-членов data и c_str для совместимости с функциями, которые работают с указателями C — но не для того, чтобы хранить эти указатели и пытаться выполнять запись с их помощью! Такие функции доступа “через заднюю дверь” всегда являются злом и должны использоваться очень редко и очень осторожно, а условия корректности возвращаемых ими дескрипторов должны быть точно документированы.

Ссылки

[C++03] §23 • [Dewhurst03] §80 • [Meyers97] #29 • [Saks99] • [Stroustrup00] §7.3 • [Sutter02] §9

43. Разумно пользуйтесь идиомой Pimpl

Резюме

C++ делает закрытые члены недоступными, но не невидимыми. Там, где это оправдывается получаемыми преимуществами, следует подумать об истинной невидимости, достигаемой применением идиомы Pimpl (указателя на реализацию) для реализации брандмауэров компилятора и повышения сокрытия информации (см. рекомендации 11 и 41).

Обсуждение

Когда имеет смысл создать "брандмауэр компилятора", который полностью изолирует вызывающий код от закрытых частей класса, воспользуйтесь идиомой Pimpl (указателя на реализацию): скройте их за непрозрачным указателем (указатель (предпочтительно подходящий интеллектуальный) на объявленный, но пока не определенный класс). Например:

```
class Map {
    // ...
private:
    struct Impl;
    shared_ptr<Impl> pimpl_;
};
```

Дающий название идиоме указатель должен использоваться для хранения всех закрытых членов, как данных, так и закрытых функций-членов. Это позволяет вам вносить произвольные изменения в закрытые детали реализации ваших классов без какой бы то ни было рекомпиляции вызывающего кода. Свобода и независимость — вот отличительные черты рассматриваемой идиомы (см. рекомендацию 41).

Примечание: объявляйте указатель на закрытую реализацию, как показано — с использованием двух объявлений. Если вы скомбинируете две строки с предварительным объявлением типа и указателя на него в одну инструкцию `struct Impl*pimpl;`, это будет вполне законно, но изменит смысл объявления: в этом случае `Impl` находится в охватывающем пространстве имен и не является вложенным типом вашего класса.

Имеется как минимум три причины для использования Pimpl, и все они вытекают из различия между доступностью (в состоянии ли вы вызвать или использовать некоторый объект) и видимостью (видим ли этот объект для вас и, таким образом, зависите ли вы от его определения) в C++. В частности, все закрытые члены класса недоступны никому, кроме функций-членов и друзей, но зато видимы всем — любому коду, которому видимо определение класса.

Первое следствие этого — потенциально большее время сборки приложения из-за обработки излишних определений типов. Для закрытых данных-членов, хранящихся по значению, и параметров закрытых функций-членов, передаваемых по значению или используемых в видимой реализации функций, типы должны быть определены, даже если они никогда не потребуются в данной единице компиляции. Это может привести к увеличению времени сборки, например:

```
class C {
    // ...
private:
    AComplicatedType act_;
};
```

Заголовочный файл, содержащий определение класса C, должен также включать заголовочный файл, содержащий определение `AComplicatedType`, который в свою очередь транзитивно включает все заголовочные файлы, которые могут потребоваться для определения

AComplicatedType, и т.д. Если заголовочные файлы имеют большие размеры, время компиляции может существенно увеличиться.

Второе следствие — создание неоднозначностей и сокрытие имен для кода, который пытается вызвать функцию. Несмотря на то, что закрытая функция-член не может быть вызвана кодом вне ее класса и его друзей, она тем не менее участвует в поиске имен и разрешении перегрузки и тем самым может сделать вызов неоднозначным или некорректным. Перед выполнением проверки доступности C++ выполняет поиск имен и разрешение перегрузки. Из-за этого видимость имеет более высокий приоритет:

```
int Twice( int );            // 1

class Calc {
public:
    string Twice( string );  // 2

private:
    char* Twice( char* );    // 3

    int Test() {
        return Twice( 21 );  // A: ошибка, функции 2 и 3 не
               // подходят (могла бы подойти функция 1, но
               // ее нельзя рассматривать, так она скрыта от
    }          // данного кода)
};

Calc c;
c.Twice( "Hello" );          // Б: ошибка, функция 3
               // недоступна (могла бы использоваться
               // функция 2, но она не рассматривается, так
               // как у функции 3 лучшее соответствие
               // аргументу)
```

В строке А обходной путь состоит в том, чтобы явно квалифицировать вызов как ::Twice(21) для того, чтобы заставить поиск имен выбрать глобальную функцию. В строке Б обходной путь состоит в добавлении явного преобразования типа c.Twice(string("Hello")) для того, чтобы заставить разрешение перегрузки выбрать соответствующую функцию. Некоторые из таких проблем, связанных с вызовами, можно решить и без применения идиомы Pimpl, например, никогда не используя закрытые перегрузки функций-членов, но не для всех проблем, разрешимых при помощи идиомы Pimpl, можно найти такие обходные пути.

Третье следствие влияет на обработку ошибок и безопасность. Рассмотрим пример Widget Тома Каргилла (Tom Cargill):

```
class Widget { // …
public:
    Widget& operator=( const Widget& );
private:
    T1 t1_;
    T2 t2_;
};
```

Коротко говоря, мы не можем написать оператор operator=, который обеспечивает строгую гарантию (или хотя бы базовую гарантию), если операции T1 или T2 могут давать необратимые сбои (см. рекомендацию 71). Хорошие новости, однако, состоят в том, что приведенная далее простая трансформация всегда обеспечивает, как минимум, базовую гарантию для безопасного присваивания, и как правило — строгую гарантию, если необходимые операции T1 и T2 (а именно — конструкторы и деструкторы) не имеют побочных эффектов. Для этого следует хранить объекты не по значению, а посредством указателей, предпочтительно спрятанными за единственным указателем на реализацию:

```
class Widget { // …
public:
    Widget& operator=( const Widget& );

private:
    struct Impl;
    shared_ptr<Impl> pimpl_;
};

Widget& Widget::operator=( const Widget& ) {
    shared_ptr<Impl> temp( new Impl( /*…*/ ) );
    // Изменяем temp->t1_ и temp->t2_; если какая-то из
    // операций дает сбой, генерируем исключение, в
    // противном случае – принимаем внесенные изменения:
    pimpl_ = temp;
    return *this;
}
```

Исключения

В то время как вы получаете все преимущества дополнительного уровня косвенности, проблема состоит только в увеличении сложности кода (см. рекомендации 6 и 8).

Ссылки

[Coplien92] §5.5 • [Dewhurst03] §8 • [Lakos96] §6.4.2 • [Meyers97] §34 • [Murray93] §3.3 • [Stroustrup94] §2.10, §24.4.2 • [Sutter00] §23, §26-30 • [Sutter02] §18, §22 • [Sutter04] §16-17

44. Предпочитайте функции, которые не являются ни членами, ни друзьями

Резюме

Там, где это возможно, предпочтительно делать функции не членами и не друзьями классов.

Обсуждение

Функции, не являющиеся членами или друзьями классов, повышают степень инкапсуляции путем снижения зависимостей: тело такой функции не может зависеть от закрытых и защищенных членов класса (см. рекомендацию 11). Они также разрушают монолитность классов, снижая связность (см. рекомендацию 33), и повышают степень обобщенности, так как сложно писать шаблоны, которые не знают, является ли интересующая нас операция членом данного типа или нет (см. рекомендацию 67).

Для определения того, должна ли функция быть реализована как член и/или друг класса, можно воспользоваться следующим алгоритмом:

```
// Если у вас нет выбора – делайте функцию членом.
Если функция представляет собой один из операторов =, ->,
     [] или (), которые должны быть членами,
то
    делайте данную функцию членом класса.
// Если функция может быть не членом и не другом либо
// имеются определенные преимущества от того, чтобы сделать
// ее не членом и другом
иначе если 1. функция требует левый аргумент иного типа
     (как, например, в случае операторов >> или <<)
     или 2. требует преобразования типов для левого аргумента,
     или 3. может быть реализована с использованием только
     открытого интерфейса класса
то
    сделайте ее не членом класса (и, при необходимости,
    в случаях 1 и 2 - другом)
    Если функция требует виртуального поведения,
    то
        добавьте виртуальную функцию-член для обеспечения
        виртуального поведения, и реализуйте функцию-не член
        с использованием этой виртуальной функции.
иначе
    сделайте ее функцией-членом.
```

Примеры

Пример. `basic_string`. Стандартный класс `basic_string` чересчур монолитен и имеет 103 функции-члена, из которых 71 без потери эффективности можно сделать функциями, не являющимися ни членами, ни друзьями класса. Многие из них дублируют функциональность, уже имеющуюся в качестве алгоритма стандартной библиотеки, либо представляют собой алгоритмы, которые могли бы использоваться более широко, если бы не были спрятаны в классе `basic_string`. (См. рекомендации 5 и 32, а также [Sutter04].)

Ссылки

[Lakos96] §3.6.1, §9.1.2 • [McConnell93] §5.1-4 • [Murray93] §2.6 • [Meyers00] • [Stroustrup00] §10.3.2, §11.3.2, §11.3.5, §11.5.2, §21.2.3.1 • [Sutter00] §20 • [Sutter04] §37-40

45. new и delete всегда должны разрабатываться вместе

Резюме

Каждая перегрузка `void* operator new(parms)` в классе должна сопровождаться соответствующей перегрузкой оператора `void operator delete(void*, parms)`, где `parms` — список типов дополнительных параметров (первый из которых всегда `std::size_t`). То же относится и к операторам для массивов `new[]` и `delete[]`.

Обсуждение

Обычно редко требуется обеспечить наличие пользовательских операторов `new` или `delete`, но если все же требуется один из них — то обычно требуются они оба. Если вы определяете специфичный для данного класса оператор `T::operator new`, который выполняет некоторое специальное выделение памяти, то, вероятнее всего, вы должны определить и специфичный для данного класса оператор `T::operator delete`, который выполняет соответствующее освобождение выделенной памяти.

Появление данной рекомендации связано с одной тонкой проблемой: дело в том, что компилятор может вызвать перегруженный оператор `T::operator delete` даже если вы никогда явно его не вызываете. Вот почему вы всегда должны предоставлять операторы `new` и `delete` (а также операторы `new[]` и `delete[]`) парами.

Пусть вы определили класс с пользовательским выделением памяти:

```
class T {
    // ...
    static void* operator new(std::size_t);
    static void* operator new(std::size_t, CustomAllocator&);

    static void operator delete(void*, std::size_t);
};
```

Вы вводите простой протокол для выделения и освобождения памяти.

- Вызывающий код может выделять объекты типа `T` либо при помощи распределителя по умолчанию (используя вызов `new T`), либо при помощи пользовательского распределителя (вызов `new(alloc) T`, где `alloc` — объект типа `CustomAllocator`).
- Единственный оператор `delete`, который может быть использован вызывающим кодом — оператор по умолчанию `operator delete(size_t)`, так что, конечно, вы должны реализовать его так, чтобы он корректно освобождал память, выделенную любым способом.

Пока все в порядке.

Однако компилятор может скрыто вызвать другую перегрузку оператора `delete`, а именно `T::operator delete(size_t, CustomAllocator&)`. Это связано с тем, что инструкция

```
T* p = new(alloc) T;
```

на самом деле разворачивается в нечто наподобие

```
// Сгенерированный компилятором код для
// инструкции T* p = new(alloc) T;
//
void* __compilerTemp = T::operator new(sizeof(T), alloc);
T* p;
```

```
try {
    p = new (__compilerTemp) T; // Создание объекта T по
                                // адресу __compilerTemp
}
catch(…) {                      // Сбой в конструкторе...
    T::operator delete(__compilerTemp, sizeof(T), alloc);
    throw;
}
```

Итак, компилятор автоматически вставляет код вызова соответствующего оператора `T::operator delete` для перегруженного оператора `T::operator new`, что совершенно логично, если выделение памяти прошло успешно, но произошел сбой в конструкторе. "Соответствующая" сигнатура оператора `delete` имеет вид `void operator delete(void*, параметры_оператора_new)`.

Теперь перейдем к самому интересному. Стандарт C++ ([C++03] §5.3.4(17)) гласит, что приведенный выше код будет генерироваться тогда и только тогда, когда реально существует соответствующая перегрузка оператора `delete`. В противном случае код вообще не будет вызывать никакого оператора `delete` при сбое в конструкторе. Другими словами, при сбоях в конструкторе мы получим утечку памяти. Из шести проверенных нами распространенных компиляторов только два выводили предупреждение в такой ситуации. Вот почему каждый перегруженный оператор `void* operator new(parms)` должен сопровождаться соответствующей перегрузкой `void operator delete(void*, parms)`.

Исключения

Размещающий оператор `new`

```
void* T::operator new(size_t, void* p) { return p; }
```

не требует наличия соответствующего оператора `delete`, поскольку реального выделения памяти при этом не происходит. Все протестированные нами компиляторы не выдавали никаких предупреждений по поводу отсутствия оператора `void T::operator delete(void*, size_t, void*)`.

Ссылки

[C++03] §5.3.4 • [Stroustrup00] §6.2.6.2, §15.6 • [Sutter00] §36

46. При наличии пользовательского new следует предоставлять все стандартные типы этого оператора

Резюме

Если класс определяет любую перегрузку оператора new, он должен перегрузить все три стандартных типа этого оператора — обычный new, размещающий и не генерирующий исключений. Если этого не сделать, то эти операторы окажутся скрытыми и недоступными пользователям вашего класса.

Обсуждение

Обычно пользовательские операторы new и delete нужны очень редко, но если они все же оказываются необходимы, то вряд ли вы захотите, чтобы они скрывали встроенные сигнатуры.

В C++, после того как вы определите имя в области видимости (например, в области видимости класса), все такие же имена в охватывающих областях видимости окажутся скрыты (например, в базовых классах или охватывающих пространствах имен), так что перегрузка никогда не работает через границы областей видимости. Когда речь идет об имени оператора new, необходимо быть особенно осторожным и внимательным, чтобы не усложнять жизнь себе и пользователям вашего класса.

Пусть вы определили следующий оператор new, специфичный для класса:

```
class C {
    // ...

    // Скрывает три стандартных вида оператора new
    static void* operator new(size_t, MemoryPool&);
};
```

Теперь, если кто-то попытается написать выражение с обычным стандартным new C, компилятор сообщит о том, что он не в состоянии найти обычный старый оператор new. Объявление перегрузки C::operator new с параметром типа MemoryPool скрывает все остальные перегрузки, включая знакомые встроенные глобальные версии, которые все мы знаем и любим:

```
void* operator new(std::size_t); // Обычный
void* operator new(std::size_t,
      std::nothrow_t) throw();   // Не генерирующий исключений
void* operator new(std::size_t,
                   void*);       // Размещающий
```

В качестве другого варианта событий предположим, что ваш класс предоставляет некоторую специфичную для данного класса версию оператора new — одну из трех. В таком случае это объявление также скроет остальные две версии:

```
class C {
    // ...
    // Скрывает две другие стандартные версии оператора new
    static void* operator new(size_t, void*);
};
```

Предпочтительно, чтобы у класса C в его область видимости были явно внесены все три стандартные версии оператора new. Обычно все они должны иметь одну и ту же видимость. (Видимость для отдельных версий может быть сделана закрытой, если вы хотите явно запретить

один из вариантов оператора new, однако цель данной рекомендации — напомнить, чтобы вы не скрыли эти версии непреднамеренно.)

Заметим, что вы должны всегда избегать сокрытия размещающего new, поскольку он интенсивно используется контейнерами стандартной библиотеки.

Все, что осталось упомянуть, — это то, что внесение оператора new в область видимости может быть сделано двумя различными способами в двух разных ситуациях. Если базовый класс вашего класса также определяет оператор new, все, что вам надо, — "раскрыть" оператор new:

```
class C : public B { // ...
    public:
    using B::operator new;
};
```

В противном случае, если не имеется базового класса или в нем не определен оператор new, вы должны написать короткую пересылающую функцию (поскольку нельзя использовать using для внесения имен из глобальной области видимости):

```
class C { // ...
public:
    static void* operator new(std::size_t s) {
        return ::operator new(s);
    }

    static void* operator new(std::size_t s,
                              std::nothrow_t nt) throw() {
        return ::operator new(s, nt);
    }

    static void* operator new(std::size_t s, void* p) {
        return ::operator new(s, p);
    }
};
```

Рассмотренная рекомендация применима также к версиям операторов для массивов — operator new[].

Избегайте вызова версии new(nothrow) в вашем коде, но тем не менее обеспечьте и ее, чтобы пользователи вашего класса не оказались в какой-то момент неприятно удивлены.

Ссылки

[Dewhurst03] §60 • [Sutter04] §22-23