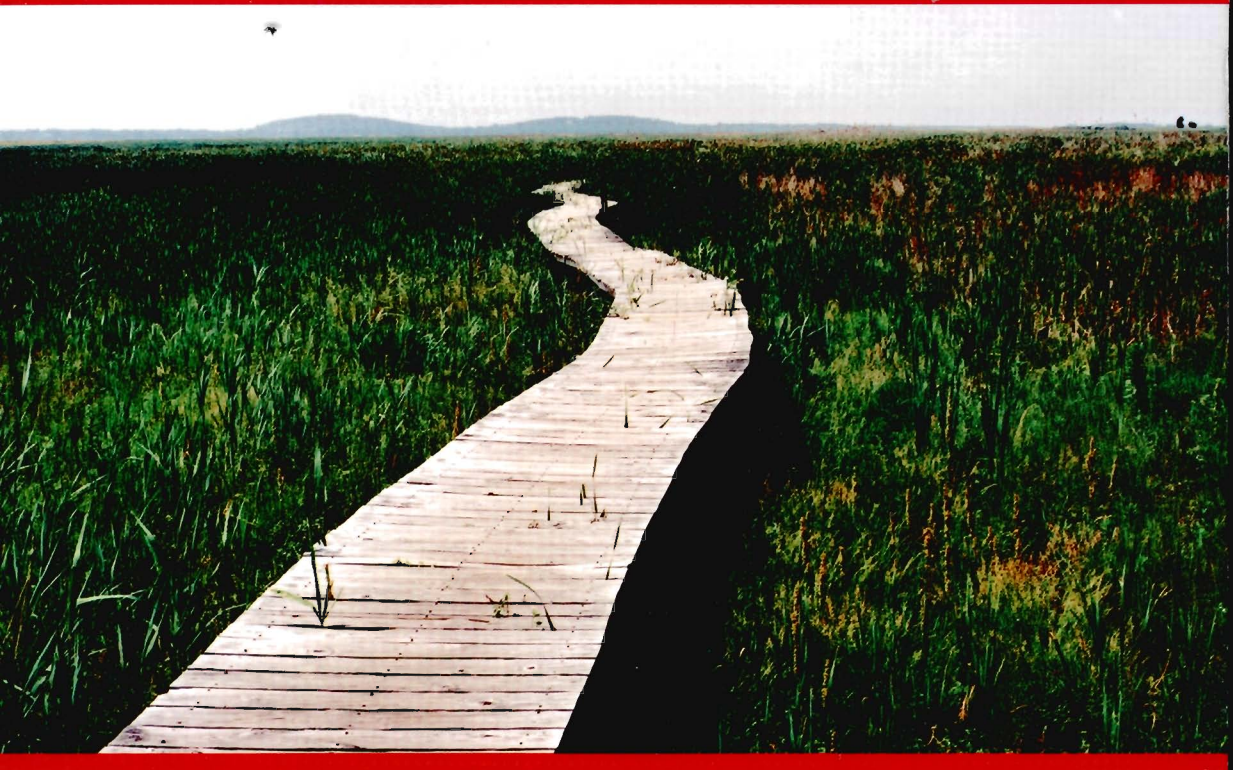
Стандарты программирования на C++

101 правило и рекомендация

Герб Саттер
Андрей Александреску



Серия *C++ In-Depth* ◆ Бьярн Страуструп

Стандарты программирования на C++

Герб Саттер и Андрей Александреску

Согласованные, высококачественные стандарты программирования повышают качество программного обеспечения, сокращают время его разработки, способствуют командной работе, снижают затраты времени на решение несущественных вопросов и облегчают сопровождение программ. Два эксперта мирового уровня в программировании на C++, творчески переработав опыт всего программистского сообщества, сумели собрать лучшее в этой книге, которую любой программист или команда могут использовать в качестве основы для разработки собственных стандартов кодирования.

Авторы не обошли вниманием ни один из разделов C++: проектирование и стиль кодирования, функции, операторы, дизайн классов, наследование, конструкторы и деструкторы, копирование, присваивание, пространства имен, модули, шаблоны, обобщенность, исключения, контейнеры и алгоритмы STL, а также многое другое. Каждая рекомендация сопровождается кратким описанием и примерами из практики. От определения типа до обработки исключений — в этой книге собран лучший опыт, последние достижения и наработки, о которых вы могли просто еще не знать, даже если уже давно используете C++. В книге вы найдете ответы на массу различных вопросов, в том числе:

- Что следует стандартизировать, а что нет?
- Каким образом обеспечить масштабируемость кода?
- Какие элементы входят в разумную стратегию обработки ошибок?
- Как (и почему) следует избегать излишних зависимостей?
- Когда и как следует совместно использовать статический и динамический полиморфизм?
- Как обеспечить "безопасное" перекрытие функций?
- Когда следует обеспечивать бессбойный обмен объектов?
- Почему и как следует предотвращать пересечение исключениями границ модулей?
- Почему вместо массивов следует использовать `vector` и `string`?
- Как выбрать верный алгоритм STL для поиска или сортировки?
- Каким правилам надо следовать для написания кода, безопасного в смысле использования типов?

Работаете ли вы в одиночку или в группе — эта книга поможет вам писать более ясный код, причем делать это более быстро и качественно.

Герб Саттер в первую очередь известен как автор бестселлера ***Решение сложных задач на C++***, а также автор сотен статей, посвященных различным аспектам разработки программного обеспечения. Герб возглавляет комитет ISO по стандартизации языка, ведет раздел и регулярно печатается в журнале ***C/C++ Users Journal***. Он работает в Microsoft над архитектурой Visual C++, отвечая за проектирование расширений C++ для программирования в .NET.

Андрей Александреску — автор знаменитой книги ***Современное проектирование на C++***; он ведет раздел и регулярно печатается в журнале ***C/C++ Users Journal***.

www.awprofessional.com/series/indepth
www.gotw.ca

Фотография на обложке – Stuart O'Sullivan

Издательский дом "Вильямс"
www.williamspublishing.com

ADDISON-WESLEY
Pearson Education

ISBN 5-8459-0859-0

9 785845 908599 05129

C++ Coding Standards

101 Rules, Guidelines, and Best Practices

Herb Sutter
Andrei Alexandrescu

ADDISON–WESLEY

Boston

Стандарты программирования на C++

101 правило и рекомендация

Герб Саттер
Андрей Александреску

Издательский дом "Вильямс"
Москва • Санкт-Петербург • Киев
2005

ББК 32.973.26-018.2.75
С21
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в Издательский дом "Вильямс"
по адресу: info@williamspublishing.com, http://www.williamspublishing.com
115419, Москва, а/я 783; 03150, Киев, а/я 152

Саттер, Герб, **Александреску**, Андрей.

С21 Стандарты программирования на C++. : Пер. с англ. — М. : Издательский дом "Вильямс", 2005. — 224 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0859-0 (рус.)

Эта книга поможет новичку стать профессионалом, так как в ней представлен сконцентрированный лучший опыт программистов на C++, обобщенный двумя экспертами мирового класса.

Начинающий программист найдет в ней простые и понятные рекомендации для ежедневного использования, подкрепленные примерами их конкретного применения на практике.

Опытные программисты найдут в ней советы и новые рекомендации, которые можно сразу же принять на вооружение. Программисты-профессионалы могут использовать эту книгу как основу для разработки собственных стандартов кодирования, как для себя лично, так и для группы, которой они руководят.

Конечно, книга рассчитана в первую очередь на профессиональных программистов с глубокими знаниями языка, однако она будет полезна любому, кто захочет углубить свои знания в данной области.

ББК 32.973.26-018.2.75

ISBN 5-8459-0859-0 (рус.)
ISBN 0-321-11358-6 (англ.)

Оглавление

Содержание

Предисловие

Идите проторенной дорогой — делайте одинаковые вещи одинаковыми способами. Накапливайте идиомы. ***Стандартизируйте****. Единственное отличие между вами и Шекспиром — в количестве используемых идиом, а не в размере словаря.*

— Алан Перлис (Alan Perlis) [выделено нами]

Лучшее в стандарте то, что он предоставляет богатый выбор.

— Приписывается разным людям

Мы бы хотели, чтобы эта книга стала основой для стандартов кодирования, используемых вашей командой, по двум основным причинам.

- *Стандарты кодирования должны отражать лучший опыт проб и ошибок всего сообщества программистов.* В них должны содержаться проверенные идиомы, основанные на опыте и твердом понимании языка. В частности, стандарт кодирования должен основываться на исчерпывающем анализе литературы по разработке программного обеспечения, и объединять воедино правила, рекомендации и наилучшие практические решения, которые в противном случае оказываются разбросанными по многочисленным источникам.
- *Природа не терпит пустоты.* Если вы не разработаете набор правил, то это сделает кто-то другой. Такие "самопальные" стандарты, как правило, грешат тем, что включают нежелательные для стандарта требования; например, многие из них, по сути, заставляют программистов использовать C++ просто как улучшенный C.

Множество таких плохих стандартов кодирования разработаны людьми, которые недостаточно хорошо понимают язык программирования C++ или пытаются чрезмерно детализировать его применение. Плохой стандарт кодирования быстро теряет кредит доверия, и в результате несогласие или неприятие программистами части его положений распространяется на весь стандарт целиком, перечеркивая содержащиеся в нем различные положительные советы и рекомендации. И это — в лучшем случае, потому что в худшем случае такой стандарт и его выполнение могут быть навязаны руководством.

Как пользоваться этой книгой

Думать. Надо добросовестно следовать умным советам, но делать это не вслепую. Во многих разделах этой книги есть подраздел "Исключения", в котором приводятся нестандартные, редко встречающиеся ситуации, когда совет из основного раздела может оказаться неприменим. Никакое количество даже самых хороших (мы на это надеемся) советов не могут заменить голову.

Каждая команда разработчиков отвечает за принятие собственных стандартов и несет ответственность за них и за неукоснительное следование им. Ваша команда — не исключение. Если вы — руководитель, предложите членам своей группы поучаствовать в разработке стандартов, которым группа будет следовать в своей работе. Люди всегда охотнее следуют правилам, которые они сами для себя вырабатывают, чем тем, которые им навязаны.

Эта книга предназначена для того, чтобы послужить основой для вашего стандарта кодирования и быть в той или иной мере включенной в него. Это — не ultima ratio в стандартах кодирования, и ваша группа может разработать (или прибавить) свои правила, в наибольшей степени подходящие для вашей конкретной группы или для конкретной решаемой задачи, так что вы не должны быть скованы этой книгой по рукам и ногам. Тем не менее, мы надеемся, что эта книга поможет вам сберечь время и усилия при разработке собственного стандарта кодирования, а также будет способствовать повышению его качества и последовательности.

Ознакомьте членов своей команды с этой книгой, и после того, как они полностью прочтут ее и познакомятся со всеми рекомендациями и их обоснованиями, решите, какие из них подходят вам, а какие в силу особенностей вашего проекта для вас неприменимы. После этого строго придерживайтесь выбранной стратегии. После того как командный стандарт принят, он не должен нарушаться иначе как по согласованному решению всей команды в целом.

И наконец, периодически творчески пересматривайте собственные стандарты с учетом практического опыта, приобретенного всей командой при работе над проектом.

Стандарты кодирования и вы

Хорошие стандарты кодирования могут принести немалую выгоду с различных точек зрения.

- *Повышение качества кода.* Работа в соответствии со стандартом приводит к однотипному решению одинаковых задач, что повышает ясность кода и упрощает его сопровождение.
- *Повышение скорости разработки.* Разработчику не приходится решать все задачи и принимать решения "с нуля".
- *Повышение уровня взаимодействия в команде.* Наличие стандарта позволяет уменьшить разногласия в команде и устранить ненужные дебаты по мелким вопросам, облегчает понимание и поддержку чужого кода членами команды.
- *Согласованность в работе.* При использовании стандарта разработчики направляют свои усилия в верном направлении, на решение действительно важных задач.

В напряженной обстановке, при жестких временных рамках люди обычно делают то, чему их учили, к чему они привыкли. Вот почему в больницах в пунктах первой помощи предпочитают опытных, тренированных сотрудников — даже хорошо обученные и знающие новички склонны к панике.

У разработчиков программного обеспечения регулярно возникают ситуации, когда что-то надо было сделать еще вчера — на позавчера. Когда на нас давит график работ (который к тому же имеет тенденцию сдвигаться в одном направлении, и то, что по плану должно было заработать завтра, от нас начинают требовать еще вчера...), мы работаем так, как приучены. Неряшливые программисты, которые даже при обычной неспешной работе не помнят о правильных принципах разработки программного обеспечения (а то и вовсе не знакомы с ними), при нехватке времени окажутся еще небрежнее, а их код будет изобиловать ошибками. Соответственно, программист, который выработал в себе хорошие привычки и регулярно ими пользуется, при "повышенном давлении" будет продолжать выдавать качественный код.

Стандарты кодирования, приведенные в этой книге, представляют собой набор рекомендаций по написанию высококачественного кода на C++. В них сконцентрирован богатый коллективный опыт всего сообщества программистов на C++. Многие из этих знаний разбросаны по частям по самым разным книгам, но не меньшее количество знаний передается изустно. Мы постарались собрать разрозненные сведения в одной книге в виде коллекции ясных, компактных правил с пояснениями, простых для понимания и следования им.

Конечно, даже самые лучшие стандарты не могут помешать написанию плохого кода. То же можно сказать о любом языке программирования, процессе или методологии. Хороший набор стандартов воспитывает хорошие привычки и дисциплину, превышающую обычные

нормы. Это служит хорошим фундаментом для дальнейшего усовершенствования и повышения квалификации. Это не преувеличение и не красивые слова — перед тем, как начать писать стихи, надо владеть словарным запасом и знать грамматику. Мы надеемся, что наша книга упростит для вас путь к поэзии программирования.

Эта книга предназначена для программистов всех уровней.

Если вы начинающий программист — мы надеемся, что рекомендации и их пояснения достаточно поучительны и помогут вам в понимании того, какие стили и идиомы C++ поддерживает наиболее естественным образом. В описании каждого правила и рекомендации приводится краткое обоснование и обсуждение, чтобы вы не просто механически запомнили правило, а поняли его суть.

Для программистов среднего и высокого уровня при описании каждого правила приводится список ссылок, который позволит вам углубленно изучить заинтересовавший вас вопрос, проведя поиск корней правила в системе типов, грамматике и объектной модели C++.

Каким бы ни был ваш уровень как программиста — вероятно, вы работаете над сложным проектом не в одиночку, а в команде. Именно в этом случае разработка стандартов кодирования окупается сполна. Вы можете использовать их для того, чтобы подтянуть всю свою команду к одному, более высокому уровню, и обеспечить повышение качества разрабатываемого кода.

Об этой книге

Основными принципами дизайна данной книги являются следующие.

- *Краткость — сестра таланта.* Чем больше стандарт кодирования по размеру, тем больше шансов, что он будет благополучно проигнорирован. Читают и используют обычно короткие стандарты. Длинные разделы, как правило, просто просматривают "по диагонали", короткие статьи обычно удостаиваются внимательного прочтения.
- *Никакой материал не должен вызывать дискуссий.* Эта книга документирует широко используемые стандарты, а не изобретает их. Если некоторая рекомендация не применима во всех ситуациях, то мы так и пишем — "подумайте о применении X" вместо "делайте X". Кроме того, к каждому правилу указаны все общепринятые исключения.
- *Весь материал должен быть обоснован.* Все рекомендации в этой книге взяты из существующих печатных работ. В конце книги приведен список использованной литературы по C++.
- *Материал не должен быть банален.* Мы не даем рекомендации, которым вы и так следуете, которые обеспечиваются компилятором или которые уже изложены в других разделах.
  - Например, "не возвращайте указатель/ссылку на локальную переменную" — хороший совет, но он не включен в данную книгу, поскольку практически все компиляторы выдают соответствующее предупреждение, к тому же этот вопрос раскрывается в первом разделе книги.
  - Рекомендация "используйте редактор (компилятор, отладчик)" — тоже хороший совет, но, конечно, вы используете эти инструменты и без нашего напоминания. Вместо этого мы рекомендуем использовать автоматизированные системы сборки программ и системы управления версиями.
  - Еще один совет — "не злоупотребляйте `goto`" — исходя из нашего опыта, и так широко известен всем программистам, так что нет смысла повторяться.

Каждый раздел состоит из следующих частей.

- *Заглавие.* Краткое название раздела, поясняющее, о чем будет идти речь.
- *Резюме.* Краткое изложение сути вопроса.

- *Обсуждение*. Расширенное пояснение рекомендации. Зачастую включает краткое обоснование, но учтите, что полную информацию по данному вопросу следует искать в приведенных ссылках.
- *Примеры* (если таковые имеются). Примеры, демонстрирующие правило или позволяющие лучше его понять и запомнить.
- *Исключения* (если таковые имеются). Описание ситуаций (обычно редких), когда приведенное правило неприменимо. Однако остерегайтесь попасть в ловушку, думая, что ваш случай особый и что в вашей ситуации это правило неприменимо, — обычно при здравом размышлении оказывается, что ничего особого в вашей ситуации нет и описанное правило может быть с успехом вами применено.
- *Ссылки*. В приведенной в этом подразделе литературе по C++ вы найдете более полный анализ рассматриваемого в разделе вопроса.

В каждой части книги имеется "наиболее важный раздел" — обычно это первый раздел части. Однако иногда это правило нарушалось в связи с необходимостью последовательного и связного изложения материала.

Благодарности

Мы от всей души благодарны редактору серии Бьярну Страуструпу (Bjarne Stroustrup), редакторам Питеру Гордону (Peter Gordon) и Дебби Лафферти (Debbie Lafferty), а также Тирреллу Албаху (Tyrrell Albaugh), Ким Бодихаймер (Kim Boedigheimer), Джону Фуллеру (John Fuller), Бернарду Гаффни (Bernard Gaffney), Курту Джонсону (Curt Johnson), Чанде Лири-Коту (Chanda Leary-Coutu), Чарли Ледди (Charles Leddy), Хитеру Муллэйн (Heather Mullane), Чути Прасертсиху (Chuti Prasertsith), Ларе Вайсонг (Lara Wysong) и всем остальным работникам издательства Addison-Wesley, помогавшим нам в нашей работе над этим проектом. Нам было очень приятно работать с ними.

На идею и оформление книги нас натолкнули несколько источников, включая такие, как [Cline99], [Peters99], а также работы легендарного и широко цитируемого Алана Перлиса (Alan Perlis).

Особая благодарность тем людям, чьи отзывы помогли нам сделать многие части книги намного лучшими, чем они были бы без этих замечаний. Особенно большое влияние на книгу оказали острые комментарии Бьярна Страуструпа. Мы очень хотим поблагодарить за активное участие в обсуждении материала и высказанные замечания таких людей, как Дэйв Абрамс (Dave Abrahams), Маршалл Клайн (Marshall Cline), Кевлин Хенни (Kevlin Henney), Говард Хиннант (Howard Hinnant), Джим Хайслоп (Jim Hyslop), Николаи Джосаттис (Nicolai Josuttis), Йон Калб (Jon Kalb), Макс Хесин (Max Khesin), Стен Липпман (Stan Lippman), Скотт Мейерс (Scott Meyers) и Дэвид Вандевурд (Daveed Vandevoorde). Кроме того, отдельное спасибо хотелось бы сказать Чаку Аллисону (Chuck Allison), Самиру Байяю (Samir Bajaj), Марку Барбуру (Marc Barbour), Тревису Брауну (Travis Brown), Нилу Кумбесу (Nil Coombes), Дамиану Дечеву (Damian Dechev), Стиву Дьюхарсту (Steve Dewhurst), Питеру Димову (Peter Dimov), Аттиле Фехеру (Attila Feher), Алану Гриффитсу (Alan Griffiths), Мичи Хеннингу (Michi Henning), Джеймсу Канзе (James Kanze), Бартошу Милевски (Bartosz Milewski), Мэтту Маркусу (Matt Marcus), Балогу Палу (Balog Pal), Владимиру Прусу (Vladimir Prus), Дэну Саксу (Dan Saks), Люку Вагнеру (Luke Wagner), Мэтью Вильсону (Matthew Wilson) и Леору Золману (Leor Zolman).

Как обычно, все оставшиеся в книге ошибки и недосмотры — на нашей совести, а не на их.

Герб Саттер (Herb Sutter)
Андрей Александреску (Andrei Alexandrescu)

Сиэтл, сентябрь 2004

Вопросы организации и стратегии

Если бы строители строили здания так же, как программисты пишут программы, — то первый же залетевший дятел разрушил бы всю цивилизацию.

— Джеральд Вайнберг (Gerald Weinberg)

Следуя великой традиции C и C++, мы начинаем отсчет с нуля. Главный совет — под номером 0 — говорит о том, что основной советчик по поводу стандартов кодирования — наши чувства и ощущения.

Остальная часть этой главы состоит из небольшого количества тщательно отобранных вопросов, которые не имеют прямого отношения к коду и посвящены важным инструментам и методам написания надежного кода.

В этом разделе книги наиболее важной мы считаем нулевую рекомендацию — "Не мелочитесь, или Что не следует стандартизировать".

0. Не мелочитесь, или Что не следует стандартизировать

Резюме

Скажем кратко: не мелочитесь.

Обсуждение

Вопросы персонального вкуса, которые не влияют на корректность и читаемость кода, не относятся к стандарту кодирования. Любой профессиональный программист сможет легко прочесть и записать код, форматирование которого немного отличается от того, которым он обычно пользуется.

Используйте одно и то же форматирование в пределах одного исходного файла или даже целого проекта, поскольку переход от одного стиля форматирования к другому в пределах одного фрагмента исходного текста достаточно сильно раздражает. Но не пытайтесь обеспечить одно и то же форматирование в разных проектах или в целом по всей компании.

Вот несколько вопросов, в которых не важно точное следование правилу, а требуется всего лишь последовательное применение стиля, используемого в файле, с которым вы работаете.

- *Не следует определять конкретный размер отступа, но следует использовать отступы для подчеркивания структуры программы.* Для отступа используйте то количество символов, которое вам нравится, но это количество должно быть одинаково, как минимум, в пределах файла.
- *Не определяйте конкретную длину строки, но она должна оставлять текст удобочитаемым.* Используйте ту длину строки, которая вам по душе, но не злоупотребляйте ею. Исследования показали, что легче всего воспринимается текст, в строке которого находится до десяти слов.
- *Следует использовать непротиворечивые соглашения об именовании, не слишком мелочно регламентируя его.* Имеется только два императивных требования по поводу именования: никогда не используйте имена, начинающиеся с подчеркивания или содержащие двойное подчеркивание, и всегда используйте для макросов только прописные буквы (`ONLY_UPPERCASE_NAMES`), при этом никогда не применяя в качестве имен макросов обычные слова или сокращения (включая распространенные параметры шаблонов, такие как `T` или `U`; запись `#define T` *`anything`* может привести к крупным неприятностям). В остальных случаях используйте непротиворечивые значимые имена и следуйте соглашениям, принятым для данного файла или модуля. (Если вы не можете сами разработать соглашение об именовании, попробуйте воспользоваться следующим: имена классов, функций и перечислений должны выглядеть как `LikeThis`, имена переменных — `likeThis`, имена закрытых членов-данных — `likeThis_`, и имена макросов — `LIKE_THIS`.)
- *Не предписывайте стиль комментариев (кроме тех случаев, когда специальный инструментарий использует их для документирования), но пишите только нужные и полезные комментарии.* Вместо комментариев пишите, где это возможно, код (см., например, руководство 16). Не пишите комментарии, которые просто повторяют код. Комментарии должны разъяснять использованный подход и обосновывать его.

И наконец, не пытайтесь заставлять использовать устаревшие правила (см. примеры 2 и 3), даже если они имеются в старых стандартах кодирования.

Примеры

Пример 1. Размещение фигурных скобок. Нет никакой разницы в плане удобочитаемости следующих фрагментов:

```
void using_k_and_r_style() {
// ...
}
void putting_each_brace_on_its_own_line()
{
// ...
}
void or_putting_each_brace_on_its_own_line_indented()
    {
// ...
    }
```

Все профессиональные программисты могут легко читать *и писать* в каждом из этих стилей без каких-либо сложностей. Но следует быть последовательным. Не размещайте скобки как придется или так, что их размещение будет скрывать вложенность областей видимости, и пытайтесь следовать стилю, принятому в том или ином файле. В данной книге размещение скобок призвано обеспечить максимальную удобочитаемость, при этом оставаясь в рамках, определенных редакторскими ограничениями.

Пример 2. Пробелы или табуляция. В некоторых командах использование табуляции запрещено (например, [BoostLRG]) на том основании, что размер табуляции варьируется от редактора к редактору, а это приводит к тому, что отступы оказываются слишком малы или слишком велики. В других командах табуляции разрешены. Важно только быть последовательным. Если вы позволяете использовать табуляцию, убедитесь, что такое решение не будет мешать ясности кода и его удобочитаемости, если члены команды будут сопровождать код друг друга (см. руководство 6). Если использование табуляции не разрешено, позвольте редактору преобразовывать пробелы в табуляции при чтении исходного файла, чтобы программисты могли работать с ними в редакторе. Однако убедитесь, что при сохранении файла табуляция будет вновь преобразована в пробелы.

Пример 3. Венгерская запись. Запись, при которой информация о типе включается в имя переменной, приносит пользу в языке программирования, небезопасном с точки зрения типов (особенно в C); возможна, хотя и не приносит никакой пользы (только недостатки) в объектно-ориентированных языках; и невозможна в обобщенном программировании. Таким образом, стандарт кодирования C++ не должен требовать использования венгерской записи, более того, может потребовать ее запрета.

Пример 4. Один вход, один выход (Single entry, single exit — "SESE"). Исторически некоторые стандарты кодирования требуют, чтобы каждая функция имела в точности один выход, что подразумевает одну инструкцию `return`. Такое требование является устаревшим в языках, поддерживающих исключения и деструкторы, так что функции обычно имеют несколько неявных выходов. Вместо этого стоит следовать стандарту наподобие рекомендации 5, которая требует от функций простоты и краткости, что делает их более простыми для понимания и более устойчивыми к ошибкам.

Ссылки

[BoostLRG] • [Brooks95] §12 • [Constantine95] §29 • [Keffer95] p. 1 • [Kernighan99] §1.1, §1.3, §1.6-7 • [Lakos96] §1.4.1, §2.7 • [McConnell93] §9, §19 • [Stroustrup94] §4.2-3 • [Stroustrup00] §4.9.3, §6.4, §7.8, §C.1 • [Sutter00] §6, §20 • [SuttHysl01]

1. Компилируйте без замечаний при максимальном уровне предупреждений

Резюме

Следует серьезно относиться к предупреждениям компилятора и использовать максимальный уровень вывода предупреждений вашим компилятором. Компиляция должна выполняться без каких-либо предупреждений. Вы должны понимать все выдаваемые предупреждения и устранять их путем изменения кода, а не снижения уровня вывода предупреждений.

Обсуждение

Ваш компилятор — ваш друг. Если он выдал предупреждение для определенной конструкции, зачастую это говорит о потенциальной проблеме в вашем коде.

Успешная сборка программы должна происходить молча (без предупреждений). Если это не так, вы быстро приобретаете привычку не обращать внимания на вывод компилятора и можете пропустить серьезную проблему (см. рекомендацию 2).

Чтобы избежать предупреждений, надо понимать, что они означают, и перефразировать ваш код так, чтобы устранить предупреждения и сделать предназначение кода более понятным как для компилятора, так и для человека.

Делайте это всегда — даже если программа выглядит корректно работающей. Делайте это, даже если убедились в мягкости предупреждения. Даже легкие предупреждения могут скрывать последующие предупреждения, указывающие на реальную опасность.

Примеры

Пример 1. Заголовочный файл стороннего производителя. Библиотечный заголовочный файл, который вы не можете изменить, может содержать конструкцию, которая приводит к (вероятно, мелкому) предупреждению. В таком случае "заверните" этот файл в свой собственный, который будет включать исходный при помощи директивы `#include` и избирательно отключать для него конкретные предупреждения. В своем проекте вы будете использовать собственный заголовочный файл, а не исходный. Например (учтите — управление выводом предупреждений варьируется от компилятора к компилятору):

```
// Файл: myproj/my_lambda.h - "обертка" для lambda.hpp из
// библиотеки Boost. Всегда включайте именно этот файл и не
// используйте lambda.hpp непосредственно. Boost.Lambda
// приводит к выводу компилятором предупреждений, о
// безвредности которых нам доподлинно известно. Когда
// разработчики сделают новую версию, которая не будет
// вызывать предупреждений, мы удалим из этого файла
// соответствующие директивы #pragma, но сам заголовочный
// файл останется.
//
#pragma warning(push) // Отключение предупреждений только
                      // для данного заголовочного файла
#pragma warning(disable:4512)
#pragma warning(disable:4180)
```

```
#include <boost/lambda/lambda.hpp>
#pragma warning(pop)  // Восстанавливаем исходный уровень
                      // вывода предупреждений
```

Пример 2. "Неиспользуемый параметр функции". Убедитесь, что вы в самом деле сознательно не используете параметр функции (Например, это "заглушка" для будущего расширения или требуемая стандартом часть сигнатуры, которую ваш код не использует). Если этот параметр вам действительно не нужен, просто удалите его имя:

```
// ... внутри пользовательского распределителя подсказка не
// используется ...

// Предупреждение: "неиспользуемый параметр 'localityHint'"
pointer allocate( size_type numObjects,
                  const void *localityHint = 0 ) {
    return static_cast<pointer>(
            mallocShared( numObjects * sizeof(T) ) );
}

// Новая версия: предупреждение устранено
pointer allocate( size_type numObjects,
                  const void * /* localityHint */ = 0 ) {
    return static_cast<pointer>(
            mallocShared( numObjects * sizeof(T) ) );
}
```

Пример 3. "Переменная определена, но не используется". Убедитесь, что вы действительно не намерены обращаться к данной переменной (к таким предупреждениям часто приводят локальные объекты, следующие идиоме "выделение ресурса есть инициализация", см. рекомендацию 13). Если обращение к объекту действительно не требуется, часто можно заставить компилятор замолчать, включив "вычисление" самой переменной в качестве выражения (такое вычисление не влияет на скорость работы программы):

```
// Предупреждение: "переменная 'lock' определена, но не
// используется"
void Fun() {
    Lock lock;
    // ...
}

// Новая версия: предупреждение не должно выводиться
void Fun() {
    Lock lock;
    lock;
    // ...
}
```

Пример 4. "Переменная может использоваться, не будучи инициализированной". Инициализируйте переменную (см. рекомендацию 19).

Пример 5. "Отсутствует return". Иногда компиляторы требуют наличия инструкции `return` несмотря на то, что поток управления не может достичь конца функции (например, при наличии бесконечного цикла, инструкции `throw`, других инструкций `return`). Такое предупреждение не стоит игнорировать, поскольку вы можете *только думать*, что управление не достигает конца функции. Например, конструкция `switch`, у которой нет выбора `default`, при внесении изменений в программу может привести к неприятностям, так что следует иметь выбор `default`, который просто выполняет `assert(false)` (см. также рекомендации 68 и 90):

```
// Предупреждение: отсутствующий "return"
int Fun( Color c ) {
    switch( c ) {
        case Red: return 2;
        case Green: return 0;
```

```
        case Blue:
        case Black: return 1;
    }
}

// Новая версия: предупреждение устранено
int Fun( Color c ) {
    switch( c ) {
        case Red: return 2;
        case Green: return 0;
        case Blue:
        case Black: return 1;
        // Значение !"string" равно false:
        default: assert(!"should never get here!");
                 return -1;
    }
}
```

Пример 6. "Несоответствие `signed`/`unsigned`". Обычно не возникает необходимость сравнивать или присваивать числа с разным типом знаковости. Измените типы сравниваемых переменных так, чтобы они соответствовали друг другу. В крайнем случае, воспользуйтесь явным преобразованием типов. (Компилятор все равно вставляет в код преобразование типов и предупреждает именно об этом, так что лучше сделать то же самостоятельно.)

Исключения

Иногда компилятор может выдавать утомительные, а порой и просто ложные предупреждения, но у вас нет способа их устранения (или такой способ заключается в нереальной или непроизводительной переделке текста программы). В таких редких случаях по решению всей команды разработчиков можно пойти на отключение конкретных мелких предупреждений, которые на самом деле не несут никакой особой информации и являются не более, чем результатом чрезмерной осторожности компилятора. Такие предупреждения можно отключить, но только данные конкретные предупреждения, а не все, максимально локализовав при этом область отключения и сопроводив ее ясным и подробным комментарием, почему такой шаг необходим.

Ссылки

[Meyers97] §48 • [Stroustrup94] §2.6.2

2. Используйте автоматические системы сборки программ

Резюме

Нажимайте на одну (единственную) кнопку: используйте полностью автоматизированные ("в одно действие") системы, которые собирают весь проект без вмешательства пользователя.

Обсуждение

Процесс сборки программы "в одно действие" очень важен. Он должен давать надежный и воспроизводимый результат трансляции ваших исходных файлов в распространяемый пакет. Имеется богатый выбор таких автоматизированных инструментов сборки, так что нет никакого оправдания тому, что вы их не используете. Выберите один из них и применяйте его в своей работе.

Мы встречались с организациями, где подобное требование игнорировалось. Некоторые полагают, что настоящий процесс сборки должен состоять в том, чтобы пощелкать мышью там и сям, запустить пару утилит для регистрации серверов COM/CORBA и вручную скопировать несколько файлов. Вряд ли у вас есть лишнее время и энергия, чтобы растрачивать их на то, что машина сделает быстрее и лучше вас. Вам нужна надежная автоматизированная система сборки программы "в одно действие".

Успешная сборка должна происходить молча, без каких бы то ни было предупреждений (см. рекомендацию 1). Идеальный процесс сборки должен выдать только одно журнальное сообщение: "Сборка успешно завершена".

Есть две модели сборки — инкрементная и полная. При инкрементной сборке компилируются только те файлы, которые претерпели изменения со времени последней инкрементной или полной сборки. Следствие: вторая из двух последовательных инкрементных сборок не должна перезаписывать никакие файлы. Если она это делает — по всей видимости, у вас в проекте имеется циклическая зависимость (см. рекомендацию 22), либо ваша система сборки выполняет ненужные операции (например, создает фиктивные временные файлы, чтобы затем просто удалить их).

Проект может иметь несколько видов полной сборки. Рассмотрите возможность параметризации процесса сборки при помощи таких важных параметров, как целевая архитектура, отладочная или коммерческая версии, вид пакета (программа-инсталлятор или просто набор файлов). Одни установки сборки могут давать только наиболее существенные исполнимые и библиотечные файлы, другие — добавлять к ним вспомогательные файлы, а окончательная сборка — создавать программу-инсталлятор, которая включает в себя все ваши файлы, файлы сторонних производителей и код инсталляции.

Со временем размер проекта обычно возрастает, растет и стоимость отказа от автоматизированной системы сборки. Если вы не используете ее с самого начала, вы теряете время и ресурсы. Все равно со временем потребность в такой системе станет непреодолимой, но при этом вы окажетесь под гораздо большим давлением, чем в начале проекта.

В больших проектах возможна даже специальная должность "хозяина сборки", который отвечает за работу этой системы.

Ссылки

[Brooks95] §13, §19 • [Dewhurst03] §1 • [GnuMake] • [Stroustrup00] §9.1

3. Используйте систему контроля версий

Резюме

Как гласит китайская пословица, плохие чернила лучше хорошей памяти: используйте системы управления версиями. Не оставляйте файлы без присмотра на долгий срок. Проверяйте их всякий раз после того, как обновленные модули проходят тесты на работоспособность. Убедитесь, что внесенные обновления не препятствуют корректной сборке программы.

Обсуждение

Практически все нетривиальные проекты требуют командной работы и/или более недели рабочего времени. В таких проектах вы будете просто *вынуждены* сравнивать различные версии одного и того же файла для выяснения того, когда (и/или кем) были внесены те или иные изменения. Вы будете *вынуждены* контролировать и руководить внесением изменений в исходные файлы проекта.

Когда над проектом работают несколько разработчиков, они вносят изменения в проект параллельно, возможно, одновременно в разные части одного и того же файла. Вам нужен инструмент, который бы автоматизировал работу с разными версиями файлов и, в определенных случаях, позволял объединять одновременно внесенные изменения. Система управления версиями (version control system, VCS) автоматизирует все необходимые действия, причем выполняя их более быстро и корректно, чем вы бы могли сделать это вручную. И вряд ли у вас есть лишнее время на игры в администратора — у вас хватает и своей работы по разработке программного обеспечения.

Даже единственный программист нередко вынужден выяснять, как и когда в программу проникла та или иная ошибка. VCS, автоматически отслеживая историю изменений каждого файла, позволяет вам "перевести стрелки часов назад" и ответить не только на вопрос, как именно выглядел файл раньше, но и когда это было.

Не портите сборку. Код, сохраненный VCS, всегда должен успешно собираться.

Огромное разнообразие инструментария данного типа не позволяет оправдать вас, если вы не используете одну из этих систем. Наиболее дешевой и популярной является `CVS` (см. ссылки). Это гибкий инструмент с возможностью обращения по TCP/IP, возможностью обеспечения повышенных мер безопасности (с использованием протокола `ssh`), возможностью администрирования с применением сценариев и даже графическим интерфейсом. Многие другие продукты VCS рассматривают `CVS` в качестве стандарта либо строят новую функциональность на ее основе.

Исключения

Проект, над которым работает один программист, причем не более недели, вероятно, в состоянии выжить и без применения VCS.

Ссылки

[BetterSCM] • [Brooks95] §11, §13 • [CVS]

4. Одна голова хорошо, а две — лучше

Резюме

Регулярно просматривайте код всей командой. Чем больше глаз — тем выше качество кода. Покажите ваш код другим и познакомьтесь с их кодом — это принесет пользу всем.

Обсуждение

Регулярное рецензирование кода другими членами команды приносит свои плоды.

- Повышение качества кода при доброжелательной критике другими.
- Выявление ошибок, непереносимого кода (если это важно) и потенциальных проблем, связанных с масштабированием проекта.
- Улучшение качества дизайна и реализации путем обмена идеями.
- Быстрое обучение новых членов команды.
- Разработка общих принципов в команде.
- Повышение уровня меритократии[1], доверия, профессиональной гордости и сотрудничества в команде.

Во многих фирмах еще не столь популярны, как хотелось бы, вознаграждения за качество кода, а также какие-либо вложения времени или средств в его повышение. Надежд на кардинальное изменение ситуации мало, но все же изменения в этой области, пусть медленно, но происходят; в частности, это связано с вопросами надежности и безопасности программного обеспечения. Коллективное рецензирование кода помогает в решении этих вопросов, в дополнение к тому, что это отличный и к тому же бесплатный метод обучения сотрудников.

Даже если ваш работодатель не намерен заботиться о рецензировании кода, постарайтесь донести мысль о его необходимости до менеджеров (маленькая подсказка: покажите им эту книгу), и в любом случае организуйте такой обмен опытом. Затраченное на это время окупится сторицей. Сделайте рецензирование кода обязательной частью цикла разработки программного обеспечения в вашей команде.

Лучшие результаты дает оперативное рецензирование кода, разрабатываемого в настоящий момент. Для этого не требуется никакого формализма — достаточно простой электронной почты. При такой организации работы легче отслеживать ваши собственные действия и избегать дублирования.

Когда вы работаете с чужим кодом, зачастую удобно иметь под рукой список того, на что именно следует обращать внимание. Мы скромно предлагаем в качестве варианта такого списка оглавление данной книги.

Мы знаем, что читаем проповедь, но мы должны были сказать это вслух. Да, ваше эго может ненавидеть показывать свои исходные тексты для всеобщей критики, но поверьте, маленькому гениальному программисту внутри вас по душе рецензирование его кода, потому что в результате он сможет писать еще более талантливые программы.

Ссылки

[Constantine95] §10, §22, §33 • [McConnell93] §24 • [MozillaCRFAQ]

---

[1] Система, при которой положение человека в обществе определяется его способностями. — *Прим. перев.*

Стиль проектирования

Дураки игнорируют сложности. Прагматики терпят их. Некоторые ухитряются их избегать. И только гении устраняют их.

— Алан Перлис (Alan Perlis)

Я также знал, но забыл афоризм Хоара о том, что преждевременная оптимизация — корень всех зол в программировании.

— Дональд Кнут (Donald Knuth),
The Errors of TeX [Knuth89]

Очень сложно отделить стиль проектирования от стиля кодирования. Мы попытались поместить в очередном разделе те вопросы, которые обычно проявляются, когда вы начинаете писать реальный код.

В этом разделе внимание уделяется принципам и практическим вопросам, область применения которых больше, чем просто отдельный класс или функция. Классические примеры — баланс между простотой и ясностью (рекомендация 6), избежание преждевременной оптимизации (рекомендация 8) и пессимизации (рекомендация 9). Эти рекомендации применимы не только на уровне кодирования отдельной функции, но и к большим областям — проектированию классов и модулей или к решениям с далеко идущими последствиями по поводу архитектуры приложений. (Они применимы также для всех программистов. Если вы считаете иначе — еще раз прочтите приведенную выше цитату Кнута и даже заучите ее на память.)

Многие из прочих рекомендаций этого и следующего разделов имеют дело с управлением зависимостями — краеугольным камнем проектирования программного обеспечения и часто повторяющейся в данной книге темой. Остановитесь и задумайтесь над произвольной методикой проектирования программного обеспечения — *любой хорошей* методикой. Какую бы методику вы ни выбрали, так или иначе вы столкнетесь с ослаблением зависимостей. Наследование? Разрабатываемый код должен делать производный класс как можно менее зависимым от базового класса. Минимизация использования глобальных переменных? Снижает дальнодействующие зависимости, осуществляемые посредством широко видимых разным частям программы данных. Абстракция? Устраняет зависимости между кодом, который управляет концепциями, и кодом, который их реализует. Сокрытие информации? Делает код клиента менее зависимым от деталей реализации. Забота об управлении зависимостями отражается в устранении совместного использования состояния (рекомендация 10), применении сокрытия информации (рекомендация 11) и прочем.

В этом разделе самой важной нам кажется рекомендация 6: " Главное — корректность, простота и ясность".

5. Один объект — одна задача

Резюме

Концентрируйтесь одновременно только на одной проблеме. Каждый объект (переменная, класс, функция, пространство имен, модуль, библиотека) должны решать одну точно поставленную задачу. С ростом объектов, естественно, увеличивается область их ответственности, но они не должны отклоняться от своего предназначения.

Обсуждение

Хорошая идея, будучи высказанной вслух, должна быть пояснена одним предложением. Аналогично, каждая сущность в программе должна иметь одно ясное предназначение.

Объект с разнородными предназначениями обычно несоразмерно трудно использовать, поскольку он представляет собой нечто большее, чем просто сумму решений, сложностей и ошибок составляющих его частей. Такой объект больше по размеру (зачастую без особых на то причин) и сложнее в применении и повторном использовании. Зачастую такие объекты имеют весьма убогий интерфейс для каждого из своих отдельных предназначений, поскольку частичное перекрытие разных областей функциональности приводит к невозможности четкой реализации в каждой из них.

Объекты с разнородными функциями обычно трудны для проектирования и реализации. "Множественная ответственность" зачастую приводит к тому, что количество возможных вариантов поведения и состояния объектов разрастается в соответствии с законами комбинаторики. Предпочтительнее использовать короткие функции с четко указанным единственным предназначением (см. также рекомендацию 39), маленькие классы, предназначенные для решения одной конкретной задачи, и компактные модули с четко очерченными границами.

Абстракции высокого уровня предпочтительно строить из меньших низкоуровневых абстракций. Избегайте объединения нескольких низкоуровневых абстракций в большой низкоуровневый конгломерат. Реализация сложного поведения из набора простых существенно легче решения обратной задачи.

Примеры

Пример 1. `realloc`. В стандарте C функция `realloc` пользуется дурной славой плохо спроектированной функции. Она делает одновременно слишком много дел: выделяет память, если ей передано значение `NULL`, освобождает ее, если передан нулевой размер, перераспределяет ее на том же месте, если это возможно, или перемещает ее, если такое перераспределение невозможно. Это уже не просто расширение функциональности. Данная функция обычно рассматривается всеми как пример недальновидного, ошибочного проектирования.

Пример 2. `basic_string`. В стандарте C++ `std::basic_string` служит таким же пользующимся дурной славой примером проектирования монолитного класса. В этот раздутый до неимоверных размеров класс добавлены все функции, о которых только можно было подумать. Этот класс пытается быть контейнером, что не совсем ему удается; в нем так и остается неразрешенным вопрос между применением итераторов и индексированием. Кроме того, в нем совершенно необоснованно дублируется множество стандартных алгоритмов. В то же время этот класс оставляет очень мало возможностей для расширения. (См. пример к рекомендации 44).

Ссылки

[Henney02a] • [Henney02b] • [McConnell93] §10.5 • [Stroustrup00] §3.8, §4.9.4, §23.4.3.1 • [Sutter00] §10, §12, §19, §23 • [Sutter02] §1 • [Sutter04] §37-40

6. Главное — корректность, простота и ясность

Резюме

Корректность лучше быстроты. Простота лучше сложности. Ясность лучше хитроумия. Безопасность лучше ненадежности (см. рекомендации 83 и 99).

Обсуждение

Сложно преувеличить значение простоты проектирования и ясности кода. Люди, которые будут сопровождать ваш код, скажут вам только спасибо за то, что вы сделали ваш код понятным. Кстати, зачастую это будете вы сами — когда будете вспоминать, о чем это вы думали полгода назад и как же работает этот код, который вы тогда написали... Прислушайтесь к следующим словам.

Программа должна быть написана для человека, который будет ее читать, и только попутно — для машины, которая будет ее выполнять.

— Гарольд Абельсон (Harold Abelson)
и Джеральд Сассман (Gerald Jay Sussman)

Пишите программы в первую очередь для людей, и только потом для машин.

— Стив Мак-Коннелл (Steve McConnell)

Самые дешевые, быстрые и надежные компоненты вычислительной системы — те, которых в ней нет.

— Гордон Белл (Gordon Bell)

Эти отсутствующие компоненты также наиболее точны (они никогда не ошибаются), наиболее надежны (никогда не ломаются) и наиболее просты в разработке, документировании, тестировании и сопровождении. Важность простоты дизайна невозможно переоценить.

— Йон Бентли (Jon Bentley)

Многие из рекомендаций этой книги естественным образом приводят к легко изменяемому дизайну и коду, а ясность является наиболее желанным качеством для программы, которую легко сопровождать. То, что вы не в состоянии понять, вы не сможете уверенно и надежно переделать.

Вероятно, наиболее распространенным соперничеством в данной области является соперничество между ясностью кода и его оптимизацией (см. рекомендации 7, 8 и 9). Когда — не если — вы находитесь перед соблазном преждевременной оптимизации для повышения производительности и, таким образом, пессимизации для повышения ясности, — вспомните, что говорит рекомендация 8: гораздо проще сделать корректную программу быстрой, чем быструю — корректной.

Избегайте "темных закутков" языка. Используйте простейшие из эффективных методов.

Примеры

Пример 1. Избегайте неуместной и/или чересчур хитроумной перегрузки операторов. В одной библиотеке пользовательского графического интерфейса пользователей без нужды заставляют писать `w+c;` для того, чтобы добавить в окно `w` дочерний управляющий элемент `c` (см. рекомендацию 26).

Пример 2. В качестве параметров конструктора лучше использовать именованные, а не временные переменные. Это позволит избежать возможных неоднозначностей объявлений, а также зачастую проясняет назначение вашего кода и тем самым упрощает его сопровождение. Кроме того, часто это просто безопаснее (см. рекомендации 13 и 31).

Ссылки

[Abelson96] • [Bentley00] §4 • [Cargill92] pp. 91-93 • [Cline99] §3.05-06 • [Constantine95] §29 • [Keffer95] p. 17 • [Lakos96] §9.1, §10.2.4 • [McConnell93] • [Meyers01] §47 • [Stroustrup00] §1.7, §2.1, §6.2.3, §23.4.2, §23.4.3.2 • [Sutter00] §40-41, §46 • [Sutter04] §29

7. Кодирование с учетом масштабируемости

Резюме

Всегда помните о возможном росте данных. Подумайте об асимптотической сложности без преждевременной оптимизации. Алгоритмы, которые работают с пользовательскими данными, должны иметь предсказуемое и, желательно, не хуже, чем линейно зависящее от количества обрабатываемых данных время работы. Когда становится важной и необходимой оптимизация, в особенности из-за роста объемов данных, в первую очередь следует улучшать *O*-сложность алгоритма, а не заниматься микрооптимизациями типа экономии на одном сложении.

Обсуждение

Эта рекомендация иллюстрирует важную точку равновесия между рекомендациями 8 и 9 — не оптимизируйте преждевременно и не пессимизируйте преждевременно. Это делает данный материал трудным в написании, поскольку он может быть неверно истолкован как совет о "преждевременной оптимизации". Это не так.

Вот предпосылки для данной рекомендации. Память и дисковая емкость растут экспоненциально; например, с 1988 по 2004 год емкость дисков росла примерно на 112% в год (почти в 1900 раз за десятилетие). Очевидным следствием этого факта является то, что любой ваш сегодняшний код завтра может иметь дело с большими объемами данных — *намного* большими! Плохое (хуже линейного) асимптотическое поведение алгоритма рано или поздно поставит на колени даже самую мощную систему, просто завалив ее достаточным количеством данных.

Защита против такого будущего означает, что мы должны избежать встраивания в наши программы того, что станет западней при работе с большими файлами, большими базами данных, с большим количеством пикселей, большим количеством окон, процессов, битов, пересылаемых по каналам связи. Одним из важных факторов успеха такой защиты является то, что стандартная библиотека C++ обеспечивает гарантированную сложность операций и алгоритмов над контейнерами STL.

Здесь и надо искать точку равновесия. Очевидно, что неверно прибегать к преждевременной оптимизации путем использования менее понятных алгоритмов в ожидании больших объемов данных, которые могут никогда не материализоваться. Не менее очевидно и то, что неверно прибегать и к преждевременной пессимизации, закрывая глаза на сложность алгоритмов (*O*-сложность), а именно — стоимость вычислений как функцию от количества элементов данных, с которыми работает алгоритм.

Данный совет состоит из двух частей. Во-первых, даже до того, как станет известно, будут ли объемы данных достаточно велики, чтобы для конкретных вычислений возникла проблема, по умолчанию следует избегать использования алгоритмов, которые работают с пользовательскими данными (которые могут расти), но не способны к масштабированию, если только использование менее масштабируемого алгоритма не приводит к существенному повышению понятности и удобочитаемости кода (см. рекомендацию 6). Но все мы часто сталкиваемся с сюрпризами. Мы пишем десять фрагментов кода, думая, что они никогда не будут иметь дела с большими наборами данных. И это действительно оказывается так — в девяти случаях из десяти. В десятом случае мы сталкиваемся с проблемами производительности. Это не раз случалось с нами, и мы знаем, что это случалось (или случится) и с вами. Конечно, мы вносили исправления и передавали их потребителям, но лучше было бы избежать таких затруднений и выполнения лишней работы. Так что при прочих равных условиях (включая понятность и удобочитаемость) воспользуйтесь следующими советами.

- *Используйте гибкие динамически распределяемые данные вместо массивов фиксированного размера.* Массив "больший, чем наибольший массив, который мне когда-либо потребуется" приводит к ошибкам и нарушению безопасности (см. рекомендацию 77). Массивы можно использовать только тогда, когда размеры данных фиксированы и известны во время компиляции.
- *Следует точно знать сложность используемого алгоритма.* Не забывайте о такой ловушке, как линейный алгоритм, который вызывает другую линейную операцию, что в результате делает алгоритм квадратичным (см., например, рекомендацию 81).
- *По возможности используйте линейные или более быстрые алгоритмы.* Идеальны алгоритмы с константной сложностью, такие как `push_back` или поиск в хэш-таблице (см. рекомендации 76 и 80). Неплохи алгоритмы со сложностью $O(\log N)$, такие как операции с контейнерами `set`/`map` и `lower_bound` или `upper_bound` с итераторами произвольного доступа (см. рекомендации 76, 85 и 86). Допустима линейная сложность $O(N)$, как, например, у `vector::insert` или `for_each` (см. рекомендации 76, 81 и 84).
- *Пытайтесь избежать применения алгоритмов с более чем линейной сложностью, где это возможно.* Например, по умолчанию следует затратить определенные усилия на поиск замены имеющегося алгоритма со сложностью $O(N \log N)$ или $O(N^2)$ (если таковая возможна), чтобы избежать непропорционального падения производительности при существенном увеличении объема данных. Так, именно в этом заключается основная причина, по которой в рекомендации 81 советуется предпочитать операции с диапазонами (которые обычно линейны) их копиям для работы с отдельными элементами (которые обычно квадратичны, так как одна линейная операция вызывает другую линейную операцию; см. пример 1 в рекомендации 81).
- *Никогда не используйте экспоненциальный алгоритм, если только вы не "приперты к стене" и не имеете другого выхода.* Ищите, не жалея сил, альтернативу, прежде чем прибегнуть к экспоненциальному алгоритму, где даже небольшое увеличение данных приводит к существенному падению производительности.

Во-вторых, после того как замеры покажут, что оптимизация действительно нужна и важна, в особенности при росте данных, сконцентрируйте усилия на снижении O-сложности, а не на микрооптимизациях наподобие экономии одного сложения.

Итак, предпочтительно использовать линейные (или лучшие) алгоритмы там, где только это возможно. Избегайте, где можете, алгоритмов с более чем линейной сложностью, и уж тем более — экспоненциальных.

Ссылки

[Bentley00] §6, §8, Appendix 4 • [Cormen01] • [Kernighan99] §7 • [Knuth97a] • [Knuth97b] • [Knuth98] • [McConnell93] §5.1-4, §10.6 • [Murray93] §9.11 • [Sedgewick98] • [Stroustrup00] §17.1.2

8. Не оптимизируйте преждевременно

Резюме

Как гласит пословица, не подгоняйте скачущую лошадь. Преждевременная оптимизация непродуктивна и быстро входит в привычку. Первое правило оптимизации: не оптимизируйте. Второе правило оптимизации (только для экспертов): не оптимизируйте ни в коем случае. Семь раз отмерь, один раз оптимизируй.

Обсуждение

В [Stroustrup00] §6 имеется замечательная цитата:

Преждевременная оптимизация — корень всех бед.

— Дональд Кнут (Donald Knuth) [цитирует Хоара (Hoare)]

С другой стороны, мы не можем игнорировать эффективность.

— Йон Бентли (Jon Bentley)

Хоар и Кнут совершенно правы (см. рекомендацию 6 и эту). Но прав и Бентли (рекомендация 9).

Мы определяем преждевременную оптимизацию как усложнение дизайна или кода (что делает его менее удобочитаемым) во имя повышения производительности, когда усилия не оправдываются доказанной необходимостью повышения производительности (например, реальными измерениями и сравнением с поставленной целью). Зачастую такие усилия вообще не приводят к повышению производительности программы.

Всегда помните:

**Гораздо, гораздо проще сделать корректную программу быстрой,
чем быструю — корректной.**

Поэтому по умолчанию не концентрируйтесь на том, чтобы сделать код быстрым, в первую очередь его надо сделать максимально понятным и удобочитаемым (рекомендация 6). Ясный код проще написать корректно, проще понять, проще переделать — и проще оптимизировать. Усложнения, включая оптимизацию, всегда можно внести позже — и только при необходимости.

Имеются две основные причины, почему преждевременная оптимизация зачастую не делает программу быстрее. Во-первых, общеизвестно, что программисты обычно плохо представляют, какой код будет быстрее или меньше по размеру, и где будет самое узкое место в разрабатываемом коде. В число таких программистов входят и авторы этой книги, и вы. Подумайте сами — современные компьютеры представляют собой исключительно сложные вычислительные модели, зачастую с несколькими работающими параллельно процессорами, глубокой иерархией кэширования, предсказанием ветвления, конвейеризацией и многимимногим другим. Компилятор, находящийся над всем этим аппаратным обеспечением, преобразует ваш исходный код в машинный, основываясь на собственном знании аппаратного обеспечения и его особенностей, с тем чтобы этот код в максимальной степени использовал все возможности аппаратного обеспечения. Над компилятором находитесь вы, с вашими представлениями о том, как должен работать тот или иной код. У вас практически нет шансов внести такую микрооптимизацию, которая в состоянии существенно повысить производительность генерируемого интеллектуальным компилятором кода. Итак, оптимизации должны предшествовать измерения, а измерениям должна предшествовать выработка целей

оптимизации. Пока необходимость оптимизации не доказана — вашим приоритетом №1 должно быть написание кода для человека. (Если кто-то потребует от вас оптимизации кода — *потребуйте доказательств* необходимости этого.)

Во-вторых, в современных программах все больше и больше операций, скорость работы которых ограничена не процессором, а, например, работой с памятью, диском или сетью, ожиданием ответа от Web-сервиса или базы данных. В лучшем случае оптимизация такого кода приведет к тому, что ваша программа будет быстрее ожидать. Это также означает, что программист зря тратит драгоценное время на улучшение того, что не требует улучшений, вместо того, чтобы заняться тем, что действительно требует его вмешательства.

Само собой разумеется, настанет день, когда вам действительно придется заняться оптимизацией вашего кода. Когда вы займетесь этим — начните с оптимизации алгоритмов (рекомендация 7) и попытайтесь инкапсулировать оптимизацию (например, в пределах функции или класса, см. рекомендации 5 и 11), четко указав в комментариях причину проводимой оптимизации и ссылку на использованный алгоритм.

Обычная ошибка новичка состоит в том, что когда он пишет новый код, то — с гордостью! — старается сделать его оптимальным ценой понятности. Чаще всего это приводит к милям "спагетти" (говоря проще — к "соплям" в программе), и даже корректно работающий код становится очень трудно читать и модифицировать (см. рекомендацию 6).

Передача параметров по ссылке (рекомендация 25), использование префиксной формы операторов ++ и -- (рекомендация 28) или подобных идиом, которые при работе должны естественным образом "стекать с кончиков ваших пальцев", преждевременной оптимизацией не являются. Это всего лишь устранение преждевременной пессимизации (рекомендация 9).

Примеры

Пример. Ирония использования `inline`. Это простейшая демонстрация скрытой стоимости преждевременной микрооптимизации. Профайлеры легко могут сказать вам (исследовав количество вызовов функций), какие функции должны быть встраиваемыми, но не являются таковыми. Но те же профайлеры не в состоянии подсказать, какие встраиваемые функции не должны быть таковыми. Очень многие программисты используют "встраивание по умолчанию" во имя оптимизации, почти всегда за счет большей связности программы достигая в лучшем случае весьма сомнительных результатов. (Сказанное означает, что делать функцию встраиваемой должен компилятор, а не программист. См. [Sutter00], [Sutter02] и [Sutter04].)

Исключения

Когда вы пишете библиотеки, трудно предсказать, какие операции будут использоваться в критичном по отношению к производительности коде. Но даже автор библиотеки должен испытать свой код на производительность в разнообразных пользовательских приложениях перед тем, как усложнять свой код оптимизацией.

Ссылки

[Bentley00] §6 • [Cline99] §13.01-09 • [Kernighan99] §7 • [Lakos96] §9.1.14 • [Meyers97] §33 • [Murray93] §9.9-10, §9.13 • [Stroustrup00] §6 introduction • [Sutter00] §30, §46 • [Sutter02] §12 • [Sutter04] §25

9. Не пессимизируйте преждевременно

Резюме

То, что просто для вас, — просто и для кода. При прочих равных условиях, в особенности — сложности и удобочитаемости кода, ряд эффективных шаблонов проектирования и идиом кодирования должны естественным образом "стекать с кончиков ваших пальцев" и быть не сложнее в написании, чем их пессимизированные альтернативы. Это не преждевременная оптимизация, а избежание излишней пессимизации.

Обсуждение

Избежание преждевременной оптимизации не влечет за собой снижения эффективности. Под преждевременной пессимизацией мы подразумеваем написание таких неоправданных потенциально неэффективных вещей, как перечисленные ниже.

- Передача параметров по значению там, где применима передача параметров по ссылке (рекомендация 25).
- Использование постфиксной версии ++ там, где с тем же успехом можно воспользоваться префиксной версией (рекомендация 28).
- Использование присваивания в конструкторах вместо списка инициализации (рекомендация 48).

Не является преждевременной оптимизацией снижение количества фиктивных временных копий объектов, в особенности во внутренних циклах, если это не приводит к усложнению кода. В рекомендации 18 поощряется максимально локальное объявление переменных, но там же приведено и описание исключения — возможный вынос переменных из цикла. В большинстве случаев такое действие не усложняет понимание предназначения кода, более того, может помочь пояснить, что именно делается в цикле и какие вычисления являются его инвариантами. Конечно же, предпочтительно использовать готовые алгоритмы вместо написания явных циклов (рекомендация 84).

Два важных способа усовершенствования программы, которые делают ее одновременно и яснее, и эффективнее — это использование абстракций (см. рекомендации 11 и 36) и библиотек (рекомендация 84). Например, использование `vector`, `list`, `map`, `find`, `sort` и других возможностей стандартной библиотеки, стандартизированных и реализованных экспертами мирового класса, не только делают ваш код яснее и легче понимаемым, но зачастую и более быстрым.

Избежание преждевременной пессимизации становится особенно важным, когда вы пишете библиотеку. При этом вы обычно не знаете контекста использования вашей библиотеки, а поэтому должны суметь сбалансировать эффективность и возможность повторного использования. Не забывайте уроков рекомендации 7 — следует куда больше заботиться о масштабируемости, чем о выигрыше пары тактов процессора.

Ссылки

[Keffer95] pp.12-13 • [Stroustrup00] §6 introduction • [Sutter00] §6

10. Минимизируйте глобальные и совместно используемые данные

Резюме

Совместное использование вызывает споры и раздоры. Избегайте совместного использования данных, в особенности глобальных данных. Совместно используемые данные усиливают связность, что приводит к снижению сопровождаемости, а зачастую и производительности.

Обсуждение

Это утверждение носит более общий характер, чем более узкое требование рекомендации 18.

Следует избегать данных со внешним связыванием в области видимости пространства имен или представляющих собой статические члены классов. Их применение усложняет логику программы и приводит к тесной связи между различными (и, что еще хуже, отдаленными) частями программы. Совместно используемые данные делают менее эффективным тестирование модулей, поскольку корректность фрагмента кода, использующего общие данные, обусловлена историей изменения этих данных, а кроме того, обуславливает функционирование некоторого, пока неизвестного, кода, который будет использовать эти данные позже.

Имена объектов в глобальном пространстве имен приводят к его дополнительному засорению.

Если вам никак не обойтись без глобальных объектов, объектов в области видимости пространства имен или статических членов классов, убедитесь в их корректной инициализации. Порядок инициализации таких объектов из разных единиц компиляции не определен, поэтому для корректной работы в таком случае требуются специальные методики (см. прилагаемые ссылки). Правила порядка инициализации достаточно сложны, поэтому лучше их избегать; но если вы все же вынуждены иметь с ними дело, то должны хорошо их изучить и использовать с величайшей осторожностью.

Объекты, находящиеся в области видимости пространств имен, статические члены или совместно используемые разными потоками или процессами, снижают уровень распараллеливания в многопоточных и многопроцессорных средах, и часто являются узким местом с точки зрения производительности и масштабируемости (см. рекомендацию 7). Старайтесь избавиться от совместного использования данных; используйте вместо него средства коммуникации (например, очередь сообщений).

Предпочтительно обеспечить низкую связность и минимизировать взаимодействие классов (см. [Cargill92]).

Исключения

Такие средства уровня программы, как `cin`, `cout` и `cerr`, являются специализированными и реализуются специальным образом. Фабрика должна поддерживать реестр функций, которые должны вызываться для создания данного типа, и такой реестр обычно один на всю программу (тем не менее предпочтительно, чтобы он был внутренним объектом по отношению к фабрике, а не глобальным совместно используемым объектом; см. рекомендацию 11).

Код, в котором объект совместно используется разными потоками, должен обеспечить сериализацию обращений к такому объекту (см. рекомендацию 12 и [Sutter04c]).

Ссылки

[Cargill92] pp. 126.136, 169-173 • [Dewhurst03] §3 • [Lakos96] §2.3.1 • [McConnell93] §5.1-4 • [Stroustrup00] §C.10.1 • [Sutter00] §47 • [Sutter02] §16, Appendix A • [Sutter04c] • [SuttHysl03]

11. Сокрытие информации

Резюме

Не выпускайте внутреннюю информацию за пределы объекта, обеспечивающего абстракцию.

Обсуждение

Для минимизации зависимостей между вызывающим кодом, который работает с абстракцией, и реализацией абстракции, внутренние данные такой реализации должны быть скрыты. В противном случае вызывающий код может обратиться к этой информации (или, что еще хуже, изменить ее), и такая утечка предназначенной сугубо для внутреннего использования информация делает вызывающий код зависимым от внутреннего представления абстракции. Открывать следует абстракции (предпочтительно из соответствующей предметной области, но, как минимум, абстракцию `get/set`), а не данные.

Сокрытие информации уменьшает стоимость проекта, сроки разработки и/или риск двумя основными путями.

- *Оно локализует изменения.* Сокрытие информации снижает область "эффекта волны" вносимого изменения и, соответственно, его стоимость.
- *Оно усиливает инварианты.* Сокрытие информации ограничивает код, ответственный за сохранение (и, в случае ошибки, за нарушение) инвариантов программы (см. рекомендацию 41).

Не позволяйте "засветиться" данным из любого объекта, который обеспечивает абстракцию (см. также рекомендацию 10). Данные — всего лишь одно из воплощений абстракции, концептуальное состояние. Если вы сконцентрируетесь на концепции, а не на ее представлениях, вы сможете предложить вдумчивый интерфейс, а "вылизать" реализацию можно и позже — например, путем применения кэширования вместо вычисления "на лету", или использования различных оптимизирующих представлений (например, полярных координат вместо декартовых).

Распространенный пример состоит в том, что члены-данные классов никогда не делаются доступными извне при помощи спецификатора `public` (см. рекомендацию 41) или посредством указателей или дескрипторов (см. рекомендацию 42). Этот принцип в той же степени применим и к большим сущностям — например, таким, как библиотеки, которые также не должны разрешать доступ к внутренней информации извне. Модули и библиотеки должны предоставлять интерфейсы, которые определяют абстракции и работу с ними, и таким образом обеспечивают большую безопасность для вызывающего кода и меньшую связность, чем при применении совместно используемых данных.

Исключения

Тестирование кода зачастую требует возможности обращения ко "внутренностям" тестируемого класса или модуля.

Совокупности значений (`struct` в стиле C), которые представляют собой просто набор данных без предоставления каких-либо абстракций, не требуют сокрытия своих данных, которые в этом случае являются интерфейсом (см. рекомендацию 41).

Ссылки

[Brooks95] §19 • [McConnell93] §6.2 • [Parnas02] • [Stroustrup00] §24.4 • [SuttHysl04a]

12. Кодирование параллельных вычислений

Резюме

Если ваше приложение использует несколько потоков или процессов, следует минимизировать количество совместно используемых объектов, где это только можно (см. рекомендацию 10), и аккуратно работать с оставшимися.

Обсуждение

Работа с потоками — отдельная большая тема. Данная рекомендация оказалась в книге, потому что эта тема очень важна и требует рассмотрения. К сожалению, одна рекомендация не в силах сделать это полно и корректно, поэтому мы только резюмируем несколько наиболее важных положений и посоветуем обратиться к указанным ниже ссылкам за дополнительной информацией. Среди наиболее важных вопросов, касающихся параллельных вычислений, такие как избежание взаимоблокировок (deadlock), неустойчивых взаимоблокировок (livelock) и условий гонки (race conditions).

Стандарт C++ ничего не говорит о потоках. Тем не менее, C++ постоянно и широко применяется для написания кода с интенсивным использованием многопоточности. Если в вашем приложении потоки совместно используют данные, на это следует обратить особое внимание.

- *Ознакомьтесь с документацией по целевой платформе на предмет локальных примитивов синхронизации.* Обычно они охватывают диапазон от простых атомарных операций с целыми числами до межпроцессных взаимоисключений и семафоров.
- *Предпочтительно "обернуть" примитивы платформы в собственные абстракции.* Это хорошая мысль, в особенности если вам требуется межплатформенная переносимость. Вы можете также воспользоваться библиотекой (например, pthreads [Butenhof97]), которая сделает это за вас.
- *Убедитесь, что используемые вами типы можно безопасно применять в многопоточных программах.* В частности, каждый тип должен как минимум
  - *гарантировать независимость объектов, которые не используются совместно.* Два потока могут свободно использовать различные объекты без каких-либо специальных действий со стороны вызывающего кода;
  - *документировать необходимые действия со стороны вызывающего кода при использовании одного объекта в разных потоках.* Многие типы требуют сериализации доступа к таким совместно используемым объектам, но есть типы, для которых это условие не является обязательным. Обычно такие типы либо спроектированы таким образом, что избегают требований блокировки, либо выполняют блокировку самостоятельно. В любом случае, вы должны быть ознакомлены с ограничениями, накладываемыми используемыми вами типами.

  Заметим, что сказанное выше применимо независимо от того, является ли тип некоторым строковым типом, контейнером STL наподобие `vector` или некоторым иным типом. (Мы заметили, что ряд авторов дают советы, из которых вытекает, что стандартные контейнеры представляют собой нечто отдельное. Это не так — контейнер представляет собой просто объект другого типа.) В частности, если вы хотите использовать компоненты стандартной библиотеки (например, `string` или контейнеры) в многопоточной программе, проконсультируйтесь сначала с документацией разработчика используемой вами стандартной библиотеки, чтобы узнать, как именно следует пользоваться ею в многопоточном приложении.

При разработке собственного типа, который предназначен для использования в многопоточной программе, вы должны сделать те же две вещи. Во-первых, вы должны гарантировать, что различные потоки могут использовать различные объекты вашего типа без использования блокировок (заметим, что обычно тип с изменяемыми статическими данными не в состоянии обеспечить такую гарантию). Во-вторых, вы должны документировать, что именно должны сделать пользователи для того, чтобы безопасно использовать один и тот же объект в разных потоках. Фундаментальный вопрос проектирования заключается в том, как распределить ответственность за корректное выполнение программы (без условий гонки и взаимоблокировок) между классом и его клиентом. Вот основные возможности.

- *Внешняя блокировка. За блокировку отвечает вызывающий код.* При таком выборе код, который использует объект, должен сам выяснять, используется ли этот объект другими потоками и, если это так, отвечает за сериализацию его использования. Например, строковые типы обычно используют внешнюю блокировку (или неизменяемость — см. третью возможность).
- *Внутренняя блокировка. Каждый объект сериализует все обращения к себе, обычно путем соответствующего блокирования всех открытых функций-членов, так что пользователю не надо предпринимать никаких дополнительных действий по сериализации использования объекта.* Например, очереди производитель/потребитель обычно используют внутреннюю блокировку, поскольку сам смысл их существования состоит в совместном использовании разными потоками, и их интерфейсы спроектированы с использованием блокировок соответствующего уровня для отдельных вызовов функций-членов (`Push`, `Pop`). В общем случае заметим, что этот вариант применим, только если вы знаете две вещи.

  Во-первых, вы должны заранее знать о том, что объекты данного типа практически всегда будут совместно использоваться разными потоками; в противном случае вы просто разработаете бесполезную блокировку. Заметим, что большинство типов не удовлетворяют этому условию; подавляющее большинство объектов даже в программах с интенсивным использованием многопоточности не разделяются разными потоками (и это хорошо — см. рекомендацию 10).

  Во-вторых, вы должны заранее быть уверены, что блокировка на уровне функций обеспечивает корректный уровень модульности, которого достаточно для большинства вызывающий функций. В частности, интерфейс типа должен быть спроектирован в пользу самодостаточных операций с невысокой степенью детализации. Если вызывающий код должен блокировать несколько операций, а *не одну*, то такой способ неприменим. В этом случае отдельные функции могут быть собраны в блокируемый модуль большего масштаба, работа с которым выполняется при помощи дополнительной (внешней) блокировки. Например, рассмотрим тип, который возвращает итератор, который может стать недействительным перед тем, как вы используете его, или предоставляет алгоритм наподобие `find`, возвращающий верный ответ, который становится неверным до того, как вы им воспользуетесь, или пользователь напишет код `if(c.empty()) c.push_back(x);` (другие примеры можно найти в [Sutter02]). В таких случаях вызывающая функция должна выполнить внешнюю блокировку на время выполнения всех отдельных вызовов функций-членов, так что отдельные блокировки для каждой функции-члена оказываются ненужной расточительностью.

  Итак, внутренняя блокировка связана с открытым интерфейсом типа. Она становится применима только тогда, когда отдельные операции типа являются сами по себе завершенными; другими словами, когда уровень абстракции типа растет и выражается и инкапсулируется более точно (например, как у очереди производитель/потребитель

по отношению к обычному контейнеру `vector`). Объединение примитивных операций для образования более крупных общих операций — этот подход требуется для того, чтобы обеспечить возможность простого вызова функции с большим внутренним содержанием. В ситуациях, когда комбинирование примитивов может быть произвольным и вы не можете определить разумный набор сценариев использования в виде одной именованной операции, имеются две альтернативы. Можно воспользоваться моделью функций обратного вызова (т.е. вызывающая функция должна вызвать одну функцию-член, передавая ей задание, которое следует выполнить, в виде команды или объекта-функции; см. рекомендации с 87 по 89). Второй метод состоит в некотором способе предоставления вызывающему коду возможности блокировки в открытом интерфейсе.

- *Проектирование, не требующее блокировок, включая неизменяемость (объекты, предназначенные только для чтения).* Можно разработать типы, для которых блокировка окажется полностью ненужной (см. ссылки). Одним из распространенных примеров являются неизменяемые объекты, которые не требуют блокировки, поскольку они никогда не изменяются. Например, будучи создан, объект неизменяемого строкового типа больше не модифицируется, а все строковые операции приводят к созданию новых строк.

Заметим, что вызывающий код ничего не должен знать о деталях реализации ваших типов (см. рекомендацию 11). Если ваш тип внутренне использует какие-то методики разделения данных (например, копирование при записи), вы не должны нести ответственность за все возможные вопросы безопасности потоков, но обязаны обеспечить корректность работы вызывающего кода при обычной работе — т.е. тип должен быть безопасен в плане многопоточности в той же мере, как если бы он не использовал методики совместного использования данных (см. [Sutter04c]). Как упоминалось, все корректно написанные типы должны позволять работу с различными объектами в разных потоках без синхронизации.

В частности, если вы разрабатываете библиотеку, предназначенную для широкого использования, вы должны предусмотреть безопасность ваших объектов в многопоточных программах, как описано выше, но при этом без дополнительных накладных расходов при работе в однопоточной программе. Например, если вы пишете библиотеку, содержащую тип, использующий копирование при записи, и вы должны обеспечить, как минимум, некоторую внутреннюю блокировку, то предпочтительно разработать ее так, чтобы в однопоточном варианте вашей библиотеки ее не было (обычно для этого используются директивы препроцессора `#ifdef`).

Если используется несколько блокировок, то избежать взаимоблокировки можно путем их запроса в одинаковом порядке (освобождение блокировок может выполняться в любом порядке). Одно из решений состоит в запросе блокировок в порядке возрастания адресов в памяти, что обеспечивает удобное, однозначное упорядочение в пределах приложения.

Ссылки

[Alexandrescu02a] • [Alexandrescu04] • [Butenhof97] • [Henney00] • [Henney01] • [Meyers04] • [Schmidt01] • [Stroustrup00] §14.9 • [Sutter02] §16 • [Sutter04c]

13. Ресурсы должны быть во владении объектов

Резюме

Не работайте вручную, если у вас есть мощные инструменты. Идиома C++ "выделение ресурса есть инициализация" (resource acquisition is initialization — RAII) представляет собой мощный инструмент для корректной работы с ресурсами. RAII позволяет компилятору автоматически обеспечить строгую гарантию того, что в других языках надо делать вручную. При выделении ресурса передайте его объекту-владельцу. Никогда не выделяйте несколько ресурсов в одной инструкции.

Обсуждение

Симметрия конструктор/деструктор, обеспечиваемая языком C++, воспроизводит симметрию, присущую парам функций захвата/освобождения ресурса, таким как `fopen/fclose`, `lock/unlock` и `new/delete`. Это делает стековые объекты (или объекты со счетчиком ссылок), в конструкторе которых происходит захват ресурса (а в деструкторе его освобождение), превосходным инструментом для автоматизации управления ресурсами.

Автоматизация легко реализуема, элегантна, недорога и по сути безопасна в плане ошибок. Если вы не будете ею пользоваться, то обречете себя на нетривиальную и кропотливую ручную работу по "спариванию" вызовов захвата и освобождения ресурсов, включающую отслеживание всех ветвлений и исключений. Это совершенно неприемлемый путь для C++, который предоставляет возможность автоматизации этой работы при помощи простой в использовании идиомы RAII.

Когда вы имеете дело с ресурсом, который требует спаривания вызовов функций захвата/освобождения, инкапсулируйте этот ресурс в объект, который выполнит эту работу за вас и освободит ресурс в своем деструкторе. Например, вместо непосредственного вызова пары функций (не членов) `OpenPort/ClosePort` можно поступить иначе:

```
class Port {
public:
    Port( const string& destination ); // Вызов OpenPort
    ~Port();                           // Вызов ClosePort
    // Порты обычно не клонируются, так что запрещаем
    // копирование и присваивание
};

void DoSomething() {
    Port port1( "server1:80" );
    // ...
}   // Забыть закрыть порт нельзя – он будет закрыт
    // автоматически при выходе из области видимости

shared_ptr<Port> port2 = /*...*/; // port2 будет закрыт
    // автоматически, когда будет уничтожен последний
    // ссылающийся на него объект shared_ptr
```

Вы можете также использовать библиотеки, которые реализуют соответствующий шаблон проектирования (см. [Alexandrescu00c]).

При реализации идиомы RAII следует особо тщательно подходить к вопросу о копирующем конструкторе и присваивании (см. рекомендацию 49): обычно генерируемые компилятором версии этих функций не подходят. Если копирование лишено смысла, копирующий конструктор и оператор присваивания можно явным образом запретить, делая их закрытыми

членами и не определяя (см. рекомендацию 53). В противном случае копирующий конструктор дублирует ресурс или использует счетчик ссылок на него, и то же делает и оператор присваивания, при необходимости освободив ресурс, которым объект владел до присваивания. Классической ошибкой является освобождение старого ресурса до того, как успешно дублирован новый (см. рекомендацию 71).

Обеспечьте, чтобы все ресурсы принадлежали объектам. Предпочтительно хранить все динамически выделенные ресурсы посредством интеллектуальных, а не обычных, указателей. Кроме того, следует выполнять каждое явное выделение ресурса (например, `new`) в отдельной инструкции, которая тут же передает ресурс управляющему объекту (например, `shared_ptr`). В противном случае может возникнуть утечка ресурсов, связанная с тем, что порядок вычисления параметров функции не определен (см. рекомендацию 31). Например:

```
void Fun(shared_ptr<Widget> sp1, shared_ptr<Widget> sp2);
// ...
Fun(shared_ptr<Widget>(new Widget),
    shared_ptr<Widget>(new Widget));
```

Такой код небезопасен. Стандарт C++ предоставляет компилятору большую свободу действий по переупорядочению выражений, которые создают два аргумента функции. В частности, компилятор может чередовать выполнение этих двух выражений: сначала для обоих объектов может быть выполнено выделение памяти (при помощи оператора `new`), а уже затем будут вызваны два конструктора `Widget`. Такая последовательность действий может привести к утечке: если один из конструкторов сгенерирует исключение, то память для *другого* объекта никогда не будет освобождена (более детальную информацию по этому вопросу вы найдете в [Sutter02]).

Эта тонкая проблема имеет простое решение: следуйте приведенному выше совету и никогда не выделяйте в одной инструкции больше одного ресурса. Следует выполнять каждое явное выделение ресурса (например, `new`) в отдельной инструкции, которая тут же передает ресурс управляющему объекту (например, `shared_ptr`), например:

```
shared_ptr<Widget> sp1(new Widget), sp2(new Widget);
Fun( sp1, sp2 );
```

См. также описание дополнительных преимуществ такого стиля в рекомендации 31.

Исключения

Можно чересчур увлечься интеллектуальными указателями. Обычные указатели вполне подходят для кода, в котором указываемый объект виден только в ограниченном объеме (например, внутри класса — типа указателя на узел дерева в классе `Tree`, использующийся для навигации по дереву).

Ссылки

[Alexandrescu00c] • [Cline99] §31.03-05 • [Dewhurst03] §24, §67 • [Meyers96] §9-10 • [Milewski01] • [Stroustrup00] §14.3-4, §25.7, §E.3, §E.6 • [Sutter00] §16 • [Sutter02] §20-21 • [Vandevoorde03] §20.1.4

Стиль кодирования

Константа для одного является переменной для другого.
— Алан Перлис (Alan Perlis)

В этом разделе мы перейдем от вопросов проектирования к вопросам, которые появляются в основном при реальном кодировании.

Правила и рекомендации из этого раздела применимы безотносительно к конкретной области языка программирования (например, функциям, классам или пространствам имен), но приводят к повышению качества вашего кода. Многие из представленных идиом позволяют вашему компилятору активнее помогать вам в работе, а вам — избежать опасных мест (включая неопределенное поведение), которые компилятор не всегда в состоянии выявить. Все это делает ваш код более надежным.

В этом разделе мы считаем наиболее важной рекомендацию 14: "Предпочитайте ошибки компиляции и компоновки ошибкам времени выполнения".

14. Предпочитайте ошибки компиляции и компоновки ошибкам времени выполнения

Резюме

Не стоит откладывать до выполнения программы выявление ошибок, которые можно обнаружить при ее сборке. Предпочтительно писать код, который использует компилятор для проверки инвариантов в процессе компиляции, вместо того, чтобы проверять их во время работы программы. Проверки времени выполнения зависят от выполнимого кода и данных, так что вы только изредка можете полностью полагаться на них. Проверки времени компиляции, напротив, не зависят от данных и предыстории исполнения, что обычно обеспечивает более высокую степень надежности.

Обсуждение

Язык C++ предлагает массу средств для "ускорения" обнаружения ошибок во время компиляции. Использование этих возможностей статических проверок дает массу преимуществ, включая следующие.

- *Статические проверки не зависят от данных и логики программы.* Статические проверки гарантируют независимость от входных данных программы или потока ее выполнения. В противоположность этому, чтобы убедиться в достаточной строгости тестирования времени выполнения, вы должны проверить его на представительном наборе входных данных. Это достаточно большая и неприятная работа для всех нетривиальных систем.
- *Статически выраженные модели более строги.* Зачастую то, что программа полагается на проверки времени компиляции, а не времени выполнения, отражает лучший дизайн, поскольку модель, создаваемая программой, корректно выражена с использованием системы типов C++. Таким образом, вы и компилятор оказываетесь партнерами с общим взглядом на инварианты программ. Зачастую проверки времени выполнения приходится использовать там, где теоретически проверку можно было бы провести статически, но сделать это невозможно из-за ограничений языка программирования (см. рекомендацию 68).
- *Статические проверки не приводят к накладным расходам времени выполнения.* При замене динамических проверок статическими создаваемая выполнимая программа оказывается быстрее, оставаясь столь же корректной.

Один из наиболее мощных инструментов статических проверок в C++ — статическая проверка типов. Споры о том, должны ли типы проверяться статически (C++, Java, ML, Haskell) или динамически (Smalltalk, Ruby, Python, Lisp), все еще активно продолжаются. В общем случае нет явного победителя, и имеются языки и стили разработки, которые дают хорошие результаты как в одном, так и во втором случае. Сторонники статической проверки аргументируют свою позицию тем, что обработка большого класса ошибок времени выполнения может быть просто устранена, что дает более надежную и качественную программу. Поклонники динамических проверок говорят, что компиляторы способны выявить только часть потенциальных ошибок, так что если вы все равно должны писать тесты для ваших модулей, вы можете вообще не волноваться о статических проверках, получив при этом менее ограничивающую среду программирования.

Понятно одно: в контексте статически типизированного языка C++, обеспечивающего строгую проверку типов и минимальную автоматическую проверку времени выполнения, программисты определенно должны использовать систему типов для своей пользы везде, где только это возможно (см. рекомендации с 90 по 100). В то же время тестирование времени выполнения целесообразно для выполнения проверок, зависящих от данных и потока выполнения программы (например, проверка выхода за границы массива или корректности входных данных) (см. рекомендации 70 и 71).

Примеры

Имеется ряд примеров, где вы можете заменить проверки времени выполнения проверками времени компиляции.

Пример 1. Логические условия времени компиляции. Если вы проверяете логическое условие времени компиляции наподобие `sizeof(int) >= 8`, используйте статические проверки (обратите также внимание на рекомендацию 91).

Пример 2. Полиморфизм времени компиляции. Подумайте о замене полиморфизма времени выполнения (виртуальные функции) полиморфизмом времени компиляции (шаблоны) при определении обобщенных функций или типов. Последний приводит к коду с лучшей статической проверкой (см. также рекомендацию 64).

Пример 3. Перечисления. Подумайте об определении перечислений (или, что еще лучше, полностью законченных типов), когда вам требуется выразить символьные константы или ограниченные целочисленные значения.

Пример 4. Понижающее преобразование типов. Если вы часто используете оператор `dynamic_cast` (или, что еще хуже, непроверяемый `static_cast`) для понижающего преобразования типов, возможно, ваш базовый класс предоставляет слишком малую функциональность? Подумайте над перепроектированием ваших интерфейсов таким образом, чтобы ваша программа могла выразить необходимые вычисления посредством базового класса.

Исключения

Некоторые условия не могут быть проверены в процессе компиляции и требуют проверки времени выполнения. В таком случае для обнаружения внутренних программных ошибок следует использовать `assert` (см. рекомендацию 68) и следовать советам из остальной части раздела, посвященного обработке ошибок, для прочих ошибок времени выполнения, таких как ошибки, зависящие от данных (см. рекомендации с 69 по 75).

Ссылки

[Alexandrescu01] §3 • [Boost] • [Meyers97] §46 • [Stroustrup00] §2.4.2 • [Sutter02] §4 • [Sutter04] §2, §19

15. Активно используйте const

Резюме

const — ваш друг: неизменяемые значения проще понимать, отслеживать и мотивировать, т.е. там, где это целесообразно, лучше использовать константы вместо переменных. Сделайте const описанием по умолчанию при определении значения — это безопасно, проверяемо во время компиляции (см. рекомендацию 14) и интегрируемо с системой типов C++. Не выполняйте преобразований типов с отбрасыванием const кроме как при вызове некорректной с точки зрения употребления const функции (см. рекомендацию 94).

Обсуждение

Константы упрощают код, поскольку вам достаточно только один раз взглянуть на ее определение, чтобы знать, чему она равна везде. Рассмотрим такой код:

```
void Fun( vector<int>& v ) {
    // ...
    const size_t len = v.size();
    // ... и еще 30 строк ...
}
```

Увидев такое определение len, вы получаете надежную информацию о семантике этой константы в пределах области ее видимости (в предположении, что код не устраняет ее константность, чего он делать не должен, как вы узнаете далее): это информация о длине v в определенной точке программы. Взглянув на одну строку, вы получили всю необходимую информацию для всей области видимости. Если бы переменная len не была определена как const, она могла бы быть позже изменена — непосредственно или косвенно.

Заметим, что описание const не является глубоким. Чтобы понять что имеется в виду, рассмотрим класс C, который имеет член типа X*. В объекте C, который является константой, член X* также является константой, но сам объект X, на который он указывает, константой не является (см. [Saks99]).

Логическую константность следует реализовывать с использованием членов, описанных как mutable. Когда константная функция-член класса оправданно требует модификации переменной-члена (т.е. когда эта переменная не влияет на наблюдаемое состояние объекта, например, если это кэшированные данные), объявите эту переменную-член как mutable. Заметим, что если все закрытые члены скрыты с использованием идиомы Pimpl (см. рекомендацию 43), описание mutable не является необходимым ни для кэшированной информации, ни для неизменного указателя на нее.

Модификатор const напоминает вирусное заболевание — появившись в вашем коде один раз, он приведет к необходимости соответствующего изменения сигнатур функций, которые еще не являются корректными в плане использования const. Это как раз не ошибка, а хорошее свойство, существенно увеличивающее мощь модификатора const, который еще не так давно был достаточно заброшен, а его возможности не вполне поняты и оценены. Переделка существующего кода для его корректности в плане использования const требует усилий, но они стоят того и даже позволяют выявить скрытые ошибки.

Корректное применение const дает отличные результаты и повышает эффективность. Чрезвычайно важно правильно и последовательно использовать модификатор const в ваших программах. Понимание того, как и где изменяется состояние программы, особенно необходимо, а модификатор const по сути документирует непосредственно в коде программы, где именно компилятор может помочь вам в этом. Правильное употребление const поможет вам лучше разобраться с вопросами проектирования и сделать ваш код более надежным и безо-

пасным. Если вы выяснили, что некоторую функцию-член невозможно сделать константной, значит, вы более детально разобрались с тем, как, где и почему эта функция модифицирует состояние объекта. Кроме того, вы сможете понять, какие члены-данные объединяют физическую и логическую константность (см. приведенные ниже примеры).

Никогда не прибегайте к преобразованию константного типа в неконстантный, кроме случаев вызова функции, некорректной в плане использования модификатора `const` (не модифицирующей параметр, который тем не менее описан как неконстантный), а также такого редкого случая, как способ замены `mutable` в старом компиляторе, не поддерживающем эту возможность.

Примеры

Пример. Избегайте `const` в объявлениях функций, принимающих параметры по значению. Два следующих объявления абсолютно эквивалентны:

```
void Fun( int x );

void Fun( const int x ); // Объявление той же самой функции:
                         // const здесь игнорируется
```

Во втором объявлении модификатор `const` избыточен. Мы рекомендуем объявлять функции без таких высокоуровневых модификаторов `const`, чтобы тот, кто читает ваши заголовочные файлы, не был дезориентирован. Однако использование такого модификатора имеет значение в *определении* функции и его применение может быть оправдано с точки зрения обнаружения непреднамеренного изменения переданного параметра:

```
void Fun( const int x ) { // Определение функции Fun
    // ...
    ++x; // Ошибка: нельзя изменять константное значение
    // ...
}
```

Ссылки

[Allison98] §10 • [Cline99] §14.02-12 • [Dewhurst03] §6, §31-32, §82 • [Keffer95] pp. 5-6 • [Koenig97] §4 • [Lakos96] §9.1.6, §9.1.12 • [Meyers97] §21 • [Murray93] §2.7 • [Stroustrup00] §7.2, §10.2.6, §16.3.1 • [Sutter00] §43

16. Избегайте макросов

Резюме

Макрос — самый неприятный инструмент C и C++, оборотень, скрывающийся под личиной функции, кот, гуляющий сам по себе и не обращающий никакого внимания на границы ваших областей видимости. Берегитесь его!

Обсуждение

Трудно найти язык, достаточно красочный, чтобы выразить все, что хочется сказать о макросах. Но тем не менее приведем несколько цитат.

> *Макросы по многим причинам — весьма неприятная вещь, которая может стать попросту опасной. В первую очередь это связано с тем, что макросы — средство замены текста, действующее во время обработки исходного текста препроцессором, т.е. еще до того, как начнется какая-либо проверка синтаксиса и семантики.* — [Sutter04] §31

> *Мне не нравится большинство видов препроцессоров и макросов. Одна из целей C++ — сделать препроцессор C излишним (§4.4, §18), поскольку я считаю его большой ошибкой.* — [Stroustrup94] §3.3.1.

> *Макросы почти никогда не являются необходимыми в C++. Используйте* `const` *(§5.4) или* `enum` *(§4.8) для определения явных констант* [см. рекомендацию 15], `inline` *(§7.1.1) для того, чтобы избежать накладных расходов на вызов функции* [но см. рекомендацию 8], `template` *(глава 13) для определения семейств функций и типов* [см. рекомендации с 64 по 67], *и* `namespace` *(§8.2) для того, чтобы избежать конфликтов имен* [см. рекомендации с 57 по 59]. — [Stroustrup00] §1.6.1

> *Первое правило по применению макросов гласит: не используйте их до тех пор, пока у вас не будет другого выхода. Практически любой макрос свидетельствует о несовершенстве языка программирования, программы или программиста.* — [Stroustrup00] §7.8

Основная проблема с макросами C++ заключается в том, что они выглядят гораздо привлекательнее, чем являются таковыми на самом деле. Макросы игнорируют области видимости, игнорируют прочие возможности и правила языка, и заменяют все символы, которые переопределяют при помощи директивы `#define`, до самого конца файла. Применение макросов внешне походит на имя или вызов функции, но не имеет с ними ничего общего. Макросы "негигиеничны", в том смысле, что они могут быть раскрыты неожиданно, причем превратиться в зависимости от контекста их использования в самые разные конструкции. Подстановка текста, выполняемая макросами, делает написание хотя бы в небольшой степени "приличного" макроса смесью искусства и черной магии.

Программисты, которые полагают, что тяжелее всего расшифровать ошибки, связанные с шаблонами, вероятно, просто никогда не имели дела с плохо написанными или неверно использованными макросами. Шаблоны являются частью системы типов C++, и тем самым позволяют компилятору куда лучше справляться с ними, чем с макросами, которые имеют мало общего с языком программирования. Хуже того, в отличие от шаблонов неверные макросы могут быть раскрыты в нечто, что в силу чистой случайности скомпилируется, не имея при этом никакого смысла. И наконец, ошибка в макросе обнаруживается только после того, как макрос раскрывается, а не при его определении.

Даже в тех редких случаях, где применение макросов оправданно (см. подраздел, посвященный исключениям), нельзя даже подумать о том, чтобы написать макрос, который является распространенным словом или аббревиатурой. Для всех макросов как можно скорее применяйте директиву `#undef`, всегда давая им необычные уродливые имена в верхнем регистре, избегая при этом размещения их в заголовочных файлах.

Примеры

Пример. Передача инстанцирования шаблона макросу. Макросы понимают в достаточной мере только круглые и квадратные скобки. В C++, однако, определена новая конструкция с угловыми скобками, используемая в шаблонах. Макросы не могут корректно обработать эту ситуацию, так что вызов

```
MACRO( Foo<int, double> )
```

макрос воспринимает так, будто ему переданы два аргумента, а именно `Foo<int` и `double>`, в то время как в действительности эта конструкция представляет собой единый объект C++.

Исключения

Макросы остаются единственно возможным решением для некоторых важных задач, таких как защита директивы `#include` (см. рекомендацию 24), использование директив `#ifdef` и `#if defined` для условной компиляции и реализация `assert` (см. рекомендацию 68).

При условной компиляции (например, системно-зависимых частей) избегайте разброса по всему тексту директив `#ifdef`. Вместо этого лучше организовать код таким образом, чтобы использование макросов обеспечивало возможность альтернативных реализаций одного общего интерфейса, который затем будет использоваться в программе.

Можно (но осторожно) использовать макросы вместо большого количества копирований и вставок близких фрагментов кода.

Заметим, что [C99] и [Boost] включают соответственно умеренные и радикальные расширения препроцессоров.

Ссылки

[Boost] • [C99] • [Dewhurst03] §25-28 • [Lakos96] §2.3.4 • [Meyers96] §1 • [Stroustrup94] §3.3.1 • [Stroustrup00] §1.6.1, §7.8 • [Sutter02] §34-35 • [Sutter04] §31 • [Sutter04a]

17. Избегайте магических чисел

Резюме

Избегайте использования в коде литеральных констант наподобие `42` или `3.1415926`. Такие константы не самоочевидны и усложняют сопровождение кода, поскольку вносят в него трудноопределимый вид дублирования. Используйте вместо них символьные имена и выражения наподобие `width*aspectRatio`.

Обсуждение

Имена добавляют информацию и вводят единую точку сопровождения; в отличие от них дублированные по всей программе обычные числа анонимны и трудно сопровождаемы. Константы должны быть перечислениями или `const`-значениями, с соответствующими областями видимости и именами.

Одно число 42 может не быть тем же числом 42, что и другое. Что еще хуже, программист может выполнять какие-то вычисления "в уме" (например: "Вот это 84 — просто удвоенное 42, которое было пятью строками ранее"), что совершенно запутывает код и делает последующую замену 42 другой константой источником огромного количества ошибок.

Лучше заменять такие жестко кодированные величины символьными константами. Строки лучше хранить отдельно от кода (например, в отдельном `.cpp`-файле или файле ресурса), что позволит непрограммистам просмотреть и обновить их, снижая количество дубликатов и помогая в интернационализации вашей программы.

Примеры

Пример 1. Важные константы из предметной области на уровне пространств имен.

```
const size_t PAGE_SIZE          = 8192,
             WORDS_PER_PAGE     = PAGE_SIZE / sizeof(int),
             INFO_BITS_PER_PAGE = 32 * CHAR_BIT;
```

Пример 2. Константы, специфичные для данного класса. Вы можете определить статические интегральные константы в определении класса; константы других типов требуют отдельного определения или применения коротких функций.

```
// Файл widget.h
class Widget {
    // Значение указано в объявлении
    static const int defaultWidth = 400;
    // Значение указано в определении
    static const double defaultPercent;
    static const char* Name() { return "Widget"; }
};

// Файл widget.cpp
    // Значение указано в определении
    const double Widget::defaultPercent = 66.67;
    // Требуется объявление
    const int Widget::defaultWidth;
```

Ссылки

[Dewhurst03] §2 • [Kernighan99] §1.5 • [Stroustrup00] §4.8, §5.4

18. Объявляйте переменные как можно локальнее

Резюме

Избегайте "раздувания" областей видимости. Переменных должно быть как можно меньше, а время их жизни — как можно короче. Эта рекомендация по сути является частным случаем рекомендации 10.

Обсуждение

Переменные, время жизни которых превышает необходимое, имеют ряд недостатков.

- *Они делают программу трудно понимаемой и сопровождаемой.* Например, должен ли код обновлять строку `path` на уровне модуля, если изменен только текущий диск?
- *Они засоряют контекст своими именами.* Непосредственным следствием является то, что переменные на уровне пространства имен, наиболее видимые среди всех остальных, одновременно являются и наихудшими (см. рекомендацию 10).
- *Они не всегда могут быть корректно инициализированы.* Никогда не объявляйте переменную до того, как вы сможете корректно ее инициализировать. Неинициализированные переменные — источник "расползающихся" ошибок во всех программах C и C++, и требуют особого внимания в связи с тем, что не всегда могут быть обнаружены компилятором (см. рекомендацию 19).

В частности, старые версии языка C до [C99] требовали, чтобы переменные были определены только в начале области видимости; такой стиль в C++ вышел из употребления. Серьезная проблема такого ограничения состоит в том, что зачастую в начале области видимости не имеется достаточной информации для инициализации переменных. В результате у вас остается два выхода — либо инициализировать переменные некоторым значением по умолчанию (например, нулем), что обычно расточительно и может привести к ошибкам (если переменная будет использована до того, как приобретет некоторое осмысленное значение), либо оставить их неинициализированными, что опасно. Неинициализированная переменная пользовательского типа будет самоинициализироваться некоторым пустым значением.

Лечение этой болезни очень простое — определяйте каждую переменную настолько локально, насколько можете, что обычно означает точку непосредственное перед ее первым использованием, когда у вас уже достаточно данных для корректной инициализации.

Исключения

Иногда с точки зрения производительности может оказаться выгодным вынесение переменной за пределы цикла (см. рекомендацию 9).

Поскольку константы не являются частью состояния программы, данная рекомендация на них не распространяется (см. рекомендацию 17).

Ссылки

[Dewhurst03] §3, §48, §66 • [Dewhurst03] §95 [McConnell93] §5.1-4, §10.1 • [Stroustrup00] §4.9.4, §6.3

19. Всегда инициализируйте переменные

Резюме

Неинициализированные переменные — распространенный источник ошибок в программах на C и C++. Избегайте их, выработав привычку очищать память перед ее использованием; инициализируйте переменные при их определении.

Обсуждение

В традициях низкоуровневой эффективности C++ (как и C), от компилятора зачастую не требуется инициализация переменных, пока вы не сделаете это явно (например, локальные переменные, члены, опущенные в списке инициализации конструктора). Такие переменные надо инициализировать явно.

Имеется несколько причин, по которым переменная может остаться неинициализированной. Но ни одна из них не является достаточно серьезной для того, чтобы оправдать опасность неопределенного поведения.

Если вы используете процедурный язык (такой как Pascal, C, Fortran или Cobol), вы можете определить переменные отдельно от кода, их использующего, и присвоить им значения позже, когда эти переменные будут использоваться. Этот подход устарел и не рекомендуется для использования (см. рекомендацию 18).

Распространенное заблуждение по поводу неинициализированных переменных заключается в том, что они приводят к краху программы, так что несколько неинициализированных переменных быстро обнаруживаются простым тестированием. На самом деле программы с неинициализированными переменными могут безукоризненно работать годами, если биты в памяти соответствуют требованиям программы. Позже вызов с другим контекстом, перекомпиляция или какие-то изменения в другой части программы могут привести к последствиям разной степени тяжести — от необъяснимого поведения до периодического аварийного завершения программы.

Примеры

Пример 1. Использование инициализирующего значения по умолчанию или оператора ?: для снижения степени смешивания потока данных и потока управления.

```
// Не рекомендуется: не инициализирует переменную
int speedupFactor;
if( condition )
    speedupFactor = 2;
else
    speedupFactor = -1;

// Лучше: инициализирует переменную
int speedupFactor = -1;
if( condition )
    speedupFactor = 2;

// Лучше: инициализирует переменную
int speedupFactor = condition ? 2 : -1;
```

Варианты, отмеченные как лучшие, не имеют промежутка между определением и инициализацией.

Пример 2. Замена сложных вычислений вызовом функции. Иногда вычисление значения происходит таким образом, что лучше инкапсулировать его в функции (см. рекомендацию 11).

```
// Не рекомендуется: не инициализирует переменную
int speedupFactor;
if( condition ) {
    // ... код ...
    speedupFactor = someValue;
} else {
    // ... код ...
    speedupFactor = someOtherValue;
}

// Лучше: инициализирует переменную
int speedupFactor = ComputeSpeedupFactor();
```

Пример 3. Инициализация массивов. Для больших составных типов, таких как массивы, корректная инициализация не всегда означает реальное обращение к данным. Пусть, например, вы используете API, который заставляет вас использовать фиксированные массивы `char` размера `MAX_PATH` (см. рекомендации 77 и 78). Если вы уверены, что массивы всегда будут рассматриваться как строки в стиле C с завершающим нулевым символом, то такого немедленного присваивания будет достаточно:

```
// Допустимо: Создание пустой строки
char path[MAX_PATH]; path[0] = '\0';
```

Более безопасная инициализация заполняет все элементы массива нулевыми значениями:

```
// Лучше: заполняем нулями весь массив
char path[MAX_PATH] = { '\0' };
```

Рекомендованы оба варианта, но в общем случае вы должны предпочитать безопасность излишней эффективности.

Исключения

Входные буферы и данные, описанные как `volatile`, которые записываются непосредственно аппаратным обеспечением или другими процессами, не требуют инициализации программой.

Ссылки

[Dewhurst03] §48 • [Stroustrup00] §4.9.5, §6.3

20. Избегайте длинных функций и глубокой вложенности

Резюме

Краткость — сестра таланта. Чересчур длинные функции и чрезмерно вложенные блоки кода зачастую препятствуют реализации принципа "одна функция — одна задача" (см. рекомендацию 5), и обычно эта проблема решается лучшим разделением задачи на отдельные части.

Обсуждение

Каждая функция должна представлять собой связную единицу работы, несущую значимое имя (см. рекомендацию 5 и обсуждение рекомендации 70). Когда функция вместо этого пытается объединить малые концептуальные элементы такого рода в одном большом теле функции, это приводит к тому, что она начинает делать слишком многое.

Чрезмерно большая по размеру функция и чрезмерная вложенность блоков (например, блоков `if`, `for`, `while` и `try`) делают функции более трудными для понимания и сопровождения, причем зачастую без каких бы то ни было оснований.

Каждый дополнительный уровень вложенности приводит к излишним интеллектуальным нагрузкам при чтении кода, поскольку при этом требуется хранить в памяти "стек" наподобие "вошли в цикл... вошли в блок `try`... вошли в условный оператор... еще в один цикл...". Вам никогда не приходилось продираться сквозь сложный код и искать, какой же именно из множества конструкций `for`, `while` и т.п. соответствует вот эта закрывающая фигурная скобка? Более хорошее и продуманное разложение задачи на функции позволит читателю вашей программы одновременно удерживать в голове существенно больший контекст.

Воспользуйтесь здравым смыслом. Ограничивайте длину и глубину ваших функций. Далее приведены некоторые добрые советы, которые помогут вам в этом.

- *Предпочитайте связность.* Пусть одна функция решает только одну задачу (см. рекомендацию 5).
- *Не повторяйтесь.* Следует предпочесть именованную функцию повтору схожих фрагментов кода.
- *Пользуйтесь оператором &&.* Избегайте вложенных последовательных конструкций `if` там, где их можно заменить оператором &&.
- *Не нагружайте `try`.* Предпочитайте использовать освобождение ресурсов в деструкторах, а не в `try`-блоках (см. рекомендацию 13).
- *Пользуйтесь алгоритмами.* Они короче, чем рукописные циклы, и зачастую лучше и эффективнее (см. рекомендацию 84).
- *Не используйте `switch` для дескрипторов типов.* Применяйте вместо этого полиморфные функции (см. рекомендацию 90).

Исключения

Функция может на законных основаниях быть длинной и/или глубокой, если ее функциональность нельзя разумно разделить на отдельные подзадачи, поскольку каждое такое потенциальное разделение требует передачи массы локальных переменных и контекста (что приводит к еще менее удобочитаемому результату). Но если несколько таких потенциальных функций получают аналогичные аргументы, они могут быть кандидатами в члены нового класса.

Ссылки

[Piwowarski82] • *[Miller56]*

21. Избегайте зависимостей инициализаций между единицами компиляции

Резюме

Объекты уровня пространств имен в разных единицах компиляции не должны зависеть друг от друга при инициализации, поскольку порядок их инициализации не определен. В противном случае вам обеспечена головная боль при попытках разобраться со сбоями в работе программы после внесения небольших изменений в ваш проект и невозможностью его переноса даже на новую версию того же самого компилятора.

Обсуждение

Когда вы определяете два объекта уровня пространства имен в разных единицах компиляции, конструктор какого из объектов будет вызван первым, не определено. Чаще всего (но не всегда) ваш инструментарий будет инициализировать их в том порядке, в котором компонуются скомпилированные объектные файлы, но полагаться на это предположение нельзя, даже если оно и выполняется, — вы же не хотите, чтобы корректность вашей программы зависела от вашего файла проекта или `makefile` (дополнительную информацию о неприятностях, связанных с зависимостью от порядка, можно почерпнуть в рекомендации 59).

Таким образом, в коде инициализации любого объекта уровня пространства имен вы не можете полагаться на то, что уже инициализирован некоторый объект, определенный в другой единице компиляции. Это же касается и динамически инициализируемых переменных примитивных типов (пример такого кода на уровне пространства имен: `bool reg_success = LibRegister("mylib");`).

Заметим, что еще до того, как будет вызван конструктор, объект на уровне пространства имен статически инициализируется нулями (в отличие от, скажем, автоматического объекта, который обычно содержит мусор). Парадоксально, но эта инициализация нулями может затруднить обнаружение ошибки, поскольку вместо аварийного завершения программы такой заполненный нулями (но на самом деле неинициализированный) объект создает видимость корректности. Вам кажется, что строка пуста, указатель имеет нулевое значение, целое число равно нулю, — в то время как на самом деле никакой код еще и не пытался инициализировать ваши объекты.

Чтобы решить эту проблему, избегайте переменных уровня пространства имен везде, где только можно; такие переменные — вообще опасная практика (см. рекомендацию 10). Если вам нужна такая переменная, которая может зависеть от другой, подумайте о применении шаблона проектирования Singleton; при аккуратном его использовании можно избежать неявных зависимостей, обеспечивая инициализацию объекта при его первом использовании. Singleton остается глобальной переменной в шкуре овцы (еще раз см. рекомендацию 10), и не работает при взаимных или циклических зависимостях (и здесь инициализация нулями также способствует общей неразберихе).

Ссылки

[Dewhurst03] §55 • [Gamma95] • [McConnell93] §5.1-4 • [Stroustrup00] §9.4.1, §10.4.9

22. Минимизируйте зависимости определений и избегайте циклических зависимостей

Резюме

Избегайте излишних зависимостей. Не включайте при помощи директивы `#include` определения там, где достаточно предварительного объявления.

Избегайте взаимозависимостей. Циклические зависимости возникают, когда два модуля непосредственно или опосредованно зависят друг от друга. Модуль представляет собой обособленную единицу; взаимозависимые модули не являются полностью отдельными модулями, будучи по сути частями одного большего модуля. Таким образом, циклические зависимости являются противниками модульности и представляют угрозу большим проектам. Избегайте их.

Обсуждение

Предпочитайте использовать предварительные объявления везде, где не требуется полное определение типа. Полное определение класса `C` требуется в двух основных случаях.

- *Когда вам необходимо знать размер объекта C.* Например, при создании объекта `C` в стеке или при использовании его в качестве непосредственно хранимого (не через указатель) члена другого класса.
- *Когда вам требуется имя или вызов члена C.* Например, когда вы вызываете функцию-член этого класса.

Не будем рассматривать в этой книге тривиальные случаи циклических зависимостей, которые приводят к ошибкам компиляции — думаем, вы успешно справитесь с ними, воспользовавшись многими хорошими советами, имеющимися в литературе и рекомендации 1. Мы же обратимся к циклическим зависимостям, которые остаются в компилируемом коде, и посмотрим, как они влияют на его качество и какие действия следует предпринять, чтобы избежать их.

В общем случае зависимости и их циклы следует рассматривать на уровне модулей. Модуль представляет собой образующий единое целое набор совместно опубликованных классов и функций (см. рекомендацию 5). Циклическая зависимость в простейшем виде представляет собой два класса, непосредственно зависящих друг от друга:

```
class Child;   // Устранение циклической зависимости

class Parent { // ...
    Child* myChild_;
};

// Возможно, в другом заголовочном файле
class Child {  // ...
    Parent* myParent_;
};
```

Классы `Parent` и `Child` зависят друг от друга. Приведенный код компилируется, но мы сталкиваемся с фундаментальной проблемой: эти два класса больше не являются независимыми, и более того, становятся взаимозависимы друг от друга. Это не всегда плохо, но должно иметь место лишь в том случае, когда оба класса являются частями одного и того же мо-

дуля (разработанного одним и тем же человеком или командой, оттестированного и выпущенного как единое целое).

В противоположность описанной ситуации, рассмотрим, что будет, если класс `Child` не должен будет хранить обратную связь с объектом `Parent`? Тогда `Child` может быть выпущен в собственном отдельном небольшом модуле (и, возможно, под другим именем), полностью независимо от `Parent` — что, конечно, существенно повышает гибкость проектирования.

Все становится еще хуже, когда зависимости охватывают несколько модулей. Такой мощный "клей", как зависимости, объединяют эти модуля в единую монолитную публикуемую единицу. Вот почему циклы являются самыми страшными врагами модульности.

Чтобы разорвать циклы, примените принцип инверсии зависимостей (Dependency Inversion Principle), описанный в [Martin96a] и [Martin00] (см. также рекомендацию 36): не делайте модули высокого уровня зависящими от модулей низкого уровня; вместо этого делайте их зависимыми от абстракций. Если вы можете определить независимые абстрактные классы для `Parent` или `Child`, вы разорвете цикл. В противном случае вы должны сделать их частями одного и того же модуля.

Частный вид зависимости, от которой страдают некоторые проекты, — это транзитивная зависимость от производных классов, которая осуществляется, когда базовый класс зависит от всех своих наследников, прямых и непрямых. Ряд реализаций шаблона проектирования Visitor приводят к такому виду зависимости, которая допустима только в исключительно стабильных иерархиях. В противном случае вам лучше изменить свой проект, например, воспользовавшись шаблоном проектирования Acyclic Visitor [Martin98].

Одним из симптомов чрезмерных зависимостей является перекомпиляция больших частей проекта при внесении небольших локальных изменений (см. рекомендацию 2).

Исключения

Циклические зависимости между классами — не всегда плохо, пока классы рассматриваются как часть одного модуля и совместно тестируются и выпускаются. Простая непосредственная реализация таких шаблонов проектирования, как Command и Visitor приводят к интерфейсам, которые естественным образом оказываются взаимозависимыми. Такие зависимости можно разрушить, но это требует более четкого проектирования.

Ссылки

[Alexandrescu01] §3 • [Boost] • [Gamma95] • [Lakos96] §0.2.1, §4.6-14, §5 • [Martin96a] • [Martin96b] • [Martin98] §7 • [Martin00] • [McConnell93] §5 • [Meyers97] §46 • [Stroustrup00] §24.3.5 • [Sutter00] §26 • [Sutter02] §37 • [Sutter03]

23. Делайте заголовочные файлы самодостаточными

Резюме

Убедитесь, что каждый написанный вами заголовочный файл компилируем самостоятельно, т.е. что он включает все заголовочные файлы, от которых зависит его содержимое.

Обсуждение

Если один заголовочный файл не работает, пока не включен другой заголовочный файл, проект получается очень неуклюжим, а на пользователя возлагается дополнительная задача следить за тем, какие заголовочные файлы надо включить в исходный текст.

Раньше некоторые эксперты советовали, чтобы заголовочные файлы не включали другие заголовочные файлы из-за накладных расходов на многократное открытие и анализ заголовочных файлов, защищенных директивами препроцессоров от повторной обработки. К счастью, сейчас этот совет устарел. Многие современные компиляторы C++ распознают соответствующую защиту заголовочных файлов автоматически (см. рекомендацию 24) и просто не открывают один и тот же заголовочный файл дважды. Некоторые компиляторы используют предкомпиляцию заголовочных файлов, которая позволяет избежать анализа часто используемых заголовочных файлов.

Однако не включайте заголовочные файлы, в которых вы *не* нуждаетесь, так как это напрасно создает паразитные зависимости.

Для гарантии самодостаточности заголовочных файлов скомпилируйте каждый из них отдельно от других и убедитесь, что это не приводит к ошибкам или предупреждениям.

Примеры

Ряд тонких моментов возникает в связи с использованием шаблонов.

Пример 1. Зависимые имена. Шаблоны компилируются в точке, где они определены, с тем исключением, что все зависимые имена или типы не компилируются до точки инстанцирования. Это означает, что `template<class T> class Widget` с членом `std::deque<T>` не приведет к ошибке времени компиляции, даже если заголовочный файл `<deque>` не включен — если при этом не происходит инстанцирование `Widget`. Поскольку очевидно, что шаблон `Widget` существует для того, чтобы быть инстанцированным, его заголовочный файл должен содержать строку `#include <deque>`.

Пример 2. Шаблоны функций-членов и функции-члены шаблонов инстанцируются только при использовании. Предположим, что `Widget` не имеет члена типа `std::deque<T>`, но функция-член `Widget` с именем `Transmogrify` использует `deque`. Тогда вызывающая `Widget` функция может инстанцировать и использовать `Widget`, даже если заголовочный файл `<deque>` не включен — до тех пор, пока не используется функция-член `Transmogrify`. По умолчанию заголовочный файл `Widget` все же должен включать `<deque>`, поскольку это необходимо, по крайней мере, для некоторых пользователей `Widget`. В редких случаях, когда большой заголовочный файл включается только для обеспечения работоспособности нескольких редко используемых функций шаблона, следует подумать о том, чтобы сделать эти функции не членами и вынести их в отдельный заголовочный файл, включающий упомянутый большой файл (см. рекомендацию 44).

Ссылки

[Lakos96] §3.2 • [Stroustrup00] §9.2.3 • [Sutter00] §26-30 • [Vandevoorde03] §9-10

24. Используйте только внутреннюю, но не внешнюю защиту директивы #include

Резюме

Предотвращайте непреднамеренное множественное включение ваших заголовочных файлов директивой `#include`, используя в них защиту с уникальными именами.

Обсуждение

Каждый заголовочный файл должен использовать внутреннюю защиту директивы `#include`, чтобы предотвратить переопределения в случае множественного включения данного файла. Например, заголовочный файл `foo.h` должен иметь такой общий вид:

```
#ifndef FOO_H_INCLUDED_
#define FOO_H_INCLUDED_
// ... Содержимое файла ...
#endif
```

Обратите внимание на следующие правила при определении защиты включения.

- *Используйте для защиты уникальные имена.* Убедитесь, что вы используете имена, уникальные, по крайней мере, в пределах вашего приложения. Выше мы использовали одно распространенное соглашение для используемых в защите имен; имена для защиты могут включать имя приложения, а некоторые инструменты генерируют имена для защиты, содержащие случайные числа.
- *Не пытайтесь хитрить.* Не размещайте никакого кода или комментариев до и после защищенной части, и следуйте показанной выше стандартной форме защиты. Современные препроцессоры могут обнаружить защиту, но могут оказаться малоинтеллектуальными и ожидать кода защиты строго в начале и в конце заголовочных файлов.

Избегайте использования устаревшей внешней защиты директивы `#include`, рекомендуемой в некоторых старых книгах:

```
#ifndef FOO_H_INCLUDED_ // НЕ рекомендуется
#include "foo.h"
#define FOO_H_INCLUDED_
#endif
```

Внешняя защита утомительна, устарела для современных компиляторов и ненадежна из-за необходимости согласования имен для защиты.

Исключения

В очень редких случаях заголовочный файл может быть предназначен для многократного включения.

Ссылки

[C++03, §2.1] • [Stroustrup00] §9.3.3

Функции и операторы

Если ваша процедура имеет десять параметров —
вероятно, вы где-то ошиблись.

— Алан Перлис (Alan Perlis)

Функции, включая перегруженные операторы, представляют собой фундаментальные единицы работы. Как вы увидите позже в разделе "Обработка ошибок и исключения" (в частности, в рекомендации 70), это непосредственно влияет на наши рассуждения о корректности и безопасности кода.

Но давайте сначала рассмотрим некоторые фундаментальные вопросы написания функций, в том числе операторов. В частности, мы обратим особое внимание на их параметры, семантику и перегрузку.

В этом разделе наиболее важной нам представляется рекомендация 26 — "Сохраняйте естественную семантику перегруженных операторов".

25. Передача параметров по значению, (интеллектуальному) указателю или ссылке

Резюме

Вы должны четко уяснить разницу между входными, выходными параметрами и параметрами, предназначенными и для ввода, и для вывода информации, а также между передачей параметров по значению и по ссылке, и корректно их использовать.

Обсуждение

Правильный выбор способа передачи аргументов в функцию — по значению, ссылке или с использованием указателей — весьма необходимый навык, который поможет вам сделать ваш код максимально безопасным и эффективным.

Хотя эффективность не должна быть нашей главной целью (см. рекомендацию 8), при прочих равных условиях, включая понятность кода, мы не должны без крайней необходимости снижать его эффективность (см. рекомендацию 9).

Вам стоит следовать приведенным ниже рекомендациям при выборе способа передачи параметров. Вот рекомендации для входных параметров (которые передают информацию в функцию, но не возвращают ее).

- Всегда описывайте все указатели или ссылки на входные параметры как `const`.
- Предпочитайте передачу примитивных типов (например, `char` или `float`) и объектов-значений с недорогими операциями копирования (например, `Point` или `complex <float>`) по значению.
- Входные аргументы прочих пользовательских типов лучше передавать как ссылки на `const`.
- Подумайте о передаче по значению вместо передачи по ссылке, если функция требует создания копии аргумента. Концептуально передача по значению эквивалентна передаче ссылки на константный объект и выполнение копирования, и может помочь компилятору в выполнении оптимизации по устранению временных переменных.

Далее приведены рекомендации для выходных параметров (а также параметров для одновременного ввода и вывода информации).

- Если аргумент необязателен, лучше передать его как (интеллектуальный) указатель (что позволит вызывающей функции передать нулевое значение как указание, что аргумент отсутствует); тот же совет применим и в случае, если функция сохраняет копию указателя или как-то иначе работает с принадлежностью аргумента.
- Если аргумент обязательный и функция не сохраняет указатель на него или каким-то иным образом влияет на его принадлежность, то такой аргумент лучше передавать по ссылке. Это указывает, что наличие данного аргумента обязательно, и заставляет вызывающую функцию отвечать за предоставление корректного объекта.

Не используйте функции с переменным количеством аргументов (см. рекомендацию 98).

Ссылки

[Alexandrescu03a] • [Cline99] §2.10-11, 14.02-12, 32.08 • [Dewhurst03] §57 • [Koenig97] §4 • [Lakos96] §9.1.11-12 • [McConnell93] §5.7 • [Meyers97] §21-22 • [Stroustrup94] §11.4.4 • [Stroustrup00] §5.5, §11.6, §16.3.4 • [Sutter00] §6, §46

26. Сохраняйте естественную семантику перегруженных операторов

Резюме

Программисты ненавидят сюрпризы. Перегружайте операторы только в случае веских на то оснований, и сохраняйте при этом их естественную семантику. Если это оказывается сложным, возможно, вы неверно используете перегрузку операторов.

Обсуждение

Хотя все (как мы надеемся) согласны с тем, что не следует реализовывать вычитание как оператор `operator+`, прочие ситуации не столь очевидны. Например, означает ли оператор `operator*` вашего класса `Tensor` скалярное или векторное умножение? Должен ли оператор `operator+=(Tensor&t, unsigned u)` прибавлять `u` к каждому из элементов `t`, или должен изменять размер `t`? В таких неоднозначных или не поддающихся интуитивному пониманию случаях следует использовать именованные функции, а не прибегать к шифрованию.

Для типов-значений (но не для всех типов; см. рекомендацию 32) следует придерживаться правила: "Если не знаешь, как поступить — поступай так, как `int`" [Meyers96]. Подражание поведению операторов встроенных типов и взаимоотношениям между ними гарантирует, что вы никого не приведете в замешательство. Если выбранная вами семантика заставляет кого-то удивленно поднять брови, может быть, перегрузка оператора — не самая лучшая идея?

Программисты ожидают, что операторы идут в связке — если выражение `a@b` имеет определенный смысл для некоторого определенного вами оператора `@` (возможно, после преобразования типов), то задайте сами себе вопрос: можно ли написать `b@a` без неприятных последствий? Можно ли написать `a@=b`? (См. рекомендацию 27.) Если оператор имеет обратный оператор (например, + и -, * и /), то поддерживаются ли они оба?

От именованных функций не ожидается наличие соответствующих взаимоотношений, так что для большей ясности кода, если возможно неверное истолкование семантики перегруженных операторов, лучше использовать именно функции.

Исключения

Имеются высокоспециализированные библиотеки (например, генераторы синтаксических анализаторов), в предметной области которых соглашения о семантике операторов существенно отличаются от их значений в C++ (например, при работе с регулярными выражениями оператор `operator*` может использоваться для выражения "ноль или большее количество"). Предпочтительно найти альтернативу необычной перегрузке оператора (например, в регулярных выражениях [C++TR104] используются строки, так что * может использоваться естественным образом, без перегрузки операторов). Если все же после тщательных размышлений вы решили использовать операторы, убедитесь, что вы четко определили единую схему для всех ваших соглашений, и при этом не затеяли опасные игры со встроенными операторами.

Ссылки

[Cline99] §23.02-06 • [C++TR104] §7 • [Dewhurst03] §85-86 • [Koenig97] §4 • [Lakos96] §9.1.1 • [Meyers96] §6 • [Stroustrup00] §11.1 • [Sutter00] §41

27. Отдавайте предпочтение каноническим формам арифметических операторов и операторов присваивания

Резюме

Если можно записать a+b, то необходимо, чтобы можно было записать и a+=b. При определении бинарных арифметических операторов одновременно предоставляйте и их присваивающие версии, причем делайте это с минимальным дублированием и максимальной эффективностью.

Обсуждение

В общем случае для некоторого бинарного оператора @ (+, -, * и т.д.) вы должны также определить его присваивающую версию, так чтобы a@=b и a=a@b имели один и тот же смысл (причем первая версия может быть более эффективна). Канонический способ достижения данной цели состоит в определении @ посредством @= следующим образом:

```
T& T::operator@=( const T& ) {
    // ... реализация ...
    return *this;
}

T operator@( const T& lhs, const T& rhs ) {
    T temp( lhs );
    return temp @= rhs;
}
```

Эти две функции работают в тандеме. Версия оператора с присваиванием выполняет всю необходимую работу и возвращает свой левый параметр. Версия без присваивания создает временную переменную из левого аргумента, и модифицирует ее с использованием формы оператора с присваиванием, после чего возвращает эту временную переменную.

Обратите внимание, что здесь `operator@` — функция-не член, так что она обладает желательным свойством возможности неявного преобразования как левого, так и правого параметра (см. рекомендацию 44). Например, если вы определите класс `String`, который имеет неявный конструктор, получающий аргумент типа `char`, то оператор `operator+(const String&, const String&)`, который не является членом класса, позволяет осуществлять операции как типа `char+String`, так и `String+char`; функция-член `String::operator+(const String&)` позволяет использовать только операцию `String+char`. Реализация, основной целью которой является эффективность, может определить ряд перегрузок оператора `operator@`, не являющихся членами класса, чтобы избежать увеличения количества временных переменных в процессе преобразований типов (см. рекомендацию 29).

Также делайте не членом функцию `operator@=` везде, где это возможно (см. рекомендацию 44). В любом случае, все операторы, не являющиеся членами, должны быть помещены в то же пространство имен, что и класс T, так что они будут легко доступны для вызывающих функций при отсутствии каких-либо сюрпризов со стороны поиска имен (см. рекомендацию 57).

Как вариант можно предусмотреть оператор `operator@`, принимающий первый параметр по значению. Таким образом вы обеспечите неявное копирование компилятором, что обеспечит ему большую свободу действий по оптимизации:

```
T& operator@=(T& lhs, const T& rhs) {
    // ... реализация ...
    return lhs;
}

T operator@(T lhs, const T& rhs) {// lhs передано по значению
    return lhs @= rhs;
}
```

Еще один вариант — оператор `operator@`, который возвращает `const`-значение. Эта методика имеет то преимущество, что при этом запрещается такой не имеющий смысла код, как `a+b=c`, но в этом случае мы теряем возможность применения потенциально полезных конструкций наподобие `a = (b+c).replace(pos,n,d)`. А это весьма выразительный код, который в одной строчке выполняет конкатенацию строк `b` и `c`, заменяет некоторые символы и присваивает полученный результат переменной `a`.

Примеры

Пример. Реализация += для строк. При конкатенации строк полезно заранее знать длину, чтобы выделять память только один раз:

```
String& String::operator+=( const String& rhs ) {
    // ... Реализация ...
    return *this;
}

String operator+( const String& lhs, const String& rhs ) {
    String temp;                    // Изначально пуста
    // Выделение достаточного количества памяти
    temp.Reserve(lhs.size()+rhs.size());
    // Конкатенация строк и возврат
    return (temp += lhs) += rhs;
}
```

Исключения

В некоторых случаях (например, оператор `operator*=` для комплексных чисел), оператор может изменять левый аргумент настолько существенно, что более выгодным может оказаться реализация оператора `operator*=` посредством оператора `operator*`, а не наоборот.

Ссылки

[Alexandrescu03a] • [Cline99] §23.06 • [Meyers96] §22 • [Sutter00] §20

28. Предпочитайте канонический вид ++ и --, и вызов префиксных операторов

Резюме

Особенность операторов инкремента и декремента состоит в том, что у них есть префиксная и постфиксная формы с немного отличающейся семантикой. Определяйте операторы `operator++` и `operator--` так, чтобы они подражали поведению своих встроенных двойников. Если только вам не требуется исходное значение — используйте префиксные версии операторов.

Обсуждение

Старая шутка гласит, что язык называется C++, а не ++C, потому что язык был улучшен (на что указывает инкремент), но многие продолжают использовать его как C (предыдущее значение до инкремента). К счастью, эту шутку можно считать устаревшей, но это отличная иллюстрация для понимания отличия между двумя формами операторов.

В случае ++ и -- постфиксные формы операторов возвращают исходное значение, в то время как префиксные формы возвращают новое значение. Лучше всего реализовывать постфиксный оператор с использованием префиксного. Вот канонический вид такого использования:

```
// ---- Префиксные операторы --------------------
T& T::operator++()        // Префиксный вид:
{
    // Выполнение         // - Выполнение
    // инкремента         //   действий
    return *this;         // - return *this;
}

T& T::operator--()        // Префиксный вид:
{
     // Выполнение        // - Выполнение
     // декремента        //   действий
     return *this;        // - return *this;
}

// ---- Постфиксные операторы -------------------

T T::operator++(int)      // Постфиксный вид:
{
    T old( *this );       // - Запоминаем старое значение
    ++*this;              // - Вызов префиксной версии
    return old;           // - Возврат старого значения
}

T T::operator--(int)      // Постфиксный вид:
{
    T old( *this );       // - Запоминаем старое значение
    --*this;              // - Вызов префиксной версии
    return old;           // - Возврат старого значения
}
```

В вызывающем коде лучше использовать префиксные операторы, если только вам не требуется исходное значение, возвращаемое постфиксной версией. Префиксная форма семантически эквивалентна, она вводится практически так же, и зачастую немного эффективнее, так как создает на один объект меньше. Это не преждевременная оптимизация, а устранение преждевременной пессимизации (см. рекомендацию 9).

Исключения

Шаблоны, используемые для научных вычислений (например, шаблоны для представления тензоров или матриц), к которым предъявляются жесткие требования по производительности. Для таких шаблонов часто приходится искать более эффективные способы реализации префиксных форм операторов.

Ссылки

[Cline99] §23.07-08 • [Dewhurst03] §87 • [Meyers96] §6 • [Stroustrup00] §19.3 • [Sutter00] §6, §20

29. Используйте перегрузку, чтобы избежать неявного преобразования типов

Резюме

Не преумножайте объекты сверх необходимости (Бритва Оккама): неявное преобразование типов обеспечивает определенное синтаксическое удобство (однако см. рекомендацию 40), но в ситуации, когда допустима оптимизация (см. рекомендацию 8) и желательно избежать создания излишних объектов, можно обеспечить перегруженные функции с сигнатурами, точно соответствующими распространенным типам аргументов, и тем самым избежать преобразования типов.

Обсуждение

Когда вы работаете в офисе и у вас заканчивается бумага, что вы делаете? Правильно — вы идете к копировальному аппарату и делаете несколько копий чистого листа бумаги.

Как бы глупо это ни звучало, но зачастую это именно то, что делает неявное преобразование типов: создание излишних временных объектов только для того, чтобы выполнить некоторые тривиальные операции над ними и тут же их выбросить (см. рекомендацию 40). В качестве примера можно рассмотреть сравнение строк:

```
class String { // ...
    String( const char* text ); // Обеспечивает неявное
                                // преобразование типов
};
bool operator==( const String&, const String& );

// ... Где-то в коде ...
if( someString == "Hello" ) { ... }
```

Ознакомившись с приведенными выше определениями, компилятор компилирует приведенное сравнение таким образом, как если бы оно было записано в виде `someString == String("Hello")`. Это слишком расточительно — копировать символы, чтобы потом просто прочесть их. Решение этой проблемы очень простое — определить перегруженные функции, чтобы избежать преобразования типов, например:

```
bool operator==( const String& lhs, const String& rhs ); // #1
bool operator==( const String& lhs, const char* rhs );   // #2
bool operator==( const char* lhs, const String& rhs );   // #3
```

Это выглядит как дублирование кода, но на самом деле это всего лишь "дублирование сигнатур", поскольку все три варианта обычно используют одну и ту же функцию. Вряд ли вы впадете в ересь преждевременной оптимизации (см. рекомендацию 8) при такой простой перегрузке, тем более что этот метод слабо применим при проектировании библиотек, когда трудно заранее предсказать, какие именно типы будут использоваться в коде, критическом по отношению к производительности.

Ссылки

[Meyers96] §21 • [Stroustrup00] §11.4, §C.6 • [Sutter00] §6

30. Избегайте перегрузки &&, | | и , (запятой)

Резюме

Мудрость — это знание того, когда надо воздержаться. Встроенные операторы &&, | | и , (запятая) трактуются компилятором специальным образом. После перегрузки они становятся обычными функциями с весьма отличной семантикой (при этом вы нарушаете рекомендации 26 и 31), а это прямой путь к трудноопределимым ошибкам и ненадежности. Не перегружайте эти операторы без крайней необходимости и глубокого понимания.

Обсуждение

Главная причина отказа от перегрузки операторов `operator&&`, `operator||` и `operator,` (запятая) состоит в том, что вы не имеете возможности реализовать полную семантику встроенных операторов в этих трех случаях, а программисты обычно ожидают ее выполнения. В частности, встроенные версии выполняют вычисление слева направо, а для операторов && и | | используются сокращенные вычисления.

Встроенные версии && и | | сначала вычисляют левую часть выражения, и если она полностью определяет результат (`false` для &&, `true` для | |), то вычислять правое выражение незачем — и оно гарантированно не будет вычисляться. Таким образом мы используем эту возможность, позволяя корректности правого выражения зависеть от успешного вычисления левого:

```
Employee* e = TryToGetEmployee();
if( e && e->Manager() )
// ...
```

Корректность этого кода обусловлена тем, что `e->Manager()` не будет вычисляться, если `e` имеет нулевое значение. Это совершенно обычно и корректно — до тех пор, пока не используется перегруженный оператор `operator&&`, поскольку в таком случае выражение, включающее &&, будет следовать правилам функции:

- вызовы функций всегда вычисляют все аргументы до выполнения кода функции;
- порядок вычисления аргументов функций не определен (см. также рекомендацию 31).

Давайте рассмотрим модернизированную версию приведенного ранее фрагмента, которая использует интеллектуальные указатели:

```
some_smart_ptr<Employee> e = TryToGetEmployee();
if( e && e->Manager() )
// ...
```

Пусть в этом коде используется перегруженный оператор `operator&&` (предоставленный автором `some_smart_ptr` или `Employee`). Тогда мы получаем код, который для читателя выглядит совершенно корректно, но потенциально может вызвать `e->Manager()` при нулевом значении `e`.

Некоторый иной код может не привести к аварийному завершению программы, но стать некорректным по другой причине — из-за зависимости от порядка вычислений двух выражений. Результат может оказаться плачевным. Например:

```
if( DisplayPrompt() && GetLine() )
// ...
```

Если оператор `operator&&` переопределен пользователем, то неизвестно, какая из функций — `DisplayPrompt` или `GetLine` — будет вызвана первой. Программа в результате может ожидать ввода пользователя до того, как будет выведено соответствующее поясняющее приглашение.

Конечно, такой код может заработать при использовании вашего конкретного компилятора и настроек сборки. Но это — очень ненадежно. Компиляторы могут выбрать любой порядок вычислений (и так они и поступают), который сочтут нужным для данного конкретного вызова, принимая во внимание такие факторы, как размер генерируемого кода, доступные регистры, сложность выражений и другие. Так что один и тот же вызов может проявлять себя по-разному в зависимости от версии компилятора, настроек компиляции и даже инструкций, окружающих данный вызов.

Та же ненадежность наблюдается и в случае оператора-запятой. Так же, как и операторы && и ||, встроенный оператор-запятая гарантирует, что выражения будут вычислены слева направо (в отличие от && и ||, здесь всегда вычисляются оба выражения). Пользовательский оператор-запятая не может гарантировать вычислений слева направо, что обычно приводит к удивительным результатам. Например, если в следующем коде используется пользовательский оператор-запятая, то неизвестно, получит ли функция g аргумент 0 или 1:

```
int i = 0;
f( i++ ), g( i );      // См. также рекомендацию 31
```

Примеры

Пример. Инициализация библиотеки при помощи перегруженного оператора `operator,` *для последовательности инициализаций.* Некоторая библиотека пытается упростить добавление нескольких значений в контейнер за один раз путем перегрузки оператора-запятой. Например, для добавления в `vector<string> letters`:

```
set_cont(letters) += "a", "b";
```

Все в порядке, пока в один прекрасный день пользователь не напишет:

```
set_cont(letters) += getstr(), getstr();
        // Порядок не определен при использовании
        // перегруженного оператора ","
```

Если функция `getstr` получает, например, ввод пользователя и он введет строки `"c"` и `"d"` в указанном порядке, то в действительности строки могут оказаться внесены в любом порядке. Это может оказаться сюрпризом, поскольку при использовании встроенного оператора `operator,` такой проблемы не возникает:

```
string s; s = getstr(), getstr(); // Порядок строго определен
                                  // при использовании
                                  // встроенного оператора ","
```

Исключения

Исключение — специализированные библиотеки шаблонов для научных вычислений, которые в соответствии с дизайном переопределяют все операторы.

Ссылки

[Dewhurst03] §14 • [Meyers96] §7, §25 • [Murray93] §2.4.3 • [Stroustrup00] §6.2.2

31. Не пишите код, который зависит от порядка вычислений аргументов функции

Резюме

Порядок вычисления аргументов функции не определен, поэтому никогда не полагайтесь на то, что аргументы будут вычисляться в той или иной очередности.

Обсуждение

На начальных этапах развития языка C регистры процессора были драгоценным ресурсом и компиляторы решали трудные задачи эффективного их использования в сложных выражениях высокоуровневых языков. Для того чтобы позволить компилятору генерировать более быстрый код, создатели C дали распределителю регистров дополнительную степень свободы. При вызове функции порядок вычисления ее аргументов оставался неопределенным. Эта аргументация, вероятно, существенно менее важна в настоящее время, но главное, что порядок вычисления аргументов функций в C++ не определен и варьируется от компилятора к компилятору (см. также рекомендацию 30).

В связи с этим необдуманные действия программиста могут привести к большим неприятностям. Рассмотрим следующий код:

```
void Transmogrify( int, int );

int count = 5;
Transmogrify( ++count, ++count ); // Порядок вычислений
                                  // неизвестен
```

Все, что мы можем сказать определенного, — это то, что при входе в тело функции `Transmogrify` значение переменной `count` будет равно 7 — но мы не можем сказать, какой аргумент будет равен 6, а какой — 7. Эта неопределенность остается и в гораздо менее очевидных случаях, таких как функции, модифицирующие свои аргументы (или некоторое глобальное состояние) в качестве побочного действия:

```
int Bump( int& x ) { return ++x; }
Transmogrify( Bump(count), Bump(count) ); // Результат
                                          // неизвестен
```

Согласно рекомендации 10, следует в первую очередь избегать глобальных и совместно используемых переменных. Но даже если вы благополучно устраните их, некоторый другой код может этого не сделать. Например, некоторые стандартные функции имеют побочные действия (например, `strtok`, а также разные перегруженные операторы `operator<<`, принимающие в качестве аргумента `ostream`).

Рецепт очень прост — использовать именованные объекты для того, чтобы обеспечить порядок вычислений (см. рекомендацию 13):

```
int bumped = ++count;
Transmogrify( bumped, ++count ); // Все в порядке
```

Ссылки

[Alexandrescu00c] • [Cline99] §31.03-05 • [Dewhurst03] §14-15 • [Meyers96] §9-10 • [Stroustrup00] §6.2.2, §14.4.1 • [Sutter00] §16 • [Sutter02] §20-21

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 большую функциональность, но никогда не должно предъявлять бо́льшие 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 );  // А: 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

Конструкторы, деструкторы и копирование

Если стандарт привел вас к обрыву, это еще не значит, что вы должны прыгнуть с него.

— Норман Даймонд (Norman Diamond)

О Большой Четверке специальных функций было сказано достаточно, чтобы вы не удивлялись тому, что им посвящен отдельный раздел. Здесь собраны знания и практика, связанные с конструкторами по умолчанию, копирующими конструкторами, копирующим присваиванием и деструкторами.

Одна из причин, по которым при работе с этими функциями следует быть особенно внимательными, заключается в том, что если вы дадите компилятору хотя бы полшанса — он тут же напишет эти функции за вас. Еще одна причина состоит в том, что C++ по умолчанию рассматривает классы как типы-значения, но далеко не все типы именно таковы (см. рекомендацию 32). Надо отчетливо понимать, когда следует писать (или запрещать) эти специальные функции явно, и следовать правилам и рекомендациям из этого раздела — это поможет вам в написании корректного, расширяемого и безопасного кода.

В этом разделе мы считаем наиболее значимой рекомендацию 51 — "Деструкторы, функции освобождения ресурсов и обмена не ошибаются".

47. Определяйте и инициализируйте переменные-члены в одном порядке

Резюме

Переменные-члены всегда инициализируются в том порядке, в котором они объявлены при определении класса; порядок их упоминания в списке инициализации конструктора игнорируется. Убедитесь, что в коде конструктора указан тот же порядок, что и в определении класса.

Обсуждение

Рассмотрим следующий код:

```
class Employee {
    string email_, firstName_, lastName_;

public:
    Employee( const char* firstName, const char* lastName )
        : firstName_(firstName), lastName_(lastName)
        , email_(firstName_+"."+lastName_+"@acme.com") {}
};
```

Этот код содержит ошибку, столь же неприятную, сколь и трудно обнаруживаемую. Поскольку член `email_` объявлен в определении класса до `first_` и `last_`, он будет инициализирован первым и будет пытаться использовать еще не инициализированные поля. Более того, если определение конструктора находится в отдельном файле, то выявить такое удаленное влияние порядка объявления переменных-членов класса на корректность конструктора окажется еще труднее.

Эта особенность языка обусловлена необходимостью гарантировать единый порядок уничтожения членов; в противном случае деструктор был бы должен уничтожать объекты в разном порядке, в зависимости от того, в каком именно порядке конструктор создавал их. Накладные расходы, необходимые для решения этой проблемы, признаны неприемлемыми.

Решение заключается в том, чтобы всегда писать инициализаторы членов в том же порядке, в котором эти члены объявлены в классе. В этом случае сразу становятся очевидными все некорректные зависимости между членами. Еще лучше полностью избегать зависимости инициализации одного члена от других.

Многие компиляторы (но не все) выдают предупреждение при нарушении этого правила.

Ссылки

[Cline99] §22.03-11 • [Dewhurst03] §52-53 • [Koenig97] §4 • [Lakos96] §10.3.5 • [Meyers97] §13 • [Murray93] §2.1.3 • [Sutter00] §47

48. В конструкторах предпочитайте инициализацию присваиванию

Резюме

В конструкторах использование инициализации вместо присваивания для установки значений переменных-членов предохраняет от ненужной работы времени выполнения при том же объеме вводимого исходного текста.

Обсуждение

Конструкторы генерируют скрытый код инициализации. Рассмотрим следующий код:

```
class A {
    string s1_, s2_;
public:
    A() { s1_ = "Hello, "; s2_ = "world"; }
};
```

В действительности сгенерированный код конструктора выглядит так, как если бы вы написали:

```
A() : s1_(), s2_() { s1_ = "Hello, "; s2_ = "world"; }
```

То есть объекты, не инициализированные вами явно, автоматически инициализируются с использованием их конструкторов по умолчанию, после чего выполняется присваивание значений с использованием их операторов присваивания. Чаще всего операторы присваивания нетривиальных объектов выполняют немного больше работы, чем конструкторы, поскольку работают с уже созданными объектами.

Таким образом, инициализация переменных-членов в списке инициализации дает код, лучше выражающий ваши намерения и обычно более быстрый и меньшего размера:

```
A() : s1_("Hello, "), s2_("world") { }
```

Эта методика не является преждевременной оптимизацией; это — избежание преждевременной пессимизации (см. рекомендацию 9).

Исключения

Всегда выполняйте захват неуправляемого ресурса (например, выделение памяти оператором `new`, результат которого не передается немедленно конструктору интеллектуального указателя) в теле конструктора, а не в списке инициализации (см. [Sutter02]). Конечно, лучше всего вообще не использовать таких небезопасных и не имеющих владельца ресурсов (см. рекомендацию 13).

Ссылки

[Dewhurst03] §51, §59 • [Keffer95] pp.13-14 • [Meyers97] §12 • [Murray93] §2.1.31 • [Sutter00] §8, §47 • [Sutter02] §18

49. Избегайте вызовов виртуальных функций в конструкторах и деструкторах

Резюме

Внутри конструкторов и деструкторов виртуальные функции теряют виртуальность. Хуже того — все прямые или косвенные вызовы нереализованных чисто виртуальных функций из конструктора или деструктора приводят к неопределенному поведению. Если ваш дизайн требует виртуальной передачи в производный класс из конструктора или деструктора базового класса, следует воспользоваться иной методикой, например, постконструкторами.

Обсуждение

В C++ полный объект конструируется по одному базовому классу за раз.

Пусть у нас есть базовый класс `B` и класс `D`, производный от `B`. При создании объекта `D`, когда выполняется конструктор `B`, динамическим типом создаваемого объекта является `B`. В частности, вызов виртуальной функции `B::Fun` приведет к выполнению функции `Fun`, определенной в классе `B`, независимо от того, перекрывает ее класс `D` или нет. И это хорошо, поскольку вызов функции-члена `D` в тот момент, когда члены объекта `D` еще не инициализированы, может привести к хаосу. Только после завершения выполнения конструктора `B` выполняется тело конструктора `D` и объект приобретает тип `D`. В качестве эмпирического правила следует помнить, что в процессе конструирования `B` нет никакого способа определить, является ли `B` отдельным объектом или базовой частью некоторого иного производного объекта.

Кроме того, следует помнить, что вызов из конструктора чисто виртуальной функции, не имеющей определения, приводит к неопределенному поведению.

С другой стороны, в некоторых случаях дизайн требует использования "постконструктора", т.е. виртуальной функции, которая должна быть вызвана после того, как полный объект оказывается сконструирован. Некоторые методики, применяемые для решения этой задачи, описаны в приводимых ниже ссылках. Вот (далеко не исчерпывающий) список возможных решений.

- *Перекладывание ответственности.* Можно просто документировать необходимость вызова в пользовательском коде постконструктора после создания объекта.
- *Отложенная постинициализация.* Можно выполнять постинициализацию при первом вызове функции-члена, для чего использовать в базовом классе флаг типа `bool`, который показывает, был уже вызван постконструктор или нет.
- *Использование семантики базового класса.* Правила языка требуют, чтобы конструктор наиболее производного класса определял, какой из конструкторов базовых классов будет вызван. Вы можете использовать это правило в своих целях (см. [Taligent94]).
- *Использование функции-фабрики.* Таким способом вы можете легко обеспечить принудительный вызов функции-постконструктора (см. примеры к данной рекомендации).

Ни одна из методик постконструирования не является идеальной. Наихудшее, что можно сделать, — это потребовать, чтобы пользователь класса вручную вызывал постконструктор. Однако даже наилучшие способы решения данной задачи требуют отличного от обычного синтаксиса конструирования объекта (что легко проверяется в процессе компиляции) и/или сотрудничества с авторами производных классов (что невозможно проверить в процессе компиляции).

Примеры

Пример. Использование функции-фабрики для принудительного вызова постконструктора. Рассмотрим следующий код:

```
class B {                          // Корень иерархии
protected:
    B() { /* ... */ }

    virtual void PostInitialize() // Вызывается сразу
        { /* ... */ }              // после конструирования

public:
template<class T>
    static shared_ptr<T> Create() // Интерфейс для
    {                             // создания объектов
        shared_ptr<T> p( new T );
        p->PostInitialize();
        return p;
    }
};

class D : public B { /* ... */ }; // Некоторый производный
                                  // класс

shared_ptr<D> p = D::Create<D>(); // Создание объекта D
```

Этот не вполне надежный дизайн основан на ряде компромиссов.

- Производные классы, такие как `D`, не должны иметь открытых конструкторов. В противном случае пользователи `D` смогут создавать объекты `D`, для которых не будет вызываться функция `PostInitialize`.
- Создание объекта использует оператор `new`, который, однако, может быть перекрыт классом `B` (см. рекомендации 45 и 46).
- Класс `D` обязательно должен определить конструктор с теми же параметрами, что и у конструктора класса `B`. Смягчить проблему может наличие нескольких перегрузок функции `Create`; эти перегрузки могут даже быть шаблонами.
- Если перечисленные требования удовлетворены, данный дизайн гарантирует, что функция `PostInitialize` будет вызвана для любого полностью сконструированного объекта класса, производного от `B`. Функция `PostInitialize` не обязательно должна быть виртуальной; однако она может свободно вызывать виртуальные функции.

Ссылки

[Alexandrescu01] §3 • [Boost] • [Dewhurst03] §75 • [Meyers97] §46 • [Stroustrup00] §15.4.3 • [Taligent94]

50. Делайте деструкторы базовых классов открытыми и виртуальными либо защищенными и невиртуальными

Резюме

Удалять или не удалять — вот в чем вопрос! Если следует обеспечить возможность удаления посредством указателя на базовый класс, то деструктор базового класса должен быть открытым и виртуальным. В противном случае он должен быть защищенным и невиртуальным.

Обсуждение

Это простое правило иллюстрирует достаточно тонкий вопрос и отражает современное использование наследования и принципов объектно-ориентированного проектирования.

Для данного базового класса `Base` вызывающий код может пытаться удалять производные от него объекты посредством указателей на `Base`. Если деструктор `Base` открытый и невиртуальный (свойства по умолчанию), он может быть случайно вызван для указателя, который в действительности указывает на производный объект, и в этом случае мы получим неопределенное поведение. Такое состояние дел привело к тому, что в старых стандартах кодирования требовалось делать деструкторы всех без исключения базовых классов виртуальными. Однако это уже перебор (даже если это и оправдано в общем случае); вместо этого следует использовать правило, согласно которому деструктор базового класса должен быть виртуальным тогда и только тогда, когда он открытый.

Написание базового класса представляет собой определение абстракции (см. рекомендации с 35 по 37). Вспомним, что для каждой функции-члена, участвующей в абстракции, вы должны решить:

- должна ли она вести себя виртуально или нет;
- должна ли она быть открыто доступна всему вызывающему коду посредством указателя на `Base` или она должна представлять собой скрытую внутреннюю деталь реализации.

Как описано в рекомендации 39, для обычных функций-членов следует сделать выбор между возможностью их вызова посредством указателя `Base*` невиртуально (но, возможно, с виртуальным поведением, если эти функции вызывают виртуальные функции, как, например, в шаблонах проектирования NVI или Template Method), виртуально либо невозможностью такого вызова вообще. Шаблон проектирования NVI представляет собой методику, которая позволяет обойтись без открытых виртуальных функций.

Деструкция может рассматриваться как просто одна из операций, хотя и со специальной семантикой, которая делает невиртуальные вызовы опасными или ошибочными. Следовательно, для деструктора базового класса выбор сужается до виртуального вызова через указатель `Base*` или невозможности вызова вообще; выбор невиртуального вызова в данном случае неприменим. Следовательно, деструктор базового класса виртуален, если он может быть вызван (т.е. открыт), и невиртуален в противном случае.

Заметим, что шаблон проектирования NVI не может быть применен к деструктору, поскольку конструкторы и деструкторы не могут осуществлять глубокие виртуальные вызовы (см. рекомендации 39 и 55).

Вывод: всегда явно пишите деструктор базового класса, поскольку неявно сгенерированный деструктор является открытым и невиртуальным.

Примеры

Клиентский код должен либо быть способен полиморфно удалять объекты посредством указателя на базовый класс, либо не должен этого делать. Каждый выбор определяет свой дизайн.

- *Пример 1. Базовые классы с полиморфным удалением.* Если должно быть разрешено полиморфное удаление, то деструктор должен быть открытым (в противном случае вызывающий код не сможет к нему обратиться) и должен быть виртуальным (в противном случае его вызов приведет к неопределенному поведению).
- *Пример 2. Базовые классы без полиморфного удаления.* Если полиморфное удаление не должно быть разрешено, деструктор должен не быть открытым (чтобы вызывающий код не мог к нему обратиться) и не должен быть виртуальным (потому что виртуальность ему не нужна).

Классы стратегий часто используются в качестве базовых для удобства, а не для обеспечения полиморфного поведения. Их деструкторы рекомендуется делать защищенными и невиртуальными.

Исключения

Некоторые компонентные архитектуры (например, COM или CORBA) не используют стандартный механизм удаления и применяют для этого различные собственные протоколы освобождения объектов. Следуйте шаблонам и идиомам этих архитектур и соответствующим образом адаптируйте данную рекомендацию.

Рассмотрим также еще один редкий случай.

- B одновременно является как базовым классом, так и конкретным классом, для которого могут создаваться объекты (так что деструктор этого класса должен быть открытым, чтобы было можно создавать и уничтожать объекты данного типа).
- У B нет виртуальных функций, и он не предназначается для полиморфного использования (так что хотя деструктор и открыт, он не обязан быть виртуальным).

Тогда несмотря на то, что деструктор оказывается открытым, имеется огромный соблазн не делать его виртуальным, поскольку такая первая виртуальная функция может привести к излишним расходам времени выполнения из-за добавления функциональности, которая нам совсем не нужна.

В этом редком случае вы можете сделать конструктор открытым и невиртуальным, но четко документировать, что производные объекты не могут использоваться полиморфно в качестве объектов базового класса. Именно это было сделано в случае класса `std::unary_function`.

В общем случае, однако, следует избегать конкретных базовых классов (см. рекомендацию 35). Например, `unary_function` представляет собой набор определений `typedef`, который не предназначен для создания объектов данного типа. В действительности бессмысленно давать ему открытый деструктор; лучше было бы последовать совету данной рекомендации и сделать его деструктор защищенным и невиртуальным.

Ссылки

[Cargill92] pp. 77-79, 207 • [Cline99] §21.06, 21.12-13 • [Henricson97] pp. 110-114 • [Koenig97] Chapters 4, 11 • [Meyers97] §14 • [Stroustrup00] §12.4.2 • [Sutter02] §27 • [Sutter04] §18

51. Деструкторы, функции освобождения ресурсов и обмена не ошибаются

Резюме

Все запуски этих функций должны быть успешными. Никогда не позволяйте ошибке выйти за пределы деструктора, функции освобождения ресурса (например, оператора `delete`) или функции обмена. В частности, типы, деструкторы которых могут генерировать исключения, категорически запрещено использовать со стандартной библиотекой C++.

Обсуждение

Перечисленные функции не должны генерировать исключений, так как они являются ключевыми для двух главных операций транзакционного программирования — отката при возникновении проблем в процессе работы и принятия результата работы, если проблем не возникло. Если нет способа безопасного возврата в предыдущее состояние при помощи операций, не генерирующих исключения, то невозможно реализовать бессбойный откат; отсутствие возможности безопасного сохранения изменения состояния при помощи операции, не генерирующей исключения, делает невозможной реализацию бессбойного принятия результата работы.

Рассмотрим следующие советы и требования, найденные в Стандарте C++.

> *Если деструктор, вызванный в процессе свертки стека, выходит с исключением, вызывается функция* `terminate` *(15.5.1). Поэтому деструкторы должны в общем случае перехватывать исключения и не позволять им распространиться за пределы деструктора.* — [C++03] §15.2(3)

> *В стандартной библиотеке C++ не определен ни один деструктор* [включая деструкторы любого типа, используемого для инстанцирования шаблона стандартной библиотеки], *генерирующий исключение.* — [C++03] §17.4.4.8(3)

Деструкторы являются специальными функциями, и компилятор автоматически вызывает их в различных контекстах. Если вы пишете класс — назовем его, к примеру, `Nefarious`[2] — деструктор которого может давать сбой (обычно посредством генерации исключения; см. рекомендацию 72), то вы столкнетесь с такими последствиями.

- *Объекты* `Nefarious` *трудно безопасно использовать в обычных функциях.* Вы не можете надежно инстанцировать автоматические объекты `Nefarious` в области видимости, если возможен выход из этой области видимости посредством исключения. Если это произойдет, деструктор `Nefarious` (вызываемый автоматически) может попытаться сгенерировать исключение, которое приведет к неожиданному завершению всей программы посредством вызова Терминатора —`std::terminate` (см. также рекомендацию 75).
- *Трудно безопасно использовать классы с членами или базовыми классами* `Nefarious`. Плохое поведение класса `Nefarious` распространяется на все классы, для которых он является базовым или у которых имеются члены этого типа.

[2] Nefarious — нечестивый, гнусный (англ.). — *Прим. перев.*

- *Вы не можете надежно создавать глобальные либо статические объекты* `Nefarious`. Исключение, которое может быть сгенерировано их деструкторами, невозможно перехватить.
- *Вы не можете надежно создавать массивы объектов* `Nefarious`. Коротко говоря, массивы при наличии деструкторов, которые могут генерировать исключения, обладают неопределенным поведением, поскольку в этой ситуации просто невозможно придумать способ разумного отката. (Подумайте сами: какой именно код должен сгенерировать компилятор для создания массива из десяти объектов `Nefarious`, если у четвертого объекта в конструкторе происходит генерация исключения, а при откате, когда вызываются деструкторы уже сконструированных объектов, один или несколько деструкторов генерируют исключения? Удовлетворительного ответа в этой ситуации просто нет.)
- *Вы не можете использовать объекты* `Nefarious` *в стандартных контейнерах.* Объекты Nefarious нельзя хранить в стандартных контейнерах или использовать их с какими-то другими частями стандартной библиотеки. Стандартная библиотека запрещает использование деструкторов, которые могут генерировать исключения.

Функции освобождения ресурсов, включая специальным образом перегруженные операторы `operator delete` и `operator delete[]`, попадают в ту же категорию, поскольку в общем случае они также используются в процессе "зачистки", в частности, в процессе обработки исключений.

Помимо деструкторов и функций освобождения ресурсов распространенные безопасные методики основаны на том, что операции обмена не генерируют исключений. В данном случае это связано не с использованием их в реализациях отката, а с их использованием в гарантированном принятии результатов работы. Например, вот идиоматическая реализация оператора `operator=` для некоторого типа `T`, который основан на выполнении копирующего конструктора, за которым следует вызов функции обмена, не генерирующей исключений:

```
T& T::operator=( const T& other ) {
    T temp( other );
    Swap( temp );
}
```

(см. также рекомендацию 56)

К счастью, область видимости сбоя при освобождении ресурса имеет определенно меньший размер. Если для сообщения об ошибке используются исключения, убедитесь, что рассматриваемые функции обрабатывают все возможные исключения и прочие ошибки, которые могут возникнуть при работе функций. (В случае исключений просто оберните все, что делает ваш деструктор, в блок `try/catch(...)`.) Это чрезвычайно важно, поскольку деструктор может быть вызван в кризисной ситуации, такой как сбой при выделении системного ресурса (например, памяти, файлов, блокировок, портов, окон или других системных объектов).

При использовании в качестве механизма обработки ошибок исключений следует документировать такое поведение, объявляя такие функции с закомментированной пустой спецификацией исключений `/* throw() */` (см. рекомендацию 75).

Ссылки

[C++03] §15.2(3), §17.4.4.8(3) • [Meyers96] §11 • [Stroustrup00] §14.4.7, §E.2-4 • [Sutter00] §8, §16 • [Sutter02] §18-19

52. Копируйте и ликвидируйте согласованно

Резюме

Если вы определили одну из следующего списка функций — копирующий конструктор, оператор копирующего присваивания или деструктор — вероятно, вам потребуется определить и обе оставшиеся функции (или по крайней мере одну из них).

Обсуждение

Если вам требуется определить одну из трех перечисленный функций, это означает, что вам требуется нечто большее, чем поведение этой функции по умолчанию, а все эти функции асимметрично взаимосвязаны.

- *Если вы пишете или запрещаете копирующий конструктор или оператор копирующего присваивания, то, вероятно, вы должны сделать то же самое и для другой функции из этой пары.* Если одна функция выполняет некоторую "специальную" работу, вероятно, другая должна делать что-то аналогичное, так как эти две функции должны иметь одинаковое действие (см. рекомендацию 53, которая поясняет этот пункт).
- *Если вы явно пишете копирующие функции, вероятно, вам надо явно написать и деструктор.* Если "специальная" работа в копирующем конструкторе заключается в выделении или дублировании некоторого ресурса (например, памяти, файла, сокета), то вы должны освободить этот ресурс в деструкторе.
- *Если вы явно пишете деструктор, вероятно, вам требуется явно написать (или явно запретить) копирование.* Если вам нужен нетривиальный деструктор, это зачастую связано с тем, что вам требуется вручную освободить ресурс, хранящийся объектом. Если так, вероятно, для этого ресурса требуется аккуратное дублирование, так что вам следует уделить внимание способу копирования и присваивания объектов, либо полностью запретить таковое.

Во многих случаях хранение корректно инкапсулированного ресурса с использованием идиомы RAII позволяет полностью избежать самостоятельной разработки указанных операций (см. рекомендацию 13).

Предпочитайте специальные члены, сгенерированные компилятором. Только они могут рассматриваться как "тривиальные", и как минимум один крупный производитель STL использует оптимизацию для классов с тривиальными специальными функциями. Похоже, вскоре это станет распространенным явлением.

Исключения

Когда любая из специальных функций объявлена только для того, чтобы сделать ее закрытой или виртуальной, но без специальной семантики, это не приводит к необходимости наличия остальных функций.

В редких случаях классы, имеющие члены странных типов (например, ссылки, `std::auto_ptr`), являются исключениями, поскольку они имеют необычную семантику копирования. В классе, хранящем ссылку или `auto_ptr`, вам, вероятно, потребуется написать копирующий конструктор и оператор присваивания, но деструктор по умолчанию будет работать корректно. (Заметим, что использование члена, являющегося ссылкой или `auto_ptr`, почти всегда ошибочно).

Ссылки

[Cline99] §30.01-14 • [Koenig97] §4 • [Stroustrup00] §5.5, §10.4 • [SuttHysl04b]

53. Явно разрешайте или запрещайте копирование

Резюме

Копируйте со знанием дела: тщательно выбирайте между использованием сгенерированных компилятором копирующего конструктора и оператора присваивания, написанием собственных версий или явным запрещением обоих, если копирование не должно быть разрешено.

Обсуждение

Распространенной ошибкой (и не только среди новичков) является игнорирование семантики копирования и присваивания при определении класса. Это характерно для маленьких вспомогательных классов, таких как предназначенные для поддержки идиомы RAII (см. рекомендацию 13).

Убедитесь, что ваш класс предоставляет осмысленное копирование (или не предоставляет его вовсе). Вот возможные варианты.

- *Явное запрещение обеих функций.* Если копирование для вашего типа лишено смысла, запретите как копирующее конструирование, так и копирующее присваивание, объявив их как закрытые нереализованные функции:

```
class T { // ...
private:                          // делаем T некопируемым
    T( const T& );                // функция не реализована
    T& operator=( const T& ); // функция не реализована
};
```

- *Явное написание обеих функций.* Если для объектов T предусмотрены копирование и копирующее присваивание, но корректное копирующее поведение отличается от поведения сгенерированных компилятором версий, то следует явно написать обе функции и сделать их не закрытыми.
- *Использование сгенерированных компилятором версий, предпочтительно с явным комментарием.* В противном случае, если копирование имеет смысл и поведение по умолчанию корректно, эти функции можно не объявлять самостоятельно и позволить компилятору самому сгенерировать их. Следует явно комментировать корректность поведения по умолчанию, чтобы читатели вашего кода знали, что вы преднамеренно не объявили данные функции.

Заметим, что запрещение копирования и копирующего присваивания означает, что вы не можете поместить объекты T в стандартные контейнеры. Это не обязательно плохо; очень может быть, что вы в любом случае не захотите хранить такие объекты в контейнерах. (Тем не менее, вы все равно можете поместить эти объекты в контейнер, если будете хранить их посредством интеллектуальных указателей; см. рекомендацию 79).

Вывод: будьте внимательны при работе с этими двумя операциями, так как компилятор имеет тенденцию к самостоятельной их генерации, а эти сгенерированные версии зачастую небезопасны для типов, не являющихся значениями (см. также рекомендацию 32).

Ссылки

[Dewhurst03] §88 • [Meyers97] §11 • [Stroustrup00] §11.2.2

54. Избегайте срезки. Подумайте об использовании в базовом классе клонирования вместо копирования

Резюме

Срезка объектов происходит автоматически, невидимо и может приводить к полному разрушению чудесного полиморфного дизайна. Подумайте о полном запрете копирующего конструктора и копирующего присваивания в базовых классах. Вместо них можно использовать виртуальную функцию-член `Clone`, если пользователям вашего класса необходимо получать полиморфные (полные, глубокие) копии.

Обсуждение

Когда вы строите иерархию классов, обычно она предназначена для получения полиморфного поведения. Вы хотите, чтобы объекты, будучи созданными, сохраняли свой тип и идентичность. Эта цель вступает в конфликт с обычной семантикой копирования объектов в C++, поскольку копирующий конструктор не является виртуальным и не может быть сделан таковым. Рассмотрим следующий пример:

```
class B { /* ... */ };

class D : public B { /* ... */ };

// Оп! Получение объекта по значению
void Transmogrify( B obj );

void Transubstantiate( B& obj ) { // Все нормально -
                                  // передача по ссылке
    Transmogrify( obj );          // Плохо! Срезка объекта!
    // ...
}

D d;
Transubstantiate( d );
```

Программист намерен работать с объектами `B` и производных классов полиморфно. Однако, по ошибке (или усталости — к тому же и кофе закончился...) программист или просто забыл написать `&` в сигнатуре `Transmogrify`, или собирался создать копию, но сделал это неверно. Код компилируется без ошибок, но когда функция `Transmogrify` вызывается с передачей ей объекта `D`, он мутирует в объект `B`. Это связано с тем, что передача по значению приводит к вызову `B::B(const B&)`, т.е. копирующего конструктора `B`, параметр которого `const B&` представляет собой автоматически преобразованную ссылку на `d`. Что приводит к полной потере динамического, полиморфного поведения, из-за которого в первую очередь и используется наследование.

Если, как автор класса `B`, вы хотите разрешить срезку, но не хотите, чтобы она могла происходить по ошибке, для такого случая существует один вариант действий, о котором мы упомянем для полноты изложения, но не рекомендуем использовать его в коде, к которому предъявляется требование переносимости: вы можете объявить копирующий конструктор `B` как `explicit`. Это может помочь избежать неявной срезки, но кроме этого запрещает все передачи параметров данного типа по значению (что может оказаться вполне приемлемым для базовых классов, объекты которых все равно не должны создаваться; см. рекомендацию 35).

```
// Объявляем копирующий конструктор как explicit (у данного
// решения имеется побочное действие, так что требуется
// улучшение этого метода)
class B { // ...
public:
    explicit B( const B& rhs );
};

class D : public B { /* ... */ };
```

Вызывающий код все равно в состоянии выполнить срезку, если это необходимо, но должен делать это явно:

```
void Transmogrify( B obj );  // Теперь эта функция вообще не
                             // может быть вызвана (!)
void Transmogrify2(const B& obj) // Идиома для намерения в
{                                // любом случае получить
    B b( obj );                  // параметр obj по значению
    // ...                       // (с возможной срезкой)
}

B b;              // Базовые классы не должны быть конкретными
D d;              // (см. рекомендацию 35), но допустим это
Transmogrify(b);  // Должна быть ошибка (см. примечание)
Transmogrify(d);  // Должна быть ошибка (см. примечание)
Transmogrify2(d); // Все в порядке
```

Примечание: на момент написания данной рекомендации некоторые компиляторы ошибочно допускали один или оба приведенных вызова функции `Transmogrify`. Эта идиома вполне стандартна, но (пока что) не полностью переносима.

Имеется лучший способ предупреждения срезки, с более высокой степенью переносимости. Пусть, например, функция наподобие `Transmogrify` в действительности хочет получить полную глубокую копию без информации о действительном производном типе переданного объекта. Более общее идиоматическое решение состоит в том, чтобы сделать копирующий конструктор базового класса защищенным (чтобы функция наподобие `Transmogrify` не могла случайно его вызвать), а вместо него воспользоваться виртуальной функцией `Clone`:

```
// Добавление функции Clone (уже лучше, но все еще требуется
// усовершенствование)
class B { // ...
public:
    virtual B* Clone() const = 0;
protected:
    B( const B& );
};

class D : public B { // ...
public:
    virtual D* Clone() const { return new D(*this); }
protected:
    D( const D& rhs ) : B( rhs ) { /* ... */ }
};
```

Теперь попытка срезки будет (переносимо) генерировать ошибку времени компиляции, а объявление функции `Clone` как чисто виртуальной заставляет непосредственный производный класс перекрыть ее. К сожалению, с данным решением все еще связаны две проблемы, которые компилятор не в состоянии обнаружить: в классе, производном от производного, функция `Clone` может оказаться неперекрытой, а перекрытие `Clone` может реализовать ее некорректно, так что копия будет не того же типа, что и оригинал. Функция `Clone` должна следовать шаблону проектирования Nonvirtual Interface (NVI; см. рекомендацию 39), который

разделяет открытую и виртуальную природы Clone и позволяет вам использовать ряд важных проверок:

```
class B { // ...
public:
  B* Clone() const { // Невиртуальная функция
    B* p = DoClone();
    assert(typeid(*p) == typeid(*this) &&
           "DoClone incorrectly overridden");
    return p;        // Проверка типа, возвращаемого DoClone
  }
protected:
    B( const B& );
private:
    virtual B* DoClone() const = 0;
};
```

Функция Clone теперь является невиртуальным интерфейсом, используемым вызывающим кодом. Производные классы должны перекрыть функцию DoClone. Дополнительная проверка обнаружит все копии, которые имеют тип, отличный от оригинала, тем самым оповещая, что в некотором производном классе не перекрыта функция DoClone; в конце концов, задача assert состоит именно в обнаружении и сообщении о таких программных ошибках (см. рекомендации 68 и 70).

Исключения

Некоторые проектные решения могут требовать, чтобы копирующие конструкторы базовых классов оставались открытыми (например, когда часть вашей иерархии представляет собой библиотеку стороннего производителя). В таком случае следует предпочесть передачу посредством (интеллектуального) указателя передаче по ссылке; как показано в рекомендации 25, передача посредством указателя существенно менее подвержена срезке и нежелательному созданию временных объектов.

Ссылки

[Dewhurst03] §30, §76, §94 • [Meyers96] §13 • [Meyers97] §22 • [Stroustrup94] §11.4.4 • [Stroustrup00] §12.2.3

55. Предпочитайте канонический вид присваивания

Резюме

При реализации оператора `operator=` предпочитайте использовать канонический вид — невиртуальный с определенной сигнатурой.

Обсуждение

Предпочтительно объявлять копирующее присваивание для типа `T` с одной из следующих сигнатур (см. [Stroustrup00] и [Alexandrescu03a]):

```
T& operator=( const T& ); // Классический вид
T& operator=( T );        // Потенциально оптимизированный
                          // вид (см. рекомендацию 27)
```

Второй вариант имеет смысл использовать, если вам в любом случае требуется копия аргумента в теле вашего оператора, как, например, при использовании идиомы, основанной на использовании функции обмена (см. рекомендацию 56).

Избегайте делать любой оператор присваивания виртуальным (см. [Meyers96] §33 и [Sutter04] §19). Если вы полагаете, что вам требуется виртуальное поведение присваивания, обратитесь сначала к указанной литературе. Если и после этого вы стоите на своем, то лучше использовать виртуальную именованную функцию, а не оператор (например, `virtual void Assign(const T&);`).

Не возвращайте `const T&`. Хотя этот тип возвращаемого значения имеет то преимущество, что защищает от странных присваиваний наподобие `(a=b)=c`, главным его недостатком является то, что вы не сможете поместить объекты типа `T` в контейнеры стандартной библиотеки; эти контейнеры требуют, чтобы оператор присваивания возвращал тип `T&`.

Всегда делайте копирующее присваивание безопасным в смысле исключений, причем предпочтительна строгая гарантия (см. рекомендацию 71).

Убедитесь, что ваш оператор присваивания безопасен в смысле присваивания самому себе. Избегайте написания оператора копирующего присваивания, который для корректной работы полагается на проверку присваивания самому себе; зачастую это говорит о недостаточной безопасности в смысле исключений. Если вы пишете копирующее присваивание с использованием идиомы обмена (см. рекомендацию 56), то вы автоматически обеспечиваете как строгую безопасность в смысле исключений, так и безопасность в смысле присваивания самому себе. Если присваивание самому себе часто встречается в программе из-за использования ссылочных синонимов или по каким-то иным причинам, проверка присваивания самому себе может использоваться в качестве средства оптимизации во избежание лишней работы.

Явно вызывайте все операторы присваивания базовых классов и всех данных-членов ([Meyers97] §16); обратите внимание, что идиома обмена автоматически заботится обо всех этих вещах. Возвращайте из оператора присваивания значение `*this` ([Meyers97] §15).

Ссылки

[Alexandrescu03a] • [Cargill92] pp41-42, 95 • [Cline99] §24.01-12 • [Koenig97] §4 • [Meyers96] §33 • [Meyers97] §17 • [Murray93] §2.2.1 • [Stroustrup00] §10.4.4.1, §10.4.6.3 • [Sutter00] §13, §38, §41 • [Sutter04] §19

56. Обеспечьте бессбойную функцию обмена

Резюме

Обычно имеет смысл предоставить для класса функцию `swap` в целях эффективного и бессбойного обмена внутреннего содержимого объекта с внутренним содержимым другого объекта. Такая функция может пригодиться для реализации ряда идиом, от простого перемещения объектов до реализации присваивания, легко обеспечивающего функцию принятия результатов работы со строгими гарантиями безопасности для вызывающего кода (см. также рекомендацию 51).

Обсуждение

Обычно функция `swap` выглядит примерно следующим образом (здесь `U` — некоторый пользовательский тип):

```
class T { // ...
public:
    void swap( T& rhs ) {
        member1_.swap( rhs.member1_ );
        std::swap( member2_, rhs.member2_ );
    }
private:
    U member1_;
    int member2_;
};
```

Для примитивных типов и стандартных контейнеров можно использовать `std::swap`. Другие классы могут реализовывать обмен в виде функций-членов с различными именами.

Рассмотрим использование `swap` для реализации копирующего присваивания посредством копирующего конструктора. Приведенная далее реализация оператора `operator=` обеспечивает строгую гарантию (см. рекомендацию 71), хотя и ценой создания дополнительного объекта, что может оказаться неприемлемым, если имеется более эффективный способ выполнения безопасного присваивания объектов типа `T`:

```
T& T::operator=(const T& other) { // Вариант 1 (традиционный)
    T temp( other );
    swap( temp );
    return *this;
}

T& T::operator=(T temp) { // Вариант 2 (см. рекомендацию 27)
    swap( temp );         // Обратите внимание на передачу
    return *this;         // temp по значению
}
```

Но что если тип `U` не имеет бессбойной функции обмена, как в случае многих существующих классов, но вам требуется поддержка функции обмена для типа `T`? Не все потеряно.

- Если копирующий конструктор и оператор копирующего присваивания `U` не дают сбоев, то с объектами типа `U` вполне справится `std::swap`.
- Если копирующий конструктор `U` может давать сбой, вы можете хранить (интеллектуальный) указатель на `U` вместо непосредственного члена. Указатели легко обмениваются. Следствием их применения являются дополнительные расходы на одно динамическое выделение памяти и дополнительную косвенность при обращении, но если вы храните все такие члены в едином Pimpl-объекте, то для всех закрытых членов дополнительные расходы вы понесете только один раз (см. рекомендацию 43).

Никогда не пользуйтесь трюком реализации копирующего присваивания посредством копирующего конструирования с использованием непосредственного вызова деструктора и размещающего new, несмотря на то, что такой трюк регулярно "всплывает" в форумах, посвященных C++ (см. также рекомендацию 99). Так что никогда не пишите:

```
T& T::operator=( const T& rhs ) { // плохо: анти-идиома
    if( this != &rhs ) {
        this->~T();               // Плохая методика!
        new (this) T( rhs );      // (см. [Sutter00] §41)
    }
    return *this;
}
```

Если объекты вашего типа можно обменять более эффективным способом, чем грубое присваивание, желательно предоставить функцию обмена, не являющуюся членом, в том же пространстве имен, где находится и ваш тип (см. рекомендацию 57). Кроме того, подумайте о специализации std::swap для ваших собственных нешаблонных типов:

```
namespace std {
    template<> void swap( MyType& lhs, MyType& rhs) {
        lhs.swap( rhs ); // для объектов MyType используется
    }                    // MyType::swap
}
```

Стандарт не позволяет вам сделать это, если MyType сам является шаблонным классом. К счастью, иметь такую специализацию хорошо, но не обязательно; основная методика состоит в обеспечении функции swap, эффективно работающей с данным типом, в виде функции, не являющейся членом класса, в том же пространстве имен, в котором находится и ваш тип.

Исключения

Обмен важен для классов с семантикой значения. Существенно менее важна она для базовых классов, поскольку эти классы в любом случае используются посредством указателей (см. рекомендации 32 и 54).

Ссылки

[C++03] §17.4.3.1(1) • [Stroustrup00] §E.3.3 • [Sutter00] §12-13, §41

Пространства имен и модули

Системы имеют подсистемы, которые в свою очередь состоят из подсистем и так до бесконечности — именно поэтому мы всегда движемся сверху вниз.

— Алан Перлис (Alan Perlis)

Пространство имен — очень важный инструмент для управления именами и снижения количества коллизий имен. То же относится и к модулям, которые, помимо этого, представляют собой инструментарий для работы с версиями. Мы определим модуль как отдельный компонент программы, содержащий тесно связанные между собой ее элементы (см. рекомендацию 5) и поддерживаемый одним и тем же программистом или группой; обычно модуль всегда компилируется одним и тем же компилятором с использованием одних и тех же опций. Модули имеются на разных уровнях детализации в широком диапазоне размеров. С одной стороны, модуль может быть минимального размера, представляя собой отдельный объектный файл, содержащий только один класс; с другой стороны, он может быть, например, отдельной динамической библиотекой, генерируемой из множества исходных файлов, содержимое которых образует подсистему внутри приложения большего размера или выпускается отдельно. Модуль может даже представлять собой огромную библиотеку, состоящую из множества небольших модулей (статических или динамических библиотек), содержащих тысячи разных типов. Несмотря на то, что такие библиотеки в стандарте C++ не упоминаются, программисты постоянно создают и используют библиотеки, и хорошо продуманная модуляризация является фундаментальной частью успешного управления зависимостями (см., например, рекомендацию 11).

Трудно представить себе программу значительного размера, которая не использует как пространства имен, так и модули. В этом разделе мы рассмотрим основные рекомендации по использованию двух этих взаимосвязанных инструментов, наряду с их взаимодействием с другими частями языка программирования и среды времени выполнения. Эти рекомендации помогут вам наиболее эффективно воспользоваться таким мощным инструментарием и избежать возможных неприятностей.

В этом разделе мы считаем наиболее значимой рекомендацию 58 — "Храните типы и функции в разных пространствах имен, если только они не предназначены для совместной работы".

57. Храните типы и их свободный интерфейс в одном пространстве имен

Резюме

Функции, не являющиеся членами и разработанные как часть интерфейса класса X (в особенности операторы и вспомогательные функции), должны быть определены в том же пространстве имен, что и X, что обеспечивает их корректный вызов.

Обсуждение

Открытый интерфейс класса образуют не только открытые функции-члены, но и функции, не являющиеся членами. Принцип Интерфейса гласит: для класса X все функции (включая функции, не являющиеся членами), которые "упоминают X" и "поставляются вместе с X" в одном и том же пространстве имен, являются логической частью X, поскольку образуют часть интерфейса X (см. рекомендацию 44 и [Sutter00]).

Язык C++ спроектирован с явным учетом Принципа Интерфейса. Причина, по которой в язык добавлен поиск, зависящий от аргумента (argument-dependent lookup — ADL), известный также как поиск Кёнига, заключается в том, чтобы обеспечить коду, использующему объект x типа X, возможность работать с частью его интерфейса, состоящей из функций, не являющихся членами (например, инструкция `cout << x` использует оператор `operator<<`, который не является членом класса X) так же легко, как и функции-члены (например, вызов `x.f()` не требует выполнения специального поиска, поскольку очевидно, что поиск `f` выполняется в области видимости X). ADL обеспечивает для свободных функций, которые получают объект X в качестве аргумента и поставляются вместе с определением X, ту же простоту использования, что и для функций-членов интерфейса X. Одним из главных мотивов принятия ADL был, в частности, класс `std::string` (см. [Sutter00]).

Рассмотрим класс X, определенный в пространстве имен N:

```
class X {
public:
    void f();
};

X operator+( const X&, const X& );
```

В вызывающей функции обычно пишется код наподобие `x3 = x1+x2`, где `x1`, `x2` и `x3` — объекты типа X. Если оператор `operator+` объявлен в том же пространстве имен, что и X, никаких проблем не возникает, и такой код отлично работает, поскольку оператор `operator+` будет легко найден с помощью ADL.

Если же оператор `operator+` не объявлен в том же пространстве имен, что и X, вызывающий код работать не будет. В этом случае имеется два способа заставить его заработать. Первый состоит в использовании явно квалифицированного оператора

```
x3 = N::operator+( x1, x2 );
```

Грустная картина — невозможность использовать естественный синтаксис оператора, который, собственно, и был главной целью введения перегрузки операторов в язык программирования. Другой способ заставить работать приведенный ранее код — использовать инструкцию `using`:

```
using N::operator+;
// или: using namespace N;
x3 = x1 + x2;
```

Применение `using` — совершенно нормальная и приемлемая вещь (см. рекомендацию 59), но все проблемы решаются гораздо проще, если автор `X` изначально поступает корректно и помещает оператор `operator+`, работающий с объектами `X`, в то же пространство имен, где находится `X`.

"Оборотная сторона" этого вопроса рассматривается в рекомендации 58.

Примеры

Пример 1. Операторы. Операторы работы с потоками `operator<<` и `operator>>` для объектов некоторого класса `X`, вероятно, относятся к наиболее ярким примерам функций, которые вполне очевидно являются частью интерфейса класса `X`, но при этом всегда представляют собой свободные функции (это обязательное условие, поскольку левый аргумент этих операторов — поток, а не объект `X`). Та же аргументация применима и к другим операторам, не являющимся членами `X`. Убедитесь, что ваши операторы находятся в том же пространстве имен, что и класс, с которым они работают. Если у вас есть возможность выбора, лучше делать операторы и все прочие функции не членами и не друзьями класса (см. рекомендацию 44).

Пример 2. Прочие функции. Если автор `X` предоставляет именованные вспомогательные функции, которые получают в качестве аргументов объекты `X`, они должны находиться в том же пространстве имен, что и `X`. В противном случае вызывающий код, использующий объекты `X`, будет не в состоянии работать с этими именованными функциями без явной квалификации их имен или применения инструкции `using`.

Ссылки

[Stroustrup00] §8.2, §10.3.2, §11.2.4 • [Sutter00] §31-34

58. Храните типы и функции в разных пространствах имен, если только они не предназначены для совместной работы

Резюме

Оберегайте ваши типы от непреднамеренного поиска, зависящего от аргументов (argument-dependent lookup — ADL, известный также как поиск Кёнига); однако преднамеренный поиск должен завершаться успешно. Этого можно добиться путем размещения типов в своих собственных пространствах имен (вместе с непосредственно связанными с ними свободными функциями; см. рекомендацию 57). Избегайте помещения типов в те же пространства имен, что и шаблоны функций или операторов).

Обсуждение

Следуя данному совету, вы сможете избежать трудно обнаруживаемых ошибок в вашем коде и необходимости разбираться с очень тонкими моментами языка, с которыми вы просто не должны сталкиваться.

Вот реальный пример, который был опубликован в группе новостей:

```
#include <vector>

namespace N {
    struct X { };
    template<typename T>
    int* operator+(T , unsigned) {/* Некоторые действия */}
}

int main() {
    std::vector<N::X> v(5);
    v[0];
}
```

Инструкция `v[0];` компилируется в некоторых реализациях стандартной библиотеки, но не во всех. Попробуем кратко пересказать эту длинную историю. Очень тонкая проблема связана с тем, что внутри большинства реализаций `vector<T>::operator[]` спрятан код наподобие `v.begin()+n`, и поиск имен для функции `operator+` может достичь пространства имен (в нашем случае `N`) типа, для которого инстанцирован вектор (в нашем случае `X`). Достигнет ли поиск этого пространства имен или нет — зависит от того, как определен `vector<T>::iterator` в данной версии реализации стандартной библиотеки. Однако если поиск достигает `N`, то здесь он находит `N::operator+`. Наконец, в зависимости от используемых типов, компилятор может просто посчитать, что для `vector<T>::iterator` оператор `N::operator+` имеет лучшее соответствие, чем оператор `std::operator+` из реализации стандартной библиотеки (который и должен был быть вызван). (Один из способов избежать такой неприятности в реализации стандартной библиотеки — не использовать код `v.begin()+n` таким образом, что он вносит непреднамеренную точку настройки: либо надо изменить код так, чтобы тип `v.begin()` никаким образом не зависел от параметра шаблона, либо вызов `operator+` следует переписать с указанием полного квалифицированного имени. См. рекомендацию 65.)

Коротко говоря, вряд ли вам удастся выявить истинную причину выводимого сообщения об ошибке. Если вам, конечно, повезет и вы получите это сообщение об ошибке, так как в случае невезения выбранный оператор `N::operator+` окажется, к несчастью, вполне подходящим

с точки зрения компилятора, и программа скомпилируется успешно, но вот результаты ее работы могут оказаться совершенно неожиданными...

Вы думаете, что вам не приходилось с этим сталкиваться? Попробуйте вспомнить, бывало ли такое в вашей практике, что ваш код, например, с использованием стандартной библиотеки приводил к удивительным и непонятным ошибкам компиляции? А после того как вы слегка меняли ваш код, порой просто меняя местами отдельные куски кода, все вдруг начинало работать и у вас оставалось только небольшое недоумение по поводу глупого компилятора, который запутался в трех строках? Практически все мы попадали в подобные ситуации, когда причиной неприятностей становилась рассматриваемая проблема, т.е. когда ADL находил имена из неподходящего пространства имен просто потому, что типы из этих пространств имен использовались поблизости друг от друга.

Эта проблема возникает не только при использовании стандартной библиотеки. В C++ с ней можно столкнуться (и это часто происходит на практике) при использовании типов, определенных в тех же пространствах имен, что и функции (в особенности шаблоны функций или операторы), не связанные с данными типами. Постарайтесь не попадаться в эту ловушку.

Основной вывод — вам не надо знать все эти тонкости. Простейший путь избежать этой категории проблем — это вообще избегать размещения свободных функций, не являющихся частью интерфейса типа X, в том же пространстве имен, где находится X, и в особенности никогда не помещать шаблоны функций или операторов в то же пространство имен, что и пользовательский тип.

Примечание. Да, стандартная библиотека C++ помещает алгоритмы и другие шаблоны функций, таких как `copy` или `distance`, в то же пространство имен, что и множество типов, таких как `pair` или `vector`. Все они находятся в одном пространстве имен. Это неудачное решение, которое вызывает описанные весьма тонкие и трудно локализуемые проблемы. К счастью, теперь у нас больше опыта и мы знаем, как следует поступать. Не делайте так, как сделано в стандартной библиотеке.

"Оборотная сторона" этого вопроса рассматривается в рекомендации 57.

Ссылки

[Stroustrup00] §10.3.2, §11.2.4 • [Sutter00] §34 • [Sutter02] §39-40

59. Не используйте `using` для пространств имен в заголовочных файлах или перед директивой `#include`

Резюме

Директива `using` для пространств имен создана для вашего удобства, а не для головной боли других. Никогда не используйте объявления или директивы `using` перед директивой `#include`.

Вывод: не используйте директивы `using` для пространств имен или `using`-объявления в заголовочных файлах. Вместо этого полностью квалифицируйте все имена. (Второе правило следует из первого, поскольку заголовочные файлы не могут знать, какие другие директивы `#include` могут появиться в тексте после них.)

Обсуждение

Вкратце: вы можете и должны свободно и без ограничений использовать объявления и директивы `using` в своих файлах реализации после директив `#include`. Несмотря на повторяющиеся заявления их противников, объявления и директивы `using` не являются злом и не противоречат цели пространств имен. Они просто делают пространства имен более удобными в использовании.

Пространства имен представляют мощное средство для устранения неоднозначности имен. В большинстве случаев различные программисты выбирают различные имена для своих типов и функций, но в том редком случае, когда они выбрали одинаковые имена, и они должны вместе использоваться в некотором коде, пространства имен позволяют избежать коллизий. Для этого достаточно, чтобы вызывающий код явно квалифицировал имя, указав, имя из какого именно пространства должно использоваться в том или ином случае. Однако в подавляющем большинстве случаев никакой неоднозначности имен не наблюдается. Поэтому вполне можно использовать директивы и объявления `using`, которые существенно облегчают использование пространств имен, снижая количество вводимого кода (программисту при использовании директив и объявлений `using` не требуется всякий раз явное упоминание того, к какому пространству имен принадлежит то или иное имя). В редких случаях коллизий имен директивы и объявления `using` не препятствуют указанию полностью квалифицированных имен для разрешения реально возникшей неоднозначности.

Однако директивы и объявления `using` предназначены только для вашего удобства и вы не должны использовать их так, чтобы они влияли на какой-то другой код. В частности, их нельзя употреблять где попало, где за ними может следовать еще какой-то сторонний код. В частности, их не следует использовать в заголовочных файлах (которые предназначены для включения в неограниченное количество файлов реализации — вы не должны вносить путаницу в значение кода в этих файлах) или перед директивой `#include` (тем самым вы по сути вносите их в текст этих заголовочных файлов).

Большинство программистов интуитивно понимают, почему директива `using` (например, `using namespace A;`) вызывает загрязнение в случае воздействия на код, следующий за ней и не осведомленный о наличии этой директивы: поскольку эта директива полностью импортирует одно пространство имен в другое, включая даже те имена, которые до сих пор не были видны, понятно, что это может легко изменить смысл следующего за директивой кода.

Но вот одна распространенная ошибка: многие считают, что использование объявления `using` на уровне пространства имен (например, `using N::Widget;`) вполне безопасно. Однако это не так. Такие объявления, как минимум, опасны, причем более тонким и хитрым способом. Рассмотрим следующий код:

```
// Фрагмент 1
namespace A {
    int f(double);
}

// Фрагмент 2
namespace B {
    using A::f;
    void g();
}

// Фрагмент 3
namespace A {
    int f(int);
}

// Фрагмент 4
void B::g() {
    f(1); // Какая перегрузка будет вызвана?
}
```

Здесь опасность заключается в том, что объявление `using` использует текущий список имен `f` в пространстве имен `A` в тот момент, когда это объявление встречается. Таким образом, какая именно перегрузка будет видима в пространстве имен `B`, зависит от того, где именно находится приведенный код фрагментов и в каком порядке он скомбинирован. (Здесь должен раздаться предупреждающий рев вашей внутренней сирены — "Зависимость от порядка есть зло!") Вторая перегрузка, `f(int)`, в большей степени соответствует вызову `f(1)`, но `f(int)` будет невидима для `B::g`, если ее объявление окажется после объявления `using`.

Рассмотрим два частных случая. Пусть фрагменты 1, 2 и 3 находятся в трех различных заголовочных файлах `s1.h`, `s2.h` и `s3.h`, а фрагмент 4 — в файле реализации `s4.cpp`, который включает указанные заголовочные файлы. Тогда семантика `B::g` зависит от порядка, в котором заголовочные файлы включены в `s4.cpp`! В частности:

- если `s3.h` идет перед `s2.h`, то `B::g` будет вызывать `A::f(int)`;
- иначе если `s1.h` идет перед `s2.h`, то `B::g` будет вызывать `A::f(double)`;
- иначе `B::g` не будет компилироваться вовсе.

В описанной ситуации имеется один вполне определенный порядок, при котором все работает так, как должно.

Давайте теперь рассмотрим ситуацию, когда фрагменты 1, 2, 3 и 4 находятся в четырех различных заголовочных файлах `s1.h`, `s2.h`, `s3.h` и `s4.h`. Теперь все становится существенно хуже: семантика `B::g` зависит от порядка включения заголовочных файлов не только в `s4.h`, но и в любой код, который включает `s4.h`! В частности, файл реализации `client_code.cpp` может пытаться включить заголовочные файлы в любом порядке:

- если `s3.h` идет перед `s2.h`, то `B::g` будет вызывать `A::f(int)`;
- иначе если `s1.h` идет перед `s2.h`, то `B::g` будет вызывать `A::f(double)`;
- иначе `B::g` не будет компилироваться вовсе.

Ситуация стала хуже потому, что два файла реализации могут включать заголовочные файлы *в разном порядке*. Что произойдет, если `client_code_1.cpp` включает `s1.h`, `s2.h` и `s4.h` в указанном порядке, а `client_code_2.cpp` включает в соответствующем порядке

s3.h, s2.h и s4.h? Тогда B::g нарушает правило одного определения (one definition rule — ODR), поскольку имеются две несогласующиеся несовместимые реализации, которые не могут быть верными одновременно: одна из них пытается вызвать A::f(int), а вторая — A::f(double).

Поэтому никогда не используйте директивы и объявления using для пространств имен в заголовочных файлах либо перед директивой #include в файле реализации. В случае нарушения этого правила вы несете ответственность за возможное изменение смысла следующего за using кода, например, вследствие загрязнения пространства имен или неполного списка импортируемых имен. (Обратите внимание на "директивы и объявления using *для пространств имен*". Указанное правило неприменимо при описании члена класса с помощью объявления using для внесения, при необходимости, имен из базового класса.)

Во всех заголовочных файлах, как и в файлах реализации до последней директивы #include, всегда используйте явные полностью квалифицированные имена. В файлах реализации после всех директив #include вы можете и должны свободно использовать директивы и объявления using. Это верный способ сочетания краткости кода с модульностью.

Исключения

Перенесение большого проекта со старой до-ANSI/ISO реализации стандартной библиотеки (все имена которой находятся в глобальном пространстве имен) к использованию новой (где практически все имена находятся в пространстве имен std) может заставить вас аккуратно разместить директиву using в заголовочном файле. Этот способ описан в [Sutter02].

Ссылки

[Stroustrup00] §9.2.1 • [Sutter02] §39-40

60. Избегайте выделения и освобождения памяти в разных модулях

Резюме

Золотое правило программиста — положи, где взял. Выделение памяти в одном модуле, а освобождение в другом делает программу более хрупкой, создавая тонкую дальнюю зависимость между этими модулями. Такие модули должны быть компилируемы одной и той же версией компилятора с одними и теми же флагами (в частности, отладочные версии и версии NDEBUG) и с одной и той же реализацией стандартной библиотеки; кроме того, с практической точки зрения лучше, чтобы модуль, выделяющий память, оставался загружен при ее освобождении.

Обсуждение

Разработчики библиотек хотят улучшить их качество, и, как прямое следствие, внутренние структуры данных и алгоритмы, используемые стандартными распределителями памяти, могут существенно различаться в разных версиях. Более того, к значительным изменениям во внутренней работе распределителей памяти могут приводить даже различные опции компилятора (например, включение или отключение отладочных возможностей).

Следовательно, о функции освобождения памяти (т.е. операторе `::operator delete` или функции `std::free`) при пересечении границ модулей практически нельзя строить какие-либо предположения, в особенности при пересечении границ модулей, при котором вы не можете гарантировать, что они будут скомпилированы одним и тем же компилятором C++ с одними и теми же опциями. Конечно, часто эти модули находятся в одном и том же файле проекта и компилируются с одними и теми же опциями, но комфорт часто приводит к забывчивости. В особенности высока цена такой забывчивости при переходе к динамически связываемым библиотекам, распределении большого проекта между несколькими группами или при замене модулей "на ходу" — в этом случае вы должны уделить максимум внимания тому, чтобы выделение и освобождение памяти выполнялось в пределах одного модуля или подсистемы.

Хорошим методом обеспечения освобождения памяти соответствующей функцией является использование `shared_ptr` (см. [C++TR104]). Интеллектуальный указатель `shared_ptr` со счетчиком ссылок может захватить свой "удалитель" в процессе конструирования. "Удалитель" — это функциональный объект (или обычный указатель на функцию), который выполняет освобождение памяти. Поскольку упомянутый функциональный объект, или указатель на функцию, является частью состояния объекта `shared_ptr`, модуль, выделивший память объекту, может одновременно определить функцию освобождения памяти, и эта функция будет корректно вызвана, даже если точка освобождения находится где-то в другом модуле — вероятно, относительно небольшой ценой (корректность важнее цены; см. также рекомендации 5, 6 и 8). Конечно, исходный модуль при этом должен оставаться загруженным.

Ссылки

[C++TR104]

61. Не определяйте в заголовочном файле объекты со связыванием

Резюме

Объекты со связыванием, включая переменные или функции уровня пространства имен, обладают выделенной для них памятью. Определение таких объектов в заголовочных файлах приводит либо к ошибкам времени компоновки, либо к бесполезному расходу памяти. Помещайте все объекты со связыванием в файлы реализации.

Обсуждение

Когда мы начинаем использовать C++, то все достаточно быстро уясняем, что заголовочный файл наподобие

```
// Избегайте определения объектов с внешним
// связыванием в заголовочном файле
int fudgeFactor;
string hello("Hello, world!");
void foo() { /* ... */ }
```

будучи включен больше чем в один исходный файл, ведет при компиляции к ошибкам дублирования символов во время компоновки. Причина проста: каждый исходный файл в действительности определяет и выделяет пространство для `fudgeFactor`, `hello` и тела `foo`, и когда приходит время сборки (компоновки, или связывания), компоновщик сталкивается с наличием нескольких объектов, которые носят одно и то же имя и борются между собой за видимость.

Решение проблемы простое — в заголовочный файл надо поместить только объявления:

```
extern int fudgeFactor;
extern string hello;
void foo(); // В случае объявления функции "extern"
            // является необязательным
```

Реальные же объявления располагаются в одном файле реализации:

```
int fudgeFactor;
string hello("Hello, world!");
void foo() { /* … */ }
```

Не определяйте в заголовочном файле и статические объекты уровня пространства имен, например:

```
// Избегайте определения объектов со статическим
// связыванием в заголовочном файле
static int fudgeFactor;
static string hello("Hello, world!");
static void foo() { /* ... */ }
```

Такое некорректное использование ключевого слова `static` более опасно, чем простое определение глобальных объектов в заголовочном файле. В случае глобальных объектов, по крайней мере, компоновщик в состоянии обнаружить наличие дублей. Но статические данные и функции могут дублироваться на законных основаниях, поскольку компилятор делает закрытую копию для каждого исходного файла. Так что если вы определите статические данные и статические функции в заголовочном файле и включите его в 50 файлов, то тела функций и пространство для данных в выходном исполняемом файле будут дублированы 50 раз (только если у вас не будет использован очень интеллектуальный компоновщик, который сможет распознать 50 одинаковых тел функций и наличие одинаковых константных данных,

которые можно безопасно объединить). Излишне говорить, что глобальные данные (такие как статические `fudgeFactor`) на самом деле не являются глобальными объектами, поскольку каждый исходный файл работает со своей копией таких данных, независимой от всех остальных копий в программе.

Не пытайтесь обойти эту ситуацию при помощи использования безымянных пространств имен в заголовочных файлах, поскольку результат будет ничуть не лучше:

```
// В заголовочном файле это приводит к тому же
// эффекту, что и использование static
namespace {
    int fudgeFactor;
    string hello("Hello, world!");
    void foo() { /* ... */ }
}
```

Исключения

В заголовочных файлах могут находиться следующие объекты с внешним связыванием.

- *Встраиваемые функции.* Они имеют внешнее связывание, но компоновщик гарантированно не отвергает многократные копии. Во всем остальном они ведут себя так же, как и обычные функции. В частности, адрес встраиваемой функции гарантированно будет единственным в пределах программы.
- *Шаблоны функций.* Аналогично встраиваемым функциям, инстанцирования ведут себя так же, как и обычные функции, с тем отличием, что их дубликаты приемлемы (и должны быть идентичны). Само собой разумеется, хороший компилятор устранит излишние копии.
- *Статические данные-члены шаблонов классов.* Они могут оказаться особенно сложными для компоновщика, но это уже не ваша проблема — вы просто определяете их в своем заголовочном файле и предоставляете сделать все остальное компилятору и компоновщику.

Кроме того, методика инициализации глобальных данных, известная как "Счетчики Шварца" ("Schwarz counters"), предписывает использование в заголовочном файле статических данных (или безымянных пространств имен). Джерри Шварц (Jerry Schwarz) использовал эту методику для инициализации стандартных потоков ввода-вывода `cin`, `cout`, `cerr` и `clog`, что и сделало ее достаточно популярной.

Ссылки

[Dewhurst03] §55 • [Ellis90] §3.3 • [Stroustrup00] §9.2, §9.4.1

62. Не позволяйте исключениям пересекать границы модулей

Резюме

Не бросайте камни в соседский огород — поскольку нет повсеместно распространенного бинарного стандарта для обработки исключений C++, не позволяйте исключениям пересекать распространяться между двумя частями кода, если только вы не в состоянии контролировать, каким компилятором и с какими опциями скомпилированы обе эти части кода. В противном случае может оказаться, что модули не поддерживают совместимые реализации распространения исключений. Обычно это правило сводится к следующему: не позволяйте исключениям пересекать границы модулей/подсистем.

Обсуждение

Стандарт C++ не определяет способ реализации распространения исключений, и не имеется никакого стандарта де-факто, признанного большинством систем. Механика распространения исключений варьируется не только в зависимости от операционной системы и компилятора, но и в зависимости от опций компиляции, использованных для компиляции данного модуля данным компилятором в данной операционной системе. Следовательно, приложение должно предотвращать несовместимость обработки исключений путем экранирования границ каждого из своих основных модулей, под которыми подразумеваются части приложения, для которых разработчик может гарантировать, что для их компиляции использован один и тот же компилятор и одни и те же опции компиляции.

Как минимум, ваше приложение должно обеспечить наличие заглушек `catch(...)` в перечисленных ниже местах, большинство из которых непосредственно связаны с модулями.

- *Вокруг `main`.* Перехватывайте и записывайте все исключения, которые иначе оказались бы неперехваченными и которые приводят к немедленному завершению работы вашей программы.
- *Вокруг функций обратного вызова из кода, который находится вне вашего контроля.* Операционные системы и библиотеки часто используют схему, при которой вы передаете указатель на функцию, которая будет вызвана позже (например, при некотором асинхронном событии). Не позволяйте исключениям распространиться за пределы вашей функции обратного вызова, поскольку вполне возможно, что код, вызывающий вашу функцию, использует иной механизм обработки исключений. Кстати говоря, он может вообще быть написан не на C++.
- *Вокруг границ потока.* В конечном итоге поток выполнения создается внутри операционной системы. Убедитесь, что ваша функция, представляющая поток, не преподнесет операционной системе сюрприз в виде исключения.
- *Вокруг границ интерфейса модуля.* Ваша подсистема предоставляет окружающему миру некоторый открытый интерфейс. Если подсистема представляет собой отдельную библиотеку, лучше, чтобы исключения оставались в ее границах, а для сообщения об ошибках использовались старые добрые коды ошибок (см. рекомендацию 72).
- *Внутри деструкторов.* Деструкторы не должны генерировать исключений (см. рекомендацию 51). Деструкторы, которые вызывают функции, способные генерировать исключения, должны защититься от возможной утечки этих исключений.

Убедитесь, что каждый модуль согласованно использует одну и ту же внутреннюю стратегию обработки ошибок (предпочтительно — исключения C++; см. рекомендацию 72) и единую стратегию обработки ошибок в интерфейсе (например, коды ошибок для API на языке C); обе эти стратегии могут быть одинаковы, но обычно это не так. Стратегии обработки ошибок могут изменяться только на границах модуля. Определите, как происходит связывание стратегий между модулями (например, как происходит взаимодействие с COM или CORBA, или что всегда следует перехватывать исключения на границе с API на языке C). Хорошим решением будет определить центральные функции, которые выполняют преобразования между исключениями и кодами ошибок, возвращаемых подсистемой. Так вы сможете легко транслировать входящие ошибки от других модулей в используемые внутренние исключения и тем самым упростить интеграцию.

Использование двух стратегий вместо одной выглядит избыточным, и вы можете поддаться соблазну отказаться от исключений и везде использовать только старые добрые коды ошибок. Но не забывайте, что обработка исключений имеет достоинства простоты использования и надежности, естественна для C++ и что избежать ее в нетривиальных программах на C++ невозможно (просто потому, что стандартный язык и библиотека генерируют исключения), так что вам следует предпочесть использовать исключения там, где это только возможно. Дополнительную информацию вы найдете в рекомендации 72.

Небольшое предостережение. Некоторые операционной системы используют механизм исключений C++ при обработке низкоуровневых системных ошибок, как, например, разыменование нулевого указателя. Следовательно, инструкция `catch(...)` может перехватить больше исключений, чем вы ожидаете, так что ваша программа может оказаться в неопределенной ситуации при выполнении перехвата `catch(...)`. Обратитесь к документации по вашей системе и либо приготовьтесь к обработке таких низкоуровневых исключений наиболее разумным способом, который сможете придумать, либо используйте в начале вашего приложения соответствующие системные вызовы для отключения такого поведения. Замена `catch(...)` последовательностью перехватов `catch(E1&){/*...*/} catch(E2&){/*...*/} ... catch(En&){/*...*/}` для всех известных типов базовых исключений `Ei` масштабируемым решением не является, поскольку вам придется обновлять этот список при добавлении новых библиотек (использующих собственные иерархии исключений) в ваше приложение.

Использование `catch(...)` в других, не перечисленных в данной рекомендации местах зачастую является признаком плохого проектирования, поскольку означает, что вы хотите перехватить абсолютно все исключения без обязательного знания о том, как следует обрабатывать конкретные исключения (см. рекомендацию 74). В хорошей программе не так много перехватов всех исключений, да и вообще инструкций `try/catch`; в идеале ошибки распространяются через весь модуль, транслируются на его границе (неизбежное зло) и обрабатываются в стратегически размещенных местах.

Ссылки

[Stroustrup00] §3.7.2, §14.7 • [Sutter00] §8-17 • [Sutter02] §17-23 • [Sutter04] §11-13

63. Используйте достаточно переносимые типы в интерфейсах модулей

Резюме

Не позволяйте типам появляться во внешнем интерфейсе модуля, если только вы не уверены в том, что все пользователи смогут корректно их понять и работать с ними. Используйте наивысший уровень абстракции, который в состоянии понять клиентский код.

Обсуждение

Чем более широко распространяется ваша библиотека, тем меньше ваш контроль над средами программирования, используемыми вашими клиентами, и тем меньше множество типов, которые ваша библиотека может надежно использовать в своем внешнем интерфейсе. Взаимодействие между модулями включает обмен бинарными данными. Увы, C++ не определяет стандартные бинарные интерфейсы; широко распространенные библиотеки для взаимодействия со внешним миром могут полагаться на такие встроенные типы, как `int` и `char`. Даже один и тот же тип на одном и том же компиляторе может оказаться бинарно несовместимым при компиляции с разными опциями.

Обычно либо вы полностью контролируете компилятор и опции компиляции, используемые для сборки модуля и его клиентов (и тогда вы можете использовать любой тип), либо вы не имеете такой возможности и должны использовать только типы, предоставляемые вычислительной платформой, или встроенные типы C++ (но даже в этом случае следует документировать размер и представление последних). В частности, использовать в интерфейсе модуля типы стандартной библиотеки можно только в том случае, если все другие модули, использующие данный, будут компилироваться в то же время и с теми же исходными файлами стандартной библиотеки.

Требуется найти определенный компромисс между проблемами используемых типов, которые могут не быть корректно восприняты всеми клиентами, и проблемами использования низкого уровня абстракции. Абстракция важна; если некоторые клиенты понимают только низкоуровневые типы и вы ограничены в использовании этими типами, то, возможно, следует подумать о дополнительных операциях, работающих с высокоуровневыми типами. Рассмотрим функцию `SummarizeFile`, которая получает в качестве аргумента файл. Имеется три варианта действий — передать указатель `char*` на строку в стиле C с именем файла; передать `string` с именем файла и передать объект `istream` или пользовательский объект `File`. Каждый из этих вариантов представляет свой уровень компромисса.

- *Вариант 1. `char*`.* Очевидно, что тип `char*` доступен наиболее широкому кругу клиентов. К сожалению, это также наиболее низкоуровневый вариант; в частности, он более проблематичен (например, вызывающий и вызываемый код должны явно решить, кто именно выделяет память для строки и кто ее освобождает), более подвержен ошибкам (например, файл может не существовать), и менее безопасен (например, может оказаться подвержен классической атаке, основанной на переполнении буфера).
- *Вариант 2. `string`.* Тип `string` доступен меньшему кругу клиентов, ограниченному использованием C++ и компиляцией с использованием той же реализации стандартной библиотеки, того же компилятора и совместимых настроек компилятора. Взамен мы получаем менее проблематичный (надо меньше беспокоиться об управлении памятью; однако см. рекомендацию 60) и более безопасный код (например, тип `string` увеличивает при необходимости свой буфер и не так подвержен атакам на основе переполнения буфера). Но и этот вариант относительно низкоуровневый, а потому так же открытый для ошибок, как и предыдущий (например, указанный файл может и не существовать).

- *Вариант 3.* `istream` *или* `File`. Если уж вы переходите к типам, являющимся классами, т.е. в любом случае требуется, чтобы клиент использовал язык программирования C++, причем тот же компилятор с теми же опциями компиляции, то воспользуйтесь преимуществами абстракции: класс `istream` (или пользовательский класс `File`, представляющий собой оболочку вокруг `istream`, позволяющую устранить зависимость от реализации стандартной библиотеки) повышает уровень абстракции и делает API существенно менее проблематичным. Функция получает объект типа `File` или соответствующий входной поток, она не должна заботиться об управлении памятью для строк, содержащих имя файла, и защищена от множества ошибок, которые вполне возможны при использовании первых двух вариантов. Остается только выполнить несколько проверок: файл должен быть открыт, а его содержимое иметь верный формат, но, в принципе, этим и ограничивается список неприятностей, которые могут произойти в данном варианте.

Даже если вы предпочтете воспользоваться во внешнем интерфейсе модуля низкоуровневой абстракцией, всегда используйте во внутренней реализации абстракции максимально высокого уровня и преобразуйте их в низкоуровневые абстракции на границах модуля. Например, если у вас имеются клиенты, не использующие C++, вы можете воспользоваться непрозрачным указателем `void*` или дескриптором типа `int` для работы с клиентом, но во внутренней реализации используйте высокоуровневые объекты. Преобразование между этими объектами и выбранными низкоуровневыми типами выполняйте только в интерфейсе модуля.

Примеры

Пример. Использование `std::string` *в интерфейсе модуля.* Пусть мы хотим, чтобы модуль предоставлял следующую функцию API:

```
std::string Translate( const std::string& );
```

Для библиотек, используемых внутри одной команды компании, это обычно неплохое решение. Но если вы планируете динамически компоновать данный модуль с вызывающим кодом, который использует иную реализацию `std::string` (например, иное размещение в памяти), то из-за такого несоответствия могут случиться разные странные и неприятные вещи.

Мы встречались с разработчиками, которые пытались использовать собственный класс-оболочку `CustomString` для объектов `std::string`, но в результате они сталкивались с той же проблемой, поскольку не имели полного контроля над процессом сборки всех клиентских приложений.

Одно из решений состоит в переходе к переносимым (вероятно, встроенным) типам, как вместо функции с аргументом `string`, так и в дополнение к ней. Такой новой функцией может быть функция

```
void Translate( const char* src, char* dest, size_t destSize );
```

Использование низкоуровневой абстракции более переносимо, но всегда добавляет сложности; здесь, например, как вызывающий, так и вызываемый код должны явно использовать обрезку строки, если размера буфера оказывается недостаточно. (Заметим, что данная версия использует буфер, выделяемый вызывающим кодом, для того чтобы избежать ловушки, связанной с выделением и освобождением памяти в разных модулях — см. рекомендацию 60.)

Ссылки

[McConnell93] §6 • [Meyers01] §15

Шаблоны и обобщенность

Место для вашей цитаты.

— Бьярн Страуструп (Bjarne Stroustrup),
[Stroustrup00] §13

Аналогично: место для вашего введения.

В этом разделе мы считаем наиболее значимой рекомендацию 64 — "Разумно сочетайте статический и динамический полиморфизм".

64. Разумно сочетайте статический и динамический полиморфизм

Резюме

Статический и динамический полиморфизм дополняют друг друга. Следует ясно представлять себе их преимущества и недостатки, чтобы использовать каждый из них там, где он дает наилучшие результаты, и сочетать их так, чтобы получить лучшее из обоих миров.

Обсуждение

Динамический полиморфизм предстает перед нами в форме классов с виртуальными функциями и объектов, работа с которыми осуществляется косвенно — через указатели или ссылки. Статический полиморфизм включает шаблоны классов и функций.

Полиморфизм означает, что данное значение может иметь несколько типов, а данная функция может принимать аргументы типов, отличающихся от точных типов ее параметров. "Полиморфизм представляет собой способ получить немного свободы динамической проверки типов, не теряя преимуществ статической проверки" — [Webber03].

Сила полиморфизма состоит в том, что один и тот же фрагмент кода может работать с разными типами, даже с теми, которые не были известны в момент написания этого кода. Такая "применимость задним числом" является краеугольным камнем полиморфизма, поскольку существенно увеличивает пригодность и возможность повторного использования кода (см. рекомендацию 37). (В противоположность этому мономорфный код работает только со строго конкретными типами, теми, для работы с которыми он изначально создавался.)

Динамический полиморфизм позволяет значению иметь несколько типов посредством открытого наследования. Например, `Derived*p` можно рассматривать как указатель не только на `Derived`, но и на объект любого типа `Base`, который прямо или косвенно является базовым для `Derived` (свойство категоризации). Динамический полиморфизм известен также как включающий полиморфизм, поскольку множество, моделируемое `Base`, включает специализации, моделируемые `Derived`.

Благодаря своим характеристикам динамический полиморфизм в C++ наилучшим образом подходит для решения следующих задач.

- *Единообразная работа, основанная на отношении надмножество/подмножество.* Работа с различными классами, удовлетворяющими отношению надмножество/подмножество (базовый/производный), может выполняться единообразно. Функция, работающая с объектом `Employee` (Служащий), будет работать и с объектами `Secretary` (Секретарь).
- *Статическая проверка типов.* В C++ все типы проверяются статически.
- *Динамическое связывание и раздельная компиляция.* Код, который использует иерархию классов, может компилироваться отдельно от этой иерархии. Это становится возможным благодаря косвенности, обеспечиваемой указателями (как на объекты, так и на функции).
- *Бинарная согласованность.* Модули могут компоноваться как статически, так и динамически, до тех пор, пока схемы виртуальных таблиц подчиняются одним и тем же правилам.

Статический полиморфизм посредством шаблонов также позволяет значению иметь несколько типов. Внутри шаблона

```
template<class T> void f( T t ) { /* ... */ }
```

`t` может иметь любой тип, который можно подставить в `f` для получения компилируемого кода. Это называется "неявным интерфейсом" в противоположность явному интерфейсу

базового класса. Таким образом достигается та же цель полиморфизма — написание кода, который работает с разными типами — но совершенно иным путем.

Статический полиморфизм наилучшим образом подходит для решения следующих задач.

- *Единообразная работа, основанная на синтаксическом и семантическом интерфейсе.* Работа с типами, которые подчиняются синтаксическому и семантическому интерфейсу, может выполняться единообразно. Интерфейсы в данном случае представляют синтаксическую сущность и не основаны на сигнатурах, так что допустима подстановка любого типа, который удовлетворяет данному синтаксису. Например, пусть дана инструкция `int i = p->f(5);`. Если `p` — указатель на класс `Base`, эта инструкция вызывает определенную функцию интерфейса, вероятно, `virtual int f(int)`. Но если `p` имеет обобщенный тип, то этот вызов может быть связан со множеством различных вещей, включая, например, вызов перегруженного оператора `operator->`, который возвращает тип, в котором определена функция `X f(double)`, где `X` — тип, который может быть преобразован в `int`.
- *Статическая проверка типов.* Все типы проверяются статически.
- *Статическое связывание (мешает раздельной компиляции).* Все типы связываются статически.
- *Эффективность.* Вычисления во время компиляции и статическое связывание позволяют достичь оптимизации и эффективности, недоступных при динамическом связывании.

Определите ваши приоритеты и используйте каждый вид полиморфизма там, где проявляются его сильные стороны.

Следует сочетать статический и динамический полиморфизм для того, чтобы получить преимущества обоих видов полиморфизма, а не для того, чтобы комбинировать их недостатки.

- *Статика помогает динамике.* Используйте статический полиморфизм для реализации динамически полиморфных интерфейсов. Например, у вас может быть абстрактный базовый класс `Command`, и вы определяете различные реализации в виде шаблона

  ```
  template</* ... */> class ConcreteCommand: public Command
  ```

  В качестве примеров можно привести реализации шаблонов проектирования Command и Visitor (см. [Alexandrescu01] и [Sutter04]).
- *Динамика помогает статике.* Обобщенный, удобный, статически связываемый интерфейс может использовать внутреннюю динамическую диспетчеризацию, что позволяет обеспечить одинаковую схему размещения объектов. Хорошими примерами могут служить реализации размеченных объединений (см. [Alexandrescu02b] и [Boost]) и параметр `Deleter` у `tr1::shared_ptr` (см. [C++TR104]).
- *Прочие сочетания.* Плохим является сочетание, при котором комбинируются слабые стороны обоих видов полиморфизма и результат получается хуже, чем при их отдельном использовании. Правильное сочетание должно комбинировать лучшее от обоих видов полиморфизма. Например, не помещайте виртуальные функции в шаблон класса, если только вы не хотите, чтобы каждый раз инстанцировались все виртуальные функции (в противоположность невиртуальным функциям шаблонных типов). В результате вы можете получить астрономический размер кода и чрезмерно ограничить ваш обобщенный тип, инстанцируя функциональность, которая никогда не используется.

Ссылки

[Alexandrescu01] §10 • [Alexandrescu02b] • [C++TR104] • [Gamma95] • [Musser01] §1.2-3, §17 • [Stroustrup00] §24.4.1 • [Sutter00] §3 • [Sutter02] §1 • [Sutter04] §17, §35 • [Vandevoorde03] §14 • [Webber03] §8.6

65. Выполняйте настройку явно и преднамеренно

Резюме

При разработке шаблона точки настройки должны быть написаны корректно, с особой тщательностью, а также ясно прокомментированы. При использовании шаблона необходимо четко знать, как именно следует настроить шаблон для работы с вашим типом, и выполнить соответствующие действия.

Обсуждение

Распространенная ловушка при написании библиотек шаблонов заключается в наличии непреднамеренных точек настройки, т.е. точек в вашем шаблоне, где может выполняться поиск пользовательского кода и его использование, но при написании такие действия вами не подразумевались. Попасть в такую ловушку очень легко — достаточно просто вызвать другую функцию или оператор обычным путем (без полной его квалификации), и если окажется, что один из его аргументов имеет тип параметра шаблона (или связанный с ним), то будет начат поиск такого кода, зависящий от аргумента. Примеров тому множество; в частности, см. рекомендацию 58.

Поэтому лучше использовать такие точки преднамеренно. Следует знать три основных пути обеспечения точек настройки в шаблоне, решить, какой именно способ вы хотите использовать в данном месте шаблона, и корректно его закодировать. Затем проверьте, не осталось ли в вашем коде случайных точек настройки там, где вы не предполагали их наличие.

Первый способ создания точки настройки — обычный "неявный интерфейс" (см. рекомендацию 64), когда ваш шаблон просто рассчитывает на то, что тип имеет соответствующий член с данным именем:

```
// Вариант 1. Создание точки настройки путем требования от
// типа T "foo-совместимости", т.е. наличия функции-члена с
// данным именем, сигнатурой и семантикой

template<typename T>
void Sample1( T t ) {
    t.foo();                    // foo - точка настройки
    typename T::value_type x; // Еще один пример: создание
}                               // точки настройки для поиска
                // типа (обычно создается посредством typedef)
```

Для реализации первого варианта автор `Sample1` должен выполнить следующие действия.

- *Вызвать функцию как член.* Просто используйте естественный синтаксис вызова функции-члена.
- *Документировать точку настройки.* Тип должен обеспечить доступную функцию-член `foo`, которая может быть вызвана с данными аргументами (в данном случае — без аргументов).

Второй вариант представляет собой использование метода "неявного интерфейса", но с функциями, не являющимися членами, поиск которых выполняется с использованием ADL[3] (т.е. ожидается, что данная функция находится в пространстве имен типа, для которого выполняется инстанцирование шаблона). Именно эта ситуация и явилась основной побудительной

[3] Поиск, зависящий от аргумента (см. стр. 118). — *Прим. перев.*

причиной для введения ADL (см. рекомендацию 57). Ваш шаблон рассчитывает на то, что для используемого типа имеется подходящая функция с заданным именем:

```
// Вариант 2: Создание точки настройки путем требования от
// типа T "foo-совместимости", т.е. наличия функции, не
// являющейся членом с данным именем, сигнатурой и
// семантикой, поиск которой выполняется посредством ADL.
// (Это единственный вариант, при котором не требуется поиск
// самого типа T.)
template<typename T>
void Sample2( T t ) {
    foo( t );  // foo - точка настройки
    cout << t; // Еще один пример - operator<< с записью в
}              // виде оператора представляет собой такую же
               // точку настройки
```

Для реализации варианта 2 автор `Sample2` должен выполнить следующие действия.

- *Вызвать функцию с использованием **неквалифицированного** имени (включая использование естественного синтаксиса в случае операторов) и убедиться, что шаблон не имеет функции-члена с тем же именем.* В случае шаблонов очень важно, чтобы вызов функции был не квалифицированным (например, не следует писать `SomeNamespace::foo(t)`) и чтобы у шаблона не было функции-члена с тем же именем, поскольку в обоих этих случаях поиск, зависящий от аргумента, выполняться не будет, что предотвратит поиск имени в пространстве имен, в котором находится тип T.
- *Документировать точку настройки.* Тип должен обеспечить наличие функции, не являющейся членом, которая может быть вызвана с данными аргументами.

Варианты 1 и 2 имеют одинаковые преимущества и применимость: пользователь может один раз написать соответствующую функцию настройки для своего типа и разместить ее там, где ее смогут найти и шаблоны других библиотек. Тем самым пользователь избегает необходимости писать множество мелких адаптеров для каждой библиотеки отдельно. Недостаток же заключается в том, что соответствующая семантика должна быть достаточно широко применима и иметь смысл для всех такого рода потенциальных применений (заметим, что в частности в эту категорию попадают операторы, что является еще одной причиной для рекомендации 26).

Третий вариант заключается в использовании специализации, когда ваш шаблон полагается на то, что пользовательский тип специализирует (при необходимости) некоторый иной предоставленный вами шаблон класса.

```
// Вариант 3: Создание точки настройки путем требования от
// типа T "foo-совместимости" путем специализации шаблона
// SampleTraits<> с предоставлением (обычно статической)
// функции с данным именем, сигнатурой и семантикой.
template<typename T>
void Sample3( T t ) {
    S3Traits<T>::foo( t );              // S3Traits<>::foo -
                                        // точка настройки
    typename S3Traits<T>::value_type x; // Другой пример -
}               // точка настройки для поиска типа (обычно
                // создается посредством typedef)
```

В этом варианте пользователь пишет адаптер, который гарантирует изолированность кода настройки для данной библиотеки в пределах этой библиотеки. Соответствующий недостаток заключается в том, что это может оказаться слишком громоздким решением; если несколько библиотек шаблонов требуют одну и ту же общую функциональность, пользователь должен будет писать несколько адаптеров, по одному для каждой библиотеки.

Для реализации этой версии автор `Sample3` должен выполнить следующие действия.

- *Предоставить шаблон класса по умолчанию в собственном пространстве имен шаблона.* Не используйте шаблоны функций, которые нельзя частично специализировать и которые приводят к перегрузкам и зависимостям от порядка (см. также рекомендацию 66).
- *Документировать точку настройки.* Пользователь должен специализировать `S3Traits` для своего собственного типа в пространстве имен библиотеки шаблонов, и документировать все члены `S3Traits` (например, `foo`) и их семантику.

При использовании любого из перечисленных вариантов следует также четко документировать семантику, требуемую от `foo`, в особенности все существенные действия (постусловия), которые должна гарантировать эта функция, и семантику сбоев (что именно происходит при сбое и каким образом должно осуществляться оповещение об ошибках).

Если точка настройки должна действовать и для встроенных типов, используйте варианты 2 и 3.

Варианты 1 и 2 следует предпочесть для тех общих операций, которые являются предоставляемыми типом сервисами. Для принятия данного решения попробуйте ответить на следующие вопросы: могут ли другие библиотеки шаблонов использовать данную возможность? является ли рассматриваемая семантика приемлемой для данного имени в общем случае? Если вы положительно ответили на эти вопросы, то, вероятно, вам действительно следует предпочесть один из этих вариантов.

Вариант 3 лучше использовать для менее общих операций, смысл которых может варьироваться. В таком случае в другом пространстве имен без каких-либо коллизий вы сможете придать тому же имени иной смысл.

Шаблон, в котором имеется несколько точек настройки, для каждой из них может выбрать свою стратегию, в наибольшей мере приемлемую в данном месте. Главное, что вы должны осознанно, с пониманием выбирать стратегию для каждой точки настройки, документировать требования к настройке (включая ожидаемые постусловия и семантику ошибок) и корректно реализовать выбранную вами стратегию.

Для того чтобы избежать непреднамеренных точек настройки, следует придерживаться следующих правил.

- *Размещайте все используемые вашим шаблоном вспомогательные функции в их собственном вложенном пространстве имен, и вызывайте их посредством полностью квалифицированных имен для запрета ADL.* Если вы вызываете вашу вспомогательную функцию и передаете ей объект типа параметра шаблона, и этот вызов не должен быть точкой настройки (т.е. вы всегда намерены вызывать вашу вспомогательную функцию, а не некоторую иную), то лучше поместить эту вспомогательную функцию во вложенное пространство имен и явно запретить ADL, полностью квалифицировав имя вызываемой функции или взяв его в скобки:

```
template<typename T>
void Sample4( T t ) {
    S4Helpers::bar( t ); // Запрет ADL: bar не является
                         // точкой настройки
    (bar)( t );          // Альтернативный способ
}
```

- *Избегайте зависимости от зависимых имен.* Говоря неформально, зависимое имя — это имя, которое каким-то образом упоминает параметр шаблона. Многие компиляторы не поддерживают "двухфазный поиск" для зависимых имен из стандарта C++, а это означает, что код шаблона, использующий зависимые имена, будет вести себя по-разному на разных компиляторах, если только не принять меры для полной определенности при использовании зависимых имен. В частности, особого внимания требует наличие *зависимых базовых классов*, когда шаблон класса наследуется от одного из параметров этого

шаблона (например, T в случае template<typename T>class C:T{};) или от типа, который построен с использованием одного из параметров шаблона (например, X<T> в случае template<typename T>class C:X<T>{};).

Коротко говоря, при обращении к любому члену зависимого базового класса необходимо всегда явно квалифицировать имя с использованием имени базового класса или при помощи this->. Этот способ можно рассматривать просто как некую магию, которая заставляет все компиляторы делать именно то, что вы от них хотите.

```
template<typename T>
class C : X<T> {
    typename X<T>::SomeType s; // Использование вложенного
                               // типа (или синонима
                               // typedef) из базового
                               // класса
public:
    void f() {
        X<T>::baz();           // Вызов функции-члена
                               // базового класса
        this->baz();           // Альтернативный способ
    }
};
```

Стандартная библиотека C++ в основном отдает предпочтение варианту 2 (например, ostream_iterator ищет оператор operator<<, а accumulate ищет оператор operator+ в пространстве имен вашего типа). В некоторых местах стандартная библиотека использует также вариант 3 (например, iterator_traits, char_traits) в основном потому, что эти классы свойств должны быть специализируемы для встроенных типов.

Заметим, что, к сожалению, стандартная библиотека C++ не всегда четко определяет точки настройки некоторых алгоритмов. Например, она ясно говорит о том, что трехпараметрическая версия accumulate должна вызывать пользовательский оператор operator+ с использованием второго варианта. Однако она не говорит, должен ли алгоритм sort вызывать пользовательскую функцию swap (обеспечивая таким образом преднамеренную точку настройки с использованием варианта 2), *может* ли он использовать пользовательскую функцию swap, и вызывает ли он функцию swap вообще; на сегодняшний день некоторые реализации sort используют пользовательскую функцию swap, в то время как другие реализации этого не делают. Важность рассматриваемой рекомендации была осознана совсем недавно, и сейчас комитет по стандартизации исправляет ситуацию, устраняя такие нечеткости из стандарта. Не повторяйте такие ошибки. (См. также рекомендацию 66.)

Ссылки

[Stroustrup00] §8.2, §10.3.2, §11.2.4 • [Sutter00] §31-34 • [Sutter04d]

66. Не специализируйте шаблоны функций

Резюме

При расширении некоторого шаблона функции (включая `std::swap`) избегайте попыток специализации шаблона. Вместо этого используйте перегрузку шаблона функции, которую следует поместить в пространство имен типа(ов), для которых разработана данная перегрузка (см. рекомендацию 57). При написании собственного шаблона функции также избегайте его специализации.

Обсуждение

Шаблоны функций вполне можно перегружать. Разрешение перегрузки рассматривает все первичные шаблоны и работает именно так, как вы и ожидаете, исходя из вашего опыта работы с перегрузкой обычных функций C++: просматриваются все видимые шаблоны и выбирается шаблон с наилучшим соответствием.

К сожалению, в случае специализации шаблона функции все оказывается несколько сложнее по двум основным причинам.

- *Специализировать шаблоны функций можно только полностью, но не частично.* Код, который выглядит как частичная специализация, на самом деле представляет собой перегрузку.
- *Специализации шаблона функции никогда не участвуют в перегрузке.* Таким образом, любая написанная вами специализация никак не повлияет на результат разрешения перегрузки и выбор используемого шаблона. Это противоречит интуитивно ожидаемому поведению разрешения перегрузки. Но, в конце концов, если вы напишете нешаблонную функцию с идентичной сигнатурой вместо специализации шаблона функции, то при разрешении перегрузки будет выбрана именно нешаблонная функция, как имеющая преимущество перед шаблоном.

Если вы пишете шаблон функции, то лучше писать его как единый шаблон, который никогда не будет специализирован или перегружен, и реализовывать шаблон функции через шаблон класса. Это и есть тот пресловутый дополнительный уровень косвенности, который позволяет обойти ограничения и миновать "темные углы" шаблонов функций. В этом случае программист, использующий ваш шаблон, сможет частично специализировать шаблон класса. Тем самым решается как проблема по поводу того, что шаблон функции не может быть частично специализирован, так и по поводу того, что специализации шаблона функции не участвуют в перегрузке.

Если вы работаете с каким-то иным старым шаблоном функции, в котором не использована описанная методика (т.е. с шаблоном функции, не реализованном посредством шаблона класса), и хотите написать собственную версию для частного случая, которая должна принимать участие в перегрузке, — делайте ее не специализацией, а обычной нешаблонной функцией (см. также рекомендации 57 и 58).

Примеры

Пример. `std::swap`. Базовый шаблон `swap` обменивает два значения `a` и `b` путем создания копии `temp` значения `a`, и присваиваний `a = b` и `b = temp`. Каким образом расширить данный шаблон для ваших собственных типов? Пусть, например, у вас есть ваш собственный тип `Widget` в вашем пространстве имен `N`:

```
namespace N {
    class Widget { /* ... */ };
}
```

Предположим, что имеется более эффективный путь обмена двух объектов `Widget`. Что вы должны сделать для того, чтобы он использовался стандартной библиотекой, — перегрузить `swap` (в том же пространстве имен, где находится `Widget`; см. рекомендацию 57) или непосредственно специализировать `std::swap`? Стандарт в данном случае невразумителен, и на практике используются разные методы (см. рекомендацию 65). Сегодня ряд реализаций корректно решают этот вопрос, предоставляя перегруженную функцию в том же пространстве имен, где находится `Widget`. Для представленного выше нешаблонного класса `Widget` это выглядит следующим образом:

```
namespace N {
    void swap( Widget&, Widget& );
}
```

Заметим, что если `Widget` является шаблоном

```
namespace N {
    template<typename T> class Widget { /* ... */ };
}
```

то специализация `std::swap` попросту невозможна, так как частичной специализации шаблона функции не существует. Лучшее, что вы можете сделать, — это добавить перегрузку функции

```
namespace ??? {
    template<typename T> void swap(Widget<T>&, Widget<T>&);
}
```

Это проблематичное решение, поскольку если вы помещаете эту функцию в пространство имен, в котором находится `Widget`, то многие реализации просто не в состоянии найти ее, но при этом стандарт запрещает располагать данную функцию в пространстве имен `std`. Эта проблема никогда бы не возникла, если бы стандарт либо указывал, что перегрузки надо искать и в пространстве имен типа шаблона, либо позволял помещать перегружаемые функции в пространство имен `std`, или (возвращаясь к основному вопросу данной рекомендации) прямо указывал, что `swap` должна реализовываться с использованием шаблона класса, который может быть частично специализирован.

Ссылки

[Austern99] §A.1.4 • [Sutter04] §7 • [Vandevoorde03] §12

67. Пишите максимально обобщенный код

Резюме

Используйте для реализации функциональности наиболее обобщенные и абстрактные средства.

Обсуждение

Когда вы пишете тот или иной код, используйте наиболее абстрактные средства, позволяющие решить поставленную задачу. Всегда думайте над тем, какие операции накладывают меньшее количество требований к интерфейсам, с которыми они работают. Такая привычка сделает ваш код более обобщенным, а следовательно, в большей степени повторно используемым и более приспособленным ко внесению изменений в его окружение.

И напротив, код, неоправданно привязанный к деталям, оказывается чрезмерно "жестким" и неспособным к повторному использованию.

- *Используйте для сравнения итераторов `!=` вместо `<`.* Оператор `!=` более общий и применим к большему классу объектов; оператор `<` требует упорядочения и может быть реализован только итераторами произвольного доступа. При использовании оператора `!=` ваш код проще переносится для работы с другими типами итераторов, такими как одно- и двунаправленные итераторы.
- *Лучше использовать итераторы, а не индексы.* Многие контейнеры не поддерживают индексный доступ; например, контейнер `list` не в состоянии эффективно реализовать его. Однако все контейнеры поддерживают итераторы. Таким образом, итераторы обеспечивают большую обобщенность кода, и при необходимости они могут использоваться совместно с индексным доступом.
- *Используйте `empty()` вместо `size() == 0`.* "Пуст/не пуст" — более примитивная концепция, чем "точный размер". Например, вы можете не знать размер потока, но всегда можете сказать о том, пуст он или нет; то же самое справедливо и для входных итераторов. Некоторые контейнеры, такие как `list`, реализуют `empty` более эффективно, чем `size`.
- *Используйте наивысший класс иерархии, предоставляющий необходимую вам функциональность.* При программировании с использованием динамических полиморфных классов не следует делать код зависимым от ненужных вам деталей и привязываться к определенным производным классам.
- *Будьте корректны при использовании `const` (см. рекомендацию 15).* Передача параметров `const&` накладывает меньше ограничений на вызывающий код, поскольку `const&` охватывает как константные, так и неконстантные объекты.

Исключения

В некоторых случаях применение индексов вместо итераторов позволяет компилятору лучше оптимизировать код. Однако перед тем как решиться на такой шаг, убедитесь, что вы действительно в нем нуждаетесь и что ваш компилятор действительно при этом лучше оптимизирует ваш код (см. рекомендацию 8).

Ссылки

[Koenig97] §12.7, §17-18 • [Meyers01] §4 • [Stroustrup00] §13, §17.1.1 • [Sutter04] §1, §5, §34

Обработка ошибок и исключения

Обработка ошибок — сложная задача, при решении которой программисту требуется вся помощь, которая только может быть предоставлена.

— Бьярн Страуструп (Bjarne Stroustrup), [Stroustrup94] §16.2

Имеется три способа написать программу без ошибок; но работает только третий способ.

— Алан Перлис (Alan Perlis)

Вопрос не в том, будем ли мы делать программные ошибки. Вопрос в том, будем ли мы что-либо предпринимать, чтобы позволить компилятору и другим используемым инструментам их обнаружить.

В этом разделе документированы добытые трудом множества программистов знания и наилучшие практические подходы, некоторые из которых стоили многих лет работы. Следуйте приведенным правилам и рекомендациям. Когда мы пишем сложную, надежную и безопасную программу — нам требуется вся помощь, которую мы только в состоянии получить.

В этом разделе мы считаем наиболее значимой рекомендацию 69 — "Определите разумную стратегию обработки ошибок и строго ей следуйте".

68. Широко применяйте assert для документирования внутренних допущений и инвариантов

Резюме

Используйте `assert` или его эквивалент для документирования внутренних допущений в модуле (т.е. там, где вызываемый и вызывающий код поддерживаются одним и тем же программистом или командой), которые должны всегда выполняться (в противном случае они являются следствием программной ошибки; например, нарушение постусловий функции, обнаруженное вызывающим кодом). (См. также рекомендацию 70.) Убедитесь, что использование `assert` не приводит к побочным действиям.

Обсуждение

Очень трудно найти ошибку в своем коде, когда вы ищете ее;
но во сто крат труднее найти ее, если вы считаете,
что ее там нет.

— Стив Мак-Коннелл (Steve McConnell)

Трудно переоценить всю мощь `assert`. Этот макрос и его альтернативы, такие как шаблоны проверки времени компиляции (или, что несколько хуже, времени выполнения), представляют собой неоценимый инструментарий для обнаружения и отладки программных ошибок при работе над проектами. Среди прочих инструментов у них, пожалуй, наилучшее отношение сложность/эффективность.

Рассматриваемые проверки обычно генерируют код только в режиме отладки (когда не определен макрос `NDEBUG`), так что от них можно освободиться при сборке окончательной версии программы. Широко используйте проверки в своих программах, но никогда не пишите выражений в `assert`, которые могут иметь побочное действие. При построении окончательной версии, когда будет определен макрос `NDEBUG`, проверки не будут генерировать никакого кода:

```
assert( ++i < limit ); // Плохо: i увеличивается только в
                       // отладочном режиме
```

Согласно теории информации, количество информации, заключающееся в событии, обратно пропорционально вероятности данного события. То есть чем менее вероятно, что какая-то проверка сработает, тем больше информации вы получите, когда она сработает.

Избегайте применения `assert(false)`, лучше использовать `assert(!"информационное сообщение")`. Большинство компиляторов вставят строку в вывод сообщения об ошибке. Подумайте также о добавлении `&&"информационное сообщение"` к более сложным проверкам вместо комментария.

Рассмотрим определение вашего собственного `assert`. Стандартный макрос `assert` просто бесцеремонно завершает вашу программу с выводом сообщения в стандартный поток вывода. Ваша среда, вероятно, обладает расширенными возможностями отладки; пусть, например, она в состоянии автоматически запустить интерактивный отладчик. В этом случае вы можете определить собственный макрос `MYASSERT` и использовать его. Может также оказаться полезным оставить большинство проверок даже в окончательной версии программы (лучше не отключать проверки по соображениям эффективности, пока необходимость этого

отключения не будет точно доказана; см. рекомендацию 8), так что существенные преимущества может предоставить наличие различных "уровней проверки", некоторые из которых могут оставаться активными и в окончательной версии программы.

Проверки зачастую связаны с условиями, которые можно было бы протестировать во время компиляции, если бы язык был достаточно выразителен для этого. Например, ваш проект может полагаться на то, что каждый объект класса `Employee` имеет ненулевой идентификатор `id_`. В идеале компилятор мог бы анализировать конструктор `Employee` и его члены и доказать при помощи статического анализа, что указанное условие всегда выполняется. В реальной ситуации вы можете использовать `assert(id_!=0)` в реализации `Employee`:

```
unsigned int Employee::GetID() {
    assert(id_!=0 && "Employee ID должен быть ненулевым") ;
    return id_;
}
```

Не используйте `assert` для сообщения об ошибках времени выполнения (см. рекомендации 70 и 72). Например, не следует применять `assert`, чтобы убедиться в корректной работе `malloc`, успешном создании окна или запуске потока программы. Однако можно использовать `assert`, чтобы убедиться, что API работает так, как документировано. Например, если вы вызываете некоторую функцию API, в документации на которую сказано, что она всегда возвращает положительное значение, но вы подозреваете наличие в ней ошибки — после вызова этой функции можно воспользоваться `assert` для проверки выполнения постусловия.

Не рекомендуется вместо проверок генерировать исключения, несмотря на то, что именно для этой цели был разработан стандартный класс `std::logic_error`. Главный недостаток использования исключений для сообщения о программных ошибках состоит в том, что при этом не требуется свертка стека — желательно вызвать отладчик именно в той строке, где обнаружено нарушение, с полным сохранением состояния программы.

Резюмируя: имеются ошибки, о которых вы знаете, что они могут произойти (см. рекомендации с 69 по 75). Все остальные ошибки произойти не должны, и если это все же случается — то это ошибка программиста. Для таких ошибок имеется `assert`.

Примеры

Пример. Проверка базовых допущений. Все мы сталкивались с ситуациями, когда происходило что-то, что "ну никак не может произойти". Но часто даже собственный опыт мало чему учит, и через некоторое время опять начинается — "это значение может быть только положительным!", "совершенно очевидно, что этот указатель не нулевой!"... Разработка программного обеспечения — работа сложная, и в программе, в которую вносятся изменения, может произойти все, что угодно. Проверки предназначены для того, чтобы убедиться в справедливости ваших предположений. Не стесняйтесь проверять тавтологии, которые не в состоянии обеспечить система:

```
string Date::DayOfWeek() const {
    // Проверка инвариантов
    assert( day_ > 0 && day_ <= 31 );
    assert( month_ > 0 && month_ <= 12 );
    // ...
}
```

Ссылки

[Abrahams01b] • [Alexandrescu03b] • [Alexandrescu03c] • [Allison98] §13 • [Cargill92] pp. 34-35 • [Cline99] §10.01-10 • [Dewhurst03] §28 • [Keffer95] pp. 24-25 • [Lakos96] §2.6, §10.2.1 • [McConnell93] §5.6 • [Stroustrup00] §24.3.7, §E.2, §E.3.5, §E.6 • [Sutter00] §47

69. Определите разумную стратегию обработки ошибок и строго ей следуйте

Резюме

Еще на ранней стадии проектирования разработайте практичную, последовательную и разумную стратегию обработки ошибок и строго следуйте ей. Убедитесь, что ваша стратегия включает следующее.

- *Идентификация*: какие условия являются ошибкой.
- *Строгость*: насколько важна каждая ошибка.
- *Обнаружение*: какой код отвечает за обнаружение ошибки.
- *Распространение*: какой механизм используется для описания и распространения уведомления об ошибке в каждом модуле.
- *Обработка*: какой код отвечает за выполнение действий, связанных с ошибкой.
- *Уведомление*: каким образом информация об ошибке вносится в журнальный файл или производится уведомление пользователя программы.

Изменяйте механизмы обработки ошибок только в пределах границ модуля.

Обсуждение

В этой рекомендации мы рассматриваем ошибки времени выполнения, возникновение которых не связано с неверным кодированием (таким ошибкам посвящена рекомендация 68).

Определите стратегию сообщения об ошибках и их обработки для вашего приложения и для каждого модуля или подсистемы, и строго следуйте ей. Стратегия должна включать, как минимум, следующие пункты.

Везде.

- *Определение ошибок*. Для каждой сущности (например, для каждой функции, класса, модуля) документируйте внутренние и внешние инварианты.

Для каждой функции.

- *Определение ошибок*. Для каждой функции документируйте ее пред- и постусловия, инварианты, за которые она отвечает, и гарантии безопасности, которые она поддерживает (см. рекомендации 70 и 71). Заметим, что деструкторы и функции освобождения ресурсов должны всегда поддерживать гарантию бессбойности, поскольку в противном случае часто невозможно надежно и безопасно выполнить освобождение захваченных ресурсов (см. рекомендацию 51).

Для каждой ошибки (см. определение "ошибки" в рекомендации 70).

- *Серьезность ошибки и категоризация*. Для каждой ошибки определите уровень серьезности. Желательно предоставить способ тонкой настройки диагностики для определенных категорий и уровней ошибок.
- *Обнаружение ошибок*. Для каждой ошибки документируйте, какой именно код отвечает за ее обнаружение, следуя советам рекомендации 70.
- *Обработка ошибок*. Для каждой ошибки определите код, который отвечает за ее обработку, следуя советам рекомендации 74.

- *Уведомление об ошибках*. Для каждой ошибки определите соответствующий метод уведомления. Сюда входят запись на диск в журнальный файл, распечатка или даже отправка SMS на мобильный телефон администратора.

Для каждого модуля.

- *Передача ошибки*. Для каждого модуля (обратите внимание: для каждого модуля, а не для каждой ошибки) определите механизм, который будет использоваться для передачи информации об ошибке (например, исключения C++, исключения COM, исключения CORBA, коды возврата).

Мы уже подчеркивали, что стратегия обработки ошибок может изменяться только на границах модулей (см. рекомендации 62 и 63). Каждый модуль должен последовательно использовать единую стратегию обработки ошибок внутри модуля (например, модули, написанные на C++, должны использовать исключения; см. рекомендацию 72), и последовательно пользоваться единой, хотя, возможно, иной стратегией обработки ошибок для своего интерфейса (например, модуль может предоставлять обычный API на языке C, чтобы обеспечить возможность его использования кодом, написанном на разных языках программирования; или использовать оболочку COM и, соответственно, исключения COM).

Все функции, являющиеся точками входа в модуль, непосредственно отвечают за преобразование между внутренней и внешней стратегиями, если они различны. Например, в модуле, который внутренне использует исключения C++, но предоставляет интерфейс в стиле C API, все функции интерфейса должны содержать перехват `catch(...)` всех исключений и преобразовывать их в коды ошибок.

Обратите внимание, в частности, на то, что функции обратного вызова и функции потоков по определению являются (или могут быть) границами модуля. Тело каждой функции обратного вызова или функции потока должно преобразовывать внутренний механизм ошибок в механизм, использующийся стратегией интерфейса (см. рекомендацию 62).

Ссылки

[Abrahams01b] • [Allison98] §13 • [McConnell93] §5.6 • [Stroustrup94] §16.2, §E.2 • [Stroustrup00] §14.9, §19.3.1 • [Sutter04b]

70. Отличайте ошибки от ситуаций, не являющихся ошибками

Резюме

Функция представляет собой единицу работы. Таким образом, сбои следует рассматривать либо как ошибки, либо как штатные ситуации, в зависимости от их влияния на функции. В функции `f` сбой является ошибкой тогда и только тогда, когда он нарушает одно из предусловий `f`, не позволяет выполнить предусловие вызываемой ею функции, препятствует достижению собственных постусловий `f` или сохранению инварианта, за поддержку которого отвечает функция `f`.

В частности, в этой рекомендации мы исключаем внутренние программные ошибки (т.е. те, где за вызывающий и вызываемый код отвечает один и тот же человек или команда, например, в пределах одного модуля). Они представляют собой отдельную категорию ошибок, для работы с которой используется такое средство, как проверки (см. рекомендацию 68).

Обсуждение

Очень важно четко различать ошибки и ситуации, не являющиеся ошибками в плане их влияния на работу функций, в особенности в целях определения гарантий безопасности (см. рекомендацию 71). Ключевыми словами данной рекомендации являются *предусловие*, *постусловие* и *инвариант*.

Функция представляет собой базовую единицу работы, независимо от того, программируете ли вы на C++ в структурном, объектно-ориентированном или обобщенном стиле. Функция делает определенные предположения о начальном состоянии (предусловия, за выполнение которых несет ответственность вызывающий код, а за проверку — вызываемый) и выполняет одно или несколько действий (документируемых, как результат выполнения функции, или ее постусловия, за выполнение которых несет ответственность данная функция). Функция может (наряду с другими функциями) нести ответственность за поддержание одного или нескольких инвариантов. В частности, не закрытая неконстантная функция-член по определению представляет собой единицу работы над объектом, и должна перевести объект из одного корректного, сохраняющего инвариант состояния в другое. Во время выполнения тела функции-члена инвариант объекта может (и практически всегда должен) нарушаться, и это вполне нормальная ситуация, лишь бы по окончании работы функции-члена инвариант вновь выполнялся. Функции более высокого уровня объединяют функции более низкого уровня в большие единицы работы.

Ошибкой является любой сбой, который не дает функции успешно завершиться. Имеется три вида ошибок.

- *Нарушение или невозможность достижения предусловия*. Функция обнаруживает нарушение одного из своих собственных предусловий (например, ограничения, накладываемого на параметр или состояние) или сталкивается с условием, которое не позволяет достичь выполнения предусловия для некоторой другой неотъемлемой функции, которая должна быть вызвана.
- *Неспособность достичь постусловия*. Функция сталкивается с ситуацией, которая не позволяет ей выполнить одно из ее собственных постусловий. Если функция возвращает значение, получение корректного возвращаемого значения является ее постусловием.

- *Неспособность восстановления инварианта.* Функция сталкивается с ситуацией, которая не позволяет ей восстановить инвариант, за поддержку которого она отвечает. Это частный случай постусловия, который в особенности относится к функциям-членам. Важнейшим постусловием всех не закрытых функций-членов является восстановление инварианта класса (см. [Stroustrup00] §E.2.)

Все прочие ситуации ошибками не являются, и, следовательно, уведомлять о них, как об ошибках, не требуется (см. примеры к данной рекомендации).

Код, который может вызвать ошибку, отвечает за ее обнаружение и уведомление о ней. В частности, вызывающий код должен обнаружить и уведомить о ситуации, когда он не в состоянии выполнить предусловия вызываемой функции (в особенности если для вызываемой функции документировано отсутствие проверок с ее стороны; так, например, оператор `vector::operator[]` не обязан выполнять проверку попадания аргумента в корректный интервал значений). Однако поскольку вызываемая функция не может полагаться на корректность работы вызывающего кода, желательно, чтобы она выполняла собственные проверки предусловий и уведомляла об обнаруженных нарушениях, генерируя ошибку (или, если функция является внутренней для (т.е. вызываемой только в пределах) модуля, то нарушение предусловий по определению является программной ошибкой и должно обрабатываться при помощи `assert` (см. рекомендацию 68)).

Добавим пару слов об определении предусловий функций. Условие является предусловием функции `f` тогда и только тогда, когда имеются основания ожидать, что весь вызывающий код проверяет выполнение данного условия перед вызовом функции `f`. Например, было бы неверно полагать предусловием нечто, что может быть проверено только путем выполнения существенной работы самой функцией, либо путем доступа к закрытой информации. Такая работа должна выполняться в функции и не дублироваться вызывающим ее кодом.

Например, функция, которая получает объект `string`, содержащий имя файла, обычно не должна делать условие существования файла предусловием, поскольку вызывающий код не в состоянии надежно гарантировать, что данный файл существует, не используя блокировки файла (при проверке существования файла без блокировки другой пользователь или процесс могут удалить или переименовать этот файл в промежутке между проверкой существования файла вызывающим кодом и попыткой открытия вызываемым кодом). Единственный корректный способ сделать существование файла предусловием — это потребовать, чтобы вызывающий код открыл его, а параметром функции сделать `ifstream` или его эквивалент (что к тому же безопаснее, поскольку работа при этом выполняется на более высоком уровне абстракции; см. рекомендацию 63), а не простое имя файла в виде объекта `string`. Многие предусловия таким образом могут быть заменены более строгим типизированием, которое превратит ошибки времени выполнения в ошибки времени компиляции (см. рекомендацию 14).

Примеры

Пример 1. `std::string::insert` *(ошибка предусловия).* При попытке вставить новый символ в объект `string` в определенной позиции `pos`, вызывающий код должен проверить корректность значения `pos`, которое не должно нарушать документированные требования к данному параметру; например, чтобы не выполнялось соотношение `pos > size()`. Функция `insert` не может успешно выполнить свою работу, если для нее не будут созданы корректные начальные условия.

Пример 2. `std::string::append` *(ошибка постусловия).* При добавлении символа к объекту `string` сбой при выделении нового буфера, если заполнен существующий, не позволит функции выполнить документированные действия и получить документированные же постусловия, так что такой сбой является ошибкой.

Пример 3. Невозможность получения возвращаемого значения (ошибка постусловия). Получение корректного возвращаемого объекта является постусловием для функции, которая возвращает значение. Если возвращаемое значение не может быть корректно создано (например, если функция возвращает `double`, но значение `double` с требуемыми математическими свойствами не существует), то это является ошибкой.

Пример 4. `std::string::find_first_of` (не ошибка в контексте `string`). При поиске символа в объекте `string`, невозможность найти искомый символ — вполне законный итог поиска, ошибкой не являющийся. Как минимум, это не ошибка при работе с классом `string` общего назначения. Если владелец данной строки предполагает, что символ должен наличествовать в строке, и его отсутствие, таким образом, является ошибкой в соответствии с высокоуровневым инвариантом, то высокоуровневый вызываемый код должен соответствующим образом уведомить об ошибке инварианта.

Пример 5. Различные условия ошибок в одной функции. Несмотря на увеличивающуюся надежность дисковых носителей, запись на диск традиционно сопровождается ожиданием ошибок. Если вы разрабатываете класс `File`, в одной-единственной функции `File::Write(const char*buffer, size_t size)`, которая требует, чтобы файл был открыт для записи, а указатель `buffer` имел ненулевое значение, вы можете предпринимать следующие действия.

- *Если `buffer` равен `NULL`*: сообщить об ошибке нарушения предусловия.
- *Если файл открыт только для чтения*: сообщить об ошибке нарушения предусловия.
- *Если запись выполнена неуспешно*: сообщить об ошибке нарушения постусловия, поскольку функция не в состоянии выполнить свою работу.

Пример 6. Различный статус одного и того же условия. Одно и то же условие может быть корректным предусловием для одной функции и не быть таковым для другой. Выбор зависит от автора функции, который определяет семантику интерфейса. В частности, `std::vector` предоставляет два пути для выполнения индексированного доступа: оператор `operator[]`, который не выполняет проверок выхода за пределы диапазона, и функцию `at`, которая такую проверку выполняет. И оператор `operator[]`, и функция `at` требуют выполнения предусловия, состоящего в том, что аргумент не должен выходить за пределы диапазона. Поскольку от оператора `operator[]` не требуется проверка его аргумента, должно быть четко документировано, что вызывающий код отвечает за то, чтобы аргумент оператора находился в допустимом диапазоне значений; понятно, что данная функция небезопасна. Функция же `at` в той же ситуации вполне безопасна, поскольку документировано, что она проверяет принадлежность своего аргумента к допустимому диапазону значений, и если аргумент выходит за пределы допустимого диапазона значений, то она сообщает об ошибке (путем генерации исключения `std::out_of_range`).

Ссылки

[Abrahams01b] • [Meyer00] • [Stroustrup00] §8.3.3, §14.1, §14.5 • [Sutter04b]

71. Проектируйте и пишите безопасный в отношении ошибок код

Резюме

В каждой функции обеспечивайте наиболее строгую гарантию безопасности, какой только можно добиться без дополнительных затрат со стороны вызывающего кода, не требующего такого уровня гарантии. Всегда обеспечивайте, как минимум, базовую гарантию безопасности.

Убедитесь, что при любых ошибках ваша программа всегда остается в корректном состоянии (в этом и заключается базовая гарантия). Остерегайтесь ошибок, нарушающих инвариант (включая утечки, но не ограничиваясь ими).

Желательно дополнительно гарантировать, что конечное состояние либо является исходным состоянием (в результате отката после происшедшей ошибки), либо корректно вычисленным целевым состоянием (если ошибок не было). Это — строгая гарантия безопасности.

Еще лучше гарантировать, что сбой в процессе операции невозможен. Хотя для большинства функций это невозможно, такую гарантию следует обеспечить для таких функций, как деструкторы и функции освобождения ресурсов. Данная гарантия — гарантия бессбойности.

Обсуждение

Базовая, строгая гарантии и гарантия бессбойности (известная также как гарантия отсутствия исключений) впервые были описаны в [Abrahams96] и получили широкую известность благодаря публикациям [GotW], [Stroustrup00, §E.2] и [Sutter00], посвященным вопросам безопасности исключений. Эти гарантии применимы к обработке любых ошибок, независимо от конкретного использованного метода, так что мы можем воспользоваться ими при описании безопасности обработки ошибок в общем случае. Гарантия бессбойности является строгим подмножеством строгой гарантии, а строгая гарантия, в свою очередь, является строгим подмножеством базовой гарантии.

В общем случае каждая функция должна обеспечивать наиболее строгую гарантию, которую она в состоянии обеспечить без излишних затрат для вызывающего кода, которому не требуется такая степень гарантии. Там, где это возможно, следует дополнительно обеспечить достаточную функциональность для того, чтобы требующий более строгую гарантию код мог получить ее (см. в примерах к данной рекомендации случай `vector::insert`).

В идеале следует писать функции, которые всегда успешно выполняются и, таким образом, могут обеспечить гарантию бессбойности. Некоторые функции должны всегда обеспечивать такую гарантию, в частности, деструкторы, функции освобождения ресурсов и функции обмена (см. рекомендацию 51).

Однако в большинстве функций могут произойти сбои. Если ошибка возможна, наиболее безопасным будет гарантировать транзакционное поведение функции: либо функция выполняется успешно и программа переходит из начального корректного состояния в корректное целевое состояние, либо — в случае сбоя — программа остается в том же состоянии, в котором находилась перед вызовом функции, т.е. видимые состояния всех объектов после сбойного вызова оказываются теми же, что и до него (например, значение глобальной целой переменной не может измениться с 42 на 43), и любое действие, которое вызывающий код мог предпринять до сбойного вызова, должно остаться возможным (с тем же смыслом) и после сбойного вызова (например, ни один итератор контейнера не должен стать недействительным; применение оператора ++ к упомянутой глобальной целой переменной даст значение 43, а не 44). Такая гарантия безопасности называется строгой.

Наконец, если обеспечить строгую гарантию оказывается слишком сложно или излишне дорого, следует обеспечить, по крайней мере, базовую гарантию: функция либо выполняется успешно и достигает целевого состояния, либо она неуспешна и оставляет программу в состоянии корректном (сохраняя инварианты, за которые отвечает данная функция), но не предсказуемом (это может быть исходное состояние, но может и не быть таковым; некоторые из постусловий могут быть не выполнены; однако все инварианты должны быть восстановлены). Ваше приложение должно быть спроектировано таким образом, чтобы суметь соответствующим образом обработать такой результат работы функции.

Вот и все; более низкого уровня гарантии не существует. Функция, которая не дает даже базовой гарантии — это просто ошибка программиста. Корректная программа должна отвечать, как минимум, базовой гарантии для всех функций. Даже те немногие корректные программы, которые сознательно идут на утечку ресурсов, в частности, в ситуациях, когда программа аварийно завершается, поступают так с учетом того, что ресурсы будут освобождены операционной системой. Всегда разрабатывайте код таким образом, чтобы корректно освобождались все ресурсы, а данные находились в согласованном состоянии даже при наличии ошибок, если только ошибка не приводит к немедленному аварийному завершению программы.

При принятии решения о том, какой уровень гарантии следует поддерживать, надо не забывать о перспективе развития проекта. Всегда проще усилить гарантию в последующих версиях, в то время как снижение степени гарантии ведет к необходимости переделки вызывающего кода, полагающегося на предоставление функцией более строгой гарантии.

Помните, что "небезопасность в отношении ошибок" и "плохое проектирование" идут рука об руку: если некоторую часть кода сложно сделать обеспечивающей базовую гарантию, то почти всегда это говорит о плохом проектировании. Например, если функция отвечает за выполнение нескольких несвязанных задач, ее трудно сделать безопасной в отношении ошибок (см. рекомендацию 5).

Остерегайтесь оператора копирующего присваивания, которому для корректной работы требуется проверка, не выполняется ли присваивание объекта самому себе. Безопасный в отношении ошибок оператор копирующего присваивания автоматически безопасен и в плане присваивания самому себе. Использовать проверку присваивания самому себе можно только в качестве оптимизации, для того чтобы избежать излишней работы (см. рекомендацию 55).

Примеры

Пример 1. Повторная попытка после сбоя. Если ваша программа включает команду для сохранения данных в файл и во время записи произошел сбой, убедитесь, что вы вернулись к состоянию, когда операция может быть повторена. В частности, не освобождайте никакие структуры данных до тех пор, пока данные не будут полностью сброшены на диск. Это не теоретизирование — нам известен один текстовый редактор, который не позволяет изменить имя файла для сохранения данных после ошибки записи.

Пример 2. Текстуры. Если вы пишете приложение, у которого можно менять внешний вид, загружая новые текстуры, то учтите, что не следует уничтожать старые текстуры до тех пор, пока не будут полностью загружены и применены новые. В противном случае при сбое во время загрузки новых текстур ваше приложение может оказаться в нестабильном состоянии.

Пример 3. `std::vector::insert`. Поскольку внутреннее представление `vector<T>` использует непрерывный блок памяти, вставка элемента в средину требует перемещения ряда имеющихся значений на одну позицию для освобождения места для вставляемого элемента. Перемещение выполняется с использованием копирующего конструктора `T::T(const T&)` и оператора присваивания `T::operator=`, и если одна из этих операций может сбоить (генерировать исключение), то единственный способ обеспечить строгую гарантию — это

сделать полную копию контейнера, выполнить операцию над копией, а затем обменять оригинал и копию с использованием бессбойной функции `vector<T>::swap`.

Однако выполнение всех указанных действий обходится излишне дорого как в плане требуемой памяти, так и процессорного времени. Таким образом, обеспечение строгой гарантии даже тогда, когда она не является необходимой, оказывается чрезмерно расточительным. Поэтому строгую гарантию вызывающий код должен при необходимости обеспечивать самостоятельно — для этого шаблон `vector` предоставляет все необходимые инструменты. (В лучшем случае, если тип содержимого контейнера не генерирует исключений в копирующем конструкторе и копирующем операторе присваивания, никаких дополнительных действий не требуется. В худшем следует создать копию вектора, выполнить вставку в копию и после успешного завершения данной операции обменять копию и исходный вектор.)

Пример 4. Запуск спутника. Рассмотрим функцию `f`, в которой частью ее работы является запуск спутника, и используемую ею функцию `LaunchSatellite`, обеспечивающую гарантию не ниже строгой. Если функция `f` может выполнить всю работу, при которой может произойти сбой, до запуска спутника, то `f` способна обеспечить строгую гарантию. Но если `f` должна выполнить некоторые операции, в процессе которых может произойти сбой, уже после запуска спутника, то обеспечение строгой гарантии оказывается невозможным — вернуть запущенный спутник на стартовую площадку уже нельзя. (Такую функцию `f` следует разделить по крайней мере на две, поскольку одна функция не должна даже пытаться выполнить несколько различных действий такой важности; см. рекомендацию 5.)

Ссылки

[Abrahams96] • [Abrahams01b] • [Alexandrescu03d] • [Josuttis99] §5.11.2 • [Stroustrup00] §14.4.3, §E.2-4, §E.6 • [Sutter00] §8-19, §40-41, §47 • [Sutter02] §17-23 • [Sutter04] §11-13 • [Sutter04b]

72. Для уведомления об ошибках следует использовать исключения

Резюме

Для уведомления об ошибках лучше использовать механизм исключений, а не коды ошибок. Применять коды состояния (например, коды ошибок, переменную `errno`) следует только тогда, когда нельзя использовать исключения (см. рекомендацию 62), а также для ситуаций, которые не являются ошибками. К другим методам, таким как экстренное завершение программы (или плановое завершение с освобождением ресурсов и т.п. действиями), следует прибегать только в ситуациях, когда восстановление после ошибки невозможно (или не требуется).

Обсуждение

То, что современные языки программирования, созданные в течение последних 20 лет, используют в качестве основного механизма сообщения об ошибках исключения, — не случайность. Практически по определению исключения предназначены для уведомления об исключениях в нормальном процессе — известных также как "ошибки", которые определены в рекомендации 70 как нарушения предусловий, постусловий и инвариантов. Так же, как и все другие механизмы уведомления об ошибках, исключения не должны генерироваться при нормальной успешной работе.

Далее мы будем использовать термин "коды состояния" для всех видов сообщения об ошибках посредством кодов (включая коды возврата, `errno`, функцию `GetLastError` и прочие стратегии возврата или получения кодов), а термин "коды ошибок" — для тех кодов состояния, которые означают ошибки. В C++ сообщение об ошибках посредством исключений имеет явные преимущества перед уведомлением посредством кодов ошибок.

- *Исключения невозможно проигнорировать.* Самое слабое место кодов ошибок заключается в том, что по умолчанию они игнорируются; чтобы уделить хотя бы минимальное внимание кодам ошибок, вы должны явно писать код, который опрашивает код ошибки и отвечает на него. Весьма распространенная среди программистов практика — случайно (или из-за лени) забыть опросить код ошибки. Исключения же невозможно просто проигнорировать; чтобы проигнорировать исключение, вы должны явно перехватить его (даже если вы сделаете это при помощи единственной инструкции `catch(...)`) и никак на него не отреагировать.
- *Исключения распространяются автоматически.* Коды ошибок по умолчанию за пределы области видимости не распространяются; для того, чтобы информировать высокоуровневую вызывающую функцию о низкоуровневой ошибке, программист должен написать промежуточный код, который передаст информацию об ошибке. Исключения автоматически распространяются за пределы области видимости до тех пор, пока они не будут перехвачены. ("*Это не самое разумное — пытаться сделать из каждой функции брандмауэр*". — [Stroustrup94, §16.8])
- *Исключения выносят обработку ошибок и восстановление после них из основного потока управления.* Проверка кода ошибки и ее обработка перемежается с основным потоком управления программы, тем самым запутывая его. Это затрудняет понимание и сопровождение как основного кода программы, так и кода обработки ошибок. Обработка исключений естественным образом перемещает обнаружение ошибок и восстановление после них в отдельные `catch`-блоки, т.е. делают обработку ошибок существенно более

модульной. Тем самым основной код программы, как и код обработки ошибок, оказывается более понятным и легче сопровождаемым.

- *Исключения оказываются наилучшим способом уведомления об ошибках в конструкторах и операторах.* Копирующие конструкторы и операторы имеют предопределенные сигнатуры, в которых просто нет места для кодов возврата. В частности, конструкторы вообще не имеют возвращаемого типа (даже `void`), а, например, каждый оператор `operator+` должен получать в точности два параметра и возвращать только один объект (предопределенного типа; см. рекомендацию 26). В случае операторов использование кодов ошибок, как минимум, возможно, если не желательно; для этого можно воспользоваться глобальной переменной наподобие `errno` или несколько более худшим методом внедрения кода состояния в состав объекта. Но для конструкторов использование кодов ошибок неприменимо, поскольку в языке C++ исключения в конструкторе и сбои в конструкторе настолько тесно связаны, что по сути являются синонимами. Если вы попытаетесь использовать подход с глобальной переменной наподобие

```
SomeType anObject; // Конструирование объекта
if( SomeType::ConstructionWasOk() ) { // Проверка результата
    // ...                            // конструирования
```

  то результат оказывается не только уродливым и подверженным ошибкам, но и ведет к "незаконнорожденным" объектам, которые признаются корректными, но на самом деле не удовлетворяют инвариантам типа. Это связано с возможными условиями гонки при использовании функции `SomeType::ConstructionWasOk` в многопоточных приложениях (см. [Stroustrup00] §E.3.5).

Главный потенциальный недостаток обработки исключений заключается в том, что требует от программиста хороших знаний о некоторых идиомах, возникающих в результате вынесения исключений из основного потока управления. Например, деструкторы и функции освобождения ресурсов не должны генерировать исключения (см. рекомендацию 51), а промежуточный код должен быть корректен при наличии исключений (см. рекомендацию 71 и ссылки); распространенная идиома для достижения этого заключается в отдельном выполнении всей работы, которая может привести к генерации исключений, и только после успешного завершения результаты работы принимаются, и состояние программы модифицируется с использованием только тех операций, которые предоставляют гарантию бессбойности (см. рекомендацию 51 и [Sutter00] §9-10, §13). Однако использование кодов ошибок также имеет собственные идиомы. Просто эти идиомы более старые и их знает большее количество программистов — но при этом, к сожалению, зачастую их просто игнорирует...

Обычно использование обработки исключений не приводит к снижению производительности. Для начала заметим, что вы всегда должны включать поддержку обработки исключений в вашем компиляторе, даже если по умолчанию она отключена; в противном случае вы не сможете получить предусмотренное стандартом поведение и уведомление об ошибках от таких операций C++, как оператор `new` или операции стандартной библиотеки наподобие вставок в контейнер (см. раздел исключений в данной рекомендации).

[Небольшое отступление. Включение поддержки обработки исключений может быть реализовано так, что оно увеличит размер выполнимого файла (что неизбежно), но при этом практически не повлияет на производительность приложения при отсутствии сгенерированных исключений, причем некоторые компиляторы именно так и поступают. Другие компиляторы приводят к определенным накладным расходам, в особенности при обеспечении режима безопасности для предотвращения атак посредством переполнения буфера в механизме обработки исключений. Однако имеются ли накладные расходы, связанные

с механизмом обработки исключений, или нет — в любом случае, вы должны включить его, иначе вы не сможете получать сообщения об ошибках от языка программирования и стандартной библиотеки.]

При включенной поддержке обработки исключений компилятором разница в производительности между генерацией исключений и возвратом кода ошибки при нормальной работе (в отсутствие ошибок) обычно незначительна. Определенную разницу в производительности можно заметить только при ошибках, но если ошибки происходят так часто, что становится ощутимой разница в производительности — скорее всего, вы используете исключения для ситуаций, которые ошибками не являются, а значит, вы не смогли корректно вычленить ошибки из прочих ситуаций (см. рекомендацию 70). Если же это действительно ошибки, которые нарушают пред-, постусловия или инварианты, и они встречаются настолько часто — значит, у вашей программы имеются гораздо более серьезные проблемы, чем снижение производительности из-за обработки исключений.

Один из симптомов неверного употребления кодов ошибок — когда коду приложения приходится постоянно проверять тривиальные истинные условия, либо (что еще хуже) когда приложение не проверяет коды ошибок, которые должно проверять.

Симптом злоупотребления исключениями — когда код приложения генерирует и перехватывает исключения настолько часто, что количества успешных и неуспешных выполнений `try`-блока оказываются величинами одного порядка. Такой `catch`-блок либо в действительности обрабатывает не истинные ошибки (которые нарушают пред-, постусловия или инварианты), либо у вашей программы имеются серьезные проблемы.

Примеры

Пример 1. Конструкторы (ошибка инварианта). Если конструктор не способен успешно создать объект своего типа (что означает, что он не способен установить все инварианты нового объекта), он должен сгенерировать исключение. Верно и обратное — исключение, сгенерированное конструктором, означает, что конструирование объекта не выполнено и что время жизни объекта никогда не начиналось.

Пример 2. Успешный рекурсивный поиск в дереве. При поиске в дереве с использованием рекурсивного алгоритма может показаться заманчивой идея вернуть результат — и легко свернуть стек — генерируя исключение с результатом поиска. Но так делать не следует: исключение означает ошибку, а результат поиска ошибкой не является (см. [Stroustrup00]). (Заметим, что, конечно, неуспешный поиск также не является ошибкой в контексте функции поиска; см. пример с функцией `find_first_of` в рекомендации 70.)

Обратитесь также к примерам рекомендации 70, заменяя в их тексте "сообщение об ошибке" на "генерация исключения".

Исключения

В редких случаях можно рассмотреть возможность использования кодов ошибок, если оказываются выполнены следующие условия.

- *Не применимы преимущества исключений.* Например, вы знаете, что практически всегда ошибка будет обрабатываться непосредственно вызывающим кодом, так что распространение ошибки никогда (или практически никогда) не будет востребовано. Это весьма редкая ситуация, поскольку обычно вызываемая функция не имеет достаточной информации о том, какой код будет ее вызывать.

- *Измерения показывают наличие реального снижения производительности при использовании исключений по сравнению с кодами ошибок.* Обычно такое снижение производительности становится заметным при частой генерации исключений во внутреннем цикле; следует напомнить, что частая генерация исключений обычно говорит о том, что в качестве ошибки рассматривается ситуация, которая ошибкой не является.

В некоторых очень редких случаях некоторые программы, работающие в реальном времени, могут собираться с отключенной обработкой исключений из-за того, что механизм этой обработки, предлагаемый компилятором, имеет в худшем случае время работы, которое не позволяет ключевым операциям выполняться в реальном времени. Конечно, такое отключение обработки исключений означает, что язык и стандартная библиотека не будут уведомлять об ошибках стандартным способом (или не будут уведомлять об ошибках вообще — см. документацию на ваш компилятор), и механизм уведомления об ошибках в вашем проекте должен быть основан не на исключениях, а на кодах ошибок. Трудно преувеличить всю нежелательность принятия такого решения. Перед тем как решиться на такой шаг, вы должны детально проанализировать, каким образом будет выполняться уведомление об ошибках в конструкторах и операторах и как предложенная схема будет работать на вашем компиляторе. Если после серьезного и глубокого анализа вы все же решите отключить механизм обработки исключений, то не делайте это во всем проекте; постарайтесь ограничиться как можно меньшим количеством модулей, собрав в них наиболее важные и чувствительные ко времени выполнения операции.

Ссылки

[Alexandrescu03d] • [Allison98] §13 • [Stroustrup94] §16 • [Stroustrup00] §8.3.3, §14.1, §14.4-5, §14.9, §E.3.5 • [Sutter00] §8-19, §40-41, §47 • [Sutter02] §17-23 • [Sutter04] §11-16 • [Sutter04b]

73. Генерируйте исключения по значению, перехватывайте — по ссылке

Резюме

Генерируйте исключения по значению (не через указатель) и перехватывайте их как ссылки (обычно константные). Эта комбинация наилучшим образом соответствует семантике исключений. При повторной генерации перехваченного исключения предпочтительно использовать просто оператор `throw;`, а не инструкцию `throw e;`.

Обсуждение

При генерации исключения генерируйте его по значению. Избегайте использовать исключение-указатель, поскольку в этом случае вам придется столкнуться с массой вопросов управления памятью: вы не можете генерировать исключение-указатель на значение в стеке, поскольку до того, как указатель достигнет точки назначения, стек будет свернут. Вы можете генерировать исключение-указатель на динамически выделенную память (если, конечно, ошибка, о которой вы сообщаете, не состоит в нехватке памяти), но при этом вы возлагаете на `catch`-блок задачу по освобождению выделенной памяти. Если вы уверены, что вам надо генерировать именно указатель, подумайте, нельзя ли заменить его интеллектуальным указателем типа `shared_ptr<T>` вместо обычного `T*`.

При генерации по значению компилятор сам отвечает за запутанный процесс управления памятью, выделенной генерируемому объекту. Все, что требуется от вас, — это принять меры для гарантии того, что копирующий конструктор класса вашего исключения не может генерировать исключений (см. рекомендацию 32).

Если только вы не генерируете интеллектуальный указатель, который добавляет уровень косвенности для сохранения полиморфизма, перехватывать исключения следует по ссылке. Перехват по значению приведет к срезке в точке перехвата (см. рекомендацию 54), что лишит вас обычно весьма важного качества полиморфизма объекта исключения. Перехват по ссылке сохраняет полиморфизм объекта исключения.

При повторной генерации исключения `e` лучше просто писать оператор `throw;` вместо инструкции `throw e;`. Дело в том, что первый способ всегда сохраняет полиморфизм объекта повторно генерируемого исключения.

Примеры

Пример. Повторная генерация измененного исключения. При повторной генерации перехваченного исключения предпочтительно использовать простой оператор `throw;`.

```
catch( MyException& e ) {      // Перехват неконстантной ссылки
  e.AppendContext("Перехват");// Внесение изменения
  throw;                       // Повторная генерация
                               // модифицированного исключения
}
```

Ссылки

[Dewhurst03] §64-65 • [Meyers96] §13 • [Stroustrup00] §14.3 • [Vandevoorde03] §20

74. Уведомляйте об ошибках, обрабатывайте и преобразовывайте их там, где следует

Резюме

Сообщайте об ошибках в тот момент, когда они обнаружены и идентифицированы как ошибки. Обрабатывайте или преобразовывайте их на самом нижнем уровне, на котором это можно сделать корректно.

Обсуждение

Сообщайте об ошибке (т.е. пишите `throw`) там, где функция обнаруживает ошибку, которую не может разрешить самостоятельно и которая делает невозможным продолжение выполнения функции (см. рекомендацию 70).

Обрабатывайте ошибку (т.е. пишите `catch`, который не генерирует повторно то же или иное исключение и не использует иной способ для дальнейшего уведомления об ошибке, например, код ошибки) там, где вы обладаете достаточной информацией, чтобы ее обработать, в том числе для обеспечения границ, определенных стратегией обработки ошибок (например, границ функции `main` или потоков; см. рекомендацию 62) и для поглощения исключений в телах деструкторов и функций освобождения ресурсов.

Преобразовывайте ошибку (т.е. пишите `catch`, который будет генерировать иное исключение или использовать иной способ для дальнейшего уведомления об ошибке, например, код ошибки) в следующих обстоятельствах.

- Для добавления высокоуровневого семантического значения. Например, в текстовом редакторе функция `Document::Open` может принимать низкоуровневую ошибку "неожиданное окончание файла" и преобразовывать ее в ошибку "неверный или поврежденный документ", добавляя соответствующую семантическую информацию.
- *Для изменения механизма обработки ошибок.* Например, в модуле, который внутренне использует исключения, но чей API в стиле C сообщает об ошибках посредством кодов ошибок, функции API должны перехватывать исключения и возвращать документированные коды ошибки, понятные вызывающему коду.

Код не должен перехватывать ошибку, если контекст не позволяет ему сделать с ней что-либо полезное. Если функция не может самостоятельно обработать ошибку (возможно, преобразовать или сознательно поглотить ее), то она не должна мешать ошибке распространяться далее, чтобы достичь вызывающего кода, который сможет ее обработать.

Исключения

Иногда оказывается полезным перехватить и повторно генерировать ту же ошибку (т.е. воспользоваться `catch` с последующим `throw`) для контроля за ошибками, несмотря на то, что в действительности обработки ошибки при этом не происходит.

Ссылки

[Stroustrup00] §3.7.2, §14.7, §14.9 • [Sutter00] §8 • [Sutter04] §11 • [Sutter04b]

75. Избегайте спецификаций исключений

Резюме

Не пишите спецификаций исключений у ваших функций, если только вас не заставляют это делать внешние обстоятельства (например, код, который вы не можете изменить, уже ввел их; см. исключения к данному разделу).

Обсуждение

Если говорить коротко — не беспокойтесь о спецификациях исключений. Основная проблема заключается в том, что спецификации исключений всего лишь "якобы" являются составной частью системы типов. Они не делают того, что от них ожидает большинство программистов, и вам почти никогда не надо то, что они на самом деле делают.

Спецификации исключений не являются частью типа функции за исключением тех случаев, когда они таки ею являются. Это не шутка — они по сути образуют "теневую" систему типов, из-за чего написание спецификаций исключений в разных местах программы оказывается:

- *неверным* — в инструкции `typedef` для указателя на функцию;
- *разрешенным* — в точно таком же коде, но без использования `typedef`;
- *необходимым* — в объявлении виртуальной функции, которая перекрывает виртуальную функцию базового класса со спецификацией исключений;
- *неявным и автоматическим* — в объявлении конструкторов, операторов присваивания, операторов присваивания и деструкторов, когда они неявно генерируются компилятором.

Весьма распространенным, но тем не менее не верным является убеждение о том, что спецификации исключений статически гарантируют, что функции будут генерировать только перечисленные исключения (возможно, не будут генерировать исключения вообще), что позволяет компилятору выполнить оптимизацию на основе данной гарантии.

В действительности же спецификации исключений делают нечто вроде бы и незначительное, но фундаментально отличающееся: они заставляют компилятор ввести в программу дополнительный код в форме неявных блоков `try/catch` вокруг тела функции для обеспечения проверки времени выполнения того, что функция действительно генерирует только перечисленные исключения или не генерирует их вообще, кроме тех случаев, когда компилятор может статически доказать, что спецификация исключений никогда не будет нарушена. В последнем случае компилятор может оптимизировать код, убрав описанную проверку. Спецификации исключений, кроме того, могут помешать дальнейшей оптимизации кода компилятором — так, например, некоторые компиляторы не могут делать встраиваемыми функции, имеющие спецификации исключений.

Однако хуже всего то, что спецификации исключений — очень "тупой" инструмент: при нарушении они по умолчанию немедленно прекращают выполнение программы. Вы можете зарегистрировать функцию-обработчик данного нарушения, но этот вряд ли вам поможет, поскольку такой обработчик только один на всю программу, и все, что он может сделать для того, чтобы избежать немедленного завершения программы, — это сгенерировать допустимое исключение. Но, напомним, такой обработчик — единственный для всего приложения, так что трудно представить, как можно сделать что-то полезное в такой ситуации или как узнать, какое исключение можно повторно сгенерировать в нем без вмешательства во все спецификации исключений (например, определить, что все спецификации исключений должны

включать некоторое общее исключение UnknownException, генерируемое обработчиком. Заметим, что это автоматически уничтожит все преимущества от использования спецификаций исключений).

Обычно вы не можете писать спецификации исключений для шаблонов функций, поскольку вы не можете в общем случае сказать заранее, какие типы исключений могут сгенерировать типы, с которыми будет работать шаблон.

Снижение производительности в обмен на введение ограничений практически всегда бесполезно. Это отличный пример преждевременной пессимизации (см. рекомендацию 9).

Нет простого пути решения описанных в данной рекомендации проблем. В частности, проблемы не решаются путем перехода к статическим проверкам. Программисты часто предлагают перейти от динамической проверки спецификаций исключений к статической, как это делается в Java и других языках программирования. Коротко говоря, это просто означает поменять шило на мыло, т.е. один ряд проблем на другой. Пользователи языков со статической проверкой спецификаций исключений не менее часто предлагают перейти к динамической проверке...

Исключения

Если вы перекрываете виртуальную функцию базового класса, которая уже имеет спецификацию исключений, и у вас нет возможности внести изменения в базовый класс и убрать спецификации исключений (или убедить сделать это автора класса), то вы должны использовать совместимую спецификацию исключений в вашей перекрывающей функции, причем следует сделать ее не менее ограничивающей, чем в базовой версии, чтобы минимизировать возможность нарушений спецификации исключений:

```
class Base { // ...      // В чужом классе имеется
    virtual f()          // спецификация исключений,
        throw(X, Y, Z); // и если вы не можете
};                       // ее удалить...

class MyDerived
: public Base { // ...  // ... то в вашем классе при
    virtual f()          // перегрузке функции она должна
        throw(X, Y, Z); // иметь совместимую (желательно
};                       // идентичную) спецификацию исключений
```

Из опыта [BoostLRG] следует, что только пустые спецификации исключений (т.е. throw()) у невстраиваемых функций "могут давать некоторое преимущество у некоторых компиляторов". Не слишком оптимистичное заявление для одного из наиболее продвинутых, разрабатываемого экспертами мирового уровня проекта...

Ссылки

[BoostLRG] • [Stroustrup00] §14.1, §14.6 • [Sutter04] §13

STL: контейнеры

Если вам нужен контейнер, по умолчанию используйте `vector`.

— Бьярн Страуструп (Bjarne Stroustrup),
[Stroustrup00] §17.7

Мы знаем, что вы уже предпочитаете использовать стандартные контейнеры вместо написанных вручную. Но какой именно контейнер следует использовать? Что должно (и что не должно) храниться в контейнерах и почему? Как следует заполнять контейнеры? Какие важные идиомы надо знать?

В этом разделе имеются ответы на все перечисленные (и не только) вопросы. И не случайно, что три первые рекомендации в нем содержат слова "Используйте `vector`...".

В этом разделе мы считаем наиболее значимой рекомендацию 79 — "Храните в контейнерах только значения или интеллектуальные указатели". К ней мы добавим — если даже вы не планируете применять [Boost] и [C++TR104] для иных целей, все равно воспользуйтесь их интеллектуальным указателем `shared_ptr`.

76. По умолчанию используйте vector. В противном случае выбирайте контейнер, соответствующий задаче

Резюме

Очень важно использовать "правильный контейнер". Если у вас есть весомые причины выбрать определенный тип контейнера, используйте тот контейнер, который наиболее подходит для вашей задачи.

Если конкретных предпочтений нет, возьмите `vector` и спокойно работайте, зная, что вы сделали верный выбор.

Обсуждение

Ниже представлены три главных аспекта, которые касаются программирования вообще, и выбора контейнера в частности.

- *Следует писать в первую очередь корректный, простой и понятный код (см. рекомендацию 6).* Предпочтителен выбор контейнера, который позволит вам написать наиболее понятный код. Так, если вам требуется вставка в определенную позицию, используйте последовательный контейнер (например, `vector`, `list`). Если вам нужен итератор произвольного доступа, выберите `vector`, `deque` или `string`. Если нужен словарный поиск наподобие `c[0] = 42;`, воспользуйтесь ассоциативным контейнером (например, `set`, `map`) — но если вам требуется упорядоченная ассоциативная коллекция, то вы не можете использовать контейнеры с использованием хеширования (нестандартные `hash_...` или стандартные `unordered_...` контейнеры).
- *Вопросы эффективности при написании должны учитываться только тогда и там, где это действительно необходимо (см. рекомендацию 8).* Если на основании проведенных измерений с реальными данными доказано, что скорость поиска действительно критична, можно воспользоваться контейнерами с использованием хеширования (нестандартными `hash_...` или стандартными `unordered_...` контейнерами), затем — отсортированным `vector`, затем — `set` или `map`, обычно в приведенном порядке. Даже в этой ситуации отличия асимптотического времени работы (с использованием "большого *O*", например, линейное и логарифмическое время работы; см. рекомендацию 7) имеет значение, только если контейнеры достаточно велики, чтобы постоянные множители перестали играть роль. Для контейнеров, содержащих небольшие объекты наподобие `double`, для преодоления влияния постоянных множителей обычно требуется по крайней мере несколько тысяч элементов.
- *Там, где это возможно и разумно, лучше писать транзакционный код со строгой гарантией безопасности код (см. рекомендацию 71), и не использовать некорректные объекты (см. рекомендацию 99).* Если для вставки и удаления элементов вам требуется транзакционная семантика, или если необходимо минимизировать недействительность итераторов, лучше воспользоваться контейнерами на основе узлов (например, `list`, `set`, `map`).

В противном случае следуйте совету Стандарта: "`vector` представляет собой тип последовательности, который нужно использовать по умолчанию" ([C++03] §23.1.1).

Если вы сомневаетесь в данном совете, спросите сами себя — действительно ли у вас есть непреодолимые причины *не* использовать стандартный контейнер `vector`, который обладает следующими свойствами.

- Гарантирует минимальные среди всех контейнеров накладные расходы памяти на хранение (ноль байтов на объект).
- Гарантирует наивысшую среди всех контейнеров скорость доступа к хранимым элементам.
- Гарантирует локальность ссылок, означающую, что объекты, находящиеся рядом друг с другом в контейнере, гарантированно находятся рядом и в памяти, что не гарантирует ни один другой контейнер.
- Гарантирует совместимость размещения в памяти с размещением, используемым языком программирования C, в отличие от остальных стандартных контейнеров (см. рекомендации 77 и 78).
- Гарантирует наличие самых гибких итераторов (произвольного доступа).
- Почти гарантированно имеет самые быстрые итераторы (указатели или классы со сравнимой производительностью, которые зачастую работают в окончательной (не отладочной) версии с той же скоростью, что и указатели).

Есть ли у вас причины не использовать такой контейнер по умолчанию? Если у вас есть такая причина, связанная с одним из трех аспектов в начале этой рекомендации, то все замечательно — просто воспользуйтесь наиболее подходящим для вас контейнером. Если такой причины нет — возьмите `vector` и спокойно работайте, зная, что вы сделали верный выбор.

И последнее — лучше использовать контейнеры и алгоритмы стандартной библиотеки, а не стороннего производителя или разработанные самостоятельно.

Примеры

Пример. Использование `vector` для небольших списков. Распространенное заблуждение — использовать `list` только потому, что "очевидно, что `list` —подходящий тип для выполнения операций со списком", таких как вставка в средину последовательности. Тип `vector` для небольших списков практически всегда превосходит тип `list`. Несмотря на то, что вставка в средину последовательности требует линейного времени работы у `vector`, и постоянного — у `list`, вектор с небольшим количеством элементов обычно справляется с этой задачей быстрее — за счет меньшего постоянного множителя; преимущества асимптотического времени работы `list` проявляются только при больших количествах элементов в контейнере.

Таким образом, используйте `vector`, пока размеры данных не потребуют иного выбора (см. рекомендацию 7), либо пока не станет существенной обеспечение строгой гарантии безопасности при возможной генерации исключений копирующим конструктором или копирующим оператором присваивания типа объектов, хранящихся в контейнере. В последнем случае может оказаться важным, что контейнер `list` обеспечивает строгую гарантию безопасности для операции вставки в коллекцию таких типов.

Ссылки

[Austern99] §5.4.1 • [C++03] §23.1.1 • [Josuttis99] §6.9 • [Meyers01] §1-2, §13, §16, §23, §25 • [Musser01] §6.1 • [Stroustrup00] §17.1, §17.6 • [Sutter00] §7, §20 • [Sutter02] §7

77. Вместо массивов используйте `vector` и `string`

Резюме

Избегайте реализации абстракции массива посредством массивов в стиле C, арифметики указателей и примитивов управления памятью. Использование `vector` или `string` не только сделает проще вашу жизнь, но и позволит написать более безопасную и масштабируемую программу.

Обсуждение

Сегодня едва ли не основными вопросами при написании программного обеспечения являются вопросы безопасности и переполнения буфера. Ограничения, связанные с фиксированным размером массивов, доставляют немало беспокойства, даже если приложению удается остаться в рамках корректности. Все эти проблемы связаны с использованием старых средств C, таких как встроенные массивы, указатели и их арифметика, а также управление памятью вручную вместо таких высокоуровневых концепций, как буферы, векторы и строки.

Вот некоторые из причин, по которым массивам в стиле C следует предпочесть стандартные средства C++.

- *Они автоматически управляют собственной памятью.* Больше не требуется никаких фиксированных буферов с размером "большим, чем любая разумная длина" ("бомба с часовым механизмом" — вот как правильно прочесть это определение), или сплошных перераспределений памяти при помощи `realloc` и соответствующих обновлений указателей.
- *У них богатый интерфейс.* Вы легко и выразительно можете реализовать сложную функциональность.
- *Они совместимы с моделью памяти в C.* `vector` и `string::c_str` могут быть переданы функциям API на языке C. В частности, в C++ `vector` служит "переходником" к C и другим языкам программирования (см. рекомендации 76 и 78).
- *Они обеспечивают расширенные возможности проверки.* Стандартные средства позволяют реализовать (в отладочном режиме) итераторы и операторы индексирования, которые способны выявить большой класс ошибок памяти. Многие из современных реализаций стандартной библиотеки предоставляют такие отладочные возможности — воспользуйтесь ими! (См. рекомендацию 83.)
- *Они практически не снижают эффективность ваших программ.* В классах `vector` и `string` при выборе между эффективностью и безопасностью решение принимается в пользу эффективности (естественно, не в отладочном режиме). Тем не менее, стандартные средства оказываются гораздо лучшей платформой для создания безопасных компонентов, чем обычные массивы и указатели.
- *Они стимулируют оптимизацию.* Современные реализации стандартной библиотеки включают оптимизации, о которых многие из нас просто никогда бы и не подумали.

Применение массива может быть оправданно, когда его размер фиксирован на стадии компиляции (например, `float[3]` для трехмерной точки; переход к четвертому измерению все равно потребует перепроектирования программы).

Ссылки

[Alexandrescu01a] • [Dewhurst03] §13, §60, §68 • [Meyers01] §13, §16 • [Stroustrup00] §3.5-7, §5.3, §20.4.1, §C.7 • [Sutter00] §36

78. Используйте vector (и string::c_str) для обмена данными с API на других языках

Резюме

vector и string::c_str служат шлюзом для сообщения с API на других языках. Однако не полагайтесь на то, что итераторы являются указателями; для получения адреса элемента, на который ссылается vector<T>::iterator iter, используйте выражение &*iter.

Обсуждение

vector (в первую очередь) и string::c_str и string::data (во вторую) представляют собой наилучшие способы обмена данными с API на других языках вообще и с библиотеками на C в частности.

Данные vector всегда хранятся последовательно, так что получение адреса первого элемента вектора дает указатель на его содержимое. Используйте &*v.begin(), &v[0] или &v.front() для получения указателя на первый элемент v. Для получения указателя на n-й элемент вектора лучше сначала провести арифметические вычисления, а затем получить адрес (например, &v.begin()[n] или &v[n]) вместо получения указателя на начало данных с последующим применением арифметики указателей (например, (&v.front())[n]). Это связано с тем, что в первом случае в отладочной реализации выполняется проверка на доступ к элементу за пределами v (см. рекомендацию 83).

Нельзя полагаться на то, что v.begin() возвращает указатель на первый элемент или, в общем случае, что итераторы вектора являются указателями. Хотя некоторые реализации STL определяют vector<T>::iterator как обычный указатель T*, итераторы могут быть (и все чаще так оно и есть) полноценными типами (еще раз см. рекомендацию 83).

Хотя в большинстве реализаций для string также используется непрерывный блок памяти, это не гарантируется стандартом, так что никогда не используйте адрес символа в строке, считая его указателем на все содержимое строки. Хорошая новость заключается в том, что функция string::c_str всегда возвращает строку в стиле C с завершающим нулевым символом (string::data также возвращает указатель на непрерывный блок памяти, но не гарантирует наличия завершающего нулевого символа).

Когда вы передаете указатель на данные объекта v типа vector, код на языке C может как читать, так и записывать элементы v; однако он не должен выходить за границы данных. Хорошо продуманный API на языке C должен получать наряду с указателем либо максимальное количество объектов (до v.size()), либо указатель на элемент, следующий за последним (&*v.begin()+v.size()).

Если у вас есть контейнер объектов типа T, отличный от vector или string, и вы хотите передать его содержимое (или заполнить его) функции API на другом языке программирования, которая ожидает указатель на массив объектов типа T, скопируйте содержимое контейнера в (или из) vector<T>, который может непосредственно сообщаться с такими функциями.

Ссылки

[Josuttis99] §6.2.3, §11.2.4 • [Meyers01] §16 • [Musser01] §B • [Stroustrup00] §16.3.1

79. Храните в контейнерах только значения или интеллектуальные указатели

Резюме

Храните к контейнерах объекты-значения. Контейнеры полагают, что их содержимое имеет тип значения, включая непосредственно хранящиеся значения, интеллектуальные указатели и итераторы.

Обсуждение

Наиболее распространенное использование контейнеров — для непосредственного хранения значений (например, `vector<int>`, `set<string>`). В случае контейнеров указателей, если контейнер владеет объектами, на которые указывает, то лучше использовать контейнер интеллектуальных указателей со счетчиком ссылок (например, `list<shared_ptr<Widget> >`); в противном случае можно выбрать контейнер обычных указателей (например, `list<Widget*>`) или иных значений, подобных указателям — таких как итераторы (например, `list<vector<Widget>::iterator>`).

Примеры

Пример 1. `auto_ptr`. Объекты `auto_ptr<T>` не являются объектами-значениями из-за своей семантики передачи владения при копировании. Использование контейнера объектов `auto_ptr` (например, `vector<auto_ptr<int> >`) должно привести к ошибке компиляции. Но даже в случае успешной компиляции никогда не пишите такой код — вместо этого вам следует использовать контейнер интеллектуальных указателей `shared_ptr`.

Пример 2. Гетерогенные контейнеры. Для того чтобы получить контейнер, хранящий и владеющий объектами различных, но связанных типов, например, типов, производных от общего базового класса, лучше использовать `container<shared_ptr<Base> >`. Альтернативой является хранение прокси-объектов, невиртуальные функции которых передают вызовы соответствующим виртуальным функциям реальных объектов.

Пример 3. Контейнеры типов, не являющихся значениями. Для того чтобы хранить объекты, несмотря на невозможность их копирования или другое их поведение, делающее их типами, не являющимися значениями (например, `DatabaseLock` или `TcpConnection`), их следует хранить опосредованно, с использованием интеллектуальных указателей (например, `container<shared_ptr<DatabaseLock> >` или `container<shared_ptr<TcpConnection> >`).

Пример 4. Необязательные значения. Если вам нужен контейнер `map<Thing,Widget>`, но некоторые `Thing` не имеют связанных с ними объектов `Widget`, можно использовать `map<Thing,shared_ptr<Widget> >`.

Пример 5. Индексные контейнеры. Если вам нужен контейнер, хранящий объекты, и доступ к ним с использованием разных видов упорядочения без пересортировки основного контейнера, вы можете воспользоваться вторичным контейнером, который "указывает в" основной контейнер, и отсортировать его различными способами с применением предикатов сравнения с разыменованием. Однако вместо контейнера указателей лучше использовать контейнер объектов типа `MainContainer::iterator` (которые являются значениями).

Ссылки

[Allison98] §14 • [Austern99] §6 • [Dewhurst03] §68 • [Josuttis99] §5.10.2 • [Koenig97] §5 • [Meyers01] §3, §7-8 • [SuttHysl04b]

80. Предпочитайте push_back другим способам расширения последовательности

Резюме

Используйте push_back везде, где это возможно. Если для вас не важна позиция вставки нового объекта, лучше всего использовать для добавления элемента в последовательность функцию push_back. Все прочие средства могут оказаться как гораздо менее быстрыми, так и менее понятными.

Обсуждение

Вы можете вставить элементы в последовательность в разных точках с использованием insert; добавить элементы в последовательность можно разными способами, включая следующие:

```
vector<int> vec;                  // vec пуст
vec.resize(vec.size() + 1, 1);    // vec содержит { 1 }
vec.insert(vec.end(), 2);         // vec содержит { 1, 2 }
vec.push_back(3);                 // vec содержит { 1, 2, 3 }
```

Среди прочих методов push_back единственный имеет *постоянное амортизированное время работы*. Время работы других методов хуже — вплоть до квадратичного. Излишне говорить, что при больших размерах данных плохое время работы препятствует масштабируемости (см. рекомендацию 7).

Магия push_back проста: эта функция увеличивает емкость экспоненциально, а не на фиксированное значение. Следовательно, количество перераспределений памяти и копирований быстро уменьшается с увеличением размера. В случае контейнера, который заполняется с использованием только лишь функции push_back, каждый элемент копируется в среднем только один раз — независимо от конечного размера контейнера.

Конечно, resize и insert могут воспользоваться той же стратегией, но это уже зависит от реализации; гарантию дает только push_back.

Алгоритмы не могут непосредственно обращаться к push_back, поскольку они не имеют доступа к контейнерам. Вы можете потребовать от алгоритма использовать push_back, воспользовавшись back_inserter.

Исключения

Если вы добавляете не один элемент, а диапазон, то даже если добавление выполняется в конец контейнера, лучше использовать функцию для вставки диапазона значений (см. рекомендацию 81).

Экспоненциальный рост приводит к расточительному выделению памяти. Для тонкой настройки роста можно явно вызвать функцию reserve — функции push_back, resize и подобные не будут перераспределять память, если ее достаточно для работы. Для получения вектора "правильного размера" следует воспользоваться идиомой "горячей посадки" (см. рекомендацию 82).

Ссылки

[Stroustrup00] §3.7-8, §16.3.5, §17.1.4.1

81. Предпочитайте операции с диапазонами операциям с отдельными элементами

Резюме

При добавлении элементов в контейнер лучше использовать операции с диапазонами (т.е. функцию `insert`, которая получает пару итераторов), а не последовательность вызовов функции для вставки одного элемента. Вызов функции для диапазона обычно проще написать, легче читать, и он более эффективен, чем явный цикл (см. также рекомендацию 84).

Обсуждение

Чем больший контекст передается функции, тем больше вероятность того, что она сможет лучше распорядиться полученной информацией. В частности, когда вы вызываете функцию и передаете ей пару итераторов, указывающих некоторый диапазон, она может выполнить оптимизацию, основанную на знании количества объектов, которые должны быть добавлены (вычисляемое как `distance(first,last)`).

То же самое можно сказать и об операциях "повторить n раз", например, о конструкторе `vector`, который получает количество повторений и значение.

Примеры

Пример 1. `vector::insert`. Пусть вам надо добавить n элементов в `vector v`. Многократный вызов `v.insert(position,x)` может привести к неоднократному перераспределению памяти, когда вектор `v` имеет недостаточно места для размещения нового элемента. Что еще хуже, вставка каждого отдельного элемента имеет линейное время работы, поскольку она должна перенести ряд элементов, чтобы освободить требуемую позицию для вставляемого элемента, а это приводит к тому, что вставка n элементов при помощи последовательных вызовов имеет квадратичное время работы! Конечно, избежать проблемы множественного перераспределения памяти можно при помощи вызова `reserve`, но это не снизит количества перемещений элементов и квадратичное время работы такого алгоритма. Быстрее и проще ясно сказать, что вам надо: `v.insert(position,first,last)`, где `first` и `last` — итераторы, определяющие диапазон элементов, которые должны быть добавлены в `v`. (Если `first` и `last` — входные итераторы, то возможности определить размер диапазона перед его действительным проходом нет, так что вектору `v` может потребоваться многократное перераспределение памяти; тем не менее, версия для вставки диапазона все равно скорее всего будет более производительной, чем вставка отдельных элементов.)

Пример 2. Создание и присваивание диапазона. Вызов конструктора (или функции присваивания), который получает диапазон итераторов, обычно выполняется быстрее, чем вызов конструктора по умолчанию (или функции `clear`) с последующими индивидуальными вставками в контейнер.

Ссылки

[Meyers01] §5 • [Stroustrup00] §16.3.8

82. Используйте подходящие идиомы для реального уменьшения емкости контейнера и удаления элементов

Резюме

Для того чтобы действительно избавиться от излишней емкости контейнера, воспользуйтесь трюком с использованием обмена, а для реального удаления элементов из контейнера — идиомой `erase-remove`.

Обсуждение

Некоторые контейнеры (например, `vector`, `string`, `deque`) могут иметь "лишнюю" емкость, которая больше не будет использоваться. Хотя стандартная библиотека C++ не предоставляет гарантированного способа для удаления излишней емкости, следующая идиома на практике оказывается вполне работоспособной:

```
container<T>(c).swap(c); // Идиома "горячей посадки" для
                         // устранения излишней емкости
                         // контейнера
```

Для того чтобы полностью опустошить `c`, удалив все элементы и убрав всю емкость, идиома должна выглядеть следующим образом:

```
container<T>().swap(c); // Идиома для удаления всего
                        // содержимого и емкости
```

Кроме того, обычно для новичков в программировании с использованием STL оказывается сюрпризом то, что алгоритм `remove` в действительности не удаляет элементы из контейнера. Понятно, что данный алгоритм на это не способен — ведь алгоритм работает только с диапазоном итераторов и не может ничего реально удалить из контейнера без вызова функции-члена контейнера, обычно `erase`. Удаление сводится к перемещению элементов, которые должны быть "удалены", и возврату итератора, указывающего на элемент, следующий за последним неудаленным. Для реального удаления элементов из контейнера после вызова `remove` следует вызвать `erase` — воспользоваться идиомой `erase-remove`. Например, для реального удаления всех элементов, равных `value`, из контейнера `c`, можно написать:

```
c.erase( remove( c.begin(), c.end() ,value ), c.end() );
```

Если контейнер имеет собственную версию `remove` или `remove_if`, желательно использовать именно ее.

Исключения

Описанная идиома "горячей усадки" не работает с реализациями `std::string` с копированием при записи. Обычно работает вызов `s.reserve(0)` или такой трюк, как `string(s.begin(),s.end()).swap(s);`, в котором использован конструктор на основе двух итераторов. На практике эти методы обычно работают и устраняют излишнюю емкость. (Было бы еще лучше, чтобы реализации `std::string` не использовали такой устаревший метод оптимизации, как копирование при записи; см. [Sutter02].)

Ссылки

[Josuttis99] §6.2.1 • [Meyers01] §17, §32, §44 • [Sutter00] §7 • [Sutter02] §7, §16

STL: алгоритмы

Предпочитайте алгоритмы циклам.

— Бьярн Страуструп (Bjarne Stroustrup),
[Stroustrup00] §18.12

Алгоритмы представляют собой циклы — только они лучше циклов. Алгоритмы — это "шаблоны" циклов, с добавлением дополнительной семантики по сравнению с простыми `for` и `do`. Конечно, начав использовать алгоритмы, вы начнете использовать и функциональные объекты и предикаты; корректно пишите их и широко используйте в своих программах.

В этом разделе мы считаем наиболее значимой рекомендацию 83 — "Используйте отладочную реализацию STL".

83. Используйте отладочную реализацию STL

Резюме

Безопасность превыше всего (см. рекомендацию 6). Используйте отладочную реализацию STL[4], даже если она имеется только для одного из ваших компиляторов, и даже если она используется только для отладочного тестирования.

Обсуждение

Так же, как легко ошибиться при работе с указателями, легко допустить и ошибку при работе с итераторами, причем такую, что программа при этом успешно компилируется, а при работе немедленно аварийно завершается (в лучшем случае) или работает неверно (в худшем). Даже если ваш компилятор не обнаружил ни одной ошибки, вы не должны полагаться на "визуальный контроль": имеются специальные инструменты — воспользуйтесь ими.

Ряд ошибок при работе с STL часто допускают даже опытные программисты.

- *Использование недействительных или неинициализированных итераторов.* В особенности легко допустить первое.
- *Передача индекса, выходящего за границы контейнера.* Например, обращение к 113-му элементу 100-элементного контейнера.
- Использование "диапазона" итераторов, который на самом деле не является диапазоном. Например, передача двух итераторов, первый из которых не предшествует второму, или когда они указывают на разные контейнеры.
- *Передача неверной позиции итератора.* Вызов функции-члена контейнера, которая получает позицию итератора (как, например, позиция, передаваемая функции `insert`), но с передачей итератора, который указывает в другой контейнер.
- *Использование неверного упорядочения.* Предоставление неверного правила для упорядочения ассоциативного контейнера или в качестве критерия сравнения в алгоритме сортировки (см. примеры в [Meyers01]). Если не использовать отладочной версии STL, такие ошибки обычно проявляются в процессе выполнения программы как неверное поведение или бесконечные циклы, но не как аварийный останов.

Большинство отладочных реализаций STL автоматически обнаруживают такие ошибки, при помощи дополнительной отладочной и системной информации, хранящейся в контейнерах и итераторах. Например, итератор может запомнить контейнер, к которому он обращается, а контейнер — запомнить все итераторы, ссылающиеся на него, так что можно отследить использование ставшего недействительным итератора. Конечно, это увеличивает размер итераторов и контейнеров, а также требует выполнения дополнительных действий при каждом изменении контейнера. Но возможность отслеживания ошибок стоит того — по крайней мере, в процессе отладки, а возможно, и в окончательной версии программы (вспомните рекомендацию 8; не отключайте полезные проверки для повышения производительности до тех пор, пока не убедитесь в необходимости такого шага непосредственными измерениями).

Даже если вы не намерены оставлять проверки в окончательной версии, и даже если у вас имеется проверочная реализация STL только для одной из используемых вами платформ, обязательно протестируйте вашу программу полным комплектом тестов с использованием отладочной реализации STL. Вы не пожалеете об этом.

---

[4] Checked implementation — под этим подразумевается реализация STL с дополнительными проверками в коде, необязательными с точки зрения стандарта. — *Прим. перев.*

Примеры

Пример 1. Использование недействительного итератора. Очень просто забыть, что итераторы стали недействительны, и воспользоваться таким недействительным итератором (см. рекомендацию 99). Рассмотрим подобный пример, адаптированный из [Meyers01], где происходит вставка элементов в начало deque:

```
deque<double>::iterator current = d.begin();
for( size_t i = 0; i < max; ++i )
  d.insert(current++, data[i] + 41 ); // Видите ли вы ошибку?
```

Быстро ответьте — увидели ли вы ошибку в приведенном фрагменте или нет? У вас только три секунды!

Время вышло! Если вы не заметили ошибку за отпущенное время, не волнуйтесь — это достаточно тонкая и понятная ошибка. Отладочная версия STL обнаружит данную ошибку на второй итерации цикла, так что вам не придется полагаться на свой невооруженный взгляд. (Исправленная версия данного кода и альтернативные варианты вы найдете в рекомендации 84.)

Пример 2. Использование диапазона итераторов, на самом деле не являющегося диапазоном. Диапазон итераторов представляет собой пару итераторов `first` и `last`, которые указывают на первый элемент диапазона, и элемент, следующий за последним элементом диапазона. Диапазон требует, чтобы итератор `last` был достижим из `first` путем некоторого количества повторных увеличений при помощи оператора инкремента итератора `first`. Известны два распространенных вида ошибки, когда происходит попытка использовать диапазон, который на самом деле диапазоном не является. Первая ошибка возникает, когда два итератора ограничивают диапазон в пределах одного контейнера, но итератор `first` на самом деле не предшествует итератору `second`:

```
for_each(c.end(),c.begin(),Something); // Не всегда очевидно
```

На каждой итерации своего внутреннего цикла алгоритм `for_each` будет проверять итератор `first` на равенство итератору `second`, и до тех пор, пока они не равны, он будет увеличивать итератор `first`. Конечно, сколько бы раз не был увеличен итератор `first`, ему никогда не стать равным итератору `second`, так что цикл по сути оказывается бесконечным. На практике в лучшем случае цикл завершится выходом за пределы контейнера `c` и немедленным аварийным завершением программы из-за нарушения защиты памяти. В худшем случае, выйдя за пределы контейнера, итератор будет считывать или даже менять данные, не являющиеся частью контейнера. Этот результат в принципе не слишком отличается от знакомых нам последствий переполнения буфера...

Вторая распространенная ошибка возникает, когда итераторы указывают в разные контейнеры:

```
for_each(c.begin(),d.end(),Something); // Не всегда очевидно
```

Результат такой ошибки аналогичен результату предыдущей. Однако поскольку итераторы отладочной версии STL помнят контейнер, с которым работают, такую ошибку можно выявить во время выполнения программы.

Ссылки

[Dinkumware-Safe] • [Horstmann95] • [Josuttis99] §5.11.1 • [Metrowerks] • [Meyers01] §21, §50 • [STLport-Debug] • [Stroustrup00] §18.3.1, §19.3.1

84. Предпочитайте вызовы алгоритмов самостоятельно разрабатываемым циклам

Резюме

Разумно используйте функциональные объекты. В очень простых случаях написанные самостоятельно циклы могут оказаться более простым и эффективным решением. Тем не менее, вызов алгоритма вместо самостоятельно разработанного цикла может оказаться более выразительным, легче сопровождаемым, менее подверженным ошибкам и не менее эффективным.

При вызове алгоритма подумайте о написании собственного функционального объекта, который инкапсулирует всю необходимую логику. Избегайте объединения связывателей параметров и простых функциональных объектов (например, `bind2nd`, `plus`), что обычно снижает ясность кода. Подумайте об использовании лямбда-библиотеки [Boost], которая автоматизирует задачу написания функциональных объектов.

Обсуждение

Программы, которые используют STL, имеют тенденцию к меньшему количеству явных циклов, чем у программ без применения STL благодаря замене низкоуровневых циклов высокоуровневыми абстрактными операциями с существенно большей семантикой. При программировании с использованием STL, следует думать не в категориях "как обработать каждый элемент", а "как обработать диапазон элементов".

Главное преимущество алгоритмов и шаблонов проектирования в общем случае заключается в том, что они позволяют нам говорить на более высоком уровне абстракции. В современном программировании мы не говорим "пусть несколько объектов следят за одним объектом и получают автоматические уведомления при изменении его состояния". Вместо этого мы говорим просто о "шаблоне проектирования Observer". Аналогично, мы говорим "Bridge", "Factory" и "Visitor". Использование словаря шаблонов проектирования позволяет повысить уровень, эффективность и корректность нашего обсуждения. Точно так же при использовании алгоритмов мы не говорим "выполняем действие над каждым элементом диапазона и записываем результат в некоторое место"; вместо этого мы говорим просто — `transform`. Аналогично, мы можем сказать `for_each`, `replace_if` и `partition`. Алгоритмы, подобно шаблонам проектирования, самодокументируемы. "Голые" циклы `for` и `while` ничего не говорят о том, для чего они предназначены, и читателю приходится изучать тела циклов, чтобы расшифровать их семантику.

Алгоритмы обычно более корректны, чем циклы. В разрабатываемых самостоятельно циклах легко допустить ошибку, например, такую как использование недействительных итераторов (см. рекомендации 83 и 99); алгоритмы в библиотеке отлажены на предмет использования недействительных итераторов и других распространенных ошибок.

И наконец, алгоритмы зачастую также более эффективны, чем простые циклы (см. [Sutter00] и [Meyers01]). В них устранены небольшие неэффективности, такие как повторные вычисления `container.end()`. Не менее важно, что стандартные алгоритмы, используемые вами, были реализованы теми же программистами, кто реализовывал и используемые вами стандартные контейнеры, и понятно, что их знание конкретных реализаций позволяет им написать алгоритмы более эффективно, чем это сможете сделать вы. Важнее всего, однако, то, что многие алгоритмы имеют высокоинтеллектуальные реализации, которые мы никогда не сможем реализовать

в собственноручно разработанном коде (не считая случаев, когда нам не нужна та степень обобщенности, которую предоставляют алгоритмы). Вообще говоря, более используемая библиотека всегда оказывается лучше отлаженной и более эффективной просто потому, что у нее больше пользователей. Вряд ли вы найдете и сможете использовать в своей программе библиотеку, настолько же широко применяемую, как и реализация вашей стандартной библиотеки. Воспользуйтесь ею. Алгоритмы STL уже написаны — так почему бы не воспользоваться ими?

Подумайте об использовании лямбда-функций [Boost]. Лямбда-функции представляют собой важный инструмент, который позволяет справиться с основным недостатком алгоритмов, а именно с удобочитаемостью. Без их применения вы должны использовать либо функциональные объекты (но тогда тела даже простых циклов находятся в отдельном месте, далеко от точки вызова), либо стандартные связыватели и функциональные объекты наподобие `bind2nd` и `plus` (достаточно запутанные, сложные и утомительные в использовании).

Примеры

Вот два примера, адаптированных из [Meyers01].

Пример 1. Преобразование ***deque****.* После того как было выполнено несколько некорректных итераций из-за недействительных итераторов (например, см. рекомендацию 83), мы пришли к окончательной версии цикла для прибавления 41 к каждому элементу массива данных типа `double` и помещения результата в дек `deque<double>`:

```
deque<double>::iterator current = d.begin();
for( size_t i = 0; i < max; ++i ) {
    // Сохраняем current действительным
    current = d.insert( current, data[i] + 41 );
    ++current; // Увеличиваем его, когда это
}              // становится безопасным
```

Вызов алгоритма позволяет легко обойти все ловушки в этом коде:

```
transform(
    data.begin(), data.end(),      // Копируем элементы data
    inserter(d, d.begin()),        // в d с начала контейнера,
    bind2nd(plus<double>(),41));  // добавляя к каждому 41
```

Впрочем, `bind2nd` и `plus` достаточно неудобны. Откровенно говоря, в действительности их мало кто использует, и связано это в первую очередь с плохой удобочитаемостью такого кода (см. рекомендацию 6).

При использовании лямбда-функций, генерирующих для нас функциональные объекты, мы можем написать совсем простой код:

```
transform(data, data+max, inserter(d,d.begin()), _1 + 41);
```

Пример 2. Найти первый элемент между ***x*** *и* ***y****.* Рассмотрим простой цикл, который выполняет поиск в `vector<int> v` первого элемента, значение которого находится между `x` и `y`. Он вычисляет итератор, который указывает либо на найденный элемент, либо на `v.end()`:

```
vector<int>::iterator i = v.begin();
for( ; i != v.end(); ++i)
    if( *i > x && *i < y ) break;
```

Вызов алгоритма достаточно проблематичен. При отсутствии лямбда-функций у нас есть два варианта — написание собственного функционального объекта или использование стандартных связывателей. Увы, в последнем случае мы не можем обойтись только стандартными связывателями и должны использовать нестандартный (хотя и достаточно распространенный) адаптер `compose2`, но даже в этом случае код получается совершенно непонятным, так что такой код на практике никто просто не напишет:

```
vector<int>::iterator i =
    find_if(v.begin(), v.end(),
            compose2(logical_and<bool>(),
                     bind2nd(greater<int>(), x),
                     bind2nd(less<int>(), y)));
```

Другой вариант, а именно — написание собственного функционального объекта — достаточно жизнеспособен. Он достаточно хорошо выглядит в точке вызова, а главный его недостаток — необходимость написания функционального объекта `BetweenValues`, который визуально удаляет логику из точки вызова:

```
template<typename T>
class BetweenValues : public unary_function<T, bool> {
public:
    BetweenValues( const T& low, const T& high )
        : low_(low), high_(high) { }
    bool operator()( const T& val ) const
        { return val > low_ && val < high_; }
private:
    T low_, high_;
};

vector<int>::iterator i =
    find_if( v.begin(), v.end(), BetweenValues<int>(x, y));
```

При применении лямбда-функций можно написать просто:

```
vector<int>::iterator i =
    find_if( v.begin(), v.end(), _1 > x && _1 < y );
```

Исключения

При использовании функциональных объектов тело цикла оказывается размещено в некотором месте, удаленном от точки вызова, что затрудняет чтение исходного текста. (Использование простых объектов со стандартными и нестандартными связывателями представляется нереалистичным.)

Лямбда-функции [Boost] решают проблему и надежно работают на современных компиляторах, но они не годятся для более старых компиляторов и могут выдавать большие запутанные сообщения об ошибках при некорректном использовании. Вызов же именованных функций, включая функции-члены, все равно требует синтаксиса с использованием связывателей.

Ссылки

[Allison98] §15 • [Austern99] §11-13 • [Boost] Lambda library • [McConnell93] §15 • [Meyers01] §43 • [Musser01] §11 • [Stroustrup00] §6.1.8, §18.5.1 • [Sutter00] §7

85. Пользуйтесь правильным алгоритмом поиска

Резюме

Данная рекомендация применима к поиску определенного значения в диапазоне. При поиске в неотсортированном диапазоне используйте алгоритмы `find/find_if` или `count/count_if`. Для поиска в отсортированном диапазоне выберите `lower_bound`, `upper_bound`, `equal_range` или (реже) `binary_search`. (Вопреки своему имени, `binary_search` обычно — неверный выбор.)

Обсуждение

В случае неотсортированных диапазонов, `find/find_if` и `count/count_if` могут за линейное время определить, находится ли данный элемент в диапазоне, и если да, то где именно. Заметим, что алгоритмы `find/find_if` обычно более эффективны, поскольку могут завершить поиск, как только искомый элемент оказывается найден.

В случае сортированных диапазонов лучше использовать алгоритмы бинарного поиска — `binary_search`, `lower_bound`, `upper_bound` и `equal_range`, которые имеют логарифмическое время работы. Увы, несмотря на свое красивое имя, `binary_search` практически всегда бесполезен, поскольку возвращает всего лишь значение типа `bool`, указывающее, найден искомый элемент или нет. Обычно вам требуется алгоритм `lower_bound` или `upper_bound`, или `equal_range`, который выдает результаты обоих алгоритмов — и `lower_bound`, и `upper_bound` (и требует в два раза больше времени).

Алгоритм `lower_bound` возвращает итератор, указывающий на первый подходящий элемент (если таковой имеется) или на позицию, где он мог бы быть (если такого элемента нет); последнее полезно для поиска верного места для вставки новых значений в отсортированную последовательность. Алгоритм `upper_bound` возвращает итератор, указывающий на элемент, следующий за последним найденным элементом (если таковой имеется), т.е. на позицию, куда можно добавить следующий эквивалентный элемент; это полезно при поиске правильного места для вставки новых значений в отсортированную последовательность, чтобы поддерживать упорядоченность, при которой равные элементы располагаются в последовательности в порядке их вставки.

Для сортированных диапазонов в качестве быстрой версии `count(first,last,value);` лучше использовать пару вызовов:

```
p = equal_range(first,last,value);
distance(p.first,p.second);
```

При поиске в ассоциативном контейнере лучше использовать вместо алгоритмов-не членов функции-члены с тем же именем. Функции-члены обычно более эффективны; например, функция-член `count` выполняется за логарифмическое время (так что, кстати, нет никаких оснований заменять ее вызовом `equal_range` с последующим `distance`, что имеет смысл для функции `count`, не являющейся членом).

Ссылки

[Austern99] §13.2-3 • [Bentley00] §13 • [Meyers01] §34, §45 • [Musser01] §22.2 • [Stroustrup00] §17.1.4.1, §18.7.2

86. Пользуйтесь правильным алгоритмом сортировки

Резюме

При сортировке вы должны четко понимать, как работает каждый из сортирующих алгоритмов, и использовать наиболее дешевый среди тех, которые пригодны для решения вашей задачи.

Обсуждение

Вам не всегда требуется полный `sort`; обычно надо меньшее, и весьма редко — большее. В общем случае стандартные алгоритмы сортировки располагаются от наиболее дешевых до наиболее дорогих в следующем порядке: `partition`, `stable_partition`, `nth_element`, `partial_sort` (и его вариант `partial_sort_copy`), `sort` и `stable_sort`. Используйте наименее дорогой из алгоритмов, которые выполняют необходимую вам работу; применение излишне мощного алгоритма — расточительство.

Время работы алгоритмов `partition`, `stable_partition` и `nth_element` — линейное, что является очень хорошим показателем.

Алгоритмы `nth_element`, `partial_sort`, `sort` и `stable_sort` требуют итераторы произвольного доступа. Вы не можете использовать их при наличии только двунаправленных итераторов (например, `list<T>::iterator`). Если вам нужны данные алгоритмы, но у вас нет итераторов произвольного доступа, вы можете воспользоваться идиомой индексного контейнера: создайте контейнер, поддерживающий итераторы произвольного доступа (например, `vector`), в котором будут храниться итераторы, указывающие на элементы интересующего вас диапазона, и затем примените к нему более мощный алгоритм с использованием разыменовывающей версии вашего предиката (в которой перед обычным сравнением выполняется разыменование итераторов).

Версии `stable_...` следует применять только тогда, когда вам необходимо сохранить относительный порядок одинаковых элементов. Заметим, что алгоритмы `partial_sort` и `nth_element` не являются устойчивыми (т.е. они не оставляют одинаковые элементы в том же относительном порядке, в котором они находились до сортировки), и у них нет стандартизированных устойчивых версий. Если вам все же требуется сохранение относительной упорядоченности элементов, вероятно, вам надо использовать `stable_sort`.

Само собой разумеется, не следует прибегать ни к каким алгоритмам сортировки, если вы можете обойтись без них. Если вы пользуетесь стандартным ассоциативным контейнером (`set/multiset` или `map/multimap`) или адаптером `priority_queue`, и вам требуется только один порядок сортировки, то не забывайте, что элементы в этих контейнерах всегда находятся в отсортированном виде.

Примеры

Пример 1. `partition`. Если вам надо разделить весь диапазон на две группы (группа элементов, удовлетворяющих предикату, за которыми следует группа элементов, предикату не удовлетворяющих), то для этого достаточно воспользоваться алгоритмом `partition`. Это все, что вам надо, чтобы ответить на вопросы наподобие приведенных далее.

- *Кто из студентов имеет средний бал не ниже 4.5?* Для ответа на этот вопрос можно воспользоваться вызовом `partition(students.begin(), students.end(), GradeAtLeast(4.5));`, который вернет итератор, указывающий на первого студента, чей средний балл ниже 4.5.
- *Какие из товаров имеют вес менее 10 кг?* Вызов `partition(products.begin(), products.end(), WeightUnder(10));` вернет итератор, указывающий на первый товар, вес которого не ниже 10 кг.

Пример 2. `nth_element`. Алгоритм `nth_element` можно использовать для того, чтобы получить один элемент в корректной n-й позиции, в которой он бы находился при полной сортировке всего диапазона, при этом все прочие элементы корректно располагаются до или после этого n-го элемента. Этого достаточно, чтобы ответить на вопросы наподобие следующих.

- *Перечислите 20 лучших покупателей.* Вызов `nth_element(s.begin(), s.begin()+19, s.end(), SalesRating);` помещает 20 наилучших покупателей в начало контейнера.
- *Какое изделие имеет медианное значение качества в данном наборе?* Искомый элемент находится в средней позиции отсортированного диапазона. Для его поиска достаточно вызова `nth_element(run.begin(), run.begin()+run.size()/2, run.end(), ItemQuality);`.
- *У какого изделия уровень качества находится на 75-м перцентиле?* Искомый элемент находится в позиции, отстоящей на 25% от начала отсортированного диапазона. Для его поиска достаточно вызова `nth_element(run.begin(), run.begin()+run.size()*.25, run.end(), ItemQuality);`.

Пример 3. `partial_sort`. Алгоритм `partial_sort` выполняет те же действия, что и `nth_element`, но кроме того обеспечивает корректное отсортированное размещение всех элементов до n-го. Алгоритм `partial_sort` используется для ответов на вопросы, аналогичные вопросам для `nth_element`, но в которых требуется, чтобы все интересующие элементы были корректно отсортированы. Этот алгоритм — все, что вам надо для ответа, например, на вопрос: "Кто из участников занял первое, второе и третье места?" Ответ можно получить при помощи вызова `partial_sort(contestants.begin(), contestants.begin()+3, contestants.end(),ScoreCompare);`, после которого участники, занявшие три первые места, окажутся в корректном порядке в трех первых элементах контейнера, и не более того.

Исключения

Хотя обычно алгоритм `partial_sort` быстрее полной сортировки (так как должен выполнять меньшее количество работы), если вам надо отсортировать почти весь (или весь) диапазон, то в этой ситуации алгоритм `sort` может оказаться быстрее.

Ссылки

[Austern99] §13.1 • [Bentley00] §11 • [Josuttis99] §9.2.2 • [Meyers01] §31 • [Musser01] §5.4, §22.26 • [Stroustrup00] §17.1.4.1, §18.7

87. Делайте предикаты чистыми функциями

Резюме

Предикат представляет собой функциональный объект, который возвращает ответ да/нет, обычно в виде значения типа `bool`. Функция является "чистой" в математическом смысле, если ее результат зависит только от ее аргументов (обратите внимание — в данном случае термин "чистая" не имеет никакого отношения к чисто виртуальным функциям).

Не позволяйте предикатам сохранять или обращаться к состоянию так, чтобы это могло влиять на результат работы оператора `operator()`; при этом понятие состояния включает как данные-члены, так и глобальные состояния. Для предикатов желательно делать оператор `operator()` константной функцией-членом (см. рекомендацию 15).

Обсуждение

Алгоритмы создают неизвестное количество копий предикатов в неизвестные моменты времени и в неизвестном порядке, так что приходится полагаться на то, что все копии эквивалентны.

Именно поэтому вы отвечаете за то, чтобы все копии предикатов были эквивалентны; это означает, что все они должны быть чистыми функциями, результат работы которых полностью и однозначно определяется аргументами, передаваемыми оператору `operator()` и не зависит ни от каких иных факторов. При передаче одних и тех же аргументов предикат всегда должен возвращать одно и то же значение.

Предикаты с состояниями могут показаться полезными, но они явно *не* очень полезны при использовании с алгоритмами стандартной библиотеки C++, и это сделано преднамеренно. В частности, предикаты с состояниями могут быть полезны только при выполнении ряда условий.

- *Предикат не копируется.* Стандартные алгоритмы не дают такой гарантии; в действительности алгоритмы, напротив, предполагают, что предикаты могут безопасно копироваться.
- *Предикаты используются в предопределенном документированном порядке.* В общем случае стандартные алгоритмы не дают никакой гарантии относительно порядка применения предикатов к элементам диапазона. При отсутствии гарантий по поводу порядка обработки элементов, операция наподобие "пометить третий элемент" (см. примеры) имеет мало смысла, поскольку не определено, какой именно элемент будет обработан третьим.

Первое условие можно обойти, написав предикат с использованием счетчика ссылок. Этот метод решает проблему копирования предикатов, поскольку в таком случае предикаты могут безопасно копироваться без изменения их семантики при применении к объектам (см. [Sutter02]). Однако обойти второе условие оказывается невозможно.

Всегда объявляйте оператор предиката `operator()` как константную функцию-член, чтобы компилятор мог помочь вам избежать неприятностей, выводя сообщение об ошибке при попытках изменить любые данные-члены, которые могут быть у предиката. Это не позволяет пресечь все злоупотребления, например, доступ к глобальным данным, но, по крайней мере, поможет избежать наиболее распространенных ошибок.

Примеры

Пример. `FlagNth`. Перед вами классический пример из [Sutter02], в котором выполняется попытка удалить третий элемент из контейнера `v`.

```
class FlagNth {
public:
    FlagNth( size_t n ) : current_(0), n_(n) { }

    // Возвращаем значение true только при третьем вызове
    template<typename T>
    bool operator()( const T& )       // Плохо: неконстантная
        { return ++current_ == n_; } // функция
private:
    size_t current_, n_;
};

// ... позже ...
v.erase( remove_if( v.begin(), v.end(), FlagNth(3) ) );
```

Увы, нет никакой гарантии, что будет удален именно третий элемент В большинстве реальных реализаций STL приведенный код наряду с третьим удалит и шестой элемент. Почему? Потому что `remove_if` обычно реализуется с использованием `find_if` и `remove_copy_if`, и копия предиката передается каждой из этих функций.

Концептуально этот пример неверен, поскольку алгоритм `remove_if` гарантирует только то, что он удалит все элементы, удовлетворяющие некоторому критерию. Он не документирует порядок, в котором совершается обход или удаление элементов из обрабатываемого диапазона, так что приведенный код использует предположение, которое не документировано и, более того, не выполняется.

Корректный способ удаления третьего элемента — выполнить итерации для его поиска и вызвать функцию `erase`.

Ссылки

[Austern99] §4.2.2 • [Josuttis99] §5.8.2, §8.1.4 • [Meyers01] §39 • [Stroustrup00] §10.2.6 • [Sutter02] §2-3

88. В качестве аргументов алгоритмов и компараторов лучше использовать функциональные объекты, а не функции

Резюме

Предпочтительно передавать алгоритмам функциональные объекты, а не функции, а компараторы ассоциативных контейнеров просто должны быть функциональными объектами. Функциональные объекты адаптируемы и, вопреки ожиданиям, обычно дают более быстрый по сравнению с функциями код.

Обсуждение

Во-первых, функциональные объекты легко сделать адаптируемыми (и такими их и следует делать — см. рекомендацию 89). Даже если у вас есть готовая функция, иногда для ее использования требуется "обертка" из `ptr_fun` или `mem_fun`. Например, такая обертка требуется при построении более сложных выражений с использованием связывателей (см. также рекомендацию 84):

```
inline bool IsHeavy( const Thing& ) { /* ... */ }
find_if(v.begin(), v.end(), not1(IsHeavy)); // Ошибка
```

Обойти эту ошибку обычно можно путем применения `ptr_fun` (или, в случае функции-члена, `mem_fun` или `mem_fun_ref`), что, к сожалению, не работает в данном конкретном случае:

```
inline bool IsHeavy( const Thing& ) { /* ... */ }
find_if(v.begin(), v.end(),
        not1(ptr_fun(IsHeavy))); // Героическая попытка...
```

Беда в том, что этот способ не будет работать, даже если вы явно укажете аргументы шаблона `ptr_fun`. Коротко говоря, проблема в том, что `ptr_fun` точно выводит типы аргументов и возвращаемый тип (в частности, тип параметра будет выведен как `const Thing&`) и создает внутренний механизм, который, в свою очередь, пытается добавить другой &, а ссылка на ссылку в настоящее время в C++ не разрешена. Имеются способы исправлений языка и/или библиотека для решения данной проблемы (например, позволяя ссылке на ссылку свернуться в обычную ссылку; см. также рекомендацию 89), но на сегодняшний день проблема остается нерешенной.

Прибегать ко всем этим ухищрениям совершенно излишне, если у вас имеется корректно написанный функциональный объект (см. рекомендацию 89), который можно использовать без какого-либо специального синтаксиса:

```
struct IsHeavy : unary_function<Thing, bool> {
    bool operator()( const Thing& ) const { /* ... */ }
};

find_if(v.begin(), v.end(), not1(IsHeavy())); // OK
```

Еще более важно то, что для определения сравнения в ассоциативных контейнерах вам нужен именно функциональный объект, а не функция. Это связано с тем, что нельзя инстанцировать шаблон с функцией в качестве параметра:

```
bool CompareThings( const Thing&, const Thing& );
set<Thing, CompareThings> s; // Ошибка
```

Вместо этого следует написать:

```
struct CompareThings
: public binary_function<Thing,Thing,bool>
{
    bool operator()( const Thing&, const Thing& ) const;
};

set<Thing, CompareThings> s; // OK
```

Наконец, имеется еще одно преимущество функциональных объектов — эффективность. Рассмотрим следующий знакомый алгоритм:

```
template<typename Iter, typename Compare>
Iter find_if( Iter first, Iter last, Compare comp );
```

Если мы передадим алгоритму в качестве компаратора функцию

```
inline bool Function( const Thing& ) { /* ... */ }
find_if( v.begin(), v.end(), Function );
```

то на самом деле будет передана ссылка на функцию. Компиляторы редко встраивают вызовы таких функций (за исключением некоторых относительно свежих компиляторов, которые в состоянии провести анализ всей программы в целом), даже если они объявлены как таковые и видимы в момент компиляции вызова `find_if`. Кроме того, как уже упоминалось, функции не адаптируемы.

Давайте передадим алгоритму `find_if` в качестве компаратора функциональный объект:

```
struct FunctionObject : unary_function<Thing, bool> {
    bool operator()( const Thing& ) const { /* ... */ }
};

find_if( v.begin(), v.end(), FunctionObject() );
```

Если мы передаем объект, который имеет (явно или неявно) встраиваемый оператор `operator()`, то такие вызовы компиляторы C++ способны делать встраиваемыми уже очень давно.

Примечание. Эта методика не является преждевременной оптимизацией (см. рекомендацию 8); ее следует рассматривать как препятствие преждевременной пессимизации (см. рекомендацию 9). Если у вас имеется готовая функция — передавайте указатель на нее (кроме тех ситуаций, когда вы должны обязательно обернуть ее в `ptr_fun` или `mem_fun`). Но если вы пишете новый код для использования в качестве аргумента алгоритма, то лучше сделать его функциональным объектом.

Ссылки

[Austern99] §4, §8, §15 • [Josuttis99] §5.9 • [Meyers01] §46 • [Musser01] §8 • [Sutter04] §25

89. Корректно пишите функциональные объекты

Резюме

Разрабатывайте функциональные объекты так, чтобы их копирование выполнялось как можно эффективнее. Там, где это возможно, делайте их максимально адаптируемыми путем наследования от `unary_function` или `binary_function`.

Обсуждение

Функциональные объекты моделируют указатели на функции. Подобно указателям на функции, они обычно передаются в функции по значению. Все стандартные алгоритмы передают объекты по значению, и то же должны делать и ваши алгоритмы, например:

```
template<class InputIter, class Func>
Function for_each(InputIter first, InputIter last, Function f);
```

Следовательно, функциональные объекты должны легко копироваться и быть мономорфными (для защиты от срезки), так что избегайте виртуальных функций (см. рекомендацию 54). Конечно, у вас могут быть большие и/или полиморфные функциональные объекты — их тоже вполне можно использовать; просто скройте их размер с помощью идиомы Pimpl (указателя на реализацию; см. рекомендацию 43). Эта идиома позволяет, как и требуется, получить внешний мономорфный класс малого размера, обеспечивающий доступ к богатой функциональности. Внешний класс должен удовлетворять следующим условиям.

- *Быть адаптируемым*. Наследуйте его от `unary_function` или `binary_function`.
- *Использовать идиому Pimpl*. Такой класс содержит указатель (например, `shared_ptr`) на (возможно, большого размера) реализацию необходимой функциональности.
- *Иметь оператор(ы) вызова функции*. Эти операторы передают вызовы объекту-реализации.

Этим ограничиваются требования к внешнему классу (не считая возможного наличия собственных (не генерируемых компилятором) конструкторов, оператора присваивания и/или деструктора.

Функциональные объекты должны быть адаптируемы. Стандартные связыватели и адаптеры полагаются на наличие определенных инструкций `typedef`, обеспечить которые легче всего при наследовании ваших функциональных объектов от `unary_function` или `binary_function`. Инстанцируйте `unary_function` или `binary_function` с теми типами, которые получает и возвращает ваш оператор `operator()` (при этом у каждого типа, не являющегося указателем, следует убрать все спецификаторы `const` верхнего уровня, а также все &).

Постарайтесь избежать наличия нескольких операторов `operator()`, поскольку это затрудняет адаптируемость. Дело в том, что обычно оказывается невозможно обеспечить корректные инструкции `typedef`, необходимые для адаптирования, поскольку один и тот же синоним типа, определяемый через инструкцию `typedef`, имеет разные значения для разных операторов `operator()`.

Не все функциональные объекты являются предикатами — предикаты представляют собой подмножество функциональных объектов (см. рекомендацию 87).

Ссылки

[Allison98] §15, §C • [Austern99] §4, §8, §15 • [Gamma95] Bridge • [Josuttis99] §8.2.4 • [Koenig97] §21, §29 • [Meyers97] §34 • [Meyers01] §38, §40, §46 • [Musser01] §2.4, §8, §23 • [Sutter00] §26-30 • [Vandevoorde03] §22

Безопасность типов

Если вы лжете компилятору, он будет мстить.

— Генри Спенсер (Henry Spencer)

Всегда будут вещи, которые мы будем хотеть сказать в наших программах и которые трудно сформулировать на любом языке программирования.

— Алан Перлис (Alan Perlis)

Последней (не по важности) темой книги является корректность типов — очень важное свойство программ, которое вы должны изо всех сил стараться поддерживать. Теоретически корректная с точки зрения типов функция не может обратиться к нетипизированной памяти или вернуть неверные значения. На практике, если ваш код поддерживает корректность типов, он тем самым избегает большого количества неприятных ошибок, от непереносимости программ до порчи содержимого памяти и неопределенного поведения программ.

Основная идея поддержки корректности типов — всегда считывать информацию в том формате, в котором она была записана. Иногда C++ позволяет легко нарушить это правило; приведенные в этом разделе рекомендации помогут вам избежать подобных ошибок.

В этом разделе мы считаем наиболее значимой рекомендацию 91 — "Работайте с типами, а не с представлениями". Система типов — ваш друг и верный союзник. Воспользуйтесь ее помощью и попытайтесь не злоупотреблять ее доверием.

90. Избегайте явного выбора типов — используйте полиморфизм

Резюме

Избегайте явного выбора типа объекта для настройки поведения. Используйте шаблоны и виртуальные функции для того, чтобы поведение объекта определялось его типом, а не вызывающим кодом.

Обсуждение

Настройка поведения в зависимости от типа объекта с использованием инструкции выбора `switch` — это ненадежный, чреватый ошибками, небезопасный метод, представляющий собой перенос методов C или Fortran в C++. Это жесткая технология, заставляющая вас всякий раз при добавлении новых возможностей переписывать уже готовый и отлаженный код. Этот метод небезопасен еще и потому, что компилятор не может подсказать вам, что вы забыли внести дополнения в какую-то из инструкций switch при добавлении нового типа.

В идеале добавление новых возможностей в программу должно осуществляться добавлением нового кода, а не изменением старого (см. рекомендацию 37). В реальной жизни это не всегда так — зачастую в дополнение к написанию нового кода мы вынуждены вносить изменения в уже имеющийся код. Такие изменения, однако, крайне нежелательны и должны быть минимизированы по двум причинам. Во-первых, изменения могут нарушить имеющуюся функциональность. Во-вторых, они препятствуют масштабируемости при росте системы и добавлении новых возможностей, поскольку количество "узлов поддержки", к которым надо возвращаться и вносить изменения, все время возрастает. Это наблюдение приводит к принципу Открытости-Закрытости, который гласит: любая сущность (например, класс или модуль) должна быть открыта для расширений, но закрыта для изменений (см. [Martin96c] и [Meyer00]).

Каким же образом мы можем написать код, который будет легко расширяем без внесения изменений? Используйте полиморфизм для написания кода в терминах абстракций (см. также рекомендацию 36), после чего при необходимости добавления функциональности это можно будет сделать путем разработки и добавления различных реализаций упомянутых абстракций. Шаблоны и виртуальные функции образуют барьер для зависимостей между кодом, использующим абстракции, и кодом, их реализующим (см. рекомендацию 64).

Конечно, управление зависимостями обусловлено выбором верных абстракций. Если абстракции несовершенны, добавление новой функциональности потребует изменений интерфейса (а не просто добавления новых реализаций интерфейса), которые обычно влекут за собой значительные изменения существующего кода. Но абстракции потому и называются "абстракциями", что предполагается их большая стабильность по сравнению с "деталями", т.е. возможными реализациями абстракций.

Совсем иначе обстоит дело с предельно детализированным кодом, который использует мало абстракций или вовсе обходится без них, работая исключительно с конкретными типами и их отдельными операциями. Добавление новой функциональности в такой код — сущее мучение.

Примеры

Пример. Рисование фигур. Классический пример — рисование различных объектов. Типичный подход в стиле C использует выбор типа. Для этого определяется член-перечисление `id_`, который хранит тип каждой фигуры — прямоугольник, окружность и т.д. Рисующий код выполняет необходимые действия в зависимости от выбранного типа:

```
class Shape { // ...
    enum { RECTANGLE, TRIANGLE, CIRCLE } id_;

    void Draw() const {
        switch( id_ ) {         // Плохой метод
        case RECTANGLE:
            // ... Код для прямоугольника ...
            break;

        case TRIANGLE:
            // ... Код для треугольника ...
            break;

        case CIRCLE:
            // ... Код для окружности ...
            break;

        default:     // Плохое решение
            assert(!"При добавлении нового типа надо "
                    "обновить эту конструкцию" );
            break;
        }
    }
};
```

Такой код сгибается под собственным весом, он хрупок, ненадежен и сложен. В частности, он страдает транзитивной циклической зависимостью, о которой говорилось в рекомендации 22. Ветвь по умолчанию конструкции `switch` — четкий симптом синдрома "не знаю, что мне делать с этим типом". И все эти болезненные неприятности полностью исчезают, стоит только вспомнить, что C++ — объектно-ориентированный язык программирования:

```
class Shape {   // ...
    virtual void Draw() const = 0; // Каждый производный
                                   // класс реализует свою функцию
};
```

В качестве альтернативы (или в качестве дополнения) рассмотрим реализацию, которая следует совету по возможности принимать решения во время компиляции (см. рекомендацию 64):

```
template<class S>
void Draw( const S& shape ) {
    shape.Draw(); // Может быть виртуальной, а может и не быть
}; // См. рекомендацию 64
```

Теперь ответственность за рисование каждой геометрической фигуры переходит к реализации самой фигуры, и синдром "не знаю, что делать с этим типом" просто невозможен.

Ссылки

[Dewhurst03] §69, §96 • [Martin96c] • [Meyer00] • [Stroustrup00] §12.2.5 • [Sutter04] §36

91. Работайте с типами, а не с представлениями

Резюме

Не пытайтесь делать какие-то предположения о том, как именно объекты представлены в памяти. Как именно следует записывать и считывать объекты из памяти — пусть решают типы объектов.

Обсуждение

Стандарт C++ дает очень мало гарантий по поводу представления типов в памяти.

- Целые числа используют двоичное представление.
- Для отрицательных чисел используется дополнительный код числа в двоичной системе.
- Обычные старые типы (Plain Old Data, POD[5]) имеют совместимое с C размещение в памяти: переменные-члены хранятся в порядке их объявления.
- Тип `int` занимает как минимум 16 битов.

В частности, достаточно распространенные соглашения *не* гарантированы ни для всех имеющихся архитектур, ни тем более для архитектур, которые могут появиться в будущем. Так что не забывайте о следующем.

- Размер `int` не равен ни 32 битам, ни какому-либо иному фиксированному размеру.
- Указатели и целые числа не всегда имеют один и тот же размер и не могут свободно преобразовываться друг в друга.
- Размещение класса в памяти не всегда приводит к размещению базового класса и членов в указанном порядке.
- Между членами класса (даже если они являются POD) могут быть промежутки в целях выравнивания.
- `offsetof` работает только для POD, но не для всех классов (хотя компилятор может и не сообщать об ошибках).
- Класс может иметь скрытые поля.
- Указатели могут быть совсем не похожи на целые числа. Если два указателя упорядочены и вы можете преобразовать их в целые числа, то получающиеся значения могут быть упорядочены иначе.
- Нельзя переносимо полагаться на конкретное размещение автоматических переменных в памяти или на направление роста стека.

[5] Неформально POD означает любой тип, представляющий собой набор простых данных, возможно, с пользовательскими функциями-членами для удобства. Говоря более строго, POD представляет собой класс или объединение, у которого нет пользовательского конструктора, копирующего присваивания, и деструктора, а также нет (нестатических) членов-данных, являющихся ссылками, указателями на члены или не являющихся POD. — *Прим. ред.*

- Указатели на функции могут иметь размер, отличный от размера указателя `void*`, несмотря на то, что некоторые API заставляют вас предположить, что их размеры одинаковы.
- Из-за вопросов выравнивания вы не можете записывать ни один объект по произвольному адресу в памяти.

Просто корректно определите типы, а затем читайте и записывайте данные с использованием указанных типов вместо работы с отдельными битами, словами и адресами. Модель памяти C++ гарантирует эффективную работу, не заставляя вас при этом работать с представлениями данных в памяти. Так и не делайте этого.

Ссылки

[Dewhurst03] §95

92. Избегайте `reinterpret_cast`

Резюме

Как гласит римская пословица, у лжи короткие ноги. Не пытайтесь использовать `reinterpret_cast`, чтобы заставить компилятор рассматривать биты объекта одного типа как биты объекта другого типа. Такое действие противоречит безопасности типов.

Обсуждение

Вспомните: *Если вы лжете компилятору, он будет мстить* (Генри Спенсер).

Преобразование `reinterpret_cast` отражает представления программиста о представлении объектов в памяти, т.е. программист берет на себя ответственность за то, что он лучше компилятора знает, что можно и что нельзя. Компилятор молча сделает то, что вы ему скажете, но применять такую грубую силу в отношениях с компилятором — последнее дело. Избегайте каких-либо предположений о представлении данных, поскольку такие предположения очень сильно влияют на безопасность и надежность вашего кода.

Кроме того, реальность такова, что результат применения `reinterpret_cast` еще хуже, чем просто насильственная интерпретация битов объекта (что само по себе достаточно нехорошо). За исключением некоторых гарантированно обратимых преобразований результат работы `reinterpret_cast` зависит от реализации, так что вы даже не знаете точно, как именно он будет работать. Это очень ненадежное и непереносимое преобразование.

Исключения

Некоторые низкоуровневые специфичные для данной системы программы могут заставить вас применить `reinterpret_cast` к потоку битов, проходящих через некоторый порт, или для преобразования целых чисел в адреса. Используйте такое небезопасное преобразование как можно реже и только в тщательно скрытых за абстракциями функциях, чтобы ваш код можно было переносить с минимальными изменениями. Если вам требуется преобразование между указателями несвязанных типов, лучше выполнять его через приведение к `void*` вместо непосредственного использования `reinterpret_cast`, т.е. вместо кода

```
T1* p1 = ... ;
T2* p2 = reinterpret_cast<T2*>( p1 );
```

лучше писать

```
T1* p1 = ... ;
void* pV = p1;
T2* p2 = static_cast<T2*>( pV );
```

Ссылки

[C++03] §5.2.10(3) • [Dewhurst03] §39 • [Stroustrup00] §5.6

93. Избегайте применения static_cast к указателям

Резюме

К указателям на динамические объекты не следует применят преобразование static_cast. Используйте безопасные альтернативы — от dynamic_cast до перепроектирования.

Обсуждение

Подумайте о замене static_cast более мощным оператором dynamic_cast, и вам не придется запоминать, в каких случаях применение static_cast безопасно, а в каких — чревато неприятностями. Хотя dynamic_cast может оказаться немного менее эффективным преобразованием, его применение позволяет обнаружить неверные преобразования типов (но не забывайте о рекомендации 8). Использование static_cast вместо dynamic_cast напоминает экономию на ночном освещении, когда выигрыш доллара в год оборачивается переломанными ногами.

При проектировании постарайтесь избегать понижающего приведения. Перепроектируйте ваш код таким образом, чтобы такое приведение стало излишним. Если вы видите, что передаете в функцию базовый класс там, где в действительности потребуется производный класс, проследите всю цепочку вызовов, чтобы понять, где же оказалась потерянной информация о типе; зачастую изменение пары прототипов оказывается замечательным решением, которое к тому же делает код более простым и понятным.

Чрезмерное применение понижающего приведения может служить признаком слишком бедного интерфейса базового класса. Такой интерфейс может привести к тому, что большая функциональность определяется в производных классах, и всякий раз при необходимости расширения интерфейса приходится использовать понижающее приведение. Одно из решений данной проблемы — перепроектирование базового интерфейса в целях повышения функциональности.

Тогда и только тогда, когда становятся существенны накладные расходы, вызванные применением dynamic_cast (см. рекомендацию 8), следует подумать о разработке собственного преобразования типов, который использует dynamic_cast при отладке и static_cast в окончательной версии (см. [Stroustrup00]):

```
template<class To, class From> To checked_cast(From* from) {
  assert(dynamic_cast<To>(from) ==
         static_cast<To>(from) && "checked_cast failed" );
  return static_cast<To>( from );
}

template<class To, class From> To checked_cast(From& from) {
  typedef tr1::remove_reference<To>::type* ToPtr; // [C++TR104]
  assert(dynamic_cast<ToPtr>(&from) ==
         static_cast<ToPtr>(&from) && "checked_cast failed" );
  return static_cast<To>( from );
}
```

Эта пара функций (по одной для указателей и ссылок) просто проверяет согласованность двух вариантов преобразования. Вы можете либо самостоятельно приспособить checked_cast для своих нужд, либо использовать имеющийся в библиотеке.

Ссылки

[Dewhurst03] §29, §35, §41 • [Meyers97] §39 • [Stroustrup00] §13.6.2 • [Sutter00] §44

94. Избегайте преобразований, отменяющих const

Резюме

Преобразование типов, отменяющее const, может привести к неопределенному поведению, а кроме того, это свидетельство плохого стиля программирования даже в том случае, когда применение такого преобразования вполне законно.

Обсуждение

Применение const — это дорога с односторонним движением, и, воспользовавшись этим спецификатором, вы не должны давать задний ход. Если вы отменяете const для объекта, который изначально был объявлен как константный, задний ход приводит вас на территорию неопределенного поведения. Например, компилятор может (и, бывает, так и поступает) поместить константные данные в память только для чтения (ROM) или в страницы памяти, защищенные от записи. Отказ от const у такого истинно константного объекта — преступный обман, зачастую караемый аварийным завершением программы из-за нарушения защиты памяти.

Даже если ваша программа не потерпит крах, отмена const представляет собой отмену обещанного и не делает того, чего от нее зачастую ожидают. Например, в приведенном фрагменте не происходит выделения массива переменной длины:

```
void Foolish( unsigned int n ) {
    const unsigned int size = 1;
    const_cast<unsigned int&>(size) = n; // Не делайте так!
    char buffer[size];                   // Размер массива
    // ...                               // все равно равен 1
}
```

В C++ имеется одно неявное преобразование const_cast из строкового литерала в char*:

```
char* weird = "Trick or treat?";
```

Компилятор молча выполняет преобразование const_cast из const char[16] в char*. Это преобразование позволено для совместимости с API в стиле C, хотя и представляет собой дыру в системе типов C++. Строковые литералы могут размещаться в памяти только для чтения, и попытка изменения такой строки может вызвать нарушение защиты памяти.

Исключения

Преобразование, отменяющее const, может оказаться необходимым для вызова функции API, некорректно указывающей константность (см. рекомендацию 15). Оно также полезно, когда функция, которая должна получать и возвращать ссылку одного и того же типа, имеет как константную, так и неконстантную перегрузки, причем одна из них вызывает другую:

```
const Object& f( const Object& );

Object& f( Object& obj ) {
    const Object& ref = obj;
    return const_cast<Object&>(f(ref)); // Преобразование
}                                       // возвращаемого типа
```

Ссылки

[Dewhurst03] §32, §40 • [Sutter00] §44

95. Не используйте преобразование типов в стиле C

Резюме

Возраст не всегда означает мудрость. Старое преобразование типов в стиле C имеет различную (и часто опасную) семантику в зависимости от контекста, спрятанную за единым синтаксисом. Замена преобразования типов в стиле C преобразованиями C++ поможет защититься от неожиданных ошибок.

Обсуждение

Одна из проблем, связанных с преобразованием типов в стиле C, заключается в том, что оно использует один и тот же синтаксис для выполнения несколько разных вещей, в зависимости от таких мелочей, как, например, какие именно заголовочные файлы включены при помощи директивы `#include`. Преобразования типов в стиле C++, сохраняя определенную опасность, присущую преобразованиям вообще, имеют четко документированное предназначение, их легко найти, дольше писать (что дает время дважды подумать при их использовании), и не позволяют незаметно выполнить опасное преобразование `reinterpret_cast` (см. рекомендацию 92).

Рассмотрим следующий код, в котором `Derived` — производный от базового класса `Base`:

```
extern void Fun( Derived* );

void Gun( Base* pb ) {
    // Будем считать, что функция Gun знает, что pb в
    // действительности указывает на объект типа Derived и
    // хочет передать его функции Fun
    Derived* pd = (Derived*)pb;  // Плохо: преобразование
    Fun( pd );                   // в стиле C
}
```

Если функция `Gun` имеет доступ к определению `Derived` (например, при включении заголовочного файла `derived.h`), то компилятор имеет всю необходимую информацию о размещении объекта, чтобы выполнить все необходимые действия по корректировке указателя при преобразовании от `Base` к `Derived`. Но если автор `Gun` забыл включить соответствующий файл определения, и функции `Gun` видно только предварительное объявление класса `Derived`, то компилятор будет полагать, что `Base` и `Derived` — несвязанные типы, и интерпретирует биты указателя `Base*` как биты указателя `Derived*`, не делая никаких коррекций, которые могут диктоваться размещением объекта в памяти!

Коротко говоря, если вы забудете включить определение класса, то ваш код может аварийно завершиться без видимых причин, при том что компилятор не сообщил ни об одной ошибке. Избавимся от проблемы следующим способом:

```
extern void Fun( Derived* );

void Gun( Base* pb ) {
    // Если мы гарантированно знаем, что pb на самом деле
    // указывает на объект типа Derived:

    // Преобразование в стиле C++
    Derived* pd = static_cast<Derived*>(pb);
```

```
    // В противном случае следует использовать

    //          = dynamic_cast<Derived*>(pb);
    Fun(pd);
}
```

Теперь, если у компилятора недостаточно статической информации об отношениях между `Base` и `Derived`, он выведет сообщение об ошибке, вместо того чтобы автоматически применить побитовое (и потенциально опасное) преобразование `reinterpret_cast` (см. рекомендацию 92).

Преобразования в стиле C++ могут защитить корректность вашего кода в процессе эволюции системы. Пусть, например, у вас есть иерархия с корнем в `Employee`, и вам надо определить уникальный идентификатор `ID` для каждого объекта `Employee`. Вы можете определить `ID` как указатель на сам объект `Employee`. Указатели однозначно идентифицируют объекты, на которые указывают, и могут сравниваться на равенство друг другу — что в точности то, что нам и надо. Итак, запишем:

```
typedef Employee* EmployeeID;

Employee& Fetch( EmployeeID id ) {
    return *id;
}
```

Пусть вы кодируете часть системы с данным дизайном. Пусть позже вам требуется сохранять ваши записи в реляционной базе данных. Понятно, что сохранение указателей — не то, что вам требуется. В результате вы изменяете дизайн так, чтобы каждый объект имел уникальный целочисленный идентификатор. Тогда целочисленный идентификатор может храниться в базе данных, а хэш-таблица отображает идентификаторы на объекты `Employee`. Теперь `typedef` выглядит следующим образом:

```
typedef int EmployeeID;

Employee& Fetch( EmployeeID id ) {
    return employeeTable_.lookup(id);
}
```

Это корректный дизайн, и вы ожидаете, что любое неверное употребление `EmployeeID` должно привести к ошибке времени компиляции. Так и получается, за исключением следующего небольшого фрагмента:

```
void TooCoolToUseNewCasts( EmployeeID id ) {
    Secretary* pSecretary = (Secretary*)id; // Плохо:
    // ...                  // преобразование в стиле C
}
```

При использовании старой инструкции `typedef` преобразование в стиле C выполняет `static_cast`, при новой будет выполнено `reinterpret_cast` с некоторым целым числом, что даст нам неопределенное поведение программы (см. рекомендацию 92).

Преобразования в стиле C++ проще искать в исходных текстах при помощи автоматического инструментария наподобие `grep` (но никакое регулярное выражение `grep` не позволит выловить синтаксис преобразования типов в стиле C). Поскольку преобразования очень опасны (в особенности `static_cast` для указателей и `reinterpret_cast`; см. рекомендацию 92), использование автоматизированного инструментария для их отслеживания — неплохая идея.

Ссылки

[Dewhurst03] §40 • [Meyers96] §2 • [Stroustrup00] §15.4.5 • [Sutter00] §44

96. Не применяйте memcpy или memcmp к не-POD типам

Резюме

Не работайте рентгеновским аппаратом (см. рекомендацию 91). Не используйте memcpy и memcmp для копирования или сравнения чего-либо структурированного более, чем обычная память.

Обсуждение

Функции memcpy и memcmp нарушают систему типов. Использовать memcpy для копирования объектов — это то же, что использовать ксерокс для копирования денег, а сравнивать объекты при помощи memcmp — то же, что сравнивать двух леопардов по количеству пятен. Инструменты и методы могут казаться подходящими для выполнения работы, но они слишком грубы для того, чтобы сделать ее правильно.

Объекты C++ предназначены для сокрытия данных (возможно, наиболее важный принцип в разработке программного обеспечения; см. рекомендацию 11). Объекты скрывают данные (см. рекомендацию 41) и предоставляют точные абстракции для копирования этих данных посредством конструкторов и операторов присваивания (см. рекомендации с 52 по 55). Пройтись по ним грубым инструментом типа memcpy — серьезное нарушение принципа сокрытия информации, которое зачастую приводит к утечкам памяти и ресурсов (в лучшем случае), аварийному завершению программы (в случае похуже) или неопределенному поведению (в самом худшем случае). Например:

```
{
    // Создаем два int в памяти
    shared_ptr<int> p1( new int ), p2( new int );

    memcpy(&p1, &p2, sizeof(p1)); // Так делать нельзя!!!

} // Утечка памяти: p2 никогда не удаляется
  // Повреждение памяти: p1 удаляется дважды
```

Неверное применение memcpy может влиять на такие фундаментальные свойства, как тип и сущность объекта. Компиляторы часто добавляют к полиморфным объектам скрытые данные (так называемый указатель на виртуальную таблицу), которые определяют сущность объекта во время выполнения программы. В случае множественного наследования в объекте содержится несколько таких таблиц, с различными смещениями внутри объекта, и большинство реализаций добавляют дополнительные внутренние указатели при виртуальном наследовании. При обычном использовании компилятор принимает меры для корректного управления всеми скрытыми полями; применение memcpy способно внести в этот механизм только хаос.

Аналогично, функция memcmp — неподходящий инструмент для сравнения чего-то более сложного, чем просто наборы битов. Иногда эта функция делает слишком мало (например, сравнение строк в стиле C — не то же, что и сравнение указателей, при помощи которых эти строки реализованы). А иногда, как это ни парадоксально, memcmp делает слишком много (например, memcmp может совершенно напрасно сравнивать байты, которые не являются частью состояния объекта, такие как заполнители, вставленные компилятором для выравнивания). В обоих случаях результат сравнения оказывается неверным.

Ссылки

[Dewhurst03] §50 • [Stroustrup94] §11.4.4

97. Не используйте объединения для преобразований

Резюме

Хитрость все равно остается ложью: объединения можно использовать для получения "преобразования типа без преобразования", записывая информацию в один член и считывая из другого. Однако это еще более опасно и менее предсказуемо, чем применение `reinterpret_cast` (см. рекомендацию 92).

Обсуждение

Не считывайте данные из поля объединения, если последняя запись была не в это же поле. Чтение из поля, отличного от поля, в которое производилась запись, имеет неопределенное поведение, и использование этого метода еще хуже, чем применение `reinterpret_cast` (см. рекомендацию 92); в последнем случае компилятор, как минимум, может предупредить программиста и не допустить "невозможной интерпретации" наподобие указателя в `char`. При использовании для этой цели объединения никакая интерпретация не приведет к ошибке времени компиляции (как и к надежному результату).

Рассмотрим фрагмент кода, предназначенного для сохранения значения одного типа (`char*`) и выборки битов этого значения в виде величины иного типа (`long`):

```
union {
    long intValue_;
    char* pointerValue_;
};

pointerValue_   = somePointer;
long int gotcha = intValue_;
```

Здесь есть две проблемы.

- *Данный код требует слишком многого.* Он полагается на то, что `sizeof(long)` и `sizeof(char*)` равны и что их битовые представления идентичны. Эти утверждения справедливы не для всех возможных реализаций (см. рекомендацию 91).
- *Он скрывает свое предназначение как от человека, так и от компилятора.* Игры с объединениями затрудняют для компилятора поиск ошибок, связанных с типами, а для человека — выявление логических ошибок.

Исключения

Если две POD-структуры являются членами объединения и начинаются с полей одних и тех же типов, можно записывать одно из таких полей, а затем считывать данные из другого.

Ссылки

[Alexandrescu02b] • [Stroustrup00] §C.8.2 • [Sutter04] §36

98. Не используйте неизвестные аргументы (троеточия)

Резюме

Наличие троеточий в C++ — опасное наследие C. Избегайте их в своих программах; используйте вместо этого высокоуровневые конструкции и библиотеки C++.

Обсуждение

Функции с переменным количеством аргументов достаточно удобны, однако использование неизвестных аргументов в стиле C — не лучший способ получения таких функций. Эти аргументы имеют много серьезных недостатков.

- *Недостаточная безопасность типов.* По сути, троеточие говорит компилятору: "Выключи все проверки. С этого момента я все беру на себя, и теперь начинает работать `reinterpret_cast`". (См. рекомендацию 92).
- *Сильное связывание и необходимость согласования вызываемого и вызывающего кода вручную.* Проверка типов языком оказывается отключена, так что вызывающий код должен использовать иные способы для сообщения о типах передаваемых аргументов. Такие протоколы (например, форматная строка `printf`) подвержены ошибкам и небезопасны, поскольку не могут быть полностью проверены ни вызывающим, ни вызываемым кодом. (См. рекомендацию 99.)
- *Неопределенное поведение для объектов типов, являющихся классами.* Передача чего бы то ни было кроме POD и примитивных типов вместо троеточий приводит к неопределенному поведению в C++. К сожалению, большинство компиляторов даже не предупреждает об этом.
- *Неизвестное количество аргументов.* Даже в случае простейших функций с переменным количеством аргументов (например, `min`) вам все равно требуется протокол, позволяющий указать количество передаваемых аргументов. (Как ни смешно, но это, пожалуй, хорошее свойство, поскольку является еще одним препятствием широкому распространению функций с переменным числом аргументов.)

Избегайте троеточий в сигнатурах ваших функций. Избегайте вызова функций с переменным количеством аргументов со своими собственными сигнатурами, даже если это вполне корректные функции из стандартной библиотеки C, такие как `sprintf`. Вызовы `sprintf` часто выглядят более компактными и простыми для понимания, чем эквивалентные вызовы с использованием форматирования `stringstream` и операторов `operator<<` — так же, как легче сесть в машину, не оборудованную ремнями и подушкой безопасности, да еще и без дверец. Удобства при использовании таких функций не стоят возникающего при этом риска. Функции в стиле `printf` представляют собой серьезную проблему безопасности (см. [Cowan01]), так что имеется целая отрасль разработки инструментария для поиска ошибок такого рода (см. [Tsai01]).

Лучше использовать безопасные в смысле типов библиотеки, которые поддерживают переменное количество аргументов иными средствами. Например, библиотека форматирования [Boost] использует новейшие средства C++ для совмещения безопасности со скоростью и удобством.

Ссылки

[Boost] • [Cowan01] • [Murray93] §2.6 • [Sutter04] §2-3 • [Tsai01]

99. Не используйте недействительные объекты и небезопасные функции

Резюме

Вы же не используете просроченные лекарства? И недействительные объекты, и "антикварные", но небезопасные функции способны навредить здоровью ваших программ.

Обсуждение

Имеется три основные категории недействительных объектов.

- *Уничтоженные объекты.* Типичными примерами таких объектов являются автоматические объекты, вышедшие из области видимости, и удаленные динамические объекты. После того как вы вызвали деструктор объекта, его время жизни истекло, и любые действия с ним небезопасны и приводят к непредсказуемым последствиям.
- *Семантически недействительные объекты.* Типичные примеры включают "висячие" указатели на удаленные объекты (например, указатель `p` после выполнения `delete p;`) и недействительные итераторы (например, `vector<T>::iterator i` после вставки в начало контейнера, к которому обращается итератор). Заметим, что висячие указатели и недействительные итераторы концептуально идентичны, и последние часто непосредственно содержат первые. Обычно небезопасно и непредсказуемо делать что-либо с такими указателями и итераторами, за исключением присваивания другого корректного значения недействительному объекту (например, `p = new Object;` или `i = v.begin();`).
- *Объекты, которые никогда не были действительными.* Примеры включают объекты, "полученные" путем преобразования указателя с использованием `reinterpret_cast` (см. рекомендацию 92) или при обращении за пределы границ массива.

Никогда не забывайте о времени жизни объекта и его корректности. Не разыменовывайте недействительные итераторы и указатели. Не делайте никаких предположений о том, что делает и чего не делает оператор `delete`; освобожденная память — это освобожденная память, и обращений к ней не должно быть ни при каких условиях. Не пытайтесь играться со временем жизни объекта путем вызова деструктора вручную (например, `obj.~T()`) с последующим вызовом размещающего `new` (см. рекомендацию 55).

Не используйте небезопасное наследство C: `strcpy`, `strncpy`, `sprintf` или прочие функции, которые выполняют запись в буфер без проверки выхода за его пределы. Функция `strcpy` не выполняет проверки границ буфера, а функция [C99] `strncpy` выполняет проверку, но не добавляет нулевой символ при выходе на границу. Обе эти функции — потенциальный источник неприятностей. Используйте современные, более безопасные и гибкие структуры и функции, такие, которые имеются в стандартной библиотеке C++ (см. рекомендацию 77). Они не всегда совершенно безопасны (в основном это связано с вопросами эффективности), но существенно меньше подвержены ошибкам и позволяют создавать гораздо более безопасный код.

Ссылки

[C99] • [Sutter00] §1 • [Sutter04] §2-3

100. Не рассматривайте массивы полиморфно

Резюме

Полиморфная работа с массивами — большая ошибка. К сожалению, обычно компилятор никак на нее не реагирует. Не попадайтесь в эту ловушку!

Обсуждение

Указатели служат одновременно для двух целей: в качестве малого размера идентификаторов объектов и в качестве итераторов для массивов (они могут обходить массивы объектов с использованием арифметики указателей). В качестве идентификаторов имеет смысл рассматривать указатель на `Derived` как указатель на `Base`. Однако как только мы переходим ко второй цели, такая замена не работает, поскольку массив объектов `Derived` — совсем не то же, что и массив объектов `Base`. Чтобы проиллюстрировать сказанное, заметим, что и мышь, и слон — оба млекопитающие, но это не значит, что колонна из ста слонов будет по длине такой же, что и колонна из ста мышей.

Размер имеет значение. При замене указателя на `Derived` указателем на `Base` компилятор точно знает, как следует подкорректировать (при необходимости) указатель, поскольку у него есть достаточное количество информации об обоих классах. Однако при выполнении арифметических операций над указателем `p` на `Base` компилятор вычисляет `p[n]` как `*(p+n*sizeof(Base))`, таким образом полагаясь на то, что объекты, находящиеся в памяти, — это объекты типа `Base`, а не некоторого производного типа, который может иметь другой размер. Представляете, какая ерунда может получиться при работе с массивом объектов `Derived`, если вы преобразуете указатель на начало этого массива в указатель типа `Base*` (что компилятор вполне допускает), а затем примените арифметические операции к этому указателю (что компилятор также пропустит, не моргнув глазом)!

Такие неприятности представляют собой результат взаимодействия двух концепций — заменимости указателей на производный класс указателями на базовый класс, и унаследованной от C арифметикой указателей, которая считает указатели мономорфными и использует при вычислениях только статическую информацию.

Для хранения массива полиморфных объектов вам нужен массив (а еще лучше — контейнер; см. рекомендацию 77) указателей на базовый класс (например, обычных указателей или, что еще лучше, интеллектуальных указателей `shared_ptr`; см. рекомендацию 79). Тогда каждый указатель массива указывает на полиморфный объект, скорее всего, объект в динамически выделенной памяти. (Если вы хотите обеспечить интерфейс контейнера полиморфных объектов, вам надо инкапсулировать весь массив и предоставить соответствующий полиморфный интерфейс для выполнения итераций.)

Кстати, одна из причин, по которой в интерфейсах следует предпочитать ссылки указателям, заключается в том, чтобы было совершенно очевидно, что речь идет только об одном объекте, а не о массиве объектов.

Ссылки

[C++TR104] • [Dewhurst03] §33, §89 • [Sutter00] §36 • [Meyers96] §3

Список литературы

Примечание: для удобства читателей весь список литературы доступен по адресу `http://www.gotw.ca/publications/c++cs/bibliography.htm`

Ссылки, выделенные полужирным шрифтом (например, **[Abrahams96]**), представляют собой гиперссылки в приведенной выше странице.

[Abelson96] Abelson H. and Sussman G. J. *Structure and Interpretation of Computer Programs (2nd Edition)* (MIT Press, 1996).

[Abrahams96] Abrahams D. *Exception Safety in STLport*. STLport website, 1996.

[Abrahams01a] Abrahams D. Exception Safety in Generic Components, in Jazayeri M., Loos R., Musser D. (eds.), *Generic Programming: International Seminar on Generic Programming, Dagstuhl Castle, Germany, April/May 1998, Selected Papers*, Lecture Notes in Computer Science 1766 (Springer, 2001).

[Abrahams01b] Abrahams D. *Error and Exception Handling*. [Boost] website, 2001.

[Alexandrescu00a] Alexandrescu A. Traits: The else-if-then of Types. *C++ Report*, 12(4), April 2000.

[Alexandrescu00b] Alexandrescu A. Traits on Steroids. *C++ Report*, 12(6), June 2000.

[Alexandrescu00c] Alexandrescu A. and Marginean P. Change the Way You Write Exception-Safe Code—Forever. *C/C++ Users Journal*, 18(12), December 2000.

[Alexandrescu01] Alexandrescu A. *Modern C++ Design*. Addison-Wesley, 2001.

Перевод: Александреску А. *Современное проектирование на C++. Серия C++ In-Depth, т.3.* — М.: Издательский дом "Вильямс", 2002.

[Alexandrescu01a] Alexandrescu A. A Policy-Based basic_string Implementation. *C/C++ Users Journal*, 19(6), June 2001.

[Alexandrescu02a] Alexandrescu A. *Multithreading and the C++ Type System*. InformIT website, February 2002.

[Alexandrescu02b] Alexandrescu A. "Discriminated Unions (I)," "... (II)," and "... (III)". *C/C++ Users Journal*, 20(4,6,8), April/June/August 2002.

[Alexandrescu03a] Alexandrescu A. Move Constructors. *C/C++ Users Journal*, 21(2), February 2003.

[Alexandrescu03b] Alexandrescu A. Assertions. *C/C++ Users Journal*, 21(4), April 2003.

[Alexandrescu03c] Alexandrescu A. and Marginean P. Enforcements. *C/C++ Users Journal*, 21(6), June 2003.

[Alexandrescu03d] Alexandrescu A. and Held D. Smart Pointers Reloaded. *C/C++ Users Journal*, 21(10), October 2003.

[Alexandrescu04] Alexandrescu A. Lock-Free Data Structures. *C/C++ Users Journal*, 22(10), October 2004.

[Allison98] Allison C. *C & C++ Code Capsules*. Prentice Hall, 1998.

[Austern99] Austern M. H. *Generic Programming and the STL*. Addison-Wesley, 1999.

[Barton94] Barton J. and Nackman L. *Scientific and Engineering C++*. Addison-Wesley, 1994.

[Bentley00] Bentley J. *Programming Pearls (2nd Edition)*. Addison-Wesley, 2000.

Перевод: Бентли Дж. *Жемчужины программирования. Второе издание.* — СПб.: Питер, 2002.

[BetterSCM] Web-узел Better SCM Initiative.

[Boost] C++ Boost.

[BoostLRG] *Boost Library Requirements and Guidelines*. (Web-узел Boost).

[Brooks95] Brooks F. *The Mythical Man-Month*. Addison-Wesley, 1975; reprinted with corrections in 1995.

[Butenhof97] Butenhof D. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[Cargill92] Cargill T. *C++ Programming Style*. Addison-Wesley, 1992.

[C90] ISO/IEC 9899:1990(E), *Programming Languages — C* (ISO C90 and ANSI C89 Standard).

[C99] ISO/IEC 9899:1999(E), *Programming Languages — C* (revised ISO and ANSI C99 Standard).

[C++98] ISO/IEC 14882:1998(E), *Programming Languages—C++* (ISO and ANSI C++ Standard).

[C++03] ISO/IEC 14882:2003(E), *Programming Languages—C++* (updated ISO and ANSI C++ Standard including the contents of [C++98] plus errata corrections).

[C++TR104] ISO/IEC JTC1/SC22/WG21/N1711. *(Draft) Technical Report on Standard Library Extensions* (ISO C++ committee working document, November 2004). Близкий к окончанию черновик с расширениями стандартной библиотеки C++, включая `shared_ptr`.

[Cline99] Cline M., Lomow G., and Girou M. *C++ FAQs (2nd Edition)*. Addison-Wesley, 1999.

[Constantine95] Constantine L. *Constantine on Peopleware*. Yourdon Press, 1995.

[Coplien92] Coplien J. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[Cormen01] Cormen T., Leiserson C., Rivest R., Stein C. *Introduction to Algorithms (2nd Edition)*. MIT Press, 2001.

Перевод: Кормен Т., Лейзерсон Ч., Ривест Р., Стейн К. *Введение в алгоритмы. 2-е изд.* — М.: Издательский дом "Вильямс" (в печати)

[CVS] Web-узел CVS.

[Cowan01] Cowan C., Barringer M., Beattie S., and Kroah-Hartman G. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. *Proceedings of the 2001 USENIX Security Symposium*, August 2001, Washington, D.C.

[Dewhurst03] Dewhurst S. *C++ Gotchas*. Addison-Wesley, 2003.

[Dinkumware-Safe] *Dinkumware Unabridged Library documentation* (Web-узел Dinkumware).

[Ellis90] Ellis M. and Stroustrup B. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

Перевод: Эллис М., Страуструп Б. *Справочное руководство по языку программирования C++ с комментариями*. — М.: Мир, 1992.

[Gamma95] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Перевод: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. *Приемы объектно-ориентированного проектирования. Паттерны проектирования*. — СПб.: Питер, 2001.

[GnuMake] Gnu make (Web-узел Gnu).

[GotW] Sutter H. *Guru of the Week* column.

[Henney00] Henney K. *C++ Patterns: Executing Around Sequences* (EuroPLoP 2000 proceedings).

[Henney01] Henney K. *C++ Patterns: Reference Accounting* (EuroPLoP 2001 proceedings).

[Henney02a] Henney K. Stringing Things Along. *Application Development Advisor*, July-August 2002.

[Henney02b] Henney K. The Next Best String. *Application Development Advisor*, October 2002.

[Henricson97] Henricson M. and Nyquist E. *Industrial Strength C++*. Prentice Hall,1997.

[Horstmann95] Horstmann C. S. Safe STL., 1995.

[Josuttis99] Josuttis N. *The C++ Standard Library*. Addison-Wesley, 1999.

Перевод: Джосьютис Н. *C++. Стандартная библиотека*. — СПб.: Питер (в печати).

[Keffer95] Keffer T. *Rogue Wave C++ Design, Implementation, and Style Guide*. Rogue Wave Software, 1995.

[Kernighan99] Kernighan B. and Pike R. *The Practice of Programming*. Addison-Wesley, 1999.

[Knuth89] Knuth D. The Errors of TeX. *Software—Practice & Experience*, 19(7), July 1989.

[Knuth97a] Knuth D. *The Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd Edition*. Addison-Wesley, 1997.

Перевод: Кнут Д. *Искусство программирования, том 1. Основные алгоритмы, 3-е изд.* — М.: Издательский дом "Вильямс", 2000.

[Knuth97b] Knuth D. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1997.

Перевод: Кнут Д. *Искусство программирования, том 1. Получисленные алгоритмы, 3-е изд.* — М.: Издательский дом "Вильямс", 2000.

[Knuth98] Knuth D. *The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd Edition.* Addison-Wesley, 1998.

Перевод: Кнут Д. *Искусство программирования, том 1. Сортировка и поиск, 2-е изд.* — М.: Издательский дом "Вильямс", 2000.

[Koenig97] Koenig A. and Moo B. *Ruminations on C++.* Addison-Wesley, 1997.

[Lakos96] Lakos J. *Large-Scale C++ Software Design.* Addison-Wesley, 1996.

[Liskov88] Liskov B. Data Abstraction and Hierarchy. *SIGPLAN Notices*, 23(5), May 1988.

[Martin96a] Martin R. C. The Dependency Inversion Principle. *C++ Report*, 8(5), May 1996.

[Martin96b] Martin R. C. Granularity. *C++ Report*, 8(9), October 1996.

[Martin96c] Martin R. C. The Open-Closed Principle. *C++ Report*, 8(1), January 1996.

[Martin98] Martin R. C., Riehle D., Buschmann F. (eds.). *Pattern Languages of Program Design 3.* Addison-Wesley, 1998.

[Martin00] Martin R. C. Abstract Classes and Pure Virtual Functions in Martin R. C. (ed.), *More C++ Gems.* Cambridge University Press, 2000.

[McConnell93] McConnell S. *Code Complete.* Microsoft Press, 1993.

[Metrowerks] Metrowerks.

[Meyer00] Meyer B. *Object-Oriented Software Construction, 2nd Edition.* Prentice Hall, 2000.

[Meyers96] Meyers S. *More Effective C++.* Addison-Wesley, 1996.

Перевод: Мейерс С. *Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов.* — М.: ДМК Пресс, 2000.

[Meyers97] Meyers S. *Effective C++, 2nd Edition.* Addison-Wesley, 1997.

Перевод: Мейерс С. *Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов.* — М.: ДМК Пресс, 2000.

[Meyers00] Meyers S. How Non-Member Functions Improve Encapsulation. *C/C++ Users Journal*, 18(2), February 2000.

[Meyers01] Meyers S. *Effective STL.* Addison-Wesley, 2001.

Перевод: Мейерс С. *Эффективное использование STL.* — СПб.: Питер, 2002.

[Meyers04] Meyers S. and Alexandrescu A. C++ and the Perils of Double-Checked Locking, Part 1 and ...Part 2. *Dr. Dobb's Journal*, 29(7,8), July and August 2004.

[Milewski01] Milewski B. *C++ In Action.* Addison-Wesley, 2001.

[Miller56] Miller G. A. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *The Psychological Review*, 1956, vol. 63.

[MozillaCRFAQ] *Frequently Asked Questions About mozilla.org's Code Review Process* (Web-узел Mozilla).

[Murray93] Murray R. C++ *Strategies and Tactics.* Addison-Wesley, 1993.

[Musser01] Musser D. R., Derge G. J., and Saini A. *STL Tutorial and Reference Guide, 2nd Edition.* Addison-Wesley, 2001.

[Parnas02] Parnas D. The Secret History of Information Hiding. *Software Pioneers: Contributions To Software Engineering*, Springer-Verlag, New York, 2002.

[Peters99] Peters T. *The Zen of Python.* Comp.lang.python, June 1999.

[Piwowarski82] Piwowarski P. A Nesting Level Complexity Measure. *ACM SIGPLAN Notices*, 9/1982.

[Saks99] Saks D. Thinking Deeply, Thinking Deeper, and Thinking Even Deeper. *C/C++ Users Journal*, 17(4,5,6), April, May, and June 1999.

[Schmidt01] Schmidt D., Stal M., Rohnert H., Buschmann F. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects,* Wiley, 2001.

[SeamonkeyCR] *Seamonkey Code Reviewer's Guide* (Web-узел Mozilla).

[Sedgewick98] Sedgewick R. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, 3rd Edition.* Addison-Wesley, 1998.

[STLport-Debug] Fomitchev B. *STLport: Debug Mode* (Web-узел STLport).

[Stroustrup94] Stroustrup B. *The Design and Evolution of C++.* Addison-Wesley, 1994.

Перевод: Страуструп Б. *Дизайн и эволюция языка C++.* — М.: ДМК Пресс, 2000.

[Stroustrup00] Stroustrup B. *The C++ Programming Language (Special 3rd Edition).* Addison-Wesley, 2000.

Перевод: Страуструп Б. *Язык программирования C++. Специальное издание.* — М.: Бином, 2001.

[Sutter99] Sutter H. ACID Programming. *Guru of the Week #61.*

[Sutter00] Sutter H. *Exceptional C++.* Addison-Wesley, 2000.

Перевод: Саттер Г. *Решение сложных задач на C++. Серия C++ In-Depth, т.4.* — М.: Издательский дом "Вильямс", 2002.

[Sutter02] Sutter H. *More Exceptional C++.* Addison-Wesley, 2002.

Перевод: Саттер Г. *Решение сложных задач на C++. Серия C++ In-Depth, т.4.* — М.: Издательский дом "Вильямс", 2002.

[Sutter03] Sutter H. Generalizing Observer. *C/C++ Users Journal*, 21(9), September 2003.

[Sutter04] Sutter H. *Exceptional C++ Style.* Addison-Wesley, 2004.

Перевод: Саттер Г. *Новые сложные задачи на C++.* — М.: Издательский дом "Вильямс" (в печати).

[Sutter04a] Sutter H. Function Types. *C/C++ Users Journal*, 22(7), July 2004.

[Sutter04b] Sutter H. When and How To Use Exceptions. *C/C++ Users Journal*, 22(8), August 2004.

[Sutter04c] Sutter H. Just Enough' Thread Safety. *C/C++ Users Journal*, 22(9), September 2004.

[Sutter04d] Sutter H. How to Provide (or Avoid) Points of Customiza-tion in Templates. *C/C++ Users Journal*, 22(11), November 2004.

[SuttHysl01] Sutter H. and Hyslop J. Hungarian wartHogs. *C/C++ Users Journal*, 19(11), November 2001.

[SuttHysl02] Sutter H. and Hyslop J. A Midsummer Night's Madness. *C/C++ Users Journal*, 20(8), August 2002.

[SuttHysl03] Sutter H. and Hyslop J. Sharing Causes Contention. *C/C++ Users Journal*, 21(4), April 2003.

[SuttHysl04a] Sutter H. and Hyslop J. Getting Abstractions. *C/C++ Users Journal*, 22(6), June 2004.

[SuttHysl04b] Sutter H. and Hyslop J. Collecting Shared Objects. *C/C++ Users Journal*, 22(8), August 2004.

[Taligent94] *Taligent's Guide to Designing Programs.* Addison-Wesley, 1994.

[Tsai01] Tsai T. and Singh N. *Libsafe 2.0: Detection of Format String Vulnerability Exploits.* Avaya Labs, March 2001.

[Vandevoorde03] Vandevoorde D. and Josuttis N. *C++ Templates.* Addison-Wesley, 2003.

Перевод: Вандевурд Д., Джосаттис Н. *Шаблоны C++. Справочник разработчика.* — М.: Издательский дом "Вильямс", 2003.

[Webber03] Webber A. B. *Modern Programming Languages: A Practical Introduction.* Franklin, Beedle & Associates, 2003.

Резюме из резюме

Вопросы организации и стратегии

0. Не мелочитесь, или Что не следует стандартизировать

Скажем кратко: не мелочитесь.

1. Компилируйте без замечаний при максимальном уровне предупреждений

Следует серьезно относиться к предупреждениям компилятора и использовать максимальный уровень вывода предупреждений вашим компилятором. Компиляция должна выполняться без каких-либо предупреждений. Вы должны понимать все выдаваемые предупреждения и устранять их путем изменения кода, а не снижения уровня вывода предупреждений.

2. Используйте автоматические системы сборки программ

Нажимайте на одну (единственную) кнопку: используйте полностью автоматизированные ("в одно действие") системы, которые собирают весь проект без вмешательства пользователя.

3. Используйте систему контроля версий

Как гласит китайская пословица, плохие чернила лучше хорошей памяти: используйте системы управления версиями. Не оставляйте файлы без присмотра на долгий срок. Проверяйте их всякий раз после того, как обновленные модули проходят тесты на работоспособность. Убедитесь, что внесенные обновления не препятствуют корректной сборке программы.

4. Одна голова хорошо, а две — лучше

Регулярно просматривайте код всей командой. Чем больше глаз — тем выше качество кода. Покажите ваш код другим и познакомьтесь с их кодом — это принесет пользу всем.

Стиль проектирования

5. Один объект — одна задача

Концентрируйтесь одновременно только на одной проблеме. Каждый объект (переменная, класс, функция, пространство имен, модуль, библиотека) должны решать одну точно поставленную задачу. С ростом объектов, естественно, увеличивается область их ответственности, но они не должны отклоняться от своего предназначения.

6. Главное — корректность, простота и ясность

Корректность лучше быстроты. Простота лучше сложности. Ясность лучше хитроумия. Безопасность лучше ненадежности (см. рекомендации 83 и 99).

7. Кодирование с учетом масштабируемости

Всегда помните о возможном росте данных. Подумайте об асимптотической сложности без преждевременной оптимизации. Алгоритмы, которые работают с пользовательскими данными, должны иметь предсказуемое и, желательно, не хуже чем линейно зависящее от количества обрабатываемых данных время работы. Когда становится важной и необходимой оптимизация, в особенности из-за роста объемов данных, в первую очередь следует улучшать О-сложность алгоритма, а не заниматься микрооптимизациями типа экономии на одном сложении.

8. Не оптимизируйте преждевременно

Как гласит пословица, не подгоняйте скачущую лошадь. Преждевременная оптимизация непродуктивна и быстро входит в привычку. Первое правило оптимизации: не оптимизируйте. Второе правило оптимизации (только для экспертов): не оптимизируйте ни в коем случае. Семь раз отмерь, один раз оптимизируй.

9. Не пессимизируйте преждевременно

То, что просто для вас, — просто и для кода. При прочих равных условиях, в особенности — сложности и удобочитаемости кода, ряд эффективных шаблонов проектирования и идиом кодирования должны естественным образом "стекать с кончиков ваших пальцев" и быть не сложнее в написании, чем их пессимизированные альтернативы. Это не преждевременная оптимизация, а избежание излишней пессимизации.

10. Минимизируйте глобальные и совместно используемые данные

Совместное использование вызывает споры и раздоры. Избегайте совместного использования данных, в особенности глобальных данных. Совместно используемые данные усиливают связность, что приводит к снижению сопровождаемости, а зачастую и производительности.

11. Сокрытие информации

Не выпускайте внутреннюю информацию за пределы объекта, обеспечивающего абстракцию.

12. Кодирование параллельных вычислений

Если ваше приложение использует несколько потоков или процессов, следует минимизировать количество совместно используемых объектов, где это только можно (см. рекомендацию 10), и аккуратно работать с оставшимися.

13. Ресурсы должны быть во владении объектов

Не работайте вручную, если у вас есть мощные инструменты. Идиома C++ "выделение ресурса есть инициализация" (resource acquisition is initialization — RAII) представляет собой мощный инструмент для корректной работы с ресурсами. RAII позволяет компилятору автоматически обеспечить строгую гарантию того, что в других языках надо делать вручную. При выделении ресурса передайте его объекту-владельцу. Никогда не выделяйте несколько ресурсов в одной инструкции.

Стиль кодирования

14. Предпочитайте ошибки компиляции и компоновки ошибкам времени выполнения

Не стоит откладывать до выполнения программы выявление ошибок, которые можно обнаружить при ее сборке. Предпочтительно писать код, который использует компилятор для проверки инвариантов в процессе компиляции, вместо того, чтобы проверять их во время работы программы. Проверки времени выполнения зависят от выполнимого кода и данных, так что вы только изредка можете полностью полагаться на них. Проверки времени компиляции, напротив, не зависят от данных и предыстории исполнения, что обычно обеспечивает более высокую степень надежности.

15. Активно используйте const

const — ваш друг: неизменяемые значения проще понимать, отслеживать и мотивировать, т.е. там, где это целесообразно, лучше использовать константы вместо переменных. Сделайте const описанием по умолчанию при определении значения — это безопасно, проверяемо во время компиляции (см. рекомендацию 14) и интегрируемо с системой типов

C++. Не выполняйте преобразований типов с отбрасыванием const кроме как при вызове некорректной с точки зрения употребления const функции (см. рекомендацию 94).

16. Избегайте макросов

Макрос — самый неприятный инструмент C и C++, оборотень, скрывающийся под личиной функции, кот, гуляющий сам по себе и не обращающий никакого внимания на границы ваших областей видимости. Берегитесь его!

17. Избегайте магических чисел

*Избегайте использования в коде литеральных констант наподобие 42 или 3.1415926. Такие константы не самоочевидны и усложняют сопровождение кода, поскольку вносят в него трудноопределимый вид дублирования. Используйте вместо них символьные имена и выражения наподобие width*aspectRatio.*

18. Объявляйте переменные как можно локальнее

Избегайте "раздувания" областей видимости. Переменных должно быть как можно меньше, а время их жизни — как можно короче. Эта рекомендация по сути является частным случаем рекомендации 10.

19. Всегда инициализируйте переменные

Неинициализированные переменные — распространенный источник ошибок в программах на C и C++. Избегайте их, выработав привычку очищать память перед ее использованием; инициализируйте переменные при их определении.

20. Избегайте длинных функций и глубокой вложенности

Краткость — сестра таланта. Чересчур длинные функции и чрезмерно вложенные блоки кода зачастую препятствуют реализации принципа "одна функция — одна задача" (см. рекомендацию 5), и обычно эта проблема решается лучшим разделением задачи на отдельные части.

21. Избегайте зависимостей инициализаций между единицами компиляции

Объекты уровня пространств имен в разных единицах компиляции не должны зависеть друг от друга при инициализации, поскольку порядок их инициализации не определен. В противном случае вам обеспечена головная боль при попытках разобраться со сбоями в работе программы после внесения небольших изменений в ваш проект и невозможностью его переноса даже на новую версию того же самого компилятора.

22. Минимизируйте зависимости определений и избегайте циклических зависимостей

Избегайте излишних зависимостей. Не включайте при помощи директивы #include определения там, где достаточно предварительного объявления.

Избегайте взаимозависимостей. Циклические зависимости возникают, когда два модуля непосредственно или опосредованно зависят друг от друга. Модуль представляет собой обособленную единицу; взаимозависимые модули не являются полностью отдельными модулями, будучи по сути частями одного большего модуля. Таким образом, циклические зависимости являются противниками модульности и представляют угрозу большим проектам. Избегайте их.

23. Делайте заголовочные файлы самодостаточными

Убедитесь, что каждый написанный вами заголовочный файл компилируем самостоятельно, т.е. что он включает все заголовочные файлы, от которых зависит его содержимое.

24. Используйте только внутреннюю, но не внешнюю защиту директивы #include

Предотвращайте непреднамеренное множественное включение ваших заголовочных файлов директивой #include, используя в них защиту с уникальными именами.

Функции и операторы

25. Передача параметров по значению, (интеллектуальному) указателю или ссылке

Вы должны четко уяснить разницу между входными, выходными параметрами и параметрами, предназначенными и для ввода, и для вывода информации, а также между передачей параметров по значению и по ссылке, и корректно их использовать.

26. Сохраняйте естественную семантику перегруженных операторов

Программисты ненавидят сюрпризы. Перегружайте операторы только в случае веских на то оснований, и сохраняйте при этом их естественную семантику. Если это оказывается сложным, возможно, вы неверно используете перегрузку операторов.

27. Отдавайте предпочтение каноническим формам арифметических операторов и операторов присваивания

Если можно записать a+b, то необходимо, чтобы можно было записать и a+=b. При определении бинарных арифметических операторов одновременно предоставляйте и их присваивающие версии, причем делайте это с минимальным дублированием и максимальной эффективностью.

28. Предпочитайте канонический вид ++ и --, и вызов префиксных операторов

Особенность операторов инкремента и декремента состоит в том, что у них есть префиксная и постфиксная формы с немного отличающейся семантикой. Определяйте операторы operator++ и operator-- так, чтобы они подражали поведению своих встроенных двойников. Если только вам не требуется исходное значение — используйте префиксные версии операторов.

29. Используйте перегрузку, чтобы избежать неявного преобразования типов

Не преумножайте объекты сверх необходимости (Бритва Оккама): неявное преобразование типов обеспечивает определенное синтаксическое удобство (однако см. рекомендацию 40), но в ситуации, когда допустима оптимизация (см. рекомендацию 8) и желательно избежать создания излишних объектов, можно обеспечить перегруженные функции с сигнатурами, точно соответствующими распространенным типам аргументов, и тем самым избежать преобразования типов.

30. Избегайте перегрузки &&, || и , (запятой)

Мудрость — это знание того, когда надо воздержаться. Встроенные операторы &&, || и , (запятая) трактуются компилятором специальным образом. После перегрузки они становятся обычными функциями с весьма отличной семантикой (при этом вы нарушаете рекомендации 26 и 31), а это прямой путь к трудноопределимым ошибкам и ненадежности. Не перегружайте эти операторы без крайней необходимости и глубокого понимания.

31. Не пишите код, который зависит от порядка вычислений аргументов функции

Порядок вычисления аргументов функции не определен, поэтому никогда не полагайтесь на то, что аргументы будут вычисляться в той или иной очередности.

Проектирование классов и наследование

32. Ясно представляйте, какой вид класса вы создаете

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

33. Предпочитайте минимальные классы монолитным

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).

34. Предпочитайте композицию наследованию

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.

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

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.

36. Предпочитайте предоставление абстрактных интерфейсов

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.

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

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).

38. Практикуйте безопасное перекрытие

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.

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

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).

40. Избегайте возможностей неявного преобразования типов

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).

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

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.

42. Не допускайте вмешательства во внутренние дела

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

43. Разумно пользуйтесь идиомой Pimpl

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).

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

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

45. new и delete всегда должны разрабатываться вместе

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[].*

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

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.

Конструкторы, деструкторы и копирование

47. Определяйте и инициализируйте переменные-члены в одном порядке

Переменные-члены всегда инициализируются в том порядке, в котором они объявлены при определении класса; порядок их упоминания в списке инициализации конструктора игнорируется. Убедитесь, что в коде конструктора указан тот же порядок, что и в определении класса.

48. В конструкторах предпочитайте инициализацию присваиванию

В конструкторах использование инициализации вместо присваивания для установки значений переменных-членов предохраняет от ненужной работы времени выполнения при том же объеме вводимого исходного текста.

49. Избегайте вызовов виртуальных функций в конструкторах и деструкторах

Внутри конструкторов и деструкторов виртуальные функции теряют виртуальность. Хуже того — все прямые или косвенные вызовы нереализованных чисто виртуальных функций из конструктора или деструктора приводят к неопределенному поведению. Если ваш дизайн требует виртуальной передачи в производный класс из конструктора или деструктора базового класса, следует воспользоваться иной методикой, например, постконструкторами.

50. Делайте деструкторы базовых классов открытыми и виртуальными либо защищенными и невиртуальными

Удалять или не удалять — вот в чем вопрос! Если следует обеспечить возможность удаления посредством указателя на базовый класс, то деструктор базового класса должен быть открытым и виртуальным. В противном случае он должен быть защищенным и невиртуальным.

51. Деструкторы, функции освобождения ресурсов и обмена не ошибаются

Все запуски этих функций должны быть успешными. Никогда не позволяйте ошибке выйти за пределы деструктора, функции освобождения ресурса (например, оператора delete) или функции обмена. В частности, типы, деструкторы которых могут генерировать исключения, категорически запрещено использовать со стандартной библиотекой C++.

52. Копируйте и ликвидируйте согласованно

Если вы определили одну из следующего списка функций — копирующий конструктор, оператор копирующего присваивания или деструктор — вероятно, вам потребуется определить и обе оставшиеся функции (или по крайней мере одну из них).

53. Явно разрешайте или запрещайте копирование

Копируйте со знанием дела: тщательно выбирайте между использованием сгенерированных компилятором копирующего конструктора и оператора присваивания, написанием собственных версий или явным запрещением обоих, если копирование не должно быть разрешено.

54. Избегайте срезки. Подумайте об использовании в базовом классе клонирования вместо копирования

Срезка объектов происходит автоматически, невидимо и может приводить к полному разрушению чудесного полиморфного дизайна. Подумайте о полном запрете копирующего конструктора и копирующего присваивания в базовых классах. Вместо них можно использовать виртуальную функцию-член Clone, если пользователям вашего класса необходимо получать полиморфные (полные, глубокие) копии.

55. Предпочитайте канонический вид присваивания

При реализации оператора operator= предпочитайте использовать канонический вид — невиртуальный с определенной сигнатурой.

56. Обеспечьте бессбойную функцию обмена

Обычно имеет смысл предоставить для класса функцию swap в целях эффективного и бессбойного обмена внутреннего содержимого объекта с внутренним содержимым другого объекта. Такая функция может пригодиться для реализации ряда идиом, от простого перемещения объектов до реализации присваивания, легко обеспечивающего функцию принятия результатов работы со строгими гарантиями безопасности для вызывающего кода (см. также рекомендацию 51).

57. Храните типы и их свободный интерфейс в одном пространстве имен

Функции, не являющиеся членами и разработанные как часть интерфейса класса X (в особенности операторы и вспомогательные функции), должны быть определены в том же пространстве имен, что и X, что обеспечивает их корректный вызов.

58. Храните типы и функции в разных пространствах имен, если только они не предназначены для совместной работы

Оберегайте ваши типы от непреднамеренного поиска, зависящего от аргументов (argument-dependent lookup — ADL, известный также как поиск Кёнига); однако преднамеренный поиск должен завершаться успешно. Этого можно добиться путем размещения типов в своих собственных пространствах имен (вместе с непосредственно связанными с ними свободными функциями; см. рекомендацию 57). Избегайте помещения типов в те же пространства имен, что и шаблоны функций или операторов).

59. Не используйте using для пространств имен в заголовочных файлах или перед директивой #include

Директива using для пространств имен создана для вашего удобства, а не для головной боли других. Никогда не используйте объявления или директивы using перед директивой #include.

Вывод: не используйте директивы using для пространств имен или using-объявления в заголовочных файлах. Вместо этого полностью квалифицируйте все имена. (Второе правило следует из первого, поскольку заголовочные файлы не могут знать, какие другие директивы #include могут появиться в тексте после них.)

60. Избегайте выделения и освобождения памяти в разных модулях

Золотое правило программиста — положи, где взял. Выделение памяти в одном модуле, а освобождение в другом делает программу более хрупкой, создавая тонкую дальнюю зависимость между этими модулями. Такие модули должны быть компилируемы одной и той же версией компилятора с одними и теми же флагами (в частности, отладочные версии и версии NDEBUG) и с одной и той же реализацией стандартной библиотеки; кроме того, с практической точки зрения лучше, чтобы модуль, выделяющий память, оставался загружен при ее освобождении.

61. Не определяйте в заголовочном файле объекты со связыванием

Объекты со связыванием, включая переменные или функции уровня пространства имен, обладают выделенной для них памятью. Определение таких объектов в заголовочных файлах приводит либо к ошибкам времени компоновки, либо к бесполезному расходу памяти. Помещайте все объекты со связыванием в файлы реализации.

62. Не позволяйте исключениям пересекать границы модулей

Не бросайте камни в соседский огород — поскольку нет повсеместно распространенного бинарного стандарта для обработки исключений C++, не позволяйте исключениям пересекать распространяться между двумя частями кода, если только вы не в состоянии контролировать, каким компилятором и с какими опциями скомпилированы обе эти части кода. В противном случае может оказаться, что модули не поддерживают совместимые реализации распространения исключений. Обычно это правило сводится к следующему: не позволяйте исключениям пересекать границы модулей/подсистем.

63. Используйте достаточно переносимые типы в интерфейсах модулей

Не позволяйте типам появляться во внешнем интерфейсе модуля, если только вы не уверены в том, что все пользователи смогут корректно их понять и работать с ними. Используйте наивысший уровень абстракции, который в состоянии понять клиентский код.

Шаблоны и обобщенность

64. Разумно сочетайте статический и динамический полиморфизм

Статический и динамический полиморфизм дополняют друг друга. Следует ясно представлять себе их преимущества и недостатки, чтобы использовать каждый из них там, где он дает наилучшие результаты, и сочетать их так, чтобы получить лучшее из обоих миров.

65. Выполняйте настройку явно и преднамеренно

При разработке шаблона точки настройки должны быть написаны корректно, с особой тщательностью, а также ясно прокомментированы. При использовании шаблона необходимо четко знать, как именно следует настроить шаблон для работы с вашим типом, и выполнить соответствующие действия.

66. Не специализируйте шаблоны функций

При расширении некоторого шаблона функции (включая std::swap) избегайте попыток специализации шаблона. Вместо этого используйте перегрузку шаблона функции, которую следует поместить в пространство имен типа(ов), для которых разработана данная перегрузка (см. рекомендацию 57). При написании собственного шаблона функции также избегайте его специализации.

67. Пишите максимально обобщенный код

Используйте для реализации функциональности наиболее обобщенные и абстрактные средства.

Обработка ошибок и исключения

68. Широко применяйте assert для документирования внутренних допущений и инвариантов

Используйте assert или его эквивалент для документирования внутренних допущений в модуле (т.е. там, где вызываемый и вызывающий код поддерживаются одним и тем же программистом или командой), которые должны всегда выполняться (в противном случае они являются следствием программной ошибки; например, нарушение постусловий функции, обнаруженное вызывающим кодом). (См. также рекомендацию 70.) Убедитесь, что использование assert не приводит к побочным действиям.

69. Определите разумную стратегию обработки ошибок и строго ей следуйте

Еще на ранней стадии проектирования разработайте практичную, последовательную и разумную стратегию обработки ошибок и строго следуйте ей. Убедитесь, что ваша стратегия включает следующее.

- *Идентификация: какие условия являются ошибкой.*
- *Строгость: насколько важна каждая ошибка.*
- *Обнаружение: какой код отвечает за обнаружение ошибки.*
- *Распространение: какой механизм используется для описания и распространения уведомления об ошибке в каждом модуле.*
- *Обработка: какой код отвечает за выполнение действий, связанных с ошибкой.*
- *Уведомление: каким образом информация об ошибке вносится в журнальный файл или производится уведомление пользователя программы.*

Изменяйте механизмы обработки ошибок только в пределах границ модуля.

70. Отличайте ошибки от ситуаций, не являющихся ошибками

Функция представляет собой единицу работы. Таким образом, сбои следует рассматривать либо как ошибки, либо как штатные ситуации, в зависимости от их влияния на функции. В функции f сбой является ошибкой тогда и только тогда, когда он нарушает одно из предусловий f, не позволяет выполнить предусловие вызываемой ею функции, препятствует достижению собственных постусловий f или сохранению инварианта, за поддержку которого отвечает функция f.

В частности, в этой рекомендации мы исключаем внутренние программные ошибки (т.е. те, где за вызывающий и вызываемый код отвечает один и тот же человек или команда, — например, в пределах одного модуля). Они представляют собой отдельную категорию ошибок, для работы с которой используется такое средство, как проверки (см. рекомендацию 68).

71. Проектируйте и пишите безопасный в отношении ошибок код

В каждой функции обеспечивайте наиболее строгую гарантию безопасности, какой только можно добиться без дополнительных затрат со стороны вызывающего кода, не требующего такого уровня гарантии. Всегда обеспечивайте, как минимум, базовую гарантию безопасности.

Убедитесь, что при любых ошибках ваша программа всегда остается в корректном состоянии (в этом и заключается базовая гарантия). Остерегайтесь ошибок, нарушающих инвариант (включая утечки, но не ограничиваясь ими).

Желательно дополнительно гарантировать, что конечное состояние либо является исходным состоянием (в результате отката после происшедшей ошибки), либо корректно вычисленным целевым состоянием (если ошибок не было). Это — строгая гарантия безопасности.

Еще лучше гарантировать, что сбой в процессе операции невозможен. Хотя для большинства функций это невозможно, такую гарантию следует обеспечить для таких функций, как деструкторы и функции освобождения ресурсов. Данная гарантия — гарантия бессбойности.

72. Для уведомления об ошибках следует использовать исключения

Для уведомления об ошибках лучше использовать механизм исключений, а не коды ошибок. Применять коды состояния (например, коды ошибок, переменную errno) следует только тогда, когда нельзя использовать исключения (см. рекомендацию 62), а также для ситуаций, которые не являются ошибками. К другим методам, таким как экстренное завершение программы (или плановое завершение с освобождением ресурсов и т.п. действиями), следует прибегать только в ситуациях, когда восстановление после ошибки невозможно (или не требуется).

73. Генерируйте исключения по значению, перехватывайте — по ссылке

Генерируйте исключения по значению (не через указатель) и перехватывайте их как ссылки (обычно константные). Эта комбинация наилучшим образом соответствует семантике исключений. При повторной генерации перехваченного исключения предпочтительно использовать просто оператор throw;, а не инструкцию throw e;.

74. Уведомляйте об ошибках, обрабатывайте и преобразовывайте их там, где следует

Сообщайте об ошибках в тот момент, когда они обнаружены и идентифицированы как ошибки. Обрабатывайте или преобразовывайте их на самом нижнем уровне, на котором это можно сделать корректно.

75. Избегайте спецификаций исключений

Не пишите спецификаций исключений у ваших функций, если только вас не заставляют это делать внешние обстоятельства (например, код, который вы не можете изменить, уже ввел их; см. исключения к данному разделу).

STL: Контейнеры

76. По умолчанию используйте vector. В противном случае выбирайте контейнер, соответствующий задаче

Очень важно использовать "правильный контейнер". Если у вас есть весомые причины выбрать определенный тип контейнера, используйте тот контейнер, который наиболее подходит для вашей задачи.

Если конкретных предпочтений нет, возьмите vector и спокойно работайте, зная, что вы сделали верный выбор.

77. Вместо массивов используйте vector и string

Избегайте реализации абстракции массива посредством массивов в стиле C, арифметики указателей и примитивов управления памятью. Использование vector или string не только сделает проще вашу жизнь, но и позволит написать более безопасную и масштабируемую программу.

78. Используйте vector (и string::c_str) для обмена данными с API на других языках

*vector и string::c_str служат шлюзом для сообщения с API на других языках. Однако не полагайтесь на то, что итераторы являются указателями; для получения адреса элемента, на который ссылается vector<T>::iterator iter, используйте выражение &*iter.*

79. Храните в контейнерах только значения или интеллектуальные указатели

Храните к контейнерах объекты-значения. Контейнеры полагают, что их содержимое имеет тип значения, включая непосредственно хранящиеся значения, интеллектуальные указатели и итераторы.

80. Предпочитайте push_back другим способам расширения последовательности

Используйте push_back везде, где это возможно. Если для вас не важна позиция вставки нового объекта, лучше всего использовать для добавления элемента в последовательность функцию push_back. Все прочие средства могут оказаться как гораздо менее быстрыми, так и менее понятными.

81. Предпочитайте операции с диапазонами операциям с отдельными элементами

При добавлении элементов в контейнер лучше использовать операции с диапазонами (т.е. функцию insert, которая получает пару итераторов), а не последовательность вызовов функции для вставки одного элемента. Вызов функции для диапазона обычно проще написать, легче читать, и он более эффективен, чем явный цикл (см. также рекомендацию 84).

82. Используйте подходящие идиомы для реального уменьшения емкости контейнера и удаления элементов

Для того чтобы действительно избавиться от излишней емкости контейнера, воспользуйтесь трюком с использованием обмена, а для реального удаления элементов из контейнера — идиомой erase-remove.

83. Используйте отладочную реализацию STL

Безопасность превыше всего (см. рекомендацию 6). Используйте отладочную реализацию STL, даже если она имеется только для одного из ваших компиляторов, и даже если она используется только для отладочного тестирования.

84. Предпочитайте вызовы алгоритмов самостоятельно разрабатываемым циклам

Разумно используйте функциональные объекты. В очень простых случаях написанные самостоятельно циклы могут оказаться более простым и эффективным решением. Тем не менее, вызов алгоритма вместо самостоятельно разработанного цикла может оказаться более выразительным, легче сопровождаемым, менее подверженным ошибкам и не менее эффективным.

При вызове алгоритма подумайте о написании собственного функционального объекта, который инкапсулирует всю необходимую логику. Избегайте объединения связывателей параметров и простых функциональных объектов (например, bind2nd, plus), что обычно снижает ясность кода. Подумайте об использовании лямбда-библиотеки [Boost], которая автоматизирует задачу написания функциональных объектов.

85. Пользуйтесь правильным алгоритмом поиска

Данная рекомендация применима к поиску определенного значения в диапазоне. При поиске в неотсортированном диапазоне используйте алгоритмы find/find_if или count/count_if. Для поиска в отсортированном диапазоне выберите lower_bound, upper_bound, equal_ range или (реже) binary_search. (Вопреки своему имени, binary_search обычно — неверный выбор.)

86. Пользуйтесь правильным алгоритмом сортировки

При сортировке вы должны четко понимать, как работает каждый из сортирующих алгоритмов, и использовать наиболее дешевый среди тех, которые пригодны для решения вашей задачи.

87. Делайте предикаты чистыми функциями

Предикат представляет собой функциональный объект, который возвращает ответ да/нет, обычно в виде значения типа bool. Функция является "чистой" в математическом смысле, если ее результат зависит только от ее аргументов (обратите внимание — в данном случае термин "чистая" не имеет никакого отношения к чисто виртуальным функциям).

Не позволяйте предикатам сохранять или обращаться к состоянию так, чтобы это могло влиять на результат работы оператора operator(); при этом понятие состояния включает как данные-члены, так и глобальные состояния. Для предикатов желательно делать оператор operator() константной функцией-членом (см. рекомендацию 15).

88. В качестве аргументов алгоритмов и компараторов лучше использовать функциональные объекты, а не функции

Предпочтительно передавать алгоритмам функциональные объекты, а не функции, а компараторы ассоциативных контейнеров просто должны быть функциональными объектами. Функциональные объекты адаптируемы и, вопреки ожиданиям, обычно дают более быстрый по сравнению с функциями код.

89. Корректно пишите функциональные объекты

Разрабатывайте функциональные объекты так, чтобы их копирование выполнялось как можно эффективнее. Там, где это возможно, делайте их максимально адаптируемыми путем наследования от unary_function или binary_function.

90. Избегайте явного выбора типов — используйте полиморфизм

Избегайте явного выбора типа объекта для настройки поведения. Используйте шаблоны и виртуальные функции для того, чтобы поведение объекта определялось его типом, а не вызывающим кодом.

91. Работайте с типами, а не с представлениями

Не пытайтесь делать какие-то предположения о том, как именно объекты представлены в памяти. Как именно следует записывать и считывать объекты из памяти — пусть решают типы объектов.

92. Избегайте reinterpret_cast

Как гласит римская пословица, у лжи короткие ноги. Не пытайтесь использовать reinterpret_cast, чтобы заставить компилятор рассматривать биты объекта одного типа как биты объекта другого типа. Такое действие противоречит безопасности типов.

93. Избегайте применения static_cast к указателям

К указателям на динамические объекты не следует применят преобразование static_cast. Используйте безопасные альтернативы — от dynamic_cast до перепроектирования.

94. Избегайте преобразований, отменяющих const

Преобразование типов, отменяющее const, может привести к неопределенному поведению, а кроме того, это свидетельство плохого стиля программирования даже в том случае, когда применение такого преобразования вполне законно.

95. Не используйте преобразование типов в стиле C

Возраст не всегда означает мудрость. Старое преобразование типов в стиле C имеет различную (и часто опасную) семантику в зависимости от контекста, спрятанную за единым синтаксисом. Замена преобразования типов в стиле C преобразованиями C++ поможет защититься от неожиданных ошибок.

96. Не применяйте memcpy или memcmp к не-POD типам

Не работайте рентгеновским аппаратом (см. рекомендацию 91). Не используйте memcpy и memcmp для копирования или сравнения чего-либо структурированного более, чем обычная память.

97. Не используйте объединения для преобразований

Хитрость все равно остается ложью: объединения можно использовать для получения "преобразования типа без преобразования", записывая информацию в один член и считывая из другого. Однако это еще более опасно и менее предсказуемо, чем применение reinterpret_cast (см. рекомендацию 92).

98. Не используйте неизвестные аргументы (троеточия)

Наличие троеточий в C++ — опасное наследие C. Избегайте их в своих программах; используйте вместо этого высокоуровневые конструкции и библиотеки C++.

99. Не используйте недействительные объекты и небезопасные функции

Вы же не используете просроченные лекарства? И недействительные объекты, и "антикварные", но небезопасные функции способны навредить здоровью ваших программ.

100. Не рассматривайте массивы полиморфно

Полиморфная работа с массивами — большая ошибка. К сожалению, обычно компилятор никак на нее не реагирует. Не попадайтесь в эту ловушку!

От издательского дома "Вильямс"

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com
WWW: http://www.williamspublishing.com

Информация для писем из

России: 115419, Москва, а/я 783
Украины: 03150, Киев, а/я 152

Предметный указатель

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ш

Э

Научно-популярное издание

Герб Саттер, Андрей Александреску

Стандарты программирования на C++

Литературный редактор *С.Г. Татаренко*
Верстка *О.В. Мишутина*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д. 43, стр. 1.

Подписано в печать 22.07.2005. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 19,35. Уч.-изд. л. 13,2.
Тираж 3000 экз. Заказ № 2182.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.