Новые сложные задачи на C++

40 новых головоломных примеров с решениями

Герб Саттер

Серия C++ In-Depth ◆ Бьярн Страуструп

C++/Языки программирования

Новые сложные задачи на C++

Герб Саттер

Стиль при разработке программного обеспечения — это поиск идеального баланса между эффективностью и функциональностью, гибкостью и размером кода. В этой книге легендарный гуру в области C++ Герб Саттер представляет 40 новых головоломных задач, в которых анализируется не только то, *что* следует написать на C++, но и *как* следует это делать, и которые призваны помочь вам в поисках идеального баланса в ваших программах.

Книга организована в виде задач и решений. В ней предложен новый взгляд на ключевые детали и взаимоотношения в C++, освещены новые стратегии, используемые в современных технологиях программирования на C++, включая такие разделы, как обобщенное программирование, STL, безопасность исключений и многое другое. В этой книге вы найдете ответы на следующие вопросы.

- Какой опыт по созданию библиотек можно вынести из STL?
- Как повысить степень обобщенности шаблонного кода?
- Почему *не следует* специализировать шаблоны функций? Что надо делать вместо этого?
- В чем заключается истинная безопасность в смысле исключений?
- Следует ли использовать спецификации исключений?
- Когда и как можно организовать "утечку" закрытой части класса?
- Какое количество памяти *в действительности* используется в стандартных контейнерах?
- Действительно ли использование описателя `const` обеспечивает повышение степени оптимизации кода?
- Как использование описателя `inline` влияет на производительность программы?
- Может ли компилироваться и работать код, который выглядит совершенно неверно и нелепо?
- В чем заключаются ошибки проектирования `std::string`?

Новые сложные задачи на C++ помогут вам в создании более надежного, производительного и интеллектуального программного обеспечения на всех этапах — от проектирования до кодирования.

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

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

Фотография на обложке – Herb Sutter

Издательский дом "Вильямс"
www.williamspublishing.com

ADDISON-WESLEY
Pearson Education

ISBN 5-8459-0823-X

Exceptional C++ Style

40 New Engineering Puzzles, Programming Problems, and Solutions

Herb Sutter

ADDISON–WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Новые сложные задачи на C++

40 новых головоломных примеров с решениями

Герб Саттер

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

ББК 32.973.26-018.2.75
С21
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

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

Научный консультант канд. техн. наук *А.Н. Кротов*

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

Саттер, Герб.

С21 Новые сложные задачи на C++. : Пер. с англ. — М. : Издательский дом "Вильямс", 2005. — 272 с. : ил. - Парал. тит. англ.

ISBN 5-8459-0823-X (рус.)

Данная книга представляет собой продолжение вышедшей ранее книги *Решение сложных задач на C++*. В форме задач и их решений рассматриваются современные методы проектирования и программирования на C++. В книге сконцентрирован богатый многолетний опыт программирования на C++ не только самого автора, но и всего сообщества программистов на C++, так что некоторые рекомендации автора могут показаться неожиданными даже опытным программистам-профессионалам. Автор рассматривает и конкретные методики, приемы и идиомы программирования, однако основная тема книги — это стиль программирования, причем в самом широком понимании этого слова. Особое внимание во всех задачах книги уделено вопросу проектирования, которое должно обеспечить максимальную надежность, безопасность, производительность и сопровождаемость создаваемого программного обеспечения.

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

ББК 32.973.26-018.2.75

ISBN 5-8459-0823-X (рус.)
ISBN 0-201-76042-8 (англ.)

Оглавление

Содержание

Тине, моей жене и лучшему другу

Предисловие

Место действия: Будапешт. Жаркий летний вечер. Мы смотрим через Дунай, на восточный берег реки.

На обложке книги вы видите фото, на котором изображена эта пастельная европейская картина. Что первое бросается вам в глаза? Почти наверняка — здание парламента в левой части фотографии. Массивное неоготическое здание приковывает взгляд своим изящным куполом, массой вычурных шпилей, десятками статуй и прочими украшениями, контрастируя с простыми строгими линиями зданий на набережной Дуная.

Откуда же такое отличие? Строительство здания парламента было завершено в 1902 году, в то время как остальные здания на набережной были построены в разрушенном Будапеште после второй мировой войны.

"Ну и что же, — скажете вы, — какое отношение это имеет к книге?"

Стиль — это всегда нечто большее, чем просто внешний вид, и за ним скрывается целая философия и мировоззрение — будь то в архитектуре строительства или в архитектуре программного обеспечения. Я думаю, что вам попадались программы, напоминающие своей "пышностью" и размерами здание парламента, равно как уверен, что вам доводилось видеть и программы, напоминающие блочно-панельное строительство.

Стиль или суть?

Что же важнее? Чему лучше и правильнее отдать предпочтение? Вы уверены, что знаете точный ответ на этот вопрос? Так, понятие "лучше" лишено смысла, пока не определена мера, которой следует мерить. Лучше для чего? Лучше в какой ситуации? Скорее всего, ответ на этот вопрос представляет собой определенный компромисс и начинается со слов "Это зависит от..."

Это книга о поиске баланса между многими мелкими аспектами дизайна и реализации программ на C++. Глубокое знание ваших инструментов и исходных материалов весьма способствует пониманию того, когда их стоит использовать.

Так *лучше* ли здание парламента и его стиль, чем у зданий, находящихся рядом с ним? Очень легко, не думая, ответить "да". Но ответ должен основываться не только на эмоциях, но и на логике. Представьте, насколько не просто построить такое здание и поддерживать его в должном состоянии.

- *Строительство.* В 1902 году, когда закончилось его строительство, это здание было самым большим в мире зданием парламента. Эта грандиозность, конечно же, сказалась на его стоимости, продолжительности строительства и количестве затраченных усилий. Так был создан "белый слон", т.е. нечто интересное само по себе, но со стоимостью, которую не оправдывает никакой интерес. Как вы думаете, сколько обычного жилья, которое пусть и не потрясает воображение,

но дает кров над головой, можно было бы построить при тех же капиталовложениях? Наверное, ответ на этот вопрос мог бы впечатлить многих. Позвольте напомнить вам, что все мы работаем в той отрасли промышленности, где давление сроков разработки ощущается особо сильно — и в принципе несравнимо с таковым во времена постройки рассматриваемого здания.

- *Поддержка.* Присмотритесь к фотографии, и вы увидите, что часть здания покрыта лесами. Реставрационные работы идут здесь годами, и на это затрачиваются такие суммы, что, пожалуй, было бы проще снести это здание и построить что-то новое. На фотографии не видно (да и не может быть видно) кое-что еще. Например, скульптуры, украшающие здание, были сделаны из плохо подобранного материала, который слишком легко разрушается, так что их реставрация и замена были начаты едва ли не сразу же после завершения строительства, и все эти украшения, "рюшечки и финтифлюшечки" — предмет постоянной заботы реставраторов уже *более века*.

Так и в программировании — очень важно найти золотую середину между стоимостью и функциональностью, между элегантностью и сопровождаемостью, между возможностями развития и украшательством.

С подобными проблемами и поиском компромиссов мы вынуждены сталкиваться ежедневно при разработке программного обеспечения на C++. Среди вопросов, которые рассматриваются в данной книге, есть и такие: делает ли безопасность кода по отношению к исключениям лучше сам код? Если да — то что именно означает "делает его лучше", и не может ли возникнуть ситуация, когда это не так уж и хорошо? А как насчет инкапсуляции? Делает ли она программу лучше? Почему? При каких условиях это не так? Если вас заинтересовали эти вопросы — книга перед вами, прочтите ее. Кстати, встраиваемые функции — это хорошая оптимизация? Следует ли к ней прибегать? (Будьте очень-очень осторожны при ответе на этот вопрос.) Что общего между возможностью экспорта в C++ и зданием парламента? А между `std::string` и монолитной архитектурой зданий на набережной Дуная?

После рассмотрения множества различных технологий и возможностей C++, в конце книги целый раздел отведен для анализа реальных примеров опубликованных исходных текстов. Мы выясним, что авторам этих фрагментов удалось, что не совсем, и как исправить баланс между затрачиваемыми усилиями и хорошим стилем.

Я надеюсь, что эта книга, а также предыдущие книги по данной теме[1] помогут вам шире взглянуть на C++, прибавят вам знаний о деталях и тонкостях языка, расскажут о его внутренних взаимосвязях и помогут вам в поиске золотой середины при разработке собственных программ.

Взгляните еще раз на фотографию на обложке книги, в правый верхний угол. Видите там воздушный шар? Вот так и мы должны подняться над городом и увидеть его весь, во всей перспективе — красоту и изящество одних строений, простоту и надежность других, понять, что стиль и суть взаимосвязаны, и они не просто сосуществуют, но и взаимодействуют и дополняют друг друга. Поднявшись над городом, мы видим не отдельные дома, но весь город в его красоте и неповторимости, и выработать именно этот взгляд на C++ — во всей его красоте и целостности — должна помочь нам данная книга.

Метод Сократа

Греческий философ Сократ обучал своих учеников, задавая им вопросы, которые были разработаны таким образом, чтобы направлять мышление учеников и помогать

[1] Вышедшие в издательстве "Вильямс" объединенными в одну книгу: *Саттер Г. Решение сложных задач на C++. Серия C++ In-Depth, т.4. М.: Издательский дом "Вильямс", 2004. — Прим. ред.*

им сделать верные выводы из того, что они уже знают, а также показать им взаимосвязь изучаемого материала с другими знаниями. Этот метод обучения стал так популярен, что сегодня мы называем его "методом Сократа". С точки зрения учащихся подход Сократа включает их в процесс обучения, заставляет думать и помогает применить уже имеющиеся знания к новой информации.

Эта книга вполне следует методу Сократа, как и ее предшественницы [Sutter00] и [Sutter02]. Предполагается, что вам приходится заниматься написанием промышленного программного обеспечения на языке C++; в книге используются вопросы и ответы для обучения эффективному применению стандарта C++ и его стандартной библиотеки, причем особое внимание уделяется разработке надежного программного обеспечения с использованием всех возможностей современного C++. Многие из рассмотренных в книге задач появились в результате работы автора и других программистов над своими программами. Цель книги — помочь читателю сделать верные выводы, как из хорошо известного ему материала, так и из только что изученного, и показать взаимосвязь между различными частями C++.

Данная книга не посвящена какому-то конкретному аспекту C++. Нельзя, однако, сказать, что она охватывает все детали C++ — для этого потребовалось бы слишком много книг, — но, тем не менее, в ней рассматривается широкая палитра возможностей C++ и стандартной библиотеки и, что немаловажно, демонстрируется, как кажущиеся на первый взгляд несвязанными между собой вещи могут совместно использоваться для получения новых решений старых и хорошо известных задач. Здесь вы найдете материал, посвященный шаблонам и пространствам имен, исключениям и наследованию, проектированию надежных классов и шаблонам проектирования, обобщенному программированию и магии макросов, — и не просто винегрет из этих вопросов, а задачи и решения, выявляющие взаимосвязь всех этих частей современного C++.

Эта книга продолжается с того места, где заканчивается изложение материала в [Sutter00] и [Sutter02], и следует той же традиции: Материал книги подается в виде задач, сгруппированных по темам. Читатели первых книг найдут здесь наполненные новым содержанием уже знакомые им темы — безопасность исключений, обобщенное программирование, методы оптимизации и управления памятью. Основное внимание уделяется вопросам обобщенного программирования и эффективного использования стандартной библиотеки C++.

Большинство задач первоначально были опубликованы в Internet и некоторых журналах, в частности, это расширенные версии задач 63–86, которые можно найти на моем узле *Guru of the Week* [GotW], а также материалы, опубликованные мною в таких журналах, как *C/C++ User Journal*, *Dr. Dobb's Journal*, бывшем *C++ Report* и др. Последние исправления и дополнения к книге можно найти на Web-узле по адресу `www.gotw.ca`.

Как читать данную книгу

Предполагается, что читатель уже хорошо знаком с основами C++. Если это не так, начните с хорошего введения и обзора по C++. Для этой цели могу порекомендовать вам такие книги, как [Stroustrup00], [Lippman98], а также [Meyers96] и [Meyers97].

Каждая задача в книге имеет заголовок, который выглядит, как показано ниже.

Задача №. Название задачи **Сложность: X**

Название задачи и уровень ее сложности подсказывают вам, для кого она предназначена. Обычно в задаче есть вопрос для новичка, что позволяет размяться прежде

чем приступить к главной части — вопросу для профессионала. Замечу, что указанный уровень сложности задач — это мое субъективное мнение относительно того, насколько сложно будет решить ту или иную задачу большинству читателей, так что вы вполне можете обнаружить, что задача с уровнем сложности 7 решена вами гораздо быстрее, чем задача с уровнем сложности 5. Со времени выхода в свет предыдущих книг я получил немало писем, в которых читатели утверждали, что задача *M* сложнее (проще) задачи *N*. Это только подтверждает мой тезис о субъективности оценок и их зависимости от конкретных знаний и опыта читателя.

Чтение книги от начала до конца — неплохое решение, но вы не обязаны поступать именно так. Вы можете, например, читать только интересующие вас разделы книги. За исключением того, что я называю "мини-сериями" (связанные между собой задачи с одинаковым названием и подзаголовками "Часть 1", "Часть 2" и т.д.), задачи в книге практически независимы друг от друга, и вы можете читать их в любом порядке. Единственная подсказка: мини-серии лучше читать вместе; во всем остальном — выбор за вами.

Все примеры кода — всего лишь отрывки программ, если не оговорено иное, и не следует ожидать, что эти отрывки будут корректно компилироваться при отсутствии остальных частей программы. Для этого вам придется самостоятельно дописывать недостающие части.

И последнее замечание — об URL: нет ничего более изменчивого, чем Web, и более мучительного, чем давать ссылки на Web в печатной книге: зачастую эти ссылки устаревают еще до того, как книга попадает в типографию. Так что ко всем приведенным в книге ссылкам следует относиться критически, и в случае их некорректности — пишите мне. Дело в том, что все ссылки даны через мой Web-узел `www.gotw.ca`, и в случае некорректности какой-либо ссылки я просто обновлю перенаправление к новому местоположению страницы (если найду ее) или укажу, что таковой больше нет (если не смогу ее найти).

Благодарности

В первую очередь я хочу поблагодарить мою жену Тину (Tina) за ее терпение, любовь и поддержку, а также всю мою семью за то, что они всегда со мной, как при написании книг, так и в любое другое время. Без их безграничного терпения и участия книга бы не получилась такой, какой вы ее видите.

Я выражаю особую признательность редактору серии Бьерну Страуструпу (Bjarne Stroustrup), а также редакторам Питеру Гордону (Peter Gordon) и Дебби Лафферти (Debbie Lafferty), Тирреллу Альбах (Tyrrell Albaugh), Бернарду Гафни (Bernard Gaffney), Курту Джонсону (Curt Johnson), Чанда Лири-Куту (Chanda Leary-Coutu), Чарльзу Леди (Charles Leddy), Мелинде Мак-Кейн (Malinda McCain), Чати Прасерсиху (Chuti Prasertsith) и всем остальным членам команды Addison-Wesley за их помощь и настойчивость в работе над данным проектом. Трудно представить себе лучшую команду для работы над данной книгой; их энтузиазм и помощь помогли сделать эту книгу тем, чем она, я надеюсь, является.

Еще одна группа людей, заслуживающих особой благодарности, — это множество экспертов, чья критика и комментарии помогли сделать материал книги более полным, более удобочитаемым и более полезным. Особую благодарность я хотел бы выразить Бьерну Страуструпу (Bjarne Stroustrup), Дэйву Абрамсу (Dave Abrahams), Стиву Адамчику (Steve Adamczyk), Андрею Александреску (Andrei Alexandrescu), Чаку Аллисону (Chuck Allison), Мэтту Остерну (Matt Austern), Йоргу Барфурту (Joerg Barfurth), Питу Беккеру (Pete Becker), Брендону Брею (Brandon Bray), Стиву Дьюхарсту (Steve Dewhurst), Джонатану Кейвзу (Jonathan Caves), Питеру Димову (Peter Dimov), Хавьеру Эстрада (Javier Estrada), Аттиле Фехеру (Attila Fehér), Марко Далла Гасперина (Marco Dalla Gasperina), Дугу Грегору (Doug Gregor), Марку Холлу (Mark Hall), Кэвлину Хен-

ни (Kevlin Henney), Говарду Хиннанту (Howard Hinnant), Кэю Хорстману (Cay Horstmann), Джиму Хайслопу (Jim Hyslop), Марку Камински (Mark E. Kaminsky), Дэннису Манклу (Dennis Mancl), Брайену Мак-Намара (Brian McNamara), Скотту Мейерсу (Scott Meyers), Джеффу Пейлу (Jeff Peil), Джону Поттеру (John Potter), П. Плагеру (P. J. Plauger), Мартину Себору (Martin Sebor), Джеймсу Слотеру (James Slaughter), Николаю Смирнову (Nikolai Smirnov), Джону Спайсеру (John Spicer), Яну Кристиану ван Винклю (Jan Christiaan van Winkel), Дэвиду Вандевурду (Daveed Vandevoorde) и Биллу Вейду (Bill Wade). Все оставшиеся в книге ошибки, описки и неточности — только на моей совести.

Еще одна благодарность — нашему щенку Франки (Frankie), который оттаскивал меня от стола и тащил дышать свежим воздухом, без чего, конечно, моя работа никогда не была бы закончена. Замечу, что Франки ничего не знает о программировании, оптимизации, архитектуре программного обеспечения, и при этом он всегда выглядит счастливым и довольным. Над этим стоит задуматься...

Герб Саттер (Herb Sutter)
Сиэтл, май 2004

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Информация для писем из:

России: 115419, Москва, а/я 783

Украины: 03150, Киев, а/я 152

Обобщенное программирование и стандартная библиотека C++

Одной из наиболее мощных возможностей C++ является поддержка обобщенного программирования. Эта возможность находит непосредственное отражение в гибкости стандартной библиотеки C++, в особенности в контейнерах, итераторах и алгоритмах, известных под названием стандартной библиотеки шаблонов (Standard Template Library — STL).

Так же, как и предыдущая книга [Sutter02], эта книга начинается с задач, которые привлекают наше внимание к некоторым хорошо знакомым частям STL, в частности к векторам и строкам. Сможете ли вы избежать широко распространенных ловушек при использовании такого базового контейнера STL, как вектор? Как вы выполните обычные манипуляции со строками в C++? Какие уроки в плане конструирования библиотек вы сможете извлечь для себя из STL?

После того как мы разберемся с предопределенными шаблонами STL, мы обратимся к более общим вопросам, связанным с шаблонами и обобщенным программированием на C++. Как можно избежать при разработке собственного шаблонного кода его *не*обобщенности? Почему специализация шаблонов функций — не лучшая идея, и что следует делать вместо этого? Как корректно и переносимо добиться тех же результатов, которые дают отношения дружественности? И что нам дает ключевое слово `export`?

Эти и другие вопросы будут рассмотрены нами в разделе, посвященном обобщенному программированию и стандартной библиотеке C++.

Задача 1. Вектор: потребление и злоупотребление

Сложность: 4

Почти все используют `std::vector`, и это хорошо. К сожалению, многие не всегда верно понимают его семантику и в результате невольно применяют его странными, а порой и опасными способами. Сколько из перечисленных ниже проблем можно найти в ваших программах?

Вопрос для новичка

1. В чем разница между строками *A* и *B*?

```
void f( vector<int>& v ) {
    v[0];        // A
    v.at(0);     // B
}
```

Вопрос для профессионала

2. Рассмотрим следующий код.

```
vector<int> v;

v.reserve( 2 );
assert( v.capacity() == 2 );
v[0] = 1;
v[1] = 2;
for(vector<int>::iterator i = v.begin(); i<v.end(); i++){
    cout << *i << endl;
}

cout << v[0];
v.reserve( 100 );
assert( v.capacity() == 100 );
cout << v[0];

v[2] = 3;
v[3] = 4;
// ...
v[99] = 100;
for( vector<int>::iterator i = v.begin(); i<v.end(); i++){
    cout << *i << endl;
}
```

Раскритикуйте этот код, как с точки зрения стиля, так и с точки зрения корректности.

Решение

Обращение к элементу вектора

1. **В чем разница между строками *A* и *B*?**

```
void f( vector<int>& v ) {
    v[0];        // A
    v.at(0);     // B
}
```

В примере 1-1, если вектор v не пуст, разницы между строками *A* и *B* нет. Если же v пуст, то в строке *B* будет гарантированно сгенерировано исключение `std::out_of_range`, но что произойдет в строке *A*, сказать невозможно.

Имеется два способа обращения к элементам, содержащимся в векторе. Первый, `vector<T>::at`, выполняет проверку диапазона значения индекса, чтобы убедиться, что требуемый элемент действительно содержится в векторе. Не имеет никакого смысла обращение к сотому элементу вектора, в котором содержится всего 10 элементов, и если вы попытаетесь сделать это, то функция `at` защитит вас от неверных действий, генерируя исключение `std::out_of_range`.

Оператор `vector<T>::operator[]` может, но не обязан выполнять проверку диапазона. В стандарте об этом ничего не сказано, так что разработчик вашей стандартной библиотеки имеет полное право как добавить такую проверку, так и обойтись без нее. Если вы используете `operator[]` для обращения к элементу, отсутствующему в векторе, вы делаете это на свой страх и риск, и стандарт ничего не говорит о том, что может произойти в данном случае (хотя описание этой ситуации может оказаться в документации к используемой вами реализации стандартной библиотеки). Возможно ваша программа аварийно завершится, или будет сгенерировано исключение, или же программа будет продолжать работать, выдавая неверные результаты, или аварийно завершится в каком-то совершенно ином месте.

Такая проверка диапазона защищает нас от множества проблем. Так почему же стандарт не требует ее выполнения в операторе `operator[]`? Краткий ответ прост: эффективность. Постоянная проверка диапазона может привести к накладным расходам (возможно, небольшим) во всех ваших программах, даже там, где гарантированно не может быть нарушения границ. Согласно принципам C++, вы не должны платить за то, чего не используете, и поэтому проверка диапазона в операторе `operator[]` не является обязательной. В конкретном случае с векторами у нас есть еще одна причина для приоритета эффективности: векторы предназначены для использования вместо встроенных массивов, и поэтому они должны быть настолько же эффективны, как и массивы (в которых не выполняются проверки диапазона). Если вы хотите, чтобы такая проверка осуществлялась, — используйте функцию `at`.

Увеличение размера вектора

Теперь обратимся к примеру 1-2, который работает с `vector<int>` при помощи некоторых простых операций.

2. **Рассмотрим следующий код.**

```
vector<int> v;

v.reserve( 2 );
assert( v.capacity() == 2 );
```

Раскритикуйте этот код, как с точки зрения стиля, так и с точки зрения корректности.

Данная проверка связана с двумя проблемами, смысловой и стилистической.

Смысловая проблема состоит в том, что проверка может сработать неверно. Почему? Потому что вызов `reserve` гарантирует, что емкость вектора становится равной как минимум 2, но может быть и больше 2. Обычно это так и есть, потому что типичная реализация вектора может всегда увеличивать внутренний буфер экспоненциально, невзирая на конкретный запрос посредством функции `reserve`. Поэтому корректная проверка должна использовать оператор сравнения >=, а не строгое равенство.

```
assert( v.capacity() >= 2 );
```

Во-вторых, стилистическая ошибка заключается в том, что проверка избыточна. Почему? Потому что стандарт гарантирует выполнение проверяемого условия. Зачем же нужна явная проверка? Она не имеет смысла, если только вы не подозреваете о

наличии ошибок в используемой вами реализации стандартной библиотеки и стремитесь избежать больших проблем.

```
v[0] = 1;
v[1] = 2;
```

Обе эти строки — грубейшие, но трудно обнаруживаемые ошибки, поскольку такая программа вполне может "работать" (в зависимости от используемой конкретной реализации библиотеки).

Имеется существенная разница между функциями size/resize и capacity/reserve, т.е. между размером и емкостью вектора.

- size говорит нам, сколько элементов содержится в контейнере в настоящее время, а resize изменяет содержимое контейнера таким образом, чтобы он содержал указанное количество элементов в контейнере путем добавления или удаления их из конца контейнера. Обе эти функции присутствуют в контейнерах list, vector и deque и отсутствуют в остальных.
- capacity возвращает количество мест для элементов в контейнере, т.е. указывает, сколько элементов можно разместить в контейнере перед тем, как добавление очередного элемента вызовет выделение нового блока памяти. Функция reserve при необходимости увеличивает (но никогда не уменьшает) размер внутреннего буфера , чтобы он был способен вместить как минимум указанное количество элементов. Обе функции предусмотрены только у контейнера vector.

В нашем случае мы использовали вызов v.reserve(2) и, таким образом, гарантировали, что v.capacity() >= 2, но мы не добавляли элементы в вектор v, так что вектор v остается пуст! На данный момент все, что можно сказать о векторе — это то, что в нем есть место как минимум для двух элементов.

➢ **Рекомендация**

Помните о разнице между size/resize и capacity/reserve.

Мы можем безопасно использовать оператор operator[] (или функцию at) только для изменения элементов, которые реально содержатся в контейнере, т.е. реально учтены в size. Вы можете удивиться, почему оператор operator[] не может быть достаточно интеллектуальным, чтобы добавить элемент в контейнер, если он еще не в контейнере. Но если бы operator[] позволял делать такие вещи, мы бы могли создавать вектор с "дырами"! Рассмотрим, например, следующий фрагмент.

```
vector<int> v;
v.reserve( 100 );
v[99] = 42;     // Ошибка, но, допустим, такое возможно...
// ... что тогда можно сказать о значениях v[0..98]?
```

Увы, поскольку не предусмотрено, чтобы оператор operator[] выполнял проверку диапазона, в большинстве реализаций выражение v[0] будет просто возвращать ссылку на еще неиспользуемую память во внутреннем буфере вектора, а именно на то место в памяти, где в конечном итоге будет находиться первый элемент вектора. Следовательно, скорее всего, инструкция v[0] = 1; будет "нормально работать", т.е., например, при выводе cout << v[0] вы, вероятно, увидите на экране 1, как и ожидалось (и совершенно необоснованно!).

Но описанный сценарий — не более чем типичный вариант того, что может случиться. На самом деле все зависит от реализации стандартной библиотеки. Стандарт ничего не говорит о том, что должно происходить при записи элемента v[0] в пустом векторе v, поскольку программист легко может узнать о том, что вектор пуст, чтобы не пытаться выполнять такую запись. В конце концов, если ему очень надо, он может обеспечить выполнение соответствующей проверки, воспользовавшись вызовом v.at(0)...

Само собой разумеется, присваивания `v[0] = 1; v[1] = 2;` будут вполне корректны и осмысленны, если заменить вызов `v.reserve(2)` вызовом `v.resize(2)`. Можно также получить корректный код, заменив присваивания вызовами `v.push_back(1); v.push_back(2);`, которые обеспечивают безопасный способ размещения элементов в конце контейнера.

```
for(vector<int>::iterator i = v.begin(); i<v.end(); i++){
    cout << *i << endl;
}
```

Во-первых, заметим, что этот цикл ничего не выводит, поскольку вектор все еще пуст. Это может удивить автора рассматриваемого кода, если, конечно, он не сообразит, что по сути он ничего не внес в контейнер, а всего лишь поиграл (так и хочется сказать — с огнем) с зарезервированным местом в памяти, которое официально вектором не использовано.

То есть формально в цикле нет ошибки, что, однако, не исключает необходимости привести ряд стилистических замечаний.

1. *Старайтесь по возможности использовать `const`-вариант итератора.* Если итератор не используется для модификации содержимого вектора, следует использовать `const_iterator`.
2. *Итераторы следует сравнивать при помощи оператора сравнения* `!=`*, но не при помощи* `<`. Конечно, так как `vector<int>::iterator` — итератор произвольного доступа (само собой разумеется, не обязательно типа `int*`!), нет никаких проблем при сравнении `<v.end()`, использованном в примере. Однако такое сравнение при помощи `<` работает только с итераторами произвольного доступа, в то время как сравнение с использованием оператора `!=` работает со всеми типами итераторов. Поэтому мы должны везде использовать именно такое сравнение, а сравнение `<` оставить только для тех случаев, где это действительно необходимо. (Заметим, что сравнение `!=` существенно облегчит переход к использованию другого типа контейнера в будущем, если это вдруг потребуется. Например, итераторы `std::list` не поддерживают сравнение с использованием оператора `<`, так как являются бинаправленными итераторами.)
3. *Лучше использовать префиксную форму операторов* `--` *и* `++` *вместо постфиксной.* Возьмите за привычку писать в циклах по умолчанию `++i` вместо `i++`, если только вам действительно не требуется старое значение `i`. Например, постфиксная форма оператора естественна при использовании кода наподобие `v[i++]` для обращения к *i*-му элементу и одновременно увеличения счетчика цикла.
4. *Избегайте излишних вычислений.* В нашем случае значение, возвращаемое при вызове `v.end()`, не изменяется в процессе работы цикла, так что вместо вычисления его заново при каждой итерации лучше вычислить его один раз перед началом цикла.

   Примечание. Если ваша реализация `vector<int>::iterator` представляет собой простой указатель `int*`, а функция `end()` встраиваемая, то при определенном уровне оптимизации накладные расходы будут сведены практически к нулю, так как интеллектуальный компилятор будет способен обнаружить, что значение, возвращаемое `end()`, не изменяется в процессе работы цикла. Современные компиляторы вполне способны справиться с такой задачей. Однако если `vector<int>::iterator` не является `int*` (например, в большинстве отладочных реализаций это тип класса), функции не являются встраиваемыми и/или компилятор не способен выполнить необходимую оптимизацию, вынесение вычислений из цикла может существенно повысить производительность кода.
5. *Предпочтительнее использовать* `'\n'` *вместо* `endl`. Использование `endl` заставляет выполнить сброс внутренних буферов потока. Если поток буферизован, а сброс буферов

всякий раз по окончании вывода строки не требуется, лучше использовать endl один раз, в конце цикла; это также повысит производительность вашей программы.

Все это были комментарии "низкого уровня"; однако есть замечание и на более высоком уровне.

6. *Вместо разработки собственных циклов, используйте там, где это ясно и просто, стандартные библиотечные алгоритмы copy и for_each. Больше полагайтесь на свой вкус.* Я говорю так потому, что это как раз тот случай, когда определенную роль играют вкус и эстетизм. В простых случаях copy и for_each могут улучшить читаемость и понятность кода по сравнению с циклами, разработанными вручную. Тем не менее, часто код с использованием for_each может оказаться менее понятным и удобочитаемым, так как тело цикла придется разбивать на отдельные функторы. Иногда такое разбиение идет коду только на пользу, а иногда совсем наоборот.

   Вот почему вкус играет здесь такую важную роль. Я бы в рассматриваемом примере заменил цикл чем-то наподобие

   ```
   copy(v.begin(),v.end(),ostream_iterator<int>(cout,"\n"));
   ```

   Кроме того, при использовании того же алгоритма copy вы не сможете ошибиться в применении !=, ++, end() и endl, поскольку вам просто не придется ничего этого делать самостоятельно. (Конечно, при этом предполагается, что вы не намерены сбрасывать буферы потока при выводе каждого целого числа. Если это для вас критично, вам действительно придется писать свой цикл вместо использования стандартного алгоритма std::copy.) Повторное использование, в случае корректного применения, не только делает код более удобочитаемым и понятным, но и повышает его качество, позволяя избежать различного рода ловушек при написании собственного кода.

   Вы можете пойти дальше и написать свой собственный алгоритм для копирования контейнера, т.е. алгоритм, который работает со всем контейнером в целом, а не с какой-то его частью, определяемой диапазоном итераторов. Такой подход автоматически заставит использовать const_iterator. Рассмотрим следующий пример.

   ```
   template<class Container, class OutputIterator>
   OutputIterator copy(const Container& c,
                       OutputIterator result){
       return std::copy( c.begin(), c.end(), result );
   }
   ```

   Здесь мы просто написали обертку для применения std::copy ко всему контейнеру целиком, и так как контейнер передается как const&, итераторы автоматически будут константными (const_iterator).

➢ Рекомендация

- Будьте максимально корректны при использовании модификатора const. В частности, используйте const_iterator, если вы не модифицируете содержимое контейнера.
- Сравнивайте итераторы при помощи оператора !=, а не <.
- Лучше использовать префиксную форму операторов -- и ++ вместо постфиксной, если только вам не требуется старое значение переменной.
- Лучше использовать существующие алгоритмы, в частности, стандартные алгоритмы (такие как for_each), а не разрабатывать собственные циклы.

Далее нам встречается код

```
cout << v[0];
```

При выполнении этого кода, вероятно, результатом будет 1. Это связано с тем, что программа просто пишет в память и читает из нее — поступая при этом совсем не так, как положено, что тем не менее не приводит к немедленным фатальным сбоям (а жаль!).

```
v.reserve( 100 );
assert( v.capacity() == 100 );
```

Здесь также должна выполняться проверка с использованием оператора >=, а не ==, и даже такая проверка будет излишней (мы уже рассматривали этот вопрос раньше).

```
cout << v[0];
```

А вот тут нас ждет сюрприз! На этот раз, скорее всего, инструкция `cout << v[0];` приведет к результату 0: значение 1 таинственно исчезает...

Почему? Будем считать, что вызов `reserve(100)` приводит к запуску перераспределения памяти для внутреннего буфера `v` (если только в результате первоначального вызова `reserve(2)` не было выделено достаточного количества памяти для размещения 100 или большего числа элементов). При этом контейнер `v` копирует в новый буфер только те элементы, которые он в действительности содержит — но на самом деле он не содержит ни одного элемента! Новый буфер изначально заполнен неопределенными значениями (чаще всего — нулями), что и становится очевидным при выполнении `cout << v[0];`.

```
v[2] = 3;
v[3] = 4;
// ...
v[99] = 100;
```

Думаю, теперь у вас нет никаких сомнений в ошибочности этого кода. Это неправильный, плохой, неверный код... но поскольку `operator[]` не требует выполнения проверки диапазона, в большинстве реализаций стандартной библиотеки этот код будет молча работать, не приводя к исключениям или аварийному завершению программы.

Если же вместо приведенного выше фрагмента написать

```
v.at(2) = 3;
v.at(3) = 4;
// ...
v.at(99) = 100;
```

то проблема станет очевидной, так как первый же вызов приведет к генерации исключения `out_of_range`.

```
for(vector<int>::iterator i = v.begin(); i<v.end(); i++){
    cout << *i << endl;
}
```

Здесь опять ничего не будет выведено, а я опять предложу вам заменить этот цикл на

```
copy(v.begin(), v.end(), ostream_iterator<int>(cout,"\n"));
```

Еще раз замечу, что такой код автоматически решает все проблемы, связанные с использованием оператора сравнения `!=`, префиксной формы `++`, вызовом `end()` в теле цикла и использованием `endl`. К тому же повторное использование стандартных алгоритмов зачастую автоматически делает код более быстрым и безопасным.

Резюме

Теперь вы знаете, в чем разница между размером и емкостью, а также между оператором `operator[]` и функцией `at()`. Если необходима проверка диапазона, всегда используйте функцию `at()`, благодаря которой вы сэкономите немало времени, которое в противном случае придется потратить на работу в отладчике.

Задача 2. Строчный двор. Часть 1: sprintf Сложность: 3

В этой и следующей задачах мы рассмотрим "чудеса" `sprintf` и разберемся, почему альтернативные варианты всегда (да, всегда) лучше.

Вопрос для новичка

1. Что такое `sprintf`? Перечислите как можно больше стандартных альтернатив `sprintf`.

Вопрос для профессионала

2. В чем сильные и слабые стороны `sprintf`? Будьте конкретны в своем ответе.

Решение

"Все животные равны,
но некоторые животные равнее других".
— Джордж Оруэлл, *Скотный двор*[2]

1. Что такое `sprintf`? Перечислите как можно больше стандартных альтернатив `sprintf`.

Рассмотрим следующий код C, который использует `sprintf` для преобразования целого числа в удобочитаемое строковое представление.

```
// Пример 2-1: Строковое представление данных в C с
// использованием sprintf. Функция PrettyFormat получает в
// качестве параметра целое число и преобразует его в строку
// в предоставленный буфер. Результат должен иметь размер не
// менее 4 символов.

void PrettyFormat( int i, char* buf ) {
    // Код, простой и понятный:
    sprintf( buf, "%4d", i );
}
```

Вопрос на засыпку: как сделать то же на C++?

Впрочем, это не совсем корректный вопрос, так как, в конце концов, этот же код вполне корректен и в C++. Вопрос на засыпку надо сформулировать так: если отбросить все ограничения стандарта C[3] (если это и в самом деле ограничения), то имеется ли лучший способ выполнить эти же действия в C++ с его классами, шаблонами и прочим?

Этот вопрос интересен тем, что пример 2-1 можно выполнить по крайней мере четырьмя разными стандартными способами, каждый из которых представляет собой определенный компромисс между ясностью, безопасностью типов, безопасностью времени исполнения и эффективностью. Кроме того, перефразируя свиней-ревизионистов из произведения Оруэлла, "все четыре способа стандартны, но некоторые из них стандартнее других", да и взяты они из разных стандартов. Они приведены ниже в том порядке, в котором мы будем их рассматривать.

1. `sprintf` [C99, C++03]
2. `snprintf` [C99]
3. `std::stringstream` [C++03]
4. `std::strstream` [C++03]

[2] Перевод с англ. Д. Иванова, В. Недошивина (Дж. Оруэлл. *Скотный двор.* — Пермь, Изд. "КАПИК", 1992.)

И наконец, если этого вам покажется мало, есть еще пятый, нестандартный (но, возможно, станет таковым) способ для простых преобразований, не требующих специального форматирования.

5. `boost::lexical_cast` [Boost]

Итак, приступим к рассмотрению.

Радости и печали sprintf

2. В чем сильные и слабые стороны `sprintf`? Будьте конкретны в своем ответе.

Рассмотренный код функции `PrettyFormat` — всего лишь один из примеров использования `sprintf`. Я использовал его лишь как повод для обсуждения, так что не надо придавать слишком большое значение этой однострочной функции. Вопрос гораздо шире — мы выбираем способ для представления нестроковых значений в виде строк.

Давайте проанализируем функцию `sprintf` более детально. У нее есть два наиболее важных достоинства и три недостатка.

1. Простота использования и ясность. Как только вы изучите, как флаги форматирования и их комбинации влияют на форматирование строки, использование `sprintf` становится простым и очевидным. Очень трудно превзойти семейство функций `printf` в задачах по форматированию текста. (Да, запомнить все флаги не просто, но обычно это относится к достаточно редко используемым флагам форматирования.)

2. Максимальная эффективность (возможность непосредственного использования существующих буферов). При использовании функции `sprintf` результат помещается непосредственно в заранее подготовленный буфер, так что функция `PrettyFormat` выполняет свою работу без динамического выделения памяти или других побочных действий. Она получает уже выделенную память и записывает результирующую строку непосредственно в эту память.

Конечно, не стоит придавать этому достоинству большее значение, чем оно того заслуживает. Ваше приложение может даже не заметить разницы между использованием `sprintf` и другими методами. Никогда ничего не оптимизируйте преждевременно, приступайте к оптимизации только тогда, когда профилирование покажет, что она действительно необходима. Начинайте с ясного и понятного кода, быстрым его можно сделать потом — если в этом появится необходимость. В нашем случае необходимо учесть, что эффективность достигается за счет инкапсуляции управления памятью.

Увы, как известно большинству пользователей функции `sprintf`, у нее есть и значительные недостатки.

3. Безопасность. Функция `sprintf` — распространенный источник ошибок, связанных с переполнением буфера, когда размера выделенного буфера не хватает для размещения выводимой строки[4]. Рассмотрим, например, следующий код.

```
char smallBuf[5];
int  value = 42;
PrettyFormat( value, smallBuf );  // Вроде бы все в порядке
assert( value == 42 );
```

В этом случае значение 42 достаточно мало для того, чтобы результат "42\0" полностью разместился в пяти байтах `smallBuf`. Но представим, что однажды код изменится на такой, как показано ниже.

```
char smallBuf[5];
int value = 12108642;
```

---

[3] В настоящее время — [C99], стандарт C++ [C++03] основан на более ранней версии C.

[4] Распространенная ошибка начинающих программистов — полагаться на спецификатор ширины вывода, в нашем случае — 4, который не работает так, как они рассчитывают, поскольку этот спецификатор указывает **минимальную** длину вывода, а не максимальную.

```
PrettyFormat( value, smallBuf );  // Ах!
assert( value == 12108642 );      // У нас проблема!
```

При этом начинается запись в память за пределами `smallBuf`, что может привести к записи в память, выделенную переменной `value`, если компилятор расположит ее в памяти непосредственно за переменной `smallBuf`.

Сделать код из примера 2-1 существенно безопаснее — достаточно трудная задача. Можно изменить код так, чтобы в функцию передавался размер буфера и проверялось значение, которое возвращается функцией `sprintf` и показывает, сколько байтов записано функцией. Это даст нам код, который выглядит примерно следующим образом.

```
// Плохое решение
//
void PrettyFormat( int i, char* buf, int buflen ) {
    if (buflen <= sprintf(buf,"%4d",i)) { // Лучше не стало
        // Ну и что? Пока мы выясним, что произошла
        // неприятность, эта неприятность уже успеет
        // испортить память. Мы просто убеждаемся, что
        // неприятность действительно произошла.
  }
}
```

Решения вообще не существует. К тому моменту, когда можно убедиться в наличии или отсутствии ошибки, она уже произошла, так что при плохом варианте развития событий мы можем просто не добраться до выполнения указанной проверки[5].

4. Безопасность типов. При использовании функции `sprintf` ошибки в применении типов являются не ошибками времени компиляции, а ошибками времени выполнения программы. Они вдвойне опасны тем, что могут остаться необнаруженными даже на этапе выполнения. Функции семейства `printf` используют переменные списки аргументов из C, а компиляторы C не проверяют типы параметров в таких списках[6]. Почти каждый программист на C хоть раз, но имел сомнительное удовольствие искать источник ошибки, вызванной неверным спецификатором формата, и очень часто такая ошибка обнаруживается только после ночи работы с отладчиком в попытках воспроизвести загадочное сообщение об ошибке, присланное пользователем.

Конечно, код в примере 2-1 тривиален, и ошибиться здесь сложно. Но кто застрахован от опечаток? Ваша рука скользнула чуть ниже, и вместо `d` вы ударили по клавише `c`, в результате чего получился следующий код.

```
sprintf( buf, "%4c", i );
```

Правда, в данном случае вы быстро обнаружите ошибку, когда на выходе вместо числа получите некоторый символ (так как в этом случае функция `sprintf` молча выведет младший байт `i` как символ). А можно промахнуться чуть левее и ударить по клавише `s`, получив следующий код.

---

[5] Заметим, что в некоторых случаях проблему переполнения буфера можно разрешить, по крайней мере, теоретически, создавая собственные форматные строки в процессе работы. Я говорю "теоретически", потому что обычно это весьма непрактично; такой код всегда невразумителен и часто подвержен ошибкам. Вот вариант Б.Страуструпа, предложенный им в [Stroustrup99] с оговоркой о том, что это профессиональное решение, не рекомендуемое к использованию новичками.

```
char fmt[10];
// Создание строки формата: обычный %s может привести к
// переполнению буфера
sprintf(fmt,"%%%ds",max-1);
// Считываем не более max-1 символов в переменную name
scanf(fmt,name);
```

[6] Использование соответствующего инструментария наподобие lint может помочь обнаружить ошибки такого рода.

```
sprintf( buf, "%4s", i );
```

Эта ошибка тоже быстро проявит себя, так как такая программа должна будет немедленно (или почти немедленно) аварийно завершиться. В данном случае целое число будет рассматриваться как указатель на символ, так что функция просто попытается вывести строку из некоторого случайного места в памяти.

Но вот более тонкая ошибка. Что произойдет, если мы ошибочно заменим `d` на `ld`?

```
sprintf( buf, "%4ld", i );
```

В этом случае строка формата говорит функции `sprintf`, что в качестве первой части форматируемых данных ожидается значение типа `long int`, а не просто `int`. Это тоже плохой С-код, причем проблема в том, что это не только не ошибка времени компиляции, но может не быть и ошибкой времени выполнения. На многих платформах результат работы программы от этого не изменится, поскольку на них размер `int` и размер `long int` совпадают. Такая ошибка может остаться незамеченной до тех пор, пока вы не перенесете вашу программу на новую платформу, где размер `int` не равен размеру `long int`, но даже тогда такая ошибка может не приводить к некорректному выводу или аварийному завершению программы.

И наконец, еще одна неприятность.

5. Невозможность работы в шаблонах. Очень трудно использовать `sprintf` в шаблонах. Рассмотрим следующий код.

```
template<typename T>
void PrettyFormat( T value, char* buf ) {
    sprintf( buf, "%/* Что должно быть здесь? */", value );
}
```

Лучшее (или худшее?), что можно сделать в этой ситуации, — это объявить основной шаблон и обеспечить специализации для всех типов, совместимых со `sprintf`.

```
// Плохо: попытка шаблонизации PrettyFormat.
//
template<typename T>
void PrettyFormat( T value, char* buf ); // Главный шаблон
                                         // не определен
template<> void PrettyFormat<int>(int value, char* buf){
    sprintf( buf, "%d", value );
}
template<> void PrettyFormat<char>(char value, char* buf){
    sprintf( buf, "%c", value );
}
//  ... и т.д. ...
```

Подведем итог нашему обсуждению функции `sprintf`, собрав информацию о ней в следующей таблице.

	sprintf
Стандартность?	Да: [C90], [C++03], [C99]
Простота использования и ясность?	Да
Эффективность, без излишних распределений памяти?	Да
Безопасность в плане переполнения буфера?	Нет
Безопасность типов?	Нет
Использование в шаблонах?	Нет

Решения, которые будут рассмотрены в следующей задаче, обеспечивают другие компромиссы между указанными параметрами.

Задача 3. Строчный двор. Часть 2: стандартные альтернативы — Сложность: 6

Продолжаем сравнительный анализ `snprintf`, `std::stringstream`, `std::strstream` и нестандартного, но элегантного `boost::lexical_cast`.

Вопрос для профессионала

1. Проведите сравнительный анализ каждой из перечисленных альтернатив функции `sprintf` и выявите их сильные и слабые стороны, используя решение задачи 2 в качестве образца:
   а) `snprintf`
   b) `std::stringstream`
   в) `std::strstream`
   г) `boost::lexical_cast`

Решение

Альтернатива №1: snprintf

1. **Проведите сравнительный анализ каждой из перечисленных альтернатив функции `sprintf` и выявите их сильные и слабые стороны, используя решение задачи 2 в качестве образца:**
   а) `snprintf`

Среди прочих вариантов, `snprintf`, естественно, ближе других к `sprintf`. В этой функции добавлена только одна, но очень важная возможность: указать максимальный размер выходного буфера, тем самым обеспечивая невозможность переполнения буфера. Если переданный буфер слишком мал, выходная строка будет усечена.

Функция `snprintf` долгое время была широко распространенным нестандартным расширением практически во всех реализациях компиляторов C. После принятия стандарта C99 [C99] функция `snprintf` официально стала частью стандарта. Пока ваш компилятор не станет полностью C99-совместимым, возможно, вам придется использовать некоторое другое имя функции, специфичное для вашего компилятора (например, в ряде компиляторов это `_snprintf`).

Честно говоря, всегда следует использовать `snprintf` вместо `sprintf` (и следовало даже до того, как `snprintf` стал стандартом). Использование функций без проверки длины буфера запрещено в большинстве приличных стандартов кодирования, и не напрасно. Использование `sprintf` приводит к массе проблем, от аварийного завершения в общем случае[7] до проблем безопасности в частности[8].

Используя `snprintf`, мы можем написать корректный код безопасной функции `PrettyFormat`, что мы уже пытались, но не смогли сделать ранее.

---

[7] Это реальная проблема, причем характерная не только для функции `sprintf()`, но и для других функций без проверки длины буфера. Попробуйте выполнить поиск в Google по ключевым словам "strcpy" и "buffer overflow", и убедитесь в том, что это действительно так.

[8] Например, передача длинного URL Web-браузеру или серверу, код которого содержит вызовы функций без проверки размера буфера, может привести к перезаписи данных в памяти за внутренним буфером. В ряде случаев это позволяет разработать такой злонамеренный запрос, содержащий код, который в результате будет выполнен программой. Просто удивительно, какое количество программ разработаны таким образом, с использованием вызовов без проверки длины!

```
// Пример 3-1: Преобразование данных в строку с
// использованием функции snprintf.
//
void PrettyFormat( int i, char* buf, int buflen ) {
    // Этот код столь же прост, как и ранее, но уже
    // более безопасен:
    snprintf( buf, buflen, "%4d", i );
}
```

Заметим, что при этом все равно остается возможность для ошибки — вызывающая функция может передать неверный размер буфера. Это означает, что 100% безопасности в плане переполнения буфера не обеспечивает и `snprintf`, но она гораздо безопаснее функции `sprintf`. На вопрос "Безопасна ли эта функция в смысле переполнения буфера?" следует однозначно ответить "Да".

Заметим, что некоторые "достандартные" версии функции `snprintf` вели себя несколько иначе. В частности, в одной из распространенных реализаций при полном заполнении буфера он не завершался нулевым символом. В такой ситуации наша функция `PrettyFormat` должна немного отличаться от исходного варианта, чтобы учесть такое нестандартное поведение.

```
// Преобразование данных в строку с использованием функции
// snprintf, которая не полностью отвечает стандарту C99.
//
void PrettyFormat( int i, char* buf, int buflen ) {
    // Этот код столь же прост, как и ранее, но уже
    // более безопасен:
    if( buflen > 0 ) {
        _snprintf( buf, buflen-1, "%4d", i );
        buf[buflen-1] = '\0';
    }
}
```

Во всех остальных отношениях функции `sprintf` и `snprintf` идентичны. Приведем следующую таблицу сравнения `snprintf` и `sprintf`.

	snprintf	sprintf
Стандартность?	Да: только в [C99], но, вероятно, будет и в C++0x	Да: [C90], [C++03], [C99]
Простота использования и ясность?	Да	Да
Эффективность, без излишних распределений памяти?	Да	Да
Безопасность в плане переполнения буфера?	Да	Нет
Безопасность типов?	Нет	Нет
Использование в шаблонах?	Нет	Нет

Из этого сравнения вполне логично вытекает следующая рекомендация.

➢ Рекомендация

Никогда не используйте функцию `sprintf`.

Если вы решите использовать возможности стандартного ввода-вывода из C, всегда используйте только те функции, которые проверяют размер буфера, такие как

snprintf, даже если эти функции в вашем компиляторе доступны только в качестве нестандартного расширения. При использовании snprintf вместо sprintf нет никаких неприятностей, зато есть очень важное преимущество.

Когда я представлял этот материал на нескольких конференциях, я был шокирован тем, что только примерно каждый десятый знал о существовании такой функции, как snprintf. Зато на каждой конференции кто-нибудь рассказывал, как в его проекте были обнаружены несколько случаев переполнения буфера, после чего sprintf были повсеместно заменены на snprintf. В результате тестирования оказывалось, что кроме явных ошибок, связанных с переполнением буфера и известных программистам, чудесным образом пропадали и ошибки, на которые им указывали годами, но которые они никак не могли локализовать.

Итак, еще раз: забудьте о существовании функции sprintf.

Альтернатива №2: std::stringstream

б) `std::stringstream`

Наиболее распространенным средством для строкового представления данных в C++ является семейство stringstream. Ниже представлен код примера 3-1 при использовании ostringstream вместо sprintf.

```
// Пример 3-2: строковое представление данных в C++ с
// использованием ostringstream.
//
void PrettyFormat( int i, string& s ) {
    // Не так уж понятно и просто:
    ostringstream temp;
    temp << setw(4) << i;
    s = temp.str();
}
```

Обратите внимание, что использование stringstream меняет достоинства и недостатки sprintf местами.

1. Простота использования и ясность. Как видите, изменения свелись не только к замене одной строки тремя, но и к введению временной переменной: Эта версия кода превосходит предыдущую по ряду параметров, но простота и ясность не входят в их число. Сложность не в том, чтобы изучить все манипуляторы потоком, — в конце концов, это задача той же сложности, что и изучить флаги форматирования sprintf; дело в том, что эти манипуляторы гораздо более громоздкие. Мне кажется, что код с длинными именами, например, `<< setprecision(9)` и `<< setw(14)`, читается не так легко, как флаги форматирования в snprintf, где то же форматирование достигается при помощи простой строки `%14.9`, даже если аккуратно отформатировать исходный текст программы.

2. Эффективность (возможность непосредственного использования существующих буферов). stringstream работает в дополнительном буфере, так что при его использовании обычно необходимо дополнительное выделение памяти для рабочего буфера и вспомогательных объектов. Я провел небольшой эксперимент, скомпилировав пример 3-2 двумя популярными компиляторами, изменив при этом `::operator new` для подсчета выполняемых при работе кода выделений памяти. На одной платформе я получил два динамических выделения памяти, а на другой — три.

Однако там, где у sprintf начинают проявляться недостатки, stringstream радует своими достоинствами.

3. Безопасность в плане переполнения буфера. Внутренний буфер basic_stringbuf потока stringstream автоматически увеличивается, если в этом возникает необходимость.

4. Безопасность типов. Использование перегруженного оператора << и разрешения перегрузки обеспечивает корректную работу с разными типами, включая пользовательские типы, которые могут иметь собственные операторы вывода в поток. При ис-

пользовании stringstream никакие ошибки времени выполнения, связанные с несоответствием типов, попросту невозможны.

5. Возможность работы в шаблонах. Поскольку теперь всегда автоматически вызывается корректный оператор <<, обобщить функцию PrettyFormat для работы с произвольными типами данных — тривиальная задача.

```
template<typename T>
void PrettyFormat( T value, string& s ) {
    ostringstream temp;
    temp << setw(4) << value;
    s = temp.str();
}
```

Итак, в результате мы получили следующую таблицу сравнения stringstream и sprintf.

	stringstream	sprintf
Стандартность?	Да: [C++03]	Да: [C90], [C++03], [C99]
Простота использования и ясность?	Нет	Да
Эффективность, без излишних распределений памяти?	Нет	Да
Безопасность в плане переполнения буфера?	Да	Нет
Безопасность типов?	Да	Нет
Использование в шаблонах?	Да	Нет

Альтернатива №3: std::strstream

в) `std::strstream`

Хорошо это или нет, но в связи с тем, что strstream в стандарте [C++03] называют устаревшим и не рекомендуемым для использования, в книгах по C++ либо в лучшем случае дается краткое описание этого класса ([Josuttis99]), либо он практически проигнорирован ([Stroustrup00]), или в книге явно указано, что из-за "второсортности" strstream его описание отсутствует ([Langer00]). Однако несмотря на то, что комитет по стандарту C++ отдал предпочтение stringstream, в котором лучше инкапсулировано управление памятью, strstream остается официальной частью стандарта, которую обязаны поддерживать все реализации C++[9].

Поскольку strstream все еще остается частью стандарта, для полноты рассмотрения мы обязаны изучить этот класс. В случае его использования пример 3-1 выглядит следующим образом.

```
// Пример 3-3: строковое представление данных в C++ с
// использованием ostrstream.
//
void PrettyFormat( int i, char* buf, int buflen ) {
```

[9] Статус strstream — нежелателен (deprecated), что означает, что комитет по стандарту C++ предупреждает: этот класс может исчезнуть из стандарта в любой момент, возможно, уже в следующей версии стандарта. Но удалить нечто из стандарта — практически очень сложная задача. Ведь как только та или иная возможность появляется в стандарте, появляется и множество кода с ее использованием, и удаление из стандарта грозит потерей обратной совместимости. Даже при официальном удалении чего-либо из стандарта конкретные реализации зачастую продолжают поддерживать это в целях обеспечения обратной совместимости. Нередко нежелательные возможности так никогда и не удаляются из стандарта. Так, в стандарте Fortran они присутствуют десятилетиями.

```
    // Неплохо, но надо не забывать об ends:
    ostrstream temp( buf, buflen );
    temp << setw(4) << i << ends;
}
```

1. Простота использования и ясность. `strstream` немного уступает `stringstream` в плане простоты использования и ясности кода. Они оба требуют создания временного объекта. При использовании `strstream` вы должны не забывать добавлять `ends` для завершения строки, и эту его особенность можно определить как нечто между "безвкусно" и "опасно". Если вы забудете сделать это, возникнет опасность чтения после конца буфера (если вы полагаетесь только на завершающий нулевой символ). Даже раскритикованная нами функция `sprintf` не поступает так, и всегда завершает строку нулевым символом. Но зато использование `strstream` таким образом, как это показано в примере 3-3, не требует вызова функции `.str()` для получения конечного результата. (Конечно, если вы поступите иначе и позволите `strstream` создать собственный буфер, вам потребуется не только вызов `.str()` для получения конечного результата, но и вызов `.freeze(false)`, поскольку иначе `strstreambuf` не будет освобождать выделенную память.)

2. Эффективность (возможность непосредственного использования существующих буферов). При создании объекта `ostrstream` с передачей ему указателя на существующий буфер нам не требуется выполнять никакого дополнительного распределения памяти; `ostrstream` будет записывать результат непосредственно в этот буфер. Это очень важное отличие от `stringstream`, в котором нет никаких подобных возможностей для помещения результата непосредственно в существующий буфер во избежание излишнего перераспределения памяти[10]. Конечно, `ostrstream` может использовать и собственный динамически выделенный буфер, если у вас нет своего — для этого вам надо просто воспользоваться конструктором `ostrstream` по умолчанию[11]. Из всех рассматриваемых в этом разделе альтернатив такая возможность эффективной работы есть только у `strstream`.

3. Безопасность в плане переполнения буфера. Как видно из примера 3-3, у `ostrstream` его внутренний буфер `strstreambuf` автоматически проверяет свою длину, чтобы обеспечить невозможность записи за пределами выделенной памяти. Если же мы используем конструктор `ostrstream` по умолчанию, то его внутренний буфер при необходимости будет автоматически увеличиваться.

4. Безопасность типов. Так же, как и `stringstream`, `strstream` полностью безопасен в этом отношении.

5. Возможность работы в шаблонах. Так же, как и `stringstream`, обеспечивает такую возможность. Вот небольшой пример, иллюстрирующий ее.

```
template<typename T>
void PrettyFormat( T value, char* buf, int buflen ) {
    ostrstream temp( buf, buflen );
    temp << setw(4) << value << ends;
}
```

Подводя итоги, получаем следующую таблицу сравнения `strstream` с `sprintf`.

---

[10] У `stringstream` есть конструктор, который получает в качестве параметра `string&`, но он просто делает копию содержимого переданной строки вместо непосредственного использования этой строки в качестве рабочей области.

[11] В табл. 3.1 исследования производительности `strstream` показаны всего лишь для двух компиляторов — Borland C++ 5.5.1 и Visual C++ 7. Дело в том, что в этих реализациях C++ по каким-то причинам при каждом вызове функции `PrettyFormat()` из примера 3-3 выполняются дополнительные выделения памяти (хотя при передаче конструктору указателя на существующий буфер выделений памяти оказывается меньше, чем когда `strstream` создает собственный буфер). Прочие среды, как и ожидалось, работают без дополнительного выделения памяти.

	strstream	sprintf
Стандартность?	Да: [C++03], хотя и объявлен нежелательным	Да: [C90], [C++03], [C99]
Простота использования и ясность?	Нет	Да
Эффективность, без излишних распределений памяти?	Да	Да
Безопасность в плане переполнения буфера?	Да	Нет
Безопасность типов?	Да	Нет
Использование в шаблонах?	Да	Нет

Несколько смущает тот факт, что настолько хорошая вещь оказалась вдруг нежелательной, но такова жизнь...

Альтернатива №4: boost::lexical_cast

г) `boost::lexical_cast`

Если вы еще незнакомы с Boost [Boost], мой совет — познакомьтесь. Это открытая библиотека для C++, написанная в основном членами комитета по стандартизации C++, которая представляет собой не только пример хорошего кода, написанного профессионалами в стиле стандартной библиотеки C++. Возможности этой библиотеки, по сути, претендуют на включение в очередной стандарт C++, так что с ними стоит ознакомиться заранее. Кроме того, вы можете использовать ее сейчас и совершенно бесплатно.

Одна из интересных возможностей, предусмотренных в Boost, — `boost::lexical_cast`, которая представляет собой оболочку вокруг `stringstream`. Boost включает также более мощные подходы, которые используют внутренние потоки и обеспечивают большое количество опций форматирования, в частности, `boost::format`, но поскольку код, написанный Кэвлином Хенни (Kevlin Henney), предельно краток и очень элегантен, я могу полностью представить его здесь (исключив обходные пути для старых компиляторов). И я делаю это, несмотря на то, что это — нестандартная возможность.

```
template<typename Target, typename Source>
Target lexical_cast( Source arg ) {
    std::stringstream interpreter;
    Target result;
    if(!(interpreter << arg) || !(interpreter >> result) ||
       !(interpreter >> std::ws).eof())
        throw bad_lexical_cast();
  return result;
}
```

Заметим, что шаблон `lexical_cast` не предназначен для конкуренции с гораздо более мощным средством форматирования строк, таким как `sprintf`. `lexical_cast` предназначен для конвертирования данных из одного типа в другой, так что он может представить конкуренцию скорее для функций C типа `atoi()` и подобных. Однако данный шаблон настолько близок рассматриваемой теме, что не упомянуть о нем просто невозможно.

Ниже представлен пример 3-1 при использовании `lexical_cast` (требование вывода как минимум 4 символов удалено).

```
// Пример 3-4: строковое представление данных в C++ с
// использованием boost::lexical_cast.
```

```
//
void PrettyFormat( int i, string& s ) {
    // Предельно просто:
    s = lexical_cast<string>( i );
}
```

1. Простота использования и ясность. Не стоит и доказывать, что данный код проще и понятнее всех ранее рассмотренных вариантов.

2. Эффективность (возможность непосредственного использования существующих буферов). Поскольку `lexical_cast` использует `stringstream`, не удивительно, что при работе он требует выделения памяти, как минимум, в том же количестве, что и `stringstream`. На одной из опробованных мною платформ в примере 3-4 было выполнено на одно выделение памяти больше, чем требуется при использовании обычного `stringstream` (см. пример 3-2); на другой платформе дополнительного выделения памяти не было.

В остальных аспектах — безопасности и возможности использования в шаблонах — `lexical_cast` соответствует `stringstream`.

Результаты сравнения `lexical_cast` и `sprintf` представлены в следующей таблице.

	lexical_cast	sprintf
Стандартность?	Нет (возможный кандидат в стандарт C++0x)	Да: [C90], [C++03], [C99]
Простота использования и ясность?	Да	Да
Эффективность, без излишних распределений памяти?	Нет	Да
Безопасность в плане переполнения буфера?	Да	Нет
Безопасность типов?	Да	Нет
Использование в шаблонах?	Да	Нет

Резюме

Конечно, ряд вопросов не был рассмотрен нами подробно. Например, все форматирование строк рассматривалось нами только в отношении обычных символов, а вопросы использования "широких" символов нами не затрагивались. Кроме того, вопросы эффективности затрагивались нами в плане непосредственного использования существующих буферов, но другая сторона этой эффективности — необходимость управления памятью программистом при использовании `sprintf`, `snprintf` и `strstream`, в то время как применение `stringstream`, `strstream` и `lexical_cast` позволяет вам избежать этого. (Здесь нет опечатки или противоречия, `strstream` внесен в оба списка в связи с тем, что его можно использовать и так, и этак.)

Помимо `lexical_cast`, рассмотренного здесь в силу его краткости и элегантности, имеется масса других нестандартных альтернатив функции `sprintf`, в том числе в самой библиотеке `boost` (например, такая замечательная и очень мощная вещь, как `boost::format`, позволяющая добавить форматирование в стиле `sprintf` к уже рассмотренным `stringstream` и `strstream`).

После обобщения результатов исследований мы получили сводную табл. 3.1, из которой понятно, что идеального варианта не существует, но, тем не менее, можно дать определенные рекомендации, которые сведены в табл. 3.2.

- Если необходимо преобразовать значение в строку `string` (или во что-то иное) — используйте `boost::lexical_cast`.

- Для простого форматирования, или при необходимости поддержки широких строк, или для использования в шаблонах выбирайте `stringstream` или `strstream`.
- Для выполнения более сложного форматирования, если не требуется поддержка широких строк или использование в шаблонах, воспользуйтесь функцией `snprintf`. То, что это функция из C, вовсе не означает, что она не может быть использованной в C++!
- Если проведенные измерения показывают, что проблема производительности действительно связана с использованием неэффективного метода форматирования, только в этом случае, только в этих конкретных местах кода имеет смысл воспользоваться одной из более быстрых альтернатив — `strstream` или `snprintf`.
- Никогда не используйте `sprintf`.

Таблица 3.1. Сравнение различных методов форматирования строк

	sprintf	snprintf	stringstream	strstream	boost::lexical_cast
	Стандартность				
[C90]	Да	Нет (распространенное расширение)	Нет	Нет	Нет
[C++03]	Да	Нет (распространенное расширение)	Да	Да, но нежелательно	Нет
[C99]	Да	Да	Нет	Нет	Нет
C++0x (предположительно)	*Да*	*Вероятно*	*Да*	*Вероятно*	*Возможно*
	Простота использования				
Простота использования и ясность?	Да	Да	Нет	Нет	Да
Эффективность, без излишних распределений памяти?	Да	Да	Нет	Да	Нет
Безопасность в плане переполнения буфера?	Нет	Да	Да	Да	Да
Безопасность типов?	Нет	Нет	Да	Да	Да
Использование в шаблонах?	Нет	Нет	Да	Да	Да

Окончание табл. 3.1

	sprintf	snprintf	stringstream	strstream	boost::lexical_cast
	Время работы по отношению к sprintf[12]				
Borland C++ 5.5.1 / Windows	1.0	1.0	12.6	8.1	19.7
Gnu g++ 2.95.2 / Cygwin + Windows	1.0	—	—	2.0	—
Microsoft VC7 / Windows	1.0	1.0	13.2	9.0	19.2
Rogue Wave 2.1.1 / SunPro 5.3 / SunOS 5.7	1.0	1.1	8.7	4.7	16.5
Rogue Wave 2.2.1 / HP aCC 3.30 / HP-UX 11.00	1.0	1.0	7.9	3.9	9.9

Таблица 3.2. Правила применения методов форматирования строк

	По умолчанию, когда эффективность не так важна	Там, где эффективность важна, и профилирование подтверждает необходимость оптимизации
Простое конвертирование в строковое представление	`boost::lexical_cast`	`std::strstream` или `snprintf`
Простое форматирование либо работа с широкими строками или в шаблонах	`std::stringstream` или `std::strstream`	`std::strstream` или `snprintf`
Сложное форматирование при отсутствии необходимости в работе с широкими строками или в шаблонах	`snprintf`	`snprintf`

В заключение отметим несколько интересных возможностей, предусмотренных в `strstream`. Не последняя из них — возможность выбора между использованием встроенного или предоставляемого программистом буфера. Его существенный технический недостаток состоит в определенной "хрупкости", связанной с необходимостью использования завершения строки при помощи `ends`. Важный (назовем его "социальным") недостаток заключается в том, что следует учитывать (пусть и небольшую) вероятность того, что в один прекрасный день и комитет по стандарту C++, и производитель вашего компилятора решат избавиться от `strstream`. Честно говоря, очень странно видеть такую хорошую вещь нежелательной. Как видим, даже в стандарте некоторые животные равнее других...

[12] Проводилось усреднение по трем запускам каждой программы, в которых выполнялось по 1 млн. вызовов кода соответствующих примеров. Результаты могут зависеть от используемых версий компиляторов и опций компиляции.

Задача 4. Функции-члены стандартной библиотеки

Сложность: 5

Повторное использование — это хорошо, но всегда ли допустимо использование возможностей стандартной библиотеки с ней самой? Вот небольшой пример, который может вас удивить. В нем рассматривается возможность стандартной библиотеки, которая может быть переносимо использована с любым вашим кодом, но только не с самой стандартной библиотекой.

Вопрос для новичка

1. Что такое `std::mem_fun`? Когда можно использовать `std::mem_fun`? Приведите пример.

Вопрос для профессионала

2. Полагая, что в комментарии в приведенном коде все правильно, ответьте на вопрос: является ли приведенный код корректным и переносимым кодом C++? Обоснуйте свой ответ.

```
std::mem_fun</*...*/>( &std::vector<int>::clear )
```

Решение

Игры с mem_fun

1. Что такое `std::mem_fun`? Когда можно использовать `std::mem_fun`? Приведите пример.

Стандартный адаптер `mem_fun` позволяет нам использовать функции-члены в алгоритмах стандартной библиотеки и другом коде, который обычно работает с функциями, не являющимися членами класса (свободными функциями). Пусть, например, мы имеем

```
class Employee {
public:
    int DoStandardRaise() { /*...*/ }
    //...
};
int GiveStandardRaise( Employee& e ) {
    return e.DoStandardRaise();
}
std::vector<Employee> emps;
```

Мы уже привыкли писать, например, следующим образом.

```
std::for_each( emps.begin(), emps.end(), &GiveStandardRaise );
```

Но что, если функции `GiveStandardRaise()` не существует или по каким-то иным причинам нам надо вызвать функцию-член непосредственно? Тогда мы можем написать следующий код.

```
std::vector<Employee> emps;
std::for_each(emps.begin(), emps.end(),
              std::mem_fun_ref(&Employee::DoStandardRaise));
```

Суффикс `_ref` в конце `mem_fun_ref` — дань историческим традициям. Когда мы пишем код наподобие приведенного, мы просто должны помнить, что надо использовать `mem_fun_ref`, если имеем дело с обычным контейнером с объектами и `for_each` работает со ссылками на эти объекты, и использовать `mem_fun`, если контейнер содержит указатели на объекты.

```
std::vector<Employee*> emp_ptrs;
std::for_each(emp_ptrs.begin(), emp_ptrs.end(),
              std::mem_fun(&Employee::DoStandardRaise));
```

Вы, вероятно, обратили внимание, что для ясности я воспользовался функцией без параметров. Вы можете воспользоваться вспомогательными классами bind... для функций с аргументом — принцип при этом остается тот же. К сожалению, вы не можете использовать этот подход для функций с двумя и более параметрами.

Вот, по сути, главное, что надо знать о mem_fun. И это подводит нас к трудной части задачи.

Используйте mem_fun, но не со стандартной библиотекой

2. **Полагая, что в комментарии в приведенном коде все правильно, ответьте на вопрос: является ли приведенный код корректным и переносимым кодом C++? Обоснуйте свой ответ.**

```
std::mem_fun</*...*/>( &std::vector<int>::clear )
```

Я специально сформулировал вопрос таким образом (с комментарием вместо аргумента шаблона), поскольку некоторые распространенные компиляторы в такой ситуации не в состоянии корректно вывести аргументы шаблона. Для них, в зависимости от вашей реализации стандартной библиотеки, может потребоваться написать что-то наподобие

```
std::mem_fun<void, std::vector<int, std::allocator<int> > > (
            &std::vector<int>::clear);
```

Со временем это ограничение должно быть ликвидировано, и компиляторы будут позволять опускать параметры шаблонов.

Вы можете удивиться моим примечанием "в зависимости от вашей реализации стандартной библиотеки". В конце концов, сигнатура std::vector<int>::clear гласит, что эта функция не получает параметров и ничего не возвращает, ведь так? Разве стандарт не говорит нам об этом?

По сути, в этом и заключается главный вопрос задачи.

Спецификация стандартной библиотеки сознательно предоставляет разработчикам определенную свободу действий по реализации функций-членов. В частности:

- сигнатура функции-члена с параметрами по умолчанию может быть заменена "двумя или более сигнатурами функций-членов с эквивалентным поведением";
- сигнатура функции-члена может иметь дополнительные параметры по умолчанию.

Второй пункт и есть главная неприятность. Все эти "может быть или не быть", игры в прятки с дополнительными параметрами и привели к появлению данной задачи.

В большинстве случаев в этом нет никаких проблем, и все эти параметры по умолчанию просто остаются незамеченными. Например, при вызове функции-члена эти скрытые параметры, которых вы не передаете, попадают в функцию в виде значений по умолчанию, и вы даже не подозреваете об их существовании. Но, к сожалению, эти параметры начинают играть важную роль, когда становится важна точная сигнатура функции — например, когда вы пытаетесь использовать mem_fun. Заметим, что это происходит, даже если ваш компилятор корректно выводит аргументы шаблонов. Это связано с двумя потенциальными проблемами.

- Если рассматриваемая функция-член получает параметр, о котором вы не подозревали, вам понадобится написать что-то наподобие std::bind2nd для того, чтобы избавиться от него (более подробное описание использования стандартных замыканий параметров функции можно найти в [Josuttis99]). Конечно, после этого ваш код не будет работать в реализации, где используется дополнительный

параметр другого типа или этого параметра нет вовсе, но в конце концов ваш код и без этого все равно не был бы переносимым, правда?

- Если функция-член в действительности имеет два или больше параметров (даже если они имеют значения по умолчанию), вы вообще не сможете воспользоваться `mem_fun`.

На практике, впрочем, все может быть не настолько плохо. Пока что мне не встречались реализации, где разработчики широко пользовались бы своим правом на свободу действий по добавлению дополнительных параметров, либо намеренных воспользоваться им в будущем. И пока это так, вы не заметите проблем с этим кодом.

Увы, на этом наша история не заканчивается. Давайте рассмотрим более общее следствие такой свободы действий.

Использование указателей на функции-члены — но не со стандартной библиотекой

Увы, есть еще один даже более фундаментальный вопрос: невозможно переносимо создать указатель на функцию-член стандартной библиотеки. Точка.

В конце концов, если вы намерены создать указатель на функцию (неважно, член или нет), вы должны знать тип этого указателя, а значит — знать точную сигнатуру функции. Поскольку знать точную сигнатуру функций-членов стандартной библиотеки невозможно (пока вы не возьмете заголовочные файлы вашей реализации стандартной библиотеки и не посмотрите, нет ли в функциях спрятанных от постороннего взгляда параметров по умолчанию; впрочем, кто может гарантировать, что они не появятся уже в следующей версии этой же библиотеки?), приходится сделать вывод, что нет надежного способа создать указатель на функцию-член стандартной библиотеки и обеспечить переносимость такого кода.

Резюме

Достаточно странно, что вы можете переносимо использовать возможность стандартной библиотеки (а именно — `mem_fun`) практически с любым кодом, кроме кода самой стандартной библиотеки. Не менее странно, что можно переносимо использовать возможность языка (а именно — указатели на функции-члены) со всеми функциями-членами, за исключением таковых в стандартной библиотеке этого языка.

Обычно свобода в реализации функций-членов стандартной библиотеки невидима, и когда вы деласте что-то с их помощью, вы никогда и не узнаете о разнице этих функций в разных реализациях. Но если вы решите использовать указатели на функции-члены или замыкания параметров, вы должны помнить, что они не могут переносимо использоваться с функциями-членами стандартной библиотеки, более того, то, что работает у вас сегодня, может перестать работать завтра, с новой версией той же стандартной библиотеки.

Задача 5. Красота обобщенности. Часть 1: Азы

Сложность: 4

Перед тем как приступить к шестой задаче, рассмотрим простой пример гибкого обобщенного кода C++ (примеры кода в этой и следующей задаче взяты из [Sutter00].

Вопрос для новичка

1. "Шаблоны C++ обеспечивают полиморфизм времени компиляции". Поясните эту фразу.

Вопрос для профессионала

2. Поясните семантику приведенной функции. Дайте максимально полный ответ и обязательно поясните, почему в ней использованы два параметра шаблона, а не один.

```
template <class T1, class T2>
void construct( T1* p, const T2& value ) {
    new (p) T1(value);
}
```

Решение

1. **"Шаблоны C++ обеспечивают полиморфизм времени компиляции". Поясните эту фразу.**

Когда мы говорим о полиморфизме в объектно-ориентированном мире, мы подразумеваем полиморфизм времени выполнения, который основан на использовании виртуальных функций. Базовый класс определяет интерфейс, представляющий собой набор виртуальных функций, а производные классы могут наследовать их из базового класса и перекрывать их таким образом, чтобы сохранялась предполагаемая интерфейсом семантика. Тогда код, который работает с базовым объектом (получая объект `Base` через указатель или по ссылке), может точно так же работать и с объектом производного класса.

```
// Пример 5-1(a): полиморфизм времени выполнения.
//
class Base {
public:
    virtual void f();
    virtual void g();
};
class Derived : public Base {
    // Перекрытие при необходимости f и/или g
};
void h( Base& b ) {
    b.f();
    b.g();
}
int main() {
    Derived d;
    h( d );
}
```

Это очень важная возможность, обеспечивающая высокую степень гибкости времени выполнения. Однако полиморфизм времени выполнения имеет и два существенных недостатка. Во-первых, типы должны быть иерархически связаны отношением

наследования; во-вторых, при вызове виртуальных функций во внутренних циклах можно заметить определенное уменьшение производительности, поскольку обычно вызов виртуальной функции выполняется посредством указателя, с дополнительным уровнем косвенности, который позволяет компилятору выяснить, какой именно вызов — функции базового или производного объекта — должен быть выполнен.

Если вы знаете используемые типы во время компиляции, вы можете избежать обоих недостатков полиморфизма времени выполнения: вы можете использовать типы, не связанные наследованием, лишь бы они обеспечивали возможность выполнения ожидаемых операций.

```
// Пример 5-1(б): полиморфизм времени компиляции.
//
class Xyzzy {
public:
    void f( bool someParm = true );
    void g();
    void GoToGazebo();
    // ... прочие функции ...
};
class Murgatroyd {
public:
    void f();
    void g( double two = 6.28, double e = 2.71828 );
    int HeavensTo( const Z& ) const;
    // ... прочие функции ...
};
template<class T>
void h( T& t ) {
    t.f();
    t.g();
}
int main() {
    Xyzzy x;
    Murgatroyd m;
    h( x );
    h( m );
}
```

До тех пор пока объекты `x` и `m` будут иметь типы, обеспечивающие возможность вызова функций `f` и `g` без аргументов, функция `h` будет корректно работать. В примере 5-1(б) в действительности функции `f` и `g` имеют разные сигнатуры в разных классах, а кроме того, эти классы, помимо интересующих нас `f` и `g`, имеют разные дополнительные функции. Но все это не оказывает никакого влияния на работу функции `h`. До тех пор пока функции `f` и `g` могут быть вызваны без аргументов, компилятор позволит функции `h` вызывать `f` и `g`. Конечно, при вызове эти функции должны делать нечто осмысленное для функции `h`!

Таким образом, шаблоны обеспечивают мощный полиморфизм времени компиляции. Неверное использование шаблонов может, конечно, привести к совершенно неудобочитаемым сообщениям об ошибках, но одновременно шаблоны являются одной из наиболее сильных возможностей C++.

2. **Поясните семантику приведенной функции. Дайте максимально полный ответ и обязательно поясните, почему в ней использованы два параметра шаблона, а не один.**

```
template <class T1, class T2>
void construct( T1* p, const T2& value ) {
    new (p) T1(value);
}
```

Функция `construct` создает объект в указанном месте в памяти и инициализирует его данным значением. Оператор `new`, использованный в данном примере, называется

размещающим new (placement new), и вместо выделения памяти для нового объекта он просто помещает его в область памяти, на которую указывает p. Любой объект, создаваемый таким образом, в общем случае должен быть уничтожен явным вызовом деструктора (как показано в задаче 6, вопрос 1), а не путем использования оператора delete.

Итак, зачем же нужны два параметра шаблона? Неужели одного недостаточно для того, чтобы создать копию объекта value? Например, если шаблон construct имеет только один параметр, вам может потребоваться явно указать тип этого параметра при копировании объекта другого типа.

```
// Пример 5-2: шаблон construct с меньшими функциональными
// возможностями, и пояснение, почему они меньше.
//
template <class T1>
void construct( T1* p, const T1& value ) {
    new (p) T1(value);
}
// Считаем, что и p1, и p2 указывают на нетипизированную
// область памяти.
//
void f( double* p1, Base* p2 ) {
    Base b;
    Derived d;
    construct( p1, 2.718 );        // OK
    construct( p2, b );            // OK
    construct( p1, 42 );           // Ошибка: T1 - double
                                   // или int?
    construct<double>( p1, 42 );   // OK
    construct( p2, d );            // Ошибка: T1 - Base
                                   // или Derived?
    construct<Base>( p2, d );      // OK
}
```

Причина, по которой два вызова помечены как ошибочные, — в неоднозначности. Компилятору недостаточно имеющейся информации для того, чтобы вывести аргумент шаблона, поэтому для корректности кода следует указывать этот аргумент явным образом. Можем ли мы позволить программисту без каких-либо предупреждений построить объект double из целого значения? Вероятно, в худшем случае это повлечет лишь потерю точности. Можем ли мы позволить построить объект типа Base из объекта Derived? Пожалуй, да. Если это допустимо, то произойдет срезка объекта Derived (но это может быть сделано намеренно).

Итак, если мы хотим позволить программисту выполнять описанные действия без явного указания типов, то нам нужна исходная версия шаблона с двумя независимыми параметрами.

Задача 6. Красота обобщенности. Часть 2: Достаточно ли универсальности? Сложность: 7

Насколько универсальной в действительности является обобщенная функция? Ответ может зависеть как от ее реализации, так и от ее интерфейса, а отлично разработанный интерфейс может быть сведен на нет простыми (и трудными в обнаружении) ошибками программирования.

Вопрос для профессионала

1. В приведенной функции есть одна малозаметная ловушка, связанная с обобщенностью. В чем она состоит и как лучше всего ее исправить?

```
template <class T>
void destroy( T* p ) {
    p->~T();
}
template <class FwdIter>
void destroy( FwdIter first, FwdIter last ) {
    while( first != last ) {
        destroy( first );
        ++first;
  }
}
```

2. В чем заключается семантика приведенной далее функции, включая требования к T? Можно ли снять какие-либо из этих требований? Если да, то продемонстрируйте, как именно, и укажите достоинства и недостатки соответствующего решения.

```
template <class T>
void swap( T& a, T& b ) {
    T temp(a); a = b; b = temp;
}
```

Решение

1. **В приведенной функции есть одна малозаметная ловушка, связанная с обобщенностью. В чем она состоит и как лучше всего ее исправить?**

```
template <class T>
void destroy( T* p ) {
    p->~T();
}
template <class FwdIter>
void destroy( FwdIter first, FwdIter last ) {
    while( first != last ) {
        destroy( first );
        ++first;
  }
}
```

Шаблон функции `destroy` предназначен для уничтожения диапазона объектов. Первая версия получает один указатель и вызывает деструктор указываемого объекта. Вторая версия получает диапазон итераторов и итеративно уничтожает объекты в указанном диапазоне.

Здесь есть одна тонкая ловушка, которая не проявлялась ни в одном примере при первоначальном появлении этого кода в книге [Sutter00]. Небольшая проблема состоит в том, что функция с двумя параметрами `destroy(FwdIter,FwdIter)` может получать любой обобщенный итератор, но при работе она вызывает функцию с одним

аргументом destroy(T*), передавая ей один из итераторов. Это автоматически приводит к тому требованию, что итератор FwdIter должен быть обычным указателем! А это означает ненужную потерю общности.

➢ Рекомендация

Запомните, что указатели (в массив) всегда являются итераторами, но итераторы не всегда представляют собой указатели.

Это также означает, что вы можете получить весьма туманные сообщения об ошибках при попытке скомпилировать код, который пытается осуществить вызов destroy(FwdIter,FwdIter) с итераторами, не являющимися указателями, поскольку ошибка будет диагностироваться в строке destroy(first). Выводимые при этом сообщения об ошибках, скажем прямо, особой ясности не прибавляют. Посмотрите, как они выглядят у одного из распространенных компиляторов.

```
'void __cdecl destroy(template-parameter-1,
template-parameter-1)' : expects 2 arguments - 1 provided

'void __cdecl destroy(template-parameter-1 *)' : could not
deduce template argument for 'template-parameter-1 *' from
'[использованный тип итератора]'
```

Это не худшие из виденных мною сообщений об ошибках, и в принципе не так сложно понять, что именно они обозначают. Первое сообщение указывает, что компилятор пытался разрешить вызов destroy(first); как вызов версии destroy с двумя аргументами. Второе говорит о попытке вызова версии с одним параметром. Обе попытки оказались неудачными, каждая по своей причине. Версию с двумя аргументами устраивает тип итератора, но ей нужны два параметра, а не один. Версия же с одним параметром требует, чтобы он обязательно был указателем. Не везет!

В действительности, мы вряд ли когда-то будем использовать функцию destroy с чем-либо кроме указателей в силу ее предназначения — превратить объекты в обычную память. Тем не менее, одно небольшое изменение — и FwdIter может быть итератором произвольного типа, а не только указателем. Так почему бы не сделать его? Все, что надо сделать, — это в версии с двумя параметрами заменить вызов

```
destroy( first );
```

вызовом

```
destroy( &*first );
```

Такая замена будет работать практически всегда. Здесь мы разыменовываем итератор для того, чтобы получить непосредственную ссылку на содержащийся в контейнере объект, а затем получаем его адрес, что дает нам требуемый указатель. В соответствии со стандартом, все итераторы должны иметь оператор *, который возвращает истинную ссылку T&. Это одна из причин, по которой стандартом C++ не поддерживаются прокси-контейнеры (дополнительную информацию по этому вопросу можно найти в книге [Sutter02][13]. (Возможны, хотя и крайне редки, ситуации, когда вызов destroy(&*first); будет работать некорректно: как указал хитроумный читатель Билл Вейд (Bill Wade), этот метод неприменим, если в T перегружен оператор &, который возвращает нечто вместо адреса объекта. Но это уже патология, и оправдания такому дизайну я просто не в состоянии придумать.)

В чем же мораль этой истории? Не забывайте о возможных нарушениях универсальности при реализации одной обобщенной функции с использованием другой. В нашем случае это вылилось в то, что версия функции с двумя параметрами оказалась не

---

[13] Задача 1.13 в издании на русском языке. — *Прим. ред.*

настолько универсальной в плане использования итераторов, как мы изначально рассчитывали. Более того, в нашей задаче мы сталкиваемся с еще одним препятствием: попытка исправить ситуацию путем замены `destroy(first);` вызовом `destroy(&*first);` приводит к новому требованию к типу `T`, которое заключается в том, что он должен предоставлять оператор & с обычной семантикой, т.е. возвращающий указатель на объект. Обеих ловушек можно избежать, если не использовать одну функцию в реализации другой.

Примечание. Я не призываю вас полностью отказаться от реализации шаблонов с использованием шаблонов; но помните о проблемах, которые может вызвать такое взаимодействие. Очевидно, что шаблоны часто могут быть реализованы при помощи других шаблонов. Например, программисты часто специализируют шаблон `std::swap` для своих типов, если они знают более эффективный способ обмена двух значений. И если вы пишете шаблон сортировки, то эта сортировка должна быть реализована с использованием вызова шаблона `swap` — в противном случае ваш шаблон не сможет воспользоваться оптимизированным способом обмена содержимого объектов.

2. **В чем заключается семантика приведенной далее функции, включая требования к `T`? Можно ли снять какие-либо из этих требований? Если да, то продемонстрируйте, как именно, и укажите достоинства и недостатки соответствующего решения.**

```
// Пример 6-2(a)
//
template <class T>
void swap( T& a, T& b ) {
    T temp(a); a = b; b = temp;
}
```

Шаблон функции `swap` просто обменивает два значения с использованием копирующего конструктора и оператора копирующего присваивания. Таким образом, этот шаблон требует, чтобы тип `T` имел конструктор копирования и оператор копирующего присваивания.

Если это все, что вы смогли сказать, — поставьте себе за этот ответ только полбалла. Одной из важнейших составляющих семантики любой функции является ее безопасность с точки зрения исключений, включая обеспечиваемые ею гарантии безопасности. В данном случае функция `swap` не является безопасной в смысле исключений, если оператор копирующего присваивания `T` может генерировать исключения. В частности, если `T::operator=` может генерировать исключения, но при этом атомарен ("все или ничего"), то если второе присваивание оказалось некорректным, мы выходим из функции по исключению, но при этом объект a оказывается уже измененным. Если же оператор `T::operator=` может генерировать исключения и не является атомарным, то возможна ситуация, когда мы выходим из функции по исключению и при этом оба параметра оказываются измененными (и один из них может не содержать ни одного из двух исходных значений). Следовательно, данный шаблон функции `swap` должен быть документирован следующим образом.

- Если оператор `T::operator=` не генерирует исключений, `swap` гарантирует атомарность операции ("все или ничего"), за исключением побочных действий операций `T` (см. также [Sutter99]).
- В противном случае, если оператор `T::operator=` может генерировать исключение :
  - если оператор `T::operator=` атомарен и осуществляется выход из функции `swap` по исключению, первый аргумент функции может быть изменен (хотя и не обязательно);
  - в противном случае, если оператор `T::operator=` не атомарен и осуществляется выход из функции `swap` по исключению, оба аргумента могут быть

изменены (хотя и не обязательно), причем один из них может не содержать ни первое, ни второе исходное значение.

Существует два способа избавиться от требования, касающиеся типа T и состоящего в том, что T должен иметь оператор присваивания, причем первый способ обеспечивает большую безопасность исключений.

1. Специализация или перегрузка swap. Пусть, например, у нас есть класс MyClass, который использует распространенную идиому не генерирующей исключений функции Swap. Тогда мы можем специализировать стандартную функцию для MyClass следующим образом.

```
// Пример 6-2(б): Специализация swap.
//
class MyClass {
public:
    void Swap( MyClass& ) /* throw() */ ;
    // ...
};
namespace std {
    template<> void swap<MyClass>(MyClass& a,
                                  MyClass& b ) { // throw()
        a.Swap( b );
    }
}
```

Мы можем также перегрузить стандартную функцию для MyClass следующим образом:

```
// Пример 6-2(в): Перегрузка swap.
//
class MyClass {
public:
    void Swap( MyClass& ) /* throw() */ ;
    // ...
};
// Примечание: не в пространстве имен std.
void swap( MyClass& a, MyClass& b ) /* throw() */ {
    a.Swap( b );
}
```

Обычно это неплохая мысль — даже если T имеет оператор присваивания, который обеспечивает корректную работу исходной версии кода!

Например, стандартная библиотека перегружает[14] swap для векторов, так что вызов swap приводит к вызову vector::swap. Это имеет большой смысл, поскольку обмен vector::swap становится более эффективным, если удается избежать копирования данных, содержащихся в векторах. Первичный шаблон в примере 6-2(а) создает новую копию (temp) одного из векторов, а затем выполняет дополнительное копирование из одного вектора в другой и из вектора temp во второй вектор, что приводит к массе операций T и времени работы $O(N)$, где N — общий размер обмениваемых векторов. Специализированная версия обычно просто выполняет присваивание нескольких указателей и целочисленных объектов, причем в течение постоянного (и обычно пренебрежимо малого) времени.

Так что если вы создаете тип, в котором есть операция наподобие swap, имеет смысл специализировать std::swap (или предоставить свою собственную функцию swap в другом пространстве имен) для вашего типа. Обычно это оказывается более эффективным способом обмена, чем работающий "в лоб" шаблон std::swap из стан-

[14] Но не "специализирует", так как вы не можете *частично* специализировать шаблоны функций. См. задачу 7, в которой более подробно рассматриваются шаблоны функций и специализации.

дартной библиотеки; к тому же это зачастую дает более безопасную с точки зрения исключений функцию.

> **Рекомендация**
>
> Подумайте о специализации `std::swap` для ваших типов, если для них возможен более эффективный способ обмена, чем стандартный.

2. Уничтожение и восстановление. Идея этого способа заключается в обмене с использованием копирующего конструктора `T` вместо оператора копирующего присваивания; конечно, этот способ работает, только если `T` имеет конструктор копирования.

```
// Пример 6-2(г): swap без присваивания.
//
template <class T>
void swap( T& a, T& b ) {
    if( &a != &b ) {          // Примечание: эта проверка
                              // теперь необходима!
        T temp( a );
        destroy( &a );
        construct( &a, b );
        destroy( &b );
        construct( &b, temp );
    }
}
```

Начнем с того, что этот метод не подходит, если конструктор копирования `T` может генерировать исключения, так как при этом мы получим все те же проблемы, что и в исходной версии, только еще усугубленные. Вполне реальна ситуация, когда объекты не только хранят некоторые промежуточные значения, но и вообще не существуют!

Если мы знаем, что конструктор копирования гарантированно не генерирует исключений, то данная версия имеет то преимущество, что позволяет работать с типами, которые не имеют оператора присваивания, но могут быть сконструированы путем копирования, а такие типы вовсе не являются редкостью. Впрочем, возможность обмена для таких типов отнюдь *не* обязательно считается достоинством. Если тип нельзя присваивать, то, вероятно, на то есть существенные причины, например, если он не имеет семантики значения, но имеет константные или ссылочные члены, то механизм, который обеспечивает (или даже навязывает) типу семантику значения, может приводить к неожиданным и некорректным результатам.

Что еще хуже, этот подход начинает игру с временем жизни объектов, а это всегда чревато неприятностями. Здесь под "игрой" я подразумеваю, что при этом изменяются не только значения, но и само существование объектов, с которыми работает данная функция. Код, использующий функцию `swap` из примера 6-2(г), может легко привести к неожиданным результатам, если пользователь забудет о необычной семантике уничтожения.

Небольшая рекомендация. Если вы вынуждены играть со временем жизни объектов и знаете, что здесь все в порядке, *и* вы точно знаете, что конструкторы копирования объектов, с которыми вы работаете, никогда не генерируют исключений, *и* вы абсолютно убеждены, что навязанная семантика значений для данных объектов в вашем приложении вполне допустима, то тогда (и только тогда) вы можете оправданно использовать описанный подход в конкретных перечисленных ситуациях. Но даже при этом не делайте такую операцию универсальным шаблоном, который может быть случайно использован для любого другого типа, и четко документируйте ее, чтобы никто случайно не воспользовался ею в другой, не столь благоприятной ситуации, потому что в контексте C++ эта методика выглядит очень экзотично.

Задача 7. Почему не специализируются шаблоны функций? Сложность: 8

Хотя заголовком этой задачи является вопрос, его легко можно переформулировать в указание. Данная задача отвечает на вопрос о том, когда и почему не следует специализировать шаблоны.

Вопрос для новичка

1. Какие два основных вида шаблонов имеются в C++, и как они могут быть специализированы?

Вопрос для профессионала

2. Какая из версий функции f будет вызвана в последней строке приведенного кода? Почему?

```
template<class T>
void f( T );

template<>
void f<int*>( int* );

template<class T>
void f( T* );
// ...
int *p;
f( p );         // Какая функция будет вызвана?
```

Решение

Перегрузка и специализация

Очень важно убедиться в правильном использовании терминологии при обсуждении данного вопроса, поэтому сначала — небольшой обзор.

1. Какие два основных вида шаблонов характерны для C++ и как они могут быть специализированы?

В C++ предусмотрено два типа шаблонов — *шаблоны классов* и *шаблоны функций*. Эти два типа шаблонов работают не совсем одинаково, и наиболее очевидное отличие заключается в перегрузке (overloading): обычные классы C++ не могут быть перегружены, так что не могут быть перегружены и их шаблоны. С другой стороны, функции в C++, имеющие одинаковые имена, могут быть перегружены, так что могут быть перегружены и шаблоны функций. Это вполне понятно и естественно, что демонстрируется следующим примером.

```
// Пример 7-1: Шаблоны классов и функций и перегрузка
//

// Шаблон класса
template<class T> class X { /* ...*/ };        // (а)

// Шаблон функции с перегрузкой
template<class T> void f( T );                 // (б)
template<class T> void f( int, T, double );    // (в)
```

Такие неспециализированные шаблоны называются также *первичными шаблонами* (primary templates).

Первичные шаблоны могут быть *специализированы*. В этом шаблоны классов и шаблоны функций существенно отличаются, как вы увидите далее в данной задаче. Шаблон класса может быть *частично* (partially) или *полностью специализирован* (fully specialized)[15]. Шаблон функции может быть специализирован только полностью, но, поскольку шаблоны функции могут быть перегружены, мы можем получить при помощи перегрузки тот же эффект, что и при помощи частичной специализации. Эти отличия продемонстрированы в следующем примере.

```
// Пример 7-1, продолжение: специализированные шаблоны
//
// частичная специализация (а) для указателей
template<class T> class X<T*> { /* ... */ };

// Полная специализация (а) для типа int
template<> class X<int> { /* ... */ };

// Отдельный первичный шаблон, перегружающий (б) and (в) -
// не частичная специализация (б), поскольку понятия
// частичной специализации шаблона функции не существует!
template<class T> void f( T* );                    // (г)

// Полная специализация (б) для типа int
template<> void f<int>( int );                     // (д)

// Обычная функция, представляющая перегрузку функций (б),
// (в) и (г), но не (д) (об этом будет сказано немного
// позже)
void f( double );                                  // (е)
```

➢ Рекомендация

Помните, что шаблон функции не может быть частично специализирован, но может быть перегружен. Попытки частичной специализации шаблонов функций приводят к созданию новых первичных шаблонов функций.

Давайте теперь обратимся исключительно к шаблонам функций и рассмотрим правила перегрузки, для того чтобы разобраться, какие из них применяются в разных ситуациях. Эти правила, если не вдаваться в тонкости, довольно просты и могут быть разделены на два случая.

- Функции, не являющиеся шаблонами, являются "привилегированными". При разрешении перегрузки обычная нешаблонная функция с типами параметров, подходящими для аргументов, имеет преимущество перед всеми соответствующими в той же степени шаблонными функциями.
- Если такой "привилегированной" функции нет, в качестве претендентов рассматриваются первичные шаблоны функций. То, какой именно первичный шаблон будет выбран, зависит от того обстоятельства, какой из них в наибольшей степени соответствует аргументам функции и является "наиболее специализированным" в соответствии с некими "загадочными" правилами. (Примечание: использование термина "специализированный" в данном контексте не имеет никакого отношения к специализации шаблонов.)
  - Очевидно, что если имеется только один "наиболее специализированный" первичный шаблон функции, то используется именно он. Если первичный шаблон специализирован для используемых типов, то будет использоваться

[15] В стандарте полная специализация называется "явной специализацией" (explicit specialization).

именно эта специализация; в противном случае будет выполнено инстанцирование первичного шаблона с корректными типами.

- В противном случае, если имеется несколько таких "наиболее специализированных" первичных шаблонов функций, то вызов неоднозначен, так как компилятор не в состоянии выбрать наилучший из них. Программист при этом должен самостоятельно сделать выбор и предпринять некоторые уточняющие действия, которые скорректируют работу компилятора.
- Если ни один первичный шаблон функции не подходит, вызов считается некорректным и программист должен исправить данный код.

Ниже представлен результат применения этих правил.

```
// Пример 7-1, продолжение: разрешение перегрузки
//
bool    b;
int     i;
double  d;
f( b );          // Вызов (б) с T = bool
f( i, 42, d );   // Вызов (в) с T = int
f( &i );         // Вызов (г) с T = int
f( i );          // Вызов (д)
f( d );          // Вызов (е)
```

До сих пор я сознательно выбирал простейшие случаи; теперь можно перейти к более сложным вариантам.

Пример Димова-Абрамса

Рассмотрим следующий код.

```
// Пример 7-2(а): явная специализация
//
template<class T>    // (а) первичный шаблон
void f( T );

template<class T>    // (б) первичный шаблон, перегружающий
void f( T* );        // (а) - шаблон функции не может быть
                     // частично специализирован, возможна
                     // только перегрузка

template<>           // (в) явная специализация (б)
void f<int>(int*);

// ...
int *p;
f( p );              // Вызов (в)
```

Вызов в последней строке в примере 7-2(а) именно такой, как мы и ожидали. Вопрос в том, *почему* мы ожидали именно такой результат. Если это ожидание — следствие неправильного представления, то следующая информация может неприятно вас удивить. "Вряд ли, — можете сказать вы, — Я написал специализацию для указателя на int, так что очевидно, что именно она и должна быть вызвана". И будете совершенно неправы.

Обратимся ко второму вопросу, представив его так, как было предложено Питером Димовым (Peter Dimov) и Дэвидом Абрамсом (David Abrahams).

2. Какая из версий функции f будет вызвана в последней строке приведенного кода? Почему?

```
// Пример 7-2(б): пример Димова-Абрамса
//
template<class T>
void f( T );
```

```
template<>
void f<int*>( int* );

template<class T>
void f( T* );
// ...
int *p;
f( p );         // Какая функция будет вызвана?
```

Ответ в данном случае — ... третья функция `f`. Давайте еще раз рассмотрим представленный код, комментируя его так, как это было сделано в примере 7-2(a), чтобы сравнить и противопоставить эти два примера.

```
template<class T>      // (a) такой же обычный первичный
void f( T );           // шаблон, как и ранее

template<>             // (в) явная специализация - на этот
void f<int*>( int* ); // раз это явная специализация (а)

template<class T>      // (б) второй первичный шаблон,
void f( T* );          // перегружающий (а)

// ...
int *p;
f( p );                // Вызов (б)! Разрешение перегрузки
                       // игнорирует специализацию и работает
                       // только с базовыми шаблонами
                       // функций.
```

Если изложенное удивляет вас — вы не одиноки в своем удивлении. В свое время это удивляло немалое количество экспертов. Ключ к пониманию приведенного кода прост, и он состоит в том, что *специализации не перегружают функции.*

Перегружаться могут только первичные шаблоны (конечно, наряду с обычными нешаблонными функциями). Рассмотрим еще раз вторую часть правил разрешения перегрузки, но на этот раз в них будут особо акцентированы некоторые моменты.

- Если такой "привилегированной" функции нет, в качестве претендентов рассматриваются *первичные шаблоны функций.* То, какой именно *первичный шаблон* будет выбран, зависит от того обстоятельства, какой из них в наибольшей степени соответствует аргументам функции и является "наиболее специализированным" [...]
  - Очевидно, что если имеется только один "наиболее специализированный" *первичный шаблон функции*, то используется именно он. Если *первичный шаблон* специализирован для используемых типов, то будет использоваться именно эта специализация; в противном случае будет выполнено инстанцирование первичного шаблона с корректными типами.

Компилятор выбирает только первичный шаблон (или нешаблонную функцию, если таковая имеется). И только после того, как первичный шаблон выбран, компилятор начинает анализировать, есть ли подходящая специализация выбранного шаблона, и если находит таковую, то использует ее.

➢ Рекомендация

Стоит запомнить, что специализации шаблона функции не участвуют в разрешении перегрузки. Специализация используется только тогда, когда первичный шаблон функции уже выбран, причем на этот выбор не влияет наличие или отсутствие специализации шаблона.

Мораль сей басни такова...

Если ваша логика подобна моей, то первый ваш вопрос после этого будет таким: "Но ведь я написал конкретную специализацию для случая, когда параметр функции имеет тип `int*`, и этот `int*` совершенно точно соответствует типу аргумента в вызове функции — так почему бы не использовать именно эту специализацию?" Увы, здесь вы ошибаетесь. Если вы хотите гарантировать вызов вашей функции в случае точного соответствия, вы должны сделать ее обычной нешаблонной функцией, а не специализацией шаблона.

Объяснение, почему специализации не участвуют в перегрузке, очень простое. Комитет по стандарту указал, что *было бы удивительно, если бы только из-за того, что вы написали специализацию для некоторого шаблона, изменялся выбор используемого шаблона*. Это объяснение вкупе с наличием способа, гарантирующего использование нашей версии функции (достаточно сделать ее не специализацией, а обычной функцией), позволяет нам более точно и ясно понять, почему специализации не влияют на выбор шаблонов.

➢ Рекомендация

Мораль №1. Если вы хотите настроить первичный шаблон функции так, чтобы при разрешении перегрузки (или всегда в случае точного соответствия типов) использовалась ваша функция, делайте ее не специализацией, а обычной нешаблонной функцией.

Следствие. Если вы используете перегрузку шаблона функции, избегайте его специализаций.

Но что если вы один из тех, кто не только использует, но и пишет шаблоны функций? Можете ли вы найти лучшее решение и заранее избежать этой и других проблем как для себя, так и для ваших пользователей? Да, можете.

```
// Пример 7-2(в): иллюстрация к морали №2
//
template<class T>
struct FImpl;

template<class T>
void f(T t) {FImpl<T>::f( t );} // Руками не трогать! :)

template<class T>
struct FImpl {
  static void f( T t );         // Специализируйте здесь
};
```

➢ Рекомендация

Мораль №2. Если вы пишете первичный шаблон функции, который с большой вероятностью будет специализирован, лучше писать его как отдельный шаблон функции, который никогда не специализируется и не перегружается и будет просто перенаправлять вызов шаблону класса со статической функцией с той же сигнатурой. Таким образом, все пользователи получают возможность специализировать этот класс как полностью, так и частично, никак не влияя при этом на разрешение перегрузки.

Резюме

Перегрузка шаблонов функций — вполне корректная вещь. Разрешение перегрузки рассматривает все первичные шаблоны функций в качестве равных претендентов,

так что здесь вполне применим ваш опыт работы с перегрузкой обычных функций C++. Компилятор рассматривает все видимые шаблоны функций и выбирает из них наиболее подходящий.

Специализация шаблонов функций существенно менее интуитивна. Во-первых, вы не можете частично их специализировать — вместо этого вы должны использовать перегрузку. Во-вторых, *специализации шаблонов функций не перегружаются*. Это означает, что любые специализации, написанные вами, не влияют на выбор используемого шаблона, что противоречит интуитивным ожиданиям большинства программистов. В конце концов, если вместо специализации вы напишете обычную функцию с той же сигнатурой, то при разрешении перегрузки будет использована именно она, поскольку обычная функция всегда имеет преимущество перед шаблоном.

Если вы пишете шаблон функции, то лучше писать его как шаблон функции, никогда не специализируемый и не перегружаемый, и реализовать посредством шаблона класса. Этот своеобразный уровень косвенности позволит вам избежать ограничений и "темных закутков" шаблонов функций, а программисты, использующие ваш шаблон, смогут как угодно — полностью ли, частично ли — специализировать *шаблон класса*, никак не влияя при этом на работу шаблона функции. Таким образом вы обходите как запрет на частичную специализацию шаблона функции, так и невозможность (иногда неожиданную) перегрузки специализации шаблона функции. Проблема решена.

Если вы используете обычный шаблон функции (не реализованный через шаблон класса), то для того, чтобы ваша специализированная версия данной функции принимала участие в перегрузке, вы должны сделать ее не специализацией шаблона, а обычной нешаблонной функцией с той же сигнатурой.

Задача 8. Дружественные шаблоны Сложность: 4

Если вы захотите объявить специализацию шаблона функции в качестве друга, как вы поступите? Согласно стандарту C++, вы должны воспользоваться одним из двух синтаксически корректных способов. В мире же реальных компиляторов один способ поддерживается слабо, зато второй работает со всеми текущими версиями распространенных компиляторов... за исключением одного.

Допустим, у нас есть шаблон функции, которая делает нечто с объектом. В частности, рассмотрим шаблон функции `boost::checked_delete` из [Boost], который удаляет переданный ему объект; среди прочего, этот шаблон вызывает деструктор объекта.

```
namespace boost {
    template<typename T> void checked_delete( T* x ) {
        // ... некоторые действия ...
        delete x;
    }
}
```

Допустим, что теперь вы хотите использовать этот шаблон с классом, у которого интересующая операция (в случае шаблона `boost::checked_delete` — деструктор) оказывается закрытой.

```
class Test {
    ~Test() { }              // Закрытая (private) функция!
};
Test* t = new Test;
boost::checked_delete( t ); // Ошибка: деструктор класса
                            // Test закрытый, поэтому он не
                            // может быть вызван в шаблоне
                            // checked_delete
```

Решение простое: надо сделать `checked_delete` другом класса `Test`. (Единственная альтернатива заключается в том, чтобы сделать деструктор класса `Test` открытым.) Что может быть проще?

И в самом деле, в стандарте C++ предусмотрено два законных и простых способа сделать это. Если, конечно, ваш компилятор будет не против...

Вопрос для новичка

1. Укажите очевидный, удовлетворяющий стандарту синтаксис объявления `boost::checked_delete` другом класса `Test`.

Вопрос для профессионала

2. Почему очевидный способ на практике оказывается ненадежным? Укажите более надежный вариант.

Решение

Эта задача — проверка на соответствие теории и практики. Сделать другом шаблон из другого пространства имен — это гораздо легче сказать (в стандарте), чем сделать (с использованием реального компилятора).

В связи с этим у меня для вас есть одна хорошая новость, одна плохая и, чтобы подбодрить вас, еще одна хорошая новость.

- Хорошая новость: существует два отличных стандартных способа выполнить поставленную задачу, причем их синтаксис вполне естественен.

- Плохая новость: ни один из них не работает на всех современных компиляторах. Даже известные как наиболее соответствующие стандарту компиляторы не позволят вам воспользоваться как минимум одним, а то и обоими стандартными способами.
- Хорошая новость: один из способов работает на всех проверенных мною компиляторах — кроме gcc.

Итак, приступим.

Исходная попытка

1. Укажите очевидный, удовлетворяющий стандарту синтаксис объявления `boost::checked_delete` другом класса `Test`.

Эта задача возникла как следствие вопроса, который задал в Usenet Стефан Борн (Stephan Born), когда пытался добиться того же, чего хотим добиться и мы. Его проблема заключалась в том, что когда он пытался сделать специализацию `boost::checked_delete` другом своего класса `Test`, код не работал на его компиляторе.

Ниже представлен его исходный текст.

```
// Пример 8-1: попытка добиться дружбы
//
class Test {
    ~Test() { }
    friend void boost::checked_delete( Test* x );
};
```

Увы, этот код не работал не только на компиляторе автора вопроса, но и на ряде других компиляторов. Коротко говоря, объявление в примере 8-1 обладает следующими характеристиками:

- оно соответствует стандарту, но полагается на "темный угол" языка;
- оно отвергается многими компиляторами, в том числе очень хорошими;
- его легко исправить таким образом, чтобы он перестал зависеть от "темных углов" и работал практически на всех современных компиляторах (кроме gcc).

Я готов приступить к пояснению четырех способов объявления друзей в C++. Это просто. Я также хочу показать вам, как поступают реальные компиляторы, и завершить рассмотрение этого вопроса написанием наиболее переносимого кода.

В "темных углах"

2. Почему очевидный способ на практике оказывается ненадежным? Укажите более надежный вариант.

При объявлении друзей имеется четыре возможности (перечисленные в [C++03], §14.5.3). Они сводятся к следующему.

Когда вы объявляете друга, не пользуясь ключевым словом `template`:

1. Если имя друга выглядит как имя специализации шаблона с явными аргументами (например, `Name<SomeType>`),

   то другом является указанная специализация шаблона.

2. Иначе, если имя друга квалифицировано с использованием имени класса или пространства имен (например `Some::Name`) *И* этот класс или пространство имен содержит подходящую нешаблонную функцию,

   то другом является эта функция.

3. Иначе, если имя друга квалифицировано с использованием имени класса или пространства имен (например `Some::Name`) *И* этот класс или пространство имен содержит подходящий шаблон функции (с выводимыми аргументами шаблона)

   то другом является специализация этого шаблона функции.

4. В противном случае имя должно быть неквалифицированным и объявляет (возможно, повторно) обычную (нешаблонную) функцию.

Понятно, что второй и четвертый пункты соответствуют нешаблонным сущностям, так что для объявления специализации шаблона в качестве друга у нас есть только два варианта: либо подогнать свой код к случаю 1, либо — к случаю 3. В нашем случае это выглядит следующим образом.

```
// Исходный текст корректен, так как соответствует случаю 3
friend void boost::checked_delete( Test* x );
```

или

```
// Добавляем "<Test>"; код при этом остается корректен, но
// соответствует случаю 1
friend void boost::checked_delete<Test>( Test* x );
```

Первый вариант по сути представляет собой сокращение второго... но только если имя квалифицировано (в нашем случае — использованием `boost::`) и в указанной области видимости нет подходящих нешаблонных функций. Хотя оба объявления корректны, первое использует то, что я называю "темными углами" языка, в них часто путаются не только программисты, но и компиляторы. Я могу назвать по крайней мере три причины, по которым следует избегать объявления такого вида, несмотря на его техническую корректность.

Причина 1: не всегда работает

Как уже упоминалось, синтаксис, используемый в третьем правиле, по сути представляет собой сокращение синтаксиса, используемого в первом правиле; в нем опускаются явные аргументы шаблона в угловых скобках. Однако такое сокращение работает только тогда, когда имя квалифицировано и указан класс или пространство имен, которые не содержат подходящей нешаблонной функции.

В частности, если пространство имен содержит (или получит позже) подходящую нешаблонную функцию, то будет выбрана именно она, поскольку ее наличие означает, что реализуется случай 2, а не 3. Достаточно тонко и неожиданно, не правда ли? Здесь очень легко допустить ошибку, так что лучше избегать использования таких тонкостей.

Причина 2: удивляет программистов

Случай 3 оказывается неожиданным и удивительным для программистов, которые пытаются разобраться в коде и понять, как он работает. Рассмотрим, например, весьма незначительно отличающийся вариант кода — все отличие состоит в том, что я убрал квалифицирующую часть `boost::`.

```
// Имя стало неквалифицированным, а это означает нечто
// совершенно иное
//
class Test {
    ~Test() { }
    friend void checked_delete( Test* x );
};
```

Если вы опустите `boost::` (т.е. сделаете вызов неквалифицированным), то оказывается, что эта ситуация соответствует совсем другому случаю, а именно — случаю 4, который вообще не работает для шаблонов. Держу пари, что любой программист

согласится со мной, что такое кардинальное изменение смысла объявления друга из-за опускания имени пространства имен, по меньшей мере, неожиданно. Таких конструкций следует избегать.

Причина 3: удивляет компиляторы

Случай 3 оказывается неожиданным и удивительным для компиляторов, он может оказаться неприменимым на практике, даже если мы проигнорируем приведенные ранее причины отказаться от данного способа.

Испытаем коды, относящиеся к случаям 1 и 3, на разных компиляторах, и проанализируем результат. Воспринимают ли компиляторы стандарт так же, как и мы (дочитавшие книгу до этого места)? Будут ли, по крайней мере, самые мощные компиляторы вести себя так, как мы от них ожидаем? Оба ответа отрицательны...

Сначала обратимся к случаю 3.

```
// Пример 8-1 - еще раз
//
namespace boost {
    template<typename T> void checked_delete( T* x ) {
        // ... остальной код ...
        delete x;
    }
}
class Test {
    ~Test() { }
    // изначальный код
    friend void boost::checked_delete( Test* x );
};

int main() {
    boost::checked_delete( new Test );
}
```

Попробуйте скомпилировать этот код на вашем компиляторе, и мы сравним наши результаты (см. табл. 8.1).

Таблица 8.1. Результат компиляции примера 8-1 разными компиляторами

Компилятор	Результат	Сообщение об ошибке
Borland 5.5	OK	
Comeau 4.3.0.1	OK	
Digital Mars 8.38	Error	Symbol Undefined ?checked_delete@@YAXPAV-Test@@@Z (void cdecl checked_delete(Test *))
EDG 3.0.1	OK	
Intel 6.0.1	OK	
gcc 2.95.3	Error	`boost::checked_delete(Test *)' should have been declared inside `boost'
gcc 3.4	Error	`void boost::checked_delete(Test*)' should have been declared inside `boost'
Metrowerks 8.2	Error	friend void boost::checked_delete(Test* x); name has not been declared in namespace/class
MS VC++ 6.0	Error	nonexistent function 'boost::checked_delete' specified as friend
MS VC++ 7.0 (2002)	OK	

Окончание табл. 8.1

Компилятор	Результат	Сообщение об ошибке
MS VC++ 7.1 (2003)	Error	'boost::checked_delete' : not a function
MS VC++ 8.0 (2005) beta	OK	

Итак, наш тест показывает, что этот синтаксис плохо распознается современными компиляторами, один из них даже неоднократно меняет свое мнение по поводу данного кода в процессе разработки новых версий. Кстати, нас не должно удивлять, что компиляторы Comeau, EDG и Intel скомпилировали этот код, поскольку все они основаны на реализации EDG C++. Получается, что из пяти различных реализаций языка три (Digital Mars, gcc и Metrowerks) не понимают такой синтаксис, два других (Borland и EDG) понимают, и еще один никак не может определиться (Microsoft).

Теперь изменим исходный текст так, чтобы он относился к случаю 1.

```
// Пример 8-2: другой способ объявления друга
//
namespace boost {
    template<typename T> void checked_delete( T* x ) {
        // ... остальной код ...
        delete x;
    }
}
class Test {
    ~Test() { }
    // Альтернативный способ
    friend void boost::checked_delete<> ( Test* x );
};

int main() {
    boost::checked_delete( new Test );
}
```

Можно использовать и эквивалентную форму записи

```
friend void boost::checked_delete<Test>( Test* x );
```

Результат исследований более оптимистичен — такой код понятен уже для большего количества компиляторов (см. табл. 8-2.).

Таблица 8.2. Результат компиляции примера 8-1 разными компиляторами

Компилятор	Результат	Сообщение об ошибке
Borland 5.5	OK	
Comeau 4.3.0.1	OK	
Digital Mars 8.38	OK	
EDG 3.0.1	OK	
Intel 6.0.1	OK	
gcc 2.95.3	Error	`boost::checked_delete(Test *)' should have been declared inside `boost'
gcc 3.1.1	Error	`void boost::checked_delete(Test*)' should have been declared inside `boost'
gcc 3.4	Error	`void boost::checked_delete(Test*)' should have been declared inside `boost'

Окончание табл. 8.2

Компилятор	Результат	Сообщение об ошибке
Metrowerks 8.2	OK	
MS VC++ 6.0	Error	nonexistent function 'boost::checked_delete' specified as friend
MS VC++ 7.0 (2002)	OK	
MS VC++ 7.1 (2003)	OK	
MS VC++ 8.0 (2005) beta	OK	

Случай 1 выглядит более привлекательно и безопасно — он работает почти на всех современных компиляторах, кроме gcc.

Отступление: проблема в пространстве имен

Заметим, что если бы интересующий нас шаблон функции не находился в другом пространстве имен, то мы бы могли спокойно использовать случай 1 практически на всех современных компиляторах.

```
// Пример 8-3: checked_delete не в пространстве имен
//

// Больше нет boost::
template<typename T> void checked_delete( T* x ) {
    // ... прочий код ...
    delete x;
}

// "boost:" больше не нужен
class Test {
    friend void checked_delete<Test>( Test* x );
};
int main() {
    checked_delete( new Test );
}
```

Результаты исследования приведены в табл. 8.3.

Таблица 8.3. Результат компиляции примера 8-3 разными компиляторами

Компилятор	Результат	Сообщение об ошибке
Borland 5.5	OK	
Comeau 4.3.0.1	OK	
Digital Mars 8.38	OK	
EDG 3.0.1	OK	
Intel 6.0.1	OK	
gcc 2.95.3	OK	
gcc 3.4	OK	
Metrowerks 8.2	OK	
MS VC++ 6.0	Error	syntax error (just can't handle it)

Окончание табл. 8.3

Компилятор	Результат	Сообщение об ошибке
MS VC++ 7.0 (2002)	Error	friend declaration incorrectly interpreted as declaring a brand-new (and undefined) ordinary nontemplate function, even though we used template syntax
MS VC++ 7.1 (2003)	OK	
MS VC++ 8.0 (2005) beta	OK	

Итак, проблема для большинства компиляторов, которые не могут скомпилировать пример 8-1, заключается в том, что в нем специализация шаблона функции объявляется *в другом пространстве имен.*

Два неверных обходных пути

Когда этот вопрос только появился в Usenet, некоторые предлагали использовать объявление `using` (или, что то же самое, директиву `using`) и сделать объявление друга неквалифицированным.

```
namespace boost {
    template<typename T> void checked_delete( T* x ) {
        // ... остальной код ...
        delete x;
    }
}
using boost::checked_delete;
// или "using namespace boost;"
class Test {
    ~Test() { }
    // Не специализация шаблона!
    friend void checked_delete( Test* x );
};
```

Это объявление друга относится к случаю 4: "4. В противном случае имя должно быть неквалифицированным и объявляет (возможно, повторно) обычную (нешаблонную) функцию." В действительности у нас получилось объявление новой нешаблонной функции `::checked_delete(Test*)` в объемлющем пространстве имен.

Если вы попытаетесь использовать этот код, многие из рассматривавшихся компиляторов отвергнут его, сообщая, что функция `checked_delete` не определена, и все они сообщат об ошибке, если вы попытаетесь использовать отношение дружбы и поместить вызов закрытого члена в шаблон `boost::checked_delete`.

И наконец, один эксперт предложил немного изменить рассмотренный код — используя одновременно `using` и синтаксис шаблона `<>`.

```
namespace boost {
    template<typename T> void checked_delete( T* x ) {
        // ... остальной код ...
        delete x;
    }
}
using boost::checked_delete;
// или "using namespace boost;"
class Test {
    ~Test() { }
    friend void checked_delete<>( Test* x );  // Корректно?
};
```

Вероятно, этот код не совсем корректен: в стандарте нет указания на то, является ли корректным такое изменение, так что вопрос остается открытым и комитет по стандартизации языка еще не принял окончательного решения по этому вопросу. Скорее всего, этот код все же будет признан некорректным, а пока все компиляторы, с которыми я имел дело, отвергли его.

Почему этот код некорректен? Будем последовательны. Директива `using` существует для того, чтобы облегчить *использование* (use) имен — для вызова функций и применения имен типов в объявлениях переменных и параметров. Объявления должны подчиняться другим правилам. Аналогично тому, как вы должны объявлять специализацию шаблона в том же пространстве имен, что и сам шаблон (это нельзя делать в другом пространстве имен посредством `using`), вы должны и объявлять специализацию шаблона другом при помощи указания исходного пространства имен шаблона (без всяких `using`).

Резюме

Для объявления специализации шаблона функции в качестве друга вы можете использовать один из двух способов.

```
// Из примера 8-1
friend void boost::checked_delete ( Test* x );
// Из примера 8-2: добавление <> или <Test>
friend void boost::checked_delete<>( Test* x ); // или "<Test>"
```

В данной задаче продемонстрировано, что цена сокращения записи путем отбрасывания <> или <Test> слишком высока — это существенная потеря переносимости.

➢ Рекомендация

Говорите то, что думаете. Указывайте, чего именно вы хотите добиться. Будьте точны. Если вы говорите о шаблоне и возникает вопрос, чего именно вы хотите добиться, — включите список (возможно, пустой) параметров шаблона.

Избегайте "темных углов" языка программирования, включая конструкции, которые несмотря на свою корректность склонны вводить в заблуждение программистов и даже компиляторы.

При объявлении специализации шаблона функции другом явно указывайте, как минимум, пустые угловые скобки <>, например:

```
namespace boost {
    template<typename T> void checked_delete( T* x );
}
class Test {
    friend void boost::checked_delete  (Test*x); // Плохо
    friend void boost::checked_delete<>(Test*x); // Хорошо
};
```

Если ваш компилятор пока что не позволяет вам использовать ни один из этих способов для объявления отношения дружбы, сделайте необходимую функцию(и) открытой[16] — добавив при этом комментарий, почему именно это сделано, и не забудьте вернуть все на место, когда новая версия вашего компилятора позволит вам справиться с рассмотренным синтаксисом.

---

[16] Имеются и другие обходные пути, но уж очень неприятные и длинные. Например, можно создать прокси-класс внутри пространства имен `boost` и сделать другом его.

Задача 9. Ограничения экспорта. Часть 1: основы

Сложность: 7

Разберемся с `export` — что о нем думают, что это такое на самом деле и почему все так старательно игнорируют эту возможность.

Вопрос для новичка

1. Что подразумевается под моделью включения для шаблонов?

Вопрос для профессионала

2. Что подразумевается под моделью разделения для шаблонов?
3. В чем заключаются основные недостатки модели включения:
   а) для обычных функций?
   б) для шаблонов?
4. Как можно преодолеть недостатки из вопроса 3 при помощи стандартной модели разделения C++:
   а) для обычных функций?
   б) для шаблонов?

Решение

Стандартная возможность экспорта шаблонов зачастую оказывается неверно понятой, так что данная и следующая задачи призваны исправить ситуацию и внести ясность в этот вопрос.

На момент написания этого материала возможность экспорта поддерживалась только в одном коммерческом компиляторе. Компилятор Comeau[17], выпущенный в 2002 году и основанный на реализации C++ Edison Design Group (EDG)[18], был первым (и пока единственным) компилятором с поддержкой экспорта. Поэтому не удивительно, что у программистов мало опыта по применению экспорта в реальных проектах; впрочем, ситуация должна кардинально измениться с появлением других компиляторов с поддержкой экспорта.

Вот о чем мы поговорим в этой и следующей задачах.

- Что такое экспорт и как следует его использовать.
- Задачи, для которых предназначается экспорт.
- Текущее состояние проблемы.
- Как экспорт влияет (зачастую неочевидно) на другие (на первый взгляд, никак не связанные с экспортом) части языка C++.
- Советы по эффективному использованию экспорта.

Рассказ о двух моделях

Стандарт C++ поддерживает две различные модели организации исходного текста шаблонов: старую, давно используемую модель включения и относительно новую модель разделения.

[17] `www.comeaucomputing.com`.
[18] `www.edg.com`.

1. Что подразумевается под моделью включения для шаблонов?

В *модели включения* (inclusion model) код шаблона с точки зрения исходного текста выглядит так же, как и встраиваемый (inline) (хотя в действительности код шаблона не обязательно должен быть встраиваемым). Весь исходный текст шаблона должен быть видимым для любого кода, который использует этот шаблон. Такая модель называется моделью включения, поскольку обычно при этом для включения заголовочных файлов с определениями шаблонов используются директивы `#include`[19].

Если вы знакомы с современными шаблонами C++, то вы знакомы и с моделью включения. Это единственная реально используемая модель, хотя бы просто потому, что в настоящее время компиляторы C++ поддерживают только ее. Все шаблоны, с которыми вы сталкивались в реальных программах, книгах и статьях до этого времени, относятся к категории модели включения.

2. Что подразумевается под моделью разделения для шаблонов?

С другой стороны, *модель разделения* (separation model) предназначается для того, чтобы позволить "разделить" компиляцию шаблонов (кавычки в данном случае использованы не случайно). В модели разделения определения шаблонов не обязательно должны быть видимы для вызывающих функций. Так и хочется добавить — "как и определения обычных функций", но это было бы не верно — несмотря на определенную схожесть, это принципиально разные вещи, как мы вскоре убедимся. Модель разделения относительно нова — она была добавлена в стандарт в средине 1990-х годов, но первая коммерческая реализация (EDG) появилась только летом 2002 года[20].

Самое главное при рассмотрении моделей включения и разделения — это понимать и помнить, что это разные модели *организации исходного текста* программы. Это не разные модели инстанцирования шаблонов, т.е. в любом случае компилятор выполняет одну и ту же работу по настройке шаблонов для конкретных аргументов. Это важно, поскольку именно в этом заключаются причины определенных ограничений экспорта (которые зачастую оказываются неожиданными для программистов, в особенности тех, кто прибегает к экспорту, полагая, что это позволит ускорить процесс сборки программы — как в случае раздельной компиляции обычных функций). При использовании любой модели компилятор вправе выполнить оптимизацию, в частности, основанную на правиле одного определения (One Definition Rule — ODR), инстанцируя шаблоны для каждой уникальной комбинации аргументов шаблонов только по одному разу, независимо от того, сколько раз и где эта комбинация встречается в вашей программе. Разработчики компиляторов имеют возможность реализовать такую оптимизацию и стратегию инстанцирования независимо от того, какая именно модель — включения или разделения — используется для физической организации исходных текстов шаблонов. Хотя для модели разделения возможность такого рода оптимизации очевидна, то же самое можно осуществить и при использовании модели включения.

Пояснение на примере

3. В чем заключаются основные недостатки модели включения:

а) для обычных функций?

б) для шаблонов?

---

[19] Или, что по сути то же самое, определения размещаются в отдельном `.cpp`-файле, который в свою очередь включается в заголовочный `.h`-файл.

[20] Заметим, что Cfront имел похожую функциональность за десятилетие до этого. Однако реализация Cfront была медленной и основывалась на эвристике, которая при проблемах, связанных с шаблонами, просто сбрасывала кэш готовых настроенных шаблонов и начинала инстанцирование всех шаблонов с нуля.

Рассмотрим пример кода шаблона функции как в модели включения, так и в модели разделения. Для сравнения я также привожу обычную функцию — как встраиваемую и в обычном варианте раздельной компиляции: это поможет нам понять отличия между раздельной компиляцией обычных функций и раздельной моделью шаблонов функций. Это совершенно разные вещи, несмотря на то, что в их названиях использовано одно и то же слово "раздельный". Именно по этой причине я брал слово "раздельный" в кавычки.

Рассмотрим следующий код с обычной встраиваемой функцией и шаблоном функции в модели включения.

```
// Пример 9-3(a): встраиваемая функция
//
// --- Файл f.h, предоставляемый пользователю ---
namespace MyLib {
    inline void f( int ) {
        // Изящный код, воплотивший в себя годы работы;
        // использует вспомогательные классы и функции
    }
}
```

Ниже приведена демонстрация использования модели включения для шаблонов:

```
// Пример 9-3(б): простенький шаблон, использующий модель
//                включения
//
// --- Файл g.h, предоставляемый пользователю ---
namespace MyLib {
    template<typename T>
    void g( T& ) {
        // Изящный код - результат многих лет работы;
        // использует вспомогательные классы и функции,
        // которые не обязательно объявлены как
        // встраиваемые, но их код располагается здесь же,
        // в этом же файле
    }
}
```

В обоих случаях код примера 9-3 вызывает вопросы, знакомые для программистов на C++.

- *Открытие исходного текста определений.* Теперь определения функций `f` и `g` (которые, возможно, представляют предмет патентного права или имеют другие причины быть скрытыми) становятся доступны любому программисту. Само по себе это может и не быть такой уж большой неприятностью, но об этом несколько позже.
- *Зависимости исходного текста.* Все функции, вызывающие `f` или `g`, зависят от деталей их внутреннего устройства, так что при любом внесении изменений в `f` или `g` требуется перекомпиляция всего кода, который к ним обращается. Кроме того, если в теле `f` или `g` используются типы, которые не упоминаются в объявлениях этих функций, то в результате вызывающие их функции также требуют доступа к полным определениям этих типов.

Использование экспорта

В состоянии ли мы решить (или хотя бы уменьшить) эти проблемы?

4. Как можно преодолеть недостатки из вопроса 3 при помощи стандартной модели разделения C++:

а) **для обычных функций?**

Для обычной функции можно ответить "легко", так как оба недостатка преодолеваются путем применения раздельной компиляции.

```
// Пример 9-4(а): раздельная компиляция функции
//
// --- Файл f.h, предоставляемый пользователю ---
namespace MyLib {
    void f( int );
}

// --- Файл f.cpp, может не быть предоставлен ---
namespace MyLib {
    void f( int ) {
        // Изящный код - результат многих лет работы;
        // использует вспомогательные классы и функции;
        // компилируется отдельно
    }
}
```

Нет ничего неожиданного в том, что данный подход решает обе проблемы, как минимум, для функций. (Та же идея может быть применена и для целых классов — о применении идиомы Pimpl вы можете прочесть в книге [Sutter00].)

- *Исходный текст определений скрыт.* Мы можем поставлять пользователям исходный текст определений, если хотим этого, но мы не обязаны это делать. Заметим, что многие популярные библиотеки поставляются с исходным текстом (возможно, за отдельную плату), причем так поступают даже производители, которые строго следят за своими имущественными правами. Исходные тексты могут потребоваться пользователям, например, для отладочных целей или по каким-либо иным причинам.
- *Отсутствие зависимостей исходного кода.* Вызывающая функция больше не зависит от деталей внутреннего устройства функции `f`, так что при ее изменении не требуется полная перекомпиляция, достаточно перекомпоновки, что зачастую на порядок (а то и более) быстрее. Кроме того (что, правда, обычно не столь сильно влияет на время сборки приложения), вызывающая `f` функция больше не зависит от типов, использованных только в теле функции `f`.

Все это хорошо известно и хорошо работает для функций. Программисты знакомы с этим методом еще со времен C, и даже еще раньше, т.е. уже много-много лет...

К настоящему же вопросу мы только подбираемся...Итак, а как обстоят дела

б) для шаблонов?

Идея, лежащая в основе экспортирования, заключается в том, чтобы получить нечто аналогичное, но для шаблонов. Некоторые могут наивно ожидать, что приведенный ниже код обладает теми же достоинствами, что и код из примера 9-4(а). Это абсолютно неоправданно. Но не расстраивайтесь, многие знатоки C++ заблуждаются точно так же.

```
// Пример 9-4(б): экспорт шаблона
//
// --- Файл g.h, предоставляемый пользователю ---
namespace MyLib {
    export template<typename T>
        void g( T& );
}
// --- Файл g.cpp, ?? предоставляемый пользователю?? ---
namespace MyLib {
    template<typename T>
    void g( T& ) {
        // Изящный код - результат многих лет работы;
        // использует вспомогательные классы и функции.
```

```
        // Теперь он "раздельно" компилируемый?
    }
}
```

Для многих оказывается неожиданным, что для шаблонов такой код не решает ни одну из указанных ранее проблем. Он может только несколько улучшить ситуацию с одной из них. Давайте еще раз рассмотрим эти проблемы.

Проблема первая: открытый исходный текст

Первая проблема остается нерешенной: исходный текст определений должен быть открыт.

Ничто в стандарте C++ не говорит (и ни из каких положений стандарта не вытекает), что вы можете не предоставлять пользователю полный исходный текст шаблона `g` только потому, что вы использовали ключевое слово `export`. На самом деле, в единственной реализации поддержки `export` компилятор требует, чтобы пользователю предоставлялось полное определение шаблона — т.е. полный исходный текст[21]. Одна из причин этого заключается в том, что компилятору C++ требуется полный контекст определения экспортируемого шаблона при его инстанцировании. Для лучшего понимания проанализируем, что, согласно стандарту C++, происходит при инстанцировании шаблона:

> *"[Зависимые] имена не связаны и их поиск выполняется в точке инстанцирования шаблона как в контексте определения шаблона, так и в контексте точки инстанцирования".* — [C++03, §14.6.2]

Зависимое имя — это имя, которое зависит от типа аргумента шаблона; они используются в большинстве полезных шаблонов. В точке инстанцирования или использования шаблона поиск зависимых имен осуществляется в двух местах. Их поиск должен выполняться в контексте инстанцирования, что достаточно просто, поскольку компилятор в этот момент обрабатывает именно эту часть программы. Но поиск этих имен должен *также* осуществляться в контексте определения шаблона, а это уже сложнее, поскольку для этого требуется не только знание полного определения шаблона, но и контекст этого определения в содержащем его файле, включая сигнатуры используемых функций и т.п. вещи, необходимые для выполнения разрешения перегрузки и других важных действий.

Рассмотрим пример 9-4(б) с точки зрения компилятора. Ваша библиотека содержит экспортируемый шаблон функции `g` с тщательно спрятанным вне заголовочного файла определением. Допустим, все хорошо, библиотека продана. Годом позже, в один прекрасный день библиотека используется в некотором пользовательском модуле `h.cpp`, где требуется инстанцирование `g<CustType>` для некоторого типа `CustType`, созданного этим утром... и что следует делать компилятору, чтобы сгенерировать объектный код? Компилятор должен, кроме прочего, выполнить просмотр определения `g` в вашем файле реализации. Как видите, экспорт не устраняет зависимости от определения шаблона, а всего лишь скрывает их.

Экспортированные шаблоны не являются "раздельно компилируемыми" в том смысле, который мы вкладываем в это понятие при работе с обычными функциями. В общем случае экспортируемые шаблоны не могут быть раздельно компилируемы в объектный код до их реального использования. Более того, до достижения точки использования шаблона мы не знаем даже, с какими типами аргументов будет инстанцирован

---

[21] При этом возникает обычный вопрос: а нельзя ли передавать зашифрованный исходный текст? Дело в том, что шифрование, при котором для дешифрования не требуется вмешательство пользователя (например, ввод пароля), легко взламывается. Некоторые компании пытались применять шифрование исходного кода, но быстро отказались от этой практики, поскольку реальная защита кода при этом не обеспечивается, зато очень сильно раздражает пользователей. Есть куда более хорошие методы защиты интеллектуальной собственности.

этот шаблон. Таким образом, экспортируемые шаблоны в лучшем случае "раздельно частично компилируемы" или "раздельно синтаксически анализируемы". Определения шаблонов должны быть реально скомпилированы при каждом инстанцировании (здесь есть определенная схожесть с библиотеками Java или .NET, в которых из байт-кода или IL может быть получено достаточно много информации об исходном тексте).

> **➢ Рекомендация**
>
> Запомните, что ключевое слово `export` не подразумевает настоящей раздельной компиляции, как это происходит в случае обычных нешаблонных функций.

Проблема вторая: зависимости и время построения

Вторая проблема тоже остается нерешенной: зависимости оказываются скрытыми, но при этом они никуда не исчезают.

Всякий раз при изменениях в теле шаблона компилятор должен заново инстанцировать все применения данного шаблона. Во время этого процесса единицы трансляции, использующие шаблон `g`, должны обрабатываться вместе с внутренней реализацией `g`, т.е. вместе с определением `g` и типов, использующихся только в теле `g`. Код шаблона будет полностью скомпилирован позже, когда станет известен контекст каждого инстанцирования.

> **➢ Рекомендация**
>
> Запомните, что ключевое слово `export` только скрывает зависимости, но не устраняет их.

Да, вызывающая функция больше не зависит *явно* от внутренних деталей `g`, поскольку определение `g` больше не вносится в единицу трансляции при помощи директивы `#include`; можно сказать, что зависимости скрыты на уровне чтения исходного кода человеком.

Однако на этом проблемы не заканчиваются, поскольку мы говорим не о тех зависимостях, которые может прочесть программист, а о тех, которые должен скомпилировать компилятор, а эти зависимости никуда не исчезают. Да, компилятор может не перекомпилировать все единицы трансляции, в которых используется данный шаблон, но он должен, как минимум, выполнить компиляцию тех модулей, которые используют шаблон с комбинациями аргументов, для которых шаблон не был использован и для которых инстанцирование должно быть выполнено "с нуля". Компилятор не в состоянии обеспечить истинную раздельную компиляцию и ограничиться только компоновкой имеющегося объектного кода.

Заметим, что компилятор может быть достаточно интеллектуальным для того, чтобы работать так же и в случае модели включения — т.е. не перестраивать все файлы, использующие данный шаблон, а ограничиться только необходимыми действиями для выполнения всех инстанцирований (если код организован так, как показано в примере 9-4(б), с тем лишь отличием, что удалено ключевое слово `export`, и в файл `g.h` добавлена директива `#include "g.cpp"`. Идея заключается в том, что компилятор может полагаться на правило одного определения, а не навязывать его, т.е. компилятор может просто полагать, что все инстанцирования с одинаковыми аргументами обязаны быть идентичны, вместо того чтобы выполнять все необходимые инстанцирования и убеждаться в том, что они действительно идентичны.

Кроме того, вспомните, что многие шаблоны используют другие шаблоны, и таким образом, компилятор должен выполнять каскадную перекомпиляцию таких шаблонов

(и их единиц трансляции), и этот рекурсивный процесс продолжается до тех пор, пока не будут выполнены все каскадные инстанцирования (реакция нормального программиста в такой ситуации: "Какое счастье, что я не занимаюсь реализацией поддержки экспорта в компиляторе!").

Как видите, даже при использовании ключевого слова `export` внесение изменений в экспортируемый шаблон не позволяет ограничиться перекомпоновкой приложения. В отличие от истинной раздельной компиляции функций, где построение приложения при внесении изменений в одну функцию существенно быстрее, чем полная его перекомпиляция, при использовании экспорта шаблонов сказать заранее, будет ли построение приложения быстрее или медленнее полной его перекомпиляции, в общем случае невозможно.

Резюме

Итак, теперь вы понимаете, почему невозможно добиться истинной "раздельной компиляции" шаблонов так же, как это делалось для нешаблонных функций. Многие программисты считают, что экспорт означает возможность поставки библиотеки шаблонов пользователю без полного исходного текста, и/или ускорение построения приложения. Ничего подобного `export` не обещает. Опыт говорит, что должен поставляться весь исходный текст или его прямой эквивалент и что скорость построения приложения оказывается такой же или меньшей, и очень редко — большей, поскольку зависимости оказываются только замаскированы, но не устранены, так что компилятору в общем случае требуется выполнить ту же (если не бо́льшую) работу, что и в модели включения.

В следующей задаче мы увидим, почему `export` усложняет C++ и его использование, вплоть до фундаментальных изменений смысла других конструкций языка, порой столь неожиданных, что их просто трудно предвидеть. В этой же задаче будут приведены некоторые советы по эффективному использованию возможностей экспорта (на тот случай, если вам когда-то доведется столкнуться с компилятором, поддерживающим данную возможность).

Задача 10. Ограничения экспорта. Часть 2: взаимосвязи, практичность и советы по использованию

Сложность: 9

Как `export` взаимодействует с другими возможностями языка C++ и как эффективно и безопасно его использовать?

Вопрос для новичка

1. Когда возможность экспорта появилась в стандарте C++ в ее современном виде? Когда она была впервые реализована?

Вопрос для профессионала

2. Поясните, каким образом экспорт изменяет смысл других конструкций C++?
3. Чем именно `export` мешает программисту?
4. В чем заключаются реальные и потенциальные преимущества `export`?

Решение

Перед вами вторая часть мини-серии. В предыдущей задаче мы рассмотрели следующие вопросы.

- Что такое `export` и для чего предназначена эта возможность C++? Мы провели анализ сходств и различий между моделями "включения" и "экспорта" исходного кода шаблонов, и выяснили, в чем состоит отличие этих моделей от встраиваемых и раздельно компилируемых функций и чем оно объясняется.
- В чем заключаются основные проблемы экспортирования и почему экспортирование работает не так, как ожидают от него программисты?

Обычно программисты ожидают от возможности экспортирования шаблонов истинной раздельной компиляции шаблонов так же, как и обычных нешаблонных функций. Основные надежды — на то, что экспорт позволит поставлять пользователям библиотеки шаблонов без полного исходного кода определений шаблонов (или их прямого эквивалента), а также что при его использовании увеличится скорость построения приложений. Как мы выяснили, ни одно из этих ожиданий ключевое слово `export` в применении к шаблонам не оправдывает.

Накопленный на сегодняшний день опыт работы с экспортом шаблонов говорит нам, что поставляться исходный текст (или его эквивалент) должен полностью, и что не известно, будет ли время построения приложения при использовании экспорта больше, меньше или останется таким же, как и без него. Почему? Главная причина в зависимостях, которые несмотря на всю маскировку никуда не удаляются, так что компилятор в общем случае должен выполнить как минимум то же количество работы, что и при использовании модели включения шаблонов. Коротко говоря, это ошибка (хотя и вполне естественная) — думать, что экспорт приводит к истинной раздельной компиляции шаблонов в том же смысле, что автор может поставлять только заголовочные файлы с объявлениями шаблонов и объектный код. Экспорт шаблонов скорее похож на библиотеки Java и .NET, в которых байт-код или IL может быть преобразован в нечто, напоминающее исходный текст. Код этих библиотек не является традиционным объектным кодом.

Здесь я хочу рассмотреть следующие вопросы.

- Текущее состояние экспорта шаблонов.

- Пути (зачастую неочевидные), которыми экспорт шаблонов приводит к изменению смысла других (казалось бы, никак не связанных с экспортом шаблонов) возможностей языка C++.
- Некоторые советы по эффективному использованию возможностей экспорта шаблонов, если вы когда-нибудь столкнетесь с компилятором, который поддерживает эту возможность.

Но сначала — краткий курс истории.

Начало: 1988–1996 гг.

1. Когда возможность экспорта появилась в стандарте C++ в ее современном виде? Когда она была впервые реализована?

Ответ на поставленный вопрос — 1996 и 2002 годы соответственно. (Если вам кажется, что дата первой реализации той или иной возможности по-хорошему должна предшествовать дате включения ее в стандарт, вы не одиноки, но экспорт шаблонов — не единственный пример, когда стандарт предвосхитил реализацию.)

Исходя из вышесказанного и вполне обоснованной критики `export`, можно начинать кампанию по побиванию камнями людей, которые окопались в комитете по стандарту C++ и принимают такие глупые решения. Но вряд ли стоит впадать в крайности и спешить с выводами. Давайте прислушаемся ко всем "за" и "против", прежде чем принимать решения.

Если экспорт не оправдал надежд такого большого количества людей, почему он вообще существует? Причина очень проста. В средине 1990-х годов большинство в комитете считало, что стандарт, в котором отсутствует раздельная компиляция шаблонов (которая давно имелась в C для функций), будет, как минимум, неполным, так что экспорт шаблонов оказался в стандарте из принципиальных соображений.

Вспомним также, что в 1995–1996 годах шаблоны были достаточно новой возможностью C++.

- Впервые шаблоны для C++ были предложены Бьярном Страуструпом в октябре 1988 года [Stroustrup88].
- В 1990 году Маргарет Эллис и Бьярн Страуструп опубликовали *The Annotated C++ Reference Manual* (ARM) [Ellis90]. В том же году приступил к работе комитет по стандартизации ISO/ANSI C++, выбрав ARM в качестве "отправной точки". Эта книга была первым справочником по C++, включавшим описание шаблонов. Это были не те шаблоны, которые известны нам сегодня. Полное описание этих простейших шаблонов занимало всего 10 страниц текста.

  В то время основной акцент делался на возможности использования параметризованных типов и функций, а основными примерами применения шаблонов были контейнер `List`, способный работать с объектами разных типов, и функция `sort`, которая могла сортировать последовательности разных типов. Кстати, даже тогда не исключалась возможность раздельной компиляции шаблонов. Так, Cfront (компилятор C++ Страуструпа) обладал определенной поддержкой "раздельной" компиляции простых шаблонов того времени; впрочем, использованный им подход не отвечал требованиям масштабируемости.
- В 1990–1996 годах разработчики компиляторов C++ работали над реализацией поддержки шаблонов в своих компиляторах и ее развитием, и в то же время комитет по стандарту C++ существенно расширил (и усложнил) семантику шаблонов. Достаточно упомянуть, что полное описание шаблонов в стандарте C++ занимает 133 страницы в 552-страничной книге [Vandevoorde03], посвященной шаблонам и их эффективному использованию (и которую я крайне рекомендую вам прочесть).

- В начале и средине 1990-х годов комитет по стандарту C++ пытался сделать шаблоны более интеллектуальными и практичными. Члены комитета оказались сами несколько удивлены чрезвычайной гибкостью и определенной "монстрообразностью" того, что получилось. Как известно, шаблоны представляют собой полный в смысле Тьюринга метаязык, который позволяет написать программу любой сложности, которая будет полностью выполнена на этапе компиляции. Современные технологии метапрограммирования с использованием шаблонов и дизайн современных библиотек оказались неожиданными для людей, которые дали программистам эти самые шаблоны. Упомянутые технологии были практически неизвестны в 1990–1996 годах. Они не использовались до 1994 года, когда Степанов впервые представил свою первую версию STL комитету. В 1995 году эта библиотека была воспринята как большое достижение — это сегодня STL "всего лишь" библиотека контейнеров и алгоритмов.

Вот почему я утверждаю, что в 1995–1996 годах шаблоны были достаточно новой возможностью C++. Современные шаблоны уже существовали почти в том же виде, что и сегодня, но даже их первооткрыватели не могли полностью представить, на что они способны. Сообщество C++ было в те годы существенно меньше сегодняшнего, только немногие компиляторы поддерживали шаблоны в объеме большем, чем было описано в ARM, а сама поддержка была слабой и по сути бесполезной. В качестве примера — в то время с первой версией STL смог справиться только один-единственный коммерческий компилятор.

Итак, все программистское сообщество в целом и комитет в частности имели только сравнительно небольшой опыт работы с шаблонами, причем в основном с более простыми шаблонами ARM. Шаблоны в 1996 году уже не были в зачаточном состоянии, но они все еще находились в стадии роста и формирования.

Как видим, комитет по стандарту C++ принял решение о включении экспорта шаблонов в стандарт еще на первых этапах развития, при ограниченном опыте работы с шаблонами.

1996 г.

Еще в 1996 году было достаточно информации, чтобы заставить нервничать множество экспертов и еще большее количество производителей компиляторов. Даже те, кто поддерживал экспорт, рассматривали его как необходимый компромисс, в то время как противники считали его источником сложности. Некоторые предлагали использовать раздельную компиляцию без применения специального ключевого слова.

В 1996 году в комитете наблюдалась скоординированная поддержка удаления понятия "раздельной" компиляции шаблонов из стандарта. В частности, утверждалось, что модель раздельной компиляции шаблонов никогда не была реализована в действительности, так что было не понятно, будет ли она работать так, как предполагается. Некоторые разработчики компиляторов C++ реализовали различные виды организации исходных кодов шаблонов, но среди них не было поддержки `export`; это была совершенно новая экспериментальная модель, за которой не было никакого реального опыта. Более того, в то время вниманию общественности было представлено несколько статей (которые ретроспективно можно назвать провидческими), в которых детально расписывались некоторые из потенциальных недостатков модели экспортирования, описанной в черновом варианте стандарта.

В частности, все производители компиляторов единодушно воспротивились включению модели раздельной компиляции в стандарт, поскольку, по их мнению, в тот момент нельзя было сказать, что из этого получится. У них была масса претензий к имеющимся редакциям этой части стандарта (как с ключевым словом `export`, так и без него), а кроме того, они не ощущали себя достаточно опытными в данном вопросе для того, чтобы реализовать экспорт шаблонов на практике или предложить

приемлемую альтернативу (не говоря уж о недостатке времени — планировалось выпустить окончательный вариант стандарта в следующем, т.е. 1997 году). Из-за всего этого производители компиляторов единогласно не хотели спешить с включением модели разделения в первый стандарт [C++98]. Вместо этого предлагалось "обкатать" идею как следует и подготовить ее к включению в следующий стандарт C++. Они не отказывались от идеи раздельной компиляции в принципе, но чувствовали, что в том виде, в котором ее предложено внести в стандарт, она еще сырована.

Тем не менее, они проиграли, и ключевое слово `export` применительно к шаблонам оказалось в стандарте[22]. Как я говорил ранее, большинство (незначительное) в комитете считало, что невозможно выпускать стандарт, в котором нет "раздельной" компиляции шаблонов, в то время как для обычных функций в C такая раздельная компиляция давно имеется и широко применяется. Несколько производителей компиляторов уже поэкспериментировали с различными видами "раздельной" компиляции шаблонов, и эта идея казалась, в принципе, обоснованной. Словом, `export` остался в стандарте из принципиальных соображений, и из этих же соображений не следует чернить его сверх меры.

Подчеркнем, что ведущие производители компиляторов были не против принципа раздельной компиляции шаблонов вообще, а против конкретной формулировки в стандарте языка. Они полагали, что следовало дать дополнительное время на анализ этого вопроса и принятие верного решения. Хотя некоторые из экспертов, которые в 1996 году голосовали за включение `export` в стандарт языка, сейчас считают это решение ошибочным, поставленные тогда цели были вполне правильными. Остается надеяться, что со временем они будут достигнуты. А пока что нам остается набираться опыта с помощью первого компилятора, который реализовал эту возможность (Comeau 4.3.01, 2002 год).

Опыт работы с экспортом

Единственный в мире производитель компилятора с поддержкой `export` для шаблонов — EDG — сообщил о том, что это наиболее сложная в реализации возможность C++, которая требует объема работы не меньшего, чем любые три другие возможности языка (например, пространства имен или шаблоны-члены классов). Потребовалось более трех человеко-лет работы только для кодирования и тестирования, не считая проектирования. Для сравнения — реализация языка Java теми же тремя программистами потребовала только два человеко-года.

Почему же поддержка `export` так сложна и трудно реализуема? Две основные причины можно сформулировать следующим образом.

1. *Экспорт базируется на поиске Кёнига.* Большинство компиляторов все еще не в состоянии реализовать корректный поиск Кёнига даже в пределах одной единицы трансляции (попросту говоря, в одном исходном файле). Экспорт же требует выполнения поиска Кёнига в нескольких единицах трансляции. (Дополнительную информацию о поиске Кёнига можно найти в [Sutter00].)
2. *Концептуально экспорт требует от компилятора одновременной работы с несколькими таблицами символов.* Инстанцирование экспортированного шаблона может вызвать каскадные инстанцирования в других единицах трансляции, и каждое инстанцирование должно иметь возможность обратиться к объектам, которые существуют (или "как бы существуют") при синтаксическом анализе определения шаблона. В C++ уже достаточно сложна работа только с одной таблицей символов,

[22] Обсуждение этого вопроса было бурным, и мнения членов комитета разделились. Так, в марте 1996 года голоса разделились как 2 к 1 против раздельной компиляции шаблонов; однако уже в июле того же года, когда ключевое слово `export` было внесено в стандарт, перевес голосов в пользу такого решения составил 2 к 1.

а в случае экспорта шаблонов приходится иметь дело с произвольным количеством таких таблиц.

До чего доводит экспорт

2. Поясните, каким образом экспорт изменяет смысл других конструкций C++?

Ключевое слово `export` порой неожиданным образом влияет на другие возможности языка. Многие из них никак не упомянуты в стандарте. В частности, `export` экспортирует не только шаблон, с которым употребляется.

- Если некоторые функции и объекты в безымянных пространствах имен используются в экспортируемых шаблонах, то теперь они должны быть доступны не только в пределах своих единиц трансляции. Аналогично, некоторые статические функции и объекты, используемые в экспортируемых шаблонах, которые ранее были видимы только в пределах своего файла, теперь должны иметь внешнее связывание (или, по крайней мере, вести себя так, как будто они им обладают). Это противоречит предназначению безымянных пространств имен и описания функций и объектов как статических, что должно делать их имена строго внутренними для соответствующих единиц трансляции. (Хотя статические функции и объекты объявлены устаревшей и нежелательной конструкцией языка, и вы должны вместо описания `static` использовать безымянные пространства имен, они остаются частью стандарта C++.)
- Разрешение перегрузки также должно быть способно выполнять разрешение с использованием имен из различных единиц трансляции — включая перегруженные имена из произвольного количества безымянных пространств имен. Главное преимущество размещения внутренних функций в безымянных пространствах имен (или объявление их статическими) состоит в их "скрытии", так что вы можете давать им простые имена, не беспокоясь о конфликтах имен или перегрузке при использовании нескольких исходных файлов. Теперь же эти функции могут участвовать в разрешении перегрузки при их использовании в экспортируемых шаблонах. Таким образом, для скрытия имен функций и объектов, используемых в экспортируемых шаблонах, недостаточно их размещения в безымянных пространствах имен, что, конечно же, противоречит изначальной идее безымянных пространств имен и статических объявлений.
- Возникают также новые разночтения и потенциальные нарушения правила одного определения. Например, класс может иметь несколько друзей в различных единицах трансляции, и в инстанцировании могут участвовать объявления этого класса из разных единиц трансляции. Если это так, то какой именно набор правил доступа следует применять? Это может показаться очень мелким вопросом, а многие подобные ошибки — безвредными, но последствия нарушения правила одного определения становятся все более существенными у ряда распространенных компиляторов (см. один из примеров в [Sutter02c]).

Трудность корректного использования

3. Чем именно `export` мешает программисту?

Корректно использовать экспортируемые шаблоны несколько сложнее, чем обычные. Вот три примера, иллюстрирующие это утверждение.

Пример 1. Становится проще, чем раньше, написать программу с трудно предсказуемым смыслом. Так же, как и в модели включения шаблонов, зачастую экспортируемый шаблон имеет различные пути инстанцирования и каждый из них имеет свой собственный контекст. На возражение наподобие того, что та же проблема наблюдается и в случае функций, определенных в заголовочном файле, т.е. встраиваемых (`inline`),

можно возразить, что проблема с шаблонами существенно большая, поскольку появляется больше возможностей для изменения смысла имен, в частности, потому что шаблоны работают с более мощными множествами имен, чем закрытые функции. Шаблоны используют *зависимые имена*, т.е. имена, которые зависят от аргументов шаблонов. Поэтому при каждом инстанцировании шаблона с одинаковыми аргументами пользователь шаблона должен обеспечить одинаковый контекст (например, множество перегруженных функций, работающих с типами аргументов шаблона). Это требуется для того, чтобы предотвратить непредумышленное инстанцирование, имеющее разный смысл в разных файлах, что противоречит правилу одного определения. Почему ожидается, что это будет большей проблемой в модели экспорта, чем в модели включения? Потому что главное, чем отличаются эти модели, — это выполнение поиска имен в разных единицах трансляции при экспортировании, чего не требуется для реализации других возможностей стандартного C++.

Пример 2. Компилятору сложнее генерировать высококачественную диагностику, помогающую программисту. Сообщения об ошибках, связанные с шаблонами, и без того пользуются дурной славой слишком громоздких и трудно понимаемых из-за длинных имен. Каскадные инстанцирования при экспортировании шаблонов вряд ли добавят ясности в эти сообщения. Кроме того, экспортирование добавляет как бы новое измерение в пространстве сообщений об ошибках. Сообщения типа "ошибка в строке X, вызванная инстанцированием функции Y, вызванным инстанцированием Z, вызванным инстанцированием..." теперь должны указывать еще и единицы трансляции, в которых это произошло. В результате каждая строка такой "поэмы об ошибках" будет содержать разные единицы трансляции. Выявление нарушений правила одного определения — уже достаточно сложная проблема, но в такой ситуации она становится во много раз сложнее. Столкнувшись с такими проблемами, многие программисты сочтут сообщения об ошибках в обычных шаблонах легко читаемыми и понятными.

Пример 3. Экспорт накладывает новые ограничения на среду разработки. Среда разработки — это не только `.cpp` и `.h`-файлы, и многие современные инструменты разработки не в состоянии работать с выглядящими циклическими зависимостями при изменении объектных файлов в процессе компоновки (в предыдущей задаче отмечалось, что экспорт только скрывает зависимости, так что при изменении файла с экспортируемым шаблоном надо перекомпилировать не только его, но и все его инстанцирования).

Как заметил один из специалистов мирового класса по шаблонам, Джон Спайсер (John Spicer) из EDG, "экспорт сложен по самой своей природе и требует огромной работы для того, чтобы осознать все его последствия. *Трудно дать простые советы по его использованию, которые уберегут программистов от неприятностей*" [выделено мной].

Потенциальные преимущества экспорта

4. В чем заключаются реальные и потенциальные преимущества `export`?

Теперь, когда программистам доступна реализация экспорта, самое время разобраться, как же следует использовать новые возможности. Вот основные преимущества, которые можно надеяться получить от использования экспорта шаблонов.

1. Ускорение построения приложений. Все еще остается открытым вопрос о влиянии экспорта шаблонов на скорость построения реальных приложений, использующих шаблоны. При широком внедрении и использовании экспорта шаблонов можно надеяться, что исследования в этой области позволят выработать как новые технологии, так и рекомендации по созданию кода, для которого будет явным увеличение скорости построения приложения. В частности, есть надежда, что единицы трансляции будут менее чувствительны к изменениям в определении шаблона, а значит, их обработка при необходимости перекомпиляции будет выполняться быстрее.

Предостережение. По причинам, изложенным в предыдущей задаче, устранить зависимости при помощи экспорта шаблонов представляется невозможным, так что они будут оставаться в той или иной скрытой форме. Кроме того, заметим, что в реализации шаблонов у EDG данное потенциальное достоинство доступно в рамках обоих моделей организации исходного кода — как в модели экспорта, так и в модели включения. Это означает, что для этой реализации экспорт шаблонов не имеет никаких преимуществ перед моделью включения.

2. Ограничение распространения макросов. Это — реальное преимущество использования экспорта. В традиционной модели включения макросы находятся в заголовочных файлах, и в результате оказываются включенными во множество единиц трансляции. Таким образом, макросы, попавшие извне в единицу трансляции до определения шаблона, могут воздействовать на это определение. В случае использования экспорта шаблонов этого не происходит, и программист получает более полный контроль над определением своего шаблона, которое находится в отдельном файле. Внешние макросы при этом не в состоянии легко воздействовать на внутренние детали определения шаблона.

Это — реальное преимущество модели экспорта шаблонов, но это преимущество обеспечивается не только экспортом. В комитете по стандартизации C++ уже рассматриваются более общие решения проблемы макросов во всех контекстах; примером такого решения могут служить предложенные Страуструпом новые директивы препроцессора `#scope` и `#endscope`. Если такое решение будет принято, оно полностью устранит описанное преимущество модели экспорта шаблонов.

Словом, нам остается только ожидать, какие преимущества экспорта шаблонов над моделью включения проявятся в ближайшие годы. Я со своей стороны хочу только порекомендовать при выявлении какого-либо преимущества экспорта шаблонов тщательно проверять, не проявляется ли это преимущество и в модели включения.

Мораль

Так следует ли использовать экспорт шаблонов, и если да, то как именно следует это делать с точки зрения безопасности? В настоящее время этот вопрос реально касается только очень небольшого количества программистов, которые работают с единственным компилятором, поддерживающим эту возможность языка; для большинства же этот вопрос имеет не более чем теоретическое значение. Там, где не можешь, — не должен и хотеть...

Если же вы — пользователь одного из компиляторов будущего, который поддерживает экспорт шаблонов, тогда главное правило для вас звучит следующим образом:

➢ Рекомендация

Если вам нужен переносимый код — не используйте `export`.

Эта рекомендация банальна, если учесть, что экспорт шаблонов поддерживает только один-единственный компилятор. Код с использованием `export` не переносим сегодня и вряд ли будет переносим в ближайшем будущем.

Но что если переносимость кода вас не волнует, ваш компилятор поддерживает экспорт шаблонов и вы хотите его использовать? Тогда — вся ответственность за последующие действия ложится на вас. Помните, что экспорт шаблонов — все еще в стадии эксперимента, так что он не всегда в состоянии обеспечить то, чего от него ожидают, а кроме того, может вносить определенные сложности при использовании других возможностей C++. Имеются определенные сложности и при написании кода с экспортируемыми шаблонами — с ними вы ознакомились в этой и предыдущей задачах.

Мой главный совет — даже если ваш компилятор поддерживает экспорт шаблонов, постарайтесь избежать использования этой пока что экспериментальной возможности. Предоставьте экспериментировать на себе кому-то другому.

➢ Рекомендация

(Пока что) избегайте использования `export`.

Но если вы все же решили выступить в роли подопытного кролика, то вот несколько советов на основании того, что нам уже известно об экспорте шаблонов, которые, возможно, облегчат вашу участь.

➢ Рекомендация

Если вы решили выборочно использовать `export` для некоторых из ваших шаблонов, то помните о следующем.

Не следует ожидать, что вы сможете не поставлять пользователям весь исходный текст (или его эквивалент). Вы будете вынуждены делать это всегда.

Не следует ожидать, что при использовании `export` произойдет существенное ускорение построения ваших программ. Более того, скорость построения может даже упасть.

Убедитесь, что используемый вами инструментарий и среда разработки отвечают новым требованиям, в частности, что они способны корректно обработать изменение объектных файлов на стадии компоновки, если такая методика используется вашей реализацией экспорта шаблонов.

Если ваши экспортируемые шаблоны используют функции или объекты из безымянных пространств имен (или объявленные как `static`), то:

- вы должны понимать, что эти функции и объекты будут вести себя так, как если бы они были объявлены в качестве `extern` и что функции будут принимать участие в разрешении перегрузки вместе с произвольным количеством функций из других безымянных пространств имен из неизвестного заранее числа исходных файлов;
- вы должны "обезображивать до неузнаваемости" имена таких функций, чтобы предохраниться от непреднамеренного изменения семантики. (Обидно, ведь предназначение безымянных пространств имен и статических функций и объектов именно в том, чтобы вам не надо было прибегать к такого рода действиям по изменению имен; однако экспорт шаблонов лишает вас этой возможности C++);

Вы должны понимать, что это далеко не полный список неприятностей и что вы можете в любой момент столкнуться с новыми проблемами. Еще раз напомню слова Джона Спайсера о том, что трудно дать простые советы, которые уберегут программистов от неприятностей. Возможно, в будущем ситуация изменится, а пока что экспорт шаблонов — terra incognita, где на каждом шагу вас могут подстерегать неизвестные пока что неприятности и проблемы. Будем надеяться, что со временем ситуация изменится.

Пока что нельзя сказать, насколько совет "избегайте экспорта" будет актуален в будущем. Время и опыт покажут, так ли это. Постепенно поддержка экспорта шаблонов будет реализована многими другими производителями компиляторов, и тогда, после определенного периода накопления опыта, будет дан окончательный ответ на вопрос, *стоит* ли использовать эту возможность.

Вопросы и приемы безопасности исключений

Обработка исключений представляет собой фундаментальный механизм сообщения об ошибках в современных языках программирования, включая C++. В [Sutter00] и [Sutter02] мы детально рассмотрели множество вопросов, связанных с определением того, что такое безопасность исключений, как следует писать безопасный с точки зрения исключений код и многие другие.

В этом разделе мы продолжим изложение материала, посвященного обработке исключений, сконцентрировав свое внимание на некоторых специфических возможностях языка, связанных с исключениями. Начнем же мы с ответов на неизменные вопросы — достаточно ли для безопасности исключений написать в нужном месте `try` и `catch`? Если нет, то что для этого требуется? И о чем не следует забывать при разработке стратегии безопасности исключений в ваших программах?

Прежде всего, одну задачу мы посвятим выяснению причин, по которым так важно написание безопасного с точки зрения исключений кода. Это позволит выработать стиль программирования, который даст возможность писать более интеллектуальный, надежный и легко сопровождаемый код — даже безотносительно обработки исключений. Однако не следует забывать, что лучшее — враг хорошего, и мы сможем убедиться в этом еще раз при рассмотрении спецификаций исключений. Мы рассмотрим целый ряд вопросов. Зачем они нужны в языке? Насколько оправданно было их введение в язык в принципе? Почему, несмотря на их наличие, лучше не использовать их в своих программах?

Задача 11. Попробуй поймай[23] Сложность: 3

Заключается ли безопасность исключений в том, чтобы написать `try` и `catch` в нужном месте? Если нет, то в чем? О чем не следует забывать при разработке стратегии безопасности исключений в ваших программах?

Вопрос для новичка

1. Что такое `try`-блок?

Вопрос для профессионала

2. "Написание безопасного с точки зрения исключений кода сводится в целом к размещению `try` и `catch` в правильных местах". Обсудите это утверждение.
3. Когда следует использовать `try` и `catch`? Когда их не следует использовать? Изложите ваш ответ в виде рекомендации стандарта кодирования.

Решение

1. Что такое `try`-блок?

`try`-блок (в некоторых книгах — "блок с контролем") представляет собой блок кода (составную инструкцию), выполнение которого может быть неудачным (приводящим к генерации исключения), за которым следует один или несколько блоков обработки исключений, которые выполняются при генерации в основном блоке исключения определенного типа, например:

```
// Пример 11-1: пример try-блока
//
try {
    if( некоторое_условие )
        throw string( "Строка" );
    else if(некоторое_иное_условие )
        throw 42;
}
catch( const string& ) {
    // Выполняется при генерации исключения, имеющего тип
    // string
}
catch( ... ) {
    // Выполняется при генерации любого другого исключения
}
```

В примере 11-1 код в основном блоке может сгенерировать исключение типа `string`, `int` или не сгенерировать никакого исключения.

2. "Написание безопасного с точки зрения исключений кода сводится в целом к размещению `try` и `catch` в правильных местах". Обсудите это утверждение.

Честно говоря, такое утверждение отражает фундаментальное непонимание безопасности исключений. Исключения представляют собой один из видов сообщения об ошибках, и мы знаем, что написание устойчивого к ошибкам исходного текста не сводится к проверке кода возврата и обработке ошибки.

[23] В оригинале — игра слов, основанная на переводе ключевых слов `try` — пробовать, и `catch` — ловить. — *Прим. перев.*

В действительности, безопасность исключений редко сводится к написанию ключевых слов (и чем меньше вы их пишете, тем лучше). Никогда нельзя забывать о том, что безопасность исключений влияет на проектирование. Вопросы безопасности надо учитывать заранее, еще на стадии проектирования программы, а не просто расставлять ключевые слова в подходящих местах.

Вот три основных момента, которые надо учитывать при написании безопасного в смысле исключений кода.

1. Где и когда следует генерировать исключения? Это вопрос о том, где именно следует размещать инструкции `throw`. В частности, мы должны ответить на следующие вопросы.

- Какие именно исключения должен генерировать код? То есть о каких ошибках мы будем сообщать при помощи механизма исключений, а не при помощи возврата кода ошибки или какого-либо иного метода?
- Какой код не должен генерировать исключений? В частности, какой код гарантирует отсутствие исключений? (См. задачу 12 и [Sutter99].)

2. Где и когда следует обрабатывать исключения? Это единственный вопрос, связанный с выбором правильного размещения `try` и `catch`, и в большинстве случаев эта проблема решается автоматически. Начнем с вопросов, на которые мы должны ответить.

- Какая часть исходного текста должна отвечать за обработку исключений? То есть у какого кода оказывается достаточно контекста и информации для обработки ошибки, о которой сообщается посредством исключения (возможно, путем преобразования исключения к другому виду)? В частности, заметим, что код обработчика должен обладать информацией, достаточной для того, чтобы выполнить все необходимые действия по освобождению распределенных ресурсов.
- Какой код *должен* обрабатывать исключения? То есть каким образом выбрать среди всех возможных вариантов размещения кода обработчика наиболее подходящий?

После того как мы ответим на эти вопросы, отметим, что использование идиомы "распределение ресурса есть инициализация" зачастую позволяет избежать использования ряда `try`-блоков путем автоматизации работы по освобождению ресурсов. Если вы "обернете" динамически распределяемые ресурсы в объект-владелец, его деструктор обычно в состоянии выполнить автоматическое освобождение распределенного ресурса в необходимый момент времени, без явного использования `try` и `catch`, не говоря о том, что код с использованием этого метода проще писать, а позже — читать и понимать.

➢ Рекомендация

Использование автоматического освобождения с помощью деструкторов предпочтительнее применения для этой цели `try`-блоков.

---

3. Будет ли поведение моего кода безопасным, если произойдет генерация исключения в какой-либо из вызываемых функций? Это — вопрос правильного управления ресурсами, позволяющего избежать утечек, содержания классов и инвариантов программы в должном порядке и прочих показателей корректности программы. Иначе говоря, надо, чтобы в случае генерации исключения до его перехвата и обработки программа оставалась работоспособной, в согласованном состоянии, и могла продолжить работу после обработки исключения. Для большинства программистов именно этот аспект безопасности исключений представляет наибольшую сложность и требует наибольших усилий при написании кода.

Заметим, что среди трех перечисленных моментов только один непосредственно связан с размещением ключевых слов `try` и `catch`, да и тех часто можно избежать

при разумном использовании деструкторов для автоматического освобождения распределенных ресурсов.

3. Когда следует использовать `try` и `catch`? Когда их не следует использовать? Изложите ваш ответ в виде рекомендации стандарта кодирования.

Вот один из вариантов ответа на этот вопрос.

1. Определите общую стратегию обработки ошибок и сообщений о них на уровне приложения или подсистемы и строго следуйте ей. В частности, стратегия должна охватывать, как минимум, следующие основные аспекты (а в действительности включать и многие другие вопросы).

- *Сообщения об ошибках.* Определите, о каких именно ошибках будет сообщаться и каким образом. Среди всех прочих методов сообщения об ошибках следует отдавать предпочтение исключениям. Вообще говоря, для каждой ситуации следует подобрать наиболее подходящий, удобный и легко сопровождаемый метод. Так, исключения наиболее подходят для конструкторов и операторов, которые не в состоянии вернуть значение, указывающее на происшедшую ошибку, или когда место ошибки и ее обработчик оказываются далеко друг от друга.
- *Распространение ошибок.* Помимо прочего, следует определить границы, которые не должно пересекать сгенерированное исключение. Обычно в роли таких границ выступает модуль или границы API.
- *Обработка ошибок.* Там, где это возможно, предоставьте возможность освобождения распределенных ресурсов посредством деструкторов владеющих ресурсами объектов вместо того, чтобы использовать механизм `try` и `catch`.

2. Генерируйте исключение в месте обнаружения ошибки и не пытайтесь обработать его самостоятельно. (Понятно, что если код в состоянии сам справиться с возникшей ошибкой, то он не должен сообщать о ней!)

Документируйте каждую операцию — какие исключения может сгенерировать операция и почему. Такое описание должно быть частью документации каждой функции и каждого модуля. От вас не требуется писать спецификацию исключений для каждой функции (более того, как вы увидите в задаче 13, вы и не должны это делать), но вы должны ясно и точно документировать, чего следует ожидать вызывающей функции, поскольку семантика ошибок является частью интерфейса функции или модуля.

3. Используйте ключевые слова `try` и `catch` там, где у вас есть достаточно информации для обработки ошибки, ее преобразования или обеспечения границы, определенной стратегией обработки ошибок. В частности, я обнаружил, что для использования `try` и `catch` имеются три основные причины.

- *Для обработки ошибки.* Это самый простой случай: произошла ошибка, мы знаем, что делать в такой ситуации, и мы выполняем эти действия. Жизнь продолжается (уже без исходного исключения, которое приказало долго жить). Если можно, то все необходимые действия лучше выполнять в деструкторе; если нет — использовать `try/catch`.
- *Для преобразования исключения.* Это означает перехват одного исключения, которое сообщает о низкоуровневой ошибке, и генерацию другого исключения, в контексте своей собственной высокоуровневой семантики. Кроме того, исходное исключение может быть преобразовано в другое представление, например, код ошибки.

  Рассмотрим, например, класс, представляющий коммуникационное соединение, работающее с различными типами узлов и протоколов. Попытка установить соединение между двумя узлами может окончиться неуспешно из-за множества причин — например, из-за физического повреждения сети или ошибки

аутентификации. Функция Open может самостоятельно обработать все эти ситуации и не сообщать о них вызывающей функции, которой не известно, что такое пакет Foo или протокол Bar. Коммуникационный класс самостоятельно обрабатывает низкоуровневые ошибки, оставаясь в согласованном состоянии, и сообщает вызывающей функции только об ошибке высокого уровня, т.е. о том, что соединение не может быть установлено.

```
void Session::Open( /* ... */ ) {
    try {
        // Все необходимые действия
    }
    catch( const ip_error& err ) {
        // - обработка IP-ошибки
        // - освобождение ресурсов
        throw Session::OpenFailed();
    }
    catch( const KerberosAuthentFail& err ) {
        // - обработка ошибки аутентификации
        // - освобождение ресурсов
        throw Session::OpenFailed();
    }
    // ... и т.д. ...
}
```

- *Для перехвата исключений на границе подсистемы.* Эта ситуация, как правило, включает преобразование исключения, обычно в код ошибки или другое представление, не основанное на механизме исключений. Например, свертка стека доходит до функции, которая является частью интерфейса вашей подсистемы для языка C, у вас есть только два варианта действий — либо немедленно вернуть код ошибки из текущей функции API, либо установить состояние ошибки, которое вызывающая функция может опросить позже при помощи дополнительной функции API GetLastError.

➢ Рекомендация

Определите общую стратегию сообщений об ошибках и их обработки в вашем приложении или подсистеме, и строго придерживайтесь ее. Глобальная стратегия должна включать стратегии сообщений об ошибках, распространения ошибок и их обработки.

Используйте оператор throw там, где обнаружена ошибка, но нет возможности обработать ее самостоятельно.

Используйте ключевые слова try и catch там, где вы обладаете достаточной информацией для обработки ошибки, ее преобразования или обеспечения границы, определенной стратегией обработки ошибок (например, catch(...) на границе подсистемы).

Резюме

Один мудрый человек сказал: "Либо веди сам, либо следуй за ведущим, либо уходи с дороги!" Относительно безопасности исключений можно перефразировать это так: "Либо генерируй исключение, либо обрабатывай его, либо уходи с дороги!"

На практике случай "ухода с дороги" зачастую оказывается основным при анализе безопасности исключений. Это и есть главная причина того, что безопасное с точки зрения исключений кодирование не сводится к расстановке try и catch в нужных местах. Задача скорее в том, чтобы суметь уйти с дороги в нужном месте.

Задача 12. Безопасность исключений: стоит ли овчинка выделки? Сложность: 7

Стоит ли прилагать такие усилия по написанию безопасного с точки зрения исключений кода? Этот вопрос — казалось бы, давно получивший однозначный ответ — все еще иногда становится предметом обсуждения.

Вопрос для профессионала

1. Вкратце опишите гарантии безопасности исключений Абрамса (базовую, строгую, отсутствия исключений).
2. В каких случаях следует разрабатывать код, отвечающий требованиям:
   а) базовой гарантии?
   б) строгой гарантии?
   в) гарантии отсутствия исключений?

Решение

Гарантии Абрамса

1. **Вкратце опишите гарантии безопасности исключений Абрамса (базовую, строгую, отсутствия исключений).**

Базовая гарантия (basic guarantee) заключается в том, что сбой при выполнении операции может изменить состояние программы, но не вызывает утечек и оставляет все объекты пригодными к дальнейшему использованию, в согласованном (но не обязательно предсказуемом) состоянии.

Строгая гарантия (strong guarantee) обеспечивает семантику транзакций: при сбое операции гарантируется неизменность состояния программы, относящегося к задействованным в операции объектам. Это означает отсутствие побочных эффектов операции, влияющих на состояние объектов, включая корректность или содержимое задействованных вспомогательных объектов, таких как итераторы, указывающие внутрь контейнеров, с которыми выполнялась операция.

И наконец, *гарантия бессбойности* (nofail guarantee) означает невозможность возникновения сбоя. В контексте исключений это означает, что операция гарантированно их не генерирует. (Абрамс и другие (в том числе в предыдущих книгах *Exceptional C++*) называли эту гарантию гарантией отсутствия исключений (nothrow guarantee). Я изменил это название на гарантию бессбойности, поскольку рассматриваемые гарантии относятся ко всем механизмам обработки ошибок, как с использованием исключений, так и без них.)

Какая именно гарантия нужна

2. **В каких случаях следует разрабатывать код, отвечающий требованиям:**
   а) **базовой гарантии?**
   б) **строгой гарантии?**
   в) **гарантии отсутствия исключений?**

Всегда следует писать код, отвечающий по крайней мере одной из перечисленных гарантий. Тому есть несколько причин.

1. Исключения всегда возможны. Их может сгенерировать стандартная библиотека. Они могут порождаться языком программирования. Наш код должен быть к этому готов. К счастью, это небольшая беда, поскольку теперь мы знаем, что с ними делать. Нам надо просто принять некоторые правила поведения и строго им следовать.

Главная проблема заключается в общем подходе к обработке ошибок. Как именно происходит оповещение о происшедшей ошибке — при помощи механизма исключений или посредством кодов ошибок — это лишь детали используемого синтаксиса, в то время как главные различия подходов заключаются в семантике. Каждый подход требует своего собственного стиля.

2. Написание безопасного с точки зрения исключений кода — правило хорошего тона. Безопасность кода и его качество взаимосвязаны. Распространенные методы написания безопасного в плане исключений кода вполне применимы и в случае отсутствия исключений. Рассмотрим основные приемы, облегчающие написание безопасного кода.

- Использование идиомы "распределение ресурса есть инициализация" для работы с ресурсами. Использование таких объектов-владельцев ресурсов, как классы `Lock` и `shared_ptr` (см. [Boost, Sutter02a]), — хорошая мысль безотносительно к безопасности исключений. Не удивительно, что к достоинствам таких классов относится и безопасность исключений. Сколько раз вам приходилось встречаться с функциями (понятно, что мы не говорим о ваших собственных функциях, разговор идет о чужом коде), в которых в ветви, приводящей к преждевременному возврату из функции, не выполнялось необходимое освобождение ресурсов? А ведь достаточно было использовать указанную идиому, и все эти действия были бы выполнены автоматически.
- Применение методики, когда вся необходимая работа выполняется "в стороне", а затем принимается при помощи кода, гарантированно не генерирующего исключений, позволяет избежать изменения внутреннего состояния объектов до тех пор, пока вы не будете уверены, что успешно выполнена вся операция целиком. Такое транзакционное программирование понятнее, чище и безопаснее даже при использовании кодов ошибок. Как часто вам приходилось встречаться с функциями (мы вновь говорим не о вашем коде), в которых при преждевременном возврате в одной из ветвей объекты оказывались в некорректном состоянии из-за того, что в них выполнялись некоторые изменения, после чего происходил сбой в выполнении операции?
- Следование принципу "один класс (одна функция) — одно действие". Функции, выполняющие несколько действий, например `Stack::Pop` или `EvaluateSalaryAndReturnName` из книги [Sutter00], очень сложно сделать строго безопасными в смысле исключений. Многие проблемы, связанные с безопасностью исключений, можно решить, просто придерживаясь указанного принципа. Это правило появилось задолго до того, как стало понятно, что оно применимо и к проблемам безопасности исключений; принцип одного действия ценен сам по себе.

Все эти методы можно и нужно использовать безотносительно к вопросам безопасности исключений.

Теперь, после всего сказанного, перед нами встает вопрос: когда и какую гарантию следует использовать? Вот правило, которому следует стандартная библиотека C++ и которое вы с успехом можете применять в собственных программах.

➢ Рекомендация

Функция всегда должна обеспечивать наиболее строгую гарантию безопасности исключений, которую можно обеспечить без ущерба для вызывающей функции, в ней не нуждающейся.

То есть, если ваша функция в состоянии обеспечить бессбойную гарантию без ущерба для вызывающей функции, которой такая степень гарантии не нужна, то эту гарантию следует обеспечить. Заметим также, что некоторое количество ключевых функций просто обязано обеспечивать гарантию бессбойности.

➢ Рекомендация

Никогда не позволяйте генерировать исключения деструкторам, функциям, освобождающим ресурсы, и функциям обмена, поскольку в противном случае зачастую оказывается невозможно надежно и безопасно выполнить освобождение распределенных ресурсов.

В противном случае, если ваша функция в состоянии обеспечить строгую гарантию без ущерба для пользователей, вы должны это сделать. Заметим, что `vector::insert` представляет собой пример функции, которая в общем случае не поддерживает строгую гарантию, поскольку для этого требуется создание полной копии содержимого вектора при каждой вставке элемента, и далеко не всем программам настолько необходима строгая гарантия безопасности, чтобы платить за нее такую большую цену. (Программы, которым крайне необходима строгая гарантия безопасности исключений, могут легко добиться этого при помощи функции-оболочки вокруг `vector::insert`, которая будет копировать вектор, выполнять вставку в копию, и в случае удачного выполнения этих операций обменивать содержимое копии и исходного вектора.)

В противном случае ваша функция должна обеспечивать базовую гарантию.

Дополнительную информацию о рассмотренных концепциях (например: что собой представляет бессбойная функция обмена `swap` или почему деструкторы не должны генерировать исключений) вы найдете в книгах [Sutter00] и [Sutter02].

Задача 13. Прагматичный взгляд на спецификации исключений — Сложность: 6

Сейчас, когда сообществом программистов на C++ накоплен определенный опыт работы со спецификациями исключений, пришло время систематизировать его. Наша задача посвящена вопросу применения спецификаций исключений с учетом особенностей различных реальных компиляторов.

Вопрос для новичка

1. Что произойдет при нарушении спецификации исключений? Почему? Каковы основные причины существования этой возможности C++?
2. Какие исключения могут быть сгенерированы каждой из перечисленных ниже функций.

```
int Func();
int Gunc() throw();
int Hunc() throw(A,B);
```

Вопрос для профессионала

3. Является ли спецификация исключений частью типа функции? Обоснуйте свой ответ.
4. Что собой представляют спецификации исключений и как они работают? Дайте точный ответ на поставленный вопрос.
5. Когда стоит использовать спецификацию исключений в функции? Почему вы используете (или не используете) эту возможность?

Решение

Как вы знаете, сейчас идет работа над новым стандартом C++ (рабочее название C++0x). Давайте оглянемся назад и постараемся осмыслить накопленный опыт работы с текущим стандартом [C++03]. Подавляющее большинство стандартных возможностей C++ просто замечательны, и именно им посвящена львиная доля публикаций, что не удивительно — кому хочется твердить о недостатках! Слабые, мало используемые возможности языка чаще всего просто игнорируются и постепенно забываются (и это далеко не всегда плохо). Вот почему встречается очень мало статей о таких невразумительных возможностях языка, как `valarray`, `bitset` и прочих — и в их число входят и спецификации исключений.

Давайте поближе познакомимся с имеющимся опытом использования стандартных спецификаций исключений C++.

Нарушение спецификации

1. Что произойдет при нарушении спецификации исключений? Почему? Каковы основные причины существования этой возможности C++?

Идея спецификаций исключений заключается в проверке времени выполнения того, что данная функция может генерировать только определенные типы исключений (либо не генерировать их вовсе). Например, приведенная ниже спецификация исключений гарантирует, что `f` может генерировать только исключения типа `A` или `B`[24]:

[24] Говоря точнее, если окружить эту функцию `try/catch` блоками для перехвата исключений `A` и `B`, то все возможные исключения будут перехвачены — в частности, такая функция может генерировать исключения, являющиеся производными классами от `A` и `B`. — *Прим. ред.*

```
int f() throw( A, B );
```

Если будет сгенерировано исключение, которого нет в списке спецификации исключений, будет вызвана функция unexpected().

```
// Пример 13-1
//
int f() throw(A,B) { // A и B не связаны с C
    throw C();       // Будет вызвана функция unexpected
}
```

Вы можете зарегистрировать ваш собственный обработчик для этого случая при помощи стандартной функции set_unexpected. Ваш обработчик не должен получать никаких параметров и не должен возвращать никаких значений.

```
void MyUnexpectedHandler() { /* ... */ }
std::set_unexpected( &MyUnexpectedHandler );
```

Остается один вопрос — что может делать ваш обработчик? Единственное, чего он не может делать, — это выйти из функции при помощи оператора return. Поэтому у него есть два варианта действий:

- преобразовать исключение в другое, допустимое спецификацией исключений, путем генерации исключения типа, имеющегося в списке спецификации исключений, вызвавшего вызов обработчика. Свертка стека при этом продолжится с того места, где она остановилась;
- вызвать функцию terminate, которая завершает работу программы. (Функция terminate также может быть заменена другой, но в любом случае она должна завершить выполнение программы.)

Применение

Идея, лежащая в основе спецификаций исключений, очень проста: в программе C++ любая функция, если не указано иное, может генерировать исключения любого типа. Рассмотрим некоторую функцию Func.

2. Какие исключения могут быть сгенерированы каждой из перечисленных ниже функций.

```
// Пример 13-2(a)
//
int Func();      // Может генерировать любые исключения
```

По умолчанию в C++ функция Func может генерировать исключения любого типа, как сказано в комментарии к ней. Однако зачастую нам известно, что функция может генерировать только исключения определенных типов. В таком случае может оказаться разумным сообщить об этом компилятору и программисту. Например:

```
// Пример 13-2(б)
//
int Gunc() throw();     // Не генерирует исключений
int Hunc() throw(A,B); // Может генерировать только A или B
```

В приведенном примере спецификации исключений использованы для того, чтобы дать дополнительную информацию о функциях, а именно — о типах исключений, которые они могут генерировать. Комментарии, приведенные рядом с функциями, переводят спецификации исключений на обычный русский язык.

Первая мысль по этому поводу — чем больше информации, тем лучше, так что указание спецификации исключений функции — это всегда не плохо. Но это не обязательно так, поскольку зачастую в излишней детализации и кроется зло: хотя намерения у спецификаций исключений и благие, вымощенный ими путь может завести нас не совсем туда, куда хотелось бы.

3. Является ли спецификация исключений частью типа функции? Обоснуйте свой ответ.

Джон Спайсер (John Spicer) из знаменитой Edison Design Group, автор большой части главы стандарта C++, посвященной шаблонам, назвал спецификации исключений C++ "призрачными типами" (shadow type). Одна из важнейших характеристик C++ — строгая типизация, и это хорошо и правильно. Но почему же мы называем спецификации исключений призрачными типами, а не частью системы типов C++?

Причина проста, хотя и имеет "двойное дно":

- спецификации исключений не являются частью типов функций;
- за исключением тех ситуаций, когда они являются частью типов.

Рассмотрим сначала пример, когда спецификации исключений не участвуют в образовании типа функции.

```
// Пример 13-3(а): спецификацию исключений нельзя
//                 использовать в инструкции typedef.
//
void f() throw(A,B);
typedef void (*PF)() throw(A,B); // Ошибка
PF pf = f;                       // Невозможно из-за ошибки
```

Спецификация исключений не может использоваться в определении типа посредством typedef. C++ не позволит вам написать такой код, так что спецификации исключений не могут участвовать в типе функции... как минимум, в контексте typedef. Но в других случаях спецификации исключений в действительности участвуют в типах функций, если их записать без использования typedef.

```
// Пример 13-3(б): все то же, но без typedef!
//
void f() throw(A,B);
void (*pf)() throw(A,B);        // ok
pf = f;                         // ok
```

Кстати, такое присваивание указателя на функцию можно выполнять и в случае, когда спецификации исключений различны, но ограничения, накладываемые спецификацией исключений, при присваивании не ослабляются.

```
// Пример 13-3(в): тоже вполне кошерно и с низким
//                 содержанием холестерина... :)
//
void f() throw(A,B);
void (*pf)() throw(A,B,C); // ok
pf = f;                    // ok, тип pf менее строгий
```

Спецификации исключений также участвуют в типах виртуальных функций, когда вы пытаетесь их перекрыть.

```
// Пример 13-3(г): спецификации исключений имеют значение
//                 для виртуальных функций.
//
class C {
    virtual void f() throw(A,B); // Некоторая спецификация
                                 // исключений
};
class D : C {
    void f();                    // Ошибка - спецификация
                                 // исключений менее строгая
};
```

Итак, первая проблема, связанная со спецификациями исключений, состоит в том, что в сегодняшнем C++ они являются "призрачными типами", которые играют по правилам, отличным от обычных правил системы типов C++.

Проблема вторая — (не)понимание

Вторая проблема — следует точно знать, что получается при использовании спецификации исключений.

4. Что собой представляют спецификации исключений и как они работают? Дайте точный ответ на поставленный вопрос.

Вот как обычно программисты представляют себе работу спецификаций исключений.

- Гарантируют, что функции будут генерировать только исключения перечисленных типов (возможно, не будут генерировать их вообще).
- Позволяют компилятору выполнить определенную оптимизацию, основанную на знании о том, что могут быть сгенерированы только перечисленные исключения.

Эти ожидания, увы, несколько обманчивы. Рассмотрим еще раз код примера 13-2(б).

```
// Пример 13-2(б): два потенциальных обмана
//
int Gunc() throw();   // Не генерирует исключений          ←?
int Hunc() throw(A,B);// Может генерировать только A или B ←?
```

Корректны ли приведенные комментарии? Не совсем. Функция `Gunc` на самом деле может сгенерировать исключение, а функция `Hunc` — сгенерировать исключение, отличное от `A` и `B`! Компилятор только гарантирует, что если это произойдет — он просто, не дрогнув, прикончит вашу программу.

Раз функции `Gunc` и `Hunc` могут сгенерировать все, что угодно, а не только обещанное, то компилятор не только не может полагаться на то, что этого не произойдет, но должен выполнять еще и роль полицейского, следя за тем, что именно генерируется в этих функциях и насколько оно соотносится с обещаниями, данными в спецификациях исключений. Если все же обещание будет нарушено, компилятор должен вызвать функцию `unexpected`, что в большинстве случаев приведет к завершению работы вашей программы. Почему? Потому что из этой функции есть только два пути, и ни один из них не является нормальным возвратом из функции. Выбирайте сами.

- Можно сгенерировать исключение, которое разрешено спецификацией исключений. В этом случае распространение исключения продолжится как обычно. Но помните, что обработчик `unexpected` — глобальный, т.е. он один-единственный на всю программу. Вряд ли такой глобальный обработчик окажется достаточно разумным, чтобы выполнить нечто разумное в каждом частном случае, так что, скорее всего, путь один — вызов `terminate`...
- Можно сгенерировать исключение, которое спецификацией исключений не разрешено. Если в спецификации исключений исходной функции имеется исключение `bad_exception`, то далее будет распространяться именно оно. Ну, а если нет, то путь один — вызов `terminate`...

Поскольку нарушение спецификации исключений в подавляющем большинстве случаев приводит к завершению работы вашей программы, я думаю, о таком поведении вполне можно сказать, что "нарушение, не дрогнув, прикончит вашу программу".

В начале ответа на четвертый вопрос мы видели, чего программисты ожидают от спецификаций исключений. Давайте внесем коррективы в эти ожидания.

- ~~Гарантируют,~~ *во время выполнения программы заставляют* ~~что~~ функции ~~будут~~ генерировать только перечисленные исключения (возможно, не ~~будут~~ генерировать их вообще).
- Позволяют *или запрещают* компилятору выполнить определенную оптимизацию, основанную на ~~знании о том, что могут быть сгенерированы только пере-~~

~~численные исключения~~ *проверке, имеется ли сгенерированное исключение в списке спецификации исключений.*

Чтобы понять, как должен работать компилятор, рассмотрим следующий код, в котором приведено тело одной из рассматриваемых функций — Hunc.

```
// Пример 13-4(a)
//
int Hunc() throw(A,B) {
    return Junc();
}
```

Компилятор должен сгенерировать код наподобие приведенного ниже, и расходы процессорного времени на его обработку точно такие же, как если бы вы ввели его самостоятельно (единственное облегчение — вам не надо набирать его самостоятельно; компилятор сам позаботится о его генерации).

```
// Пример 13-4(б): обработанный компилятором
//                 код из примера 13-4(a)
//
int Hunc()
try {
    return Junc();
}
catch( A ) {
    throw;
}
catch( B ) {
    throw;
}
catch( ... ) {
    std::unexpected(); // Отсюда нет возврата! В лучшем
                       // случае здесь будет сгенерировано
                       // исключение A или B
}
```

Вполне очевидно, что вместо того чтобы позволить компилятору оптимизировать код на основании информации о типах генерируемых исключений, спецификации исключений заставляют компилятор *выполнять лишнюю работу* по проверке сгенерированного исключения на предмет его принадлежности к списку спецификации.

Копнем поглубже

Большинство людей удивляет тот факт, что спецификации исключений могут вызвать снижение производительности. Одна из причин такого снижения была только что продемонстрирована — это неявная генерация try/catch-блоков, хотя при использовании эффективных компиляторов соответствующие потери производительности невелики.

Есть как минимум еще две причины снижения производительности программы из-за спецификаций исключений.

- Некоторые компиляторы автоматически делают невстраиваемыми функции, объявленные как inline, если у них имеются спецификации исключений. Это такая же эвристика, как и отказ некоторых компиляторов во встраиваемости функциям, которые имеют слишком много вложенных инструкций или содержащим циклы.
- Некоторые компиляторы вообще не в состоянии оптимизировать механизм исключений и добавляют автоматически сгенерированные try/catch-блоки даже к функциям, тела которых не генерируют исключений.

Подведем итог данному обсуждению о снижении производительности программы *при использовании спецификаций исключений, упомянув и об увеличении времени*

разработки — из-за повышения степени связности. Например, удалив один тип из списка в спецификации исключений виртуальной функции базового класса, мы заставим компилятор ругаться на перекрытия этой функции в производных классах (разработкой которых занимались ваши коллеги). Попробуйте сделать это как-нибудь вечером в пятницу, а в понедельник утром посчитайте количество гневных писем, пришедших вам по электронной почте.

В связи с эти вполне логичным представляется следующий вопрос.

5. Когда стоит использовать спецификацию исключений в функции? Почему вы используете (или не используете) эту возможность?

Вот, пожалуй, наилучшие советы, рожденные опытом всего сообщества программистов на C++.

➢ Рекомендация

Совет №1. Никогда не указывайте спецификации исключений.

Совет №2. Возможное исключение из совета №1 — это пустая спецификация исключений, но на вашем месте я бы избегал и ее.

Опыт разработчиков Boost свидетельствует о том, что единственное место, где спецификация исключений может дать некоторое преимущество на некоторых компиляторах — это пустая спецификация исключений у невстраиваемой функции. Это невеселое заключение, но его стоит иметь в виду, если вы намереваетесь писать переносимый код, который будет использоваться на разных компиляторах.

На практике все еще хуже, поскольку оказывается, что распространенные реализации C++ ухитряются по-разному обрабатывать спецификации исключений. Как минимум один популярный компилятор C++ (Microsoft — до текущей на момент написания этой задачи версии 7.1 (2003)) выполняет синтаксический анализ спецификаций исключений, но в действительности не учитывает их в работе, превращая их по сути в некоторое подобие комментариев. Но это еще не все. Например, оптимизация, которую выполняет компилятор Microsoft C++ 7.x, полагается на то, что спецификация исключений будет обеспечена внутри функции. Идея заключается в том, что если функция попытается сгенерировать нечто запретное, то внутренний обработчик остановит выполнение программы и управление никогда не вернется вызывающей функции. Так что если контроль возвращается вызывающей функции, то можно считать, что генерации исключений не было, а значит, в этом случае можно вообще убрать внешние `try/catch`-блоки.

В таком компиляторе, который, с одной стороны, не обеспечивает выполнение спецификаций исключений, а с другой — опирается в своей работе на то, что они будут выполнены, — значение спецификации исключений `throw()` изменено. В результате вместо соответствующего стандарту действия "проверь меня и останови, если я нечаянно что-то сгенерирую" выполняется следующее — "поверь мне, я никогда ничего не сгенерирую, так что можешь оптимизировать мой вызов". Поэтому будьте осторожны — даже при использовании пустой спецификации исключений ознакомьтесь с документацией на компилятор и проверьте, как именно он обрабатывает эту ситуацию. В противном случае поведение компилятора может оказаться для вас неприятным сюрпризом.

Резюме

Коротко: не утруждайте себя написанием спецификаций исключений.
Более подробно:

- Спецификации исключений могут привести к неожиданному изменению производительности программы, например, если компилятор не делает функции со спецификациями исключений встраиваемыми.

- Вызов функции `unexpected` во время выполнения программы — далеко не всегда желательный способ обработки ошибок, перехватить которые призвана спецификация исключений.
- В общем случае вы не в состоянии написать корректную спецификацию исключений для шаблона функции, поскольку не знаете, какие исключения могут сгенерировать типы, с которыми будет работать ваш шаблон.

Когда данный материал не так давно был представлен мною на конференции, я спросил, кто из присутствующих использовал в своей практике спецификации исключений. Утвердительно ответила половина слушателей. Тогда один из слушателей сказал, что мне надо было бы спросить, сколько из этих людей теперь откажутся от них, что я и сделал. И поднялось примерно то же количество рук. Это весьма показательно. Разработчики библиотеки Boost, программисты мирового класса, имеющие большой опыт работы со спецификациями исключений, наилучшей стратегией считают отказ от этой возможности [BoostES].

Многие люди с благими намерениями хотели видеть спецификации исключений в языке — вот почему они там оказались. А дальше ситуация разворачивалась, как во множестве анекдотов, повествующих о том, как волшебника, золотую рыбку или кого-то еще незадачливый герой просил исполнить его желание, но когда он получал желаемое, которое строго соответствовало его просьбе, вдруг оказывалось, что это совсем не то, чего хотел этот герой.

Будьте умеренны в своих желаниях — ведь вы можете получить то, что просите!

Разработка классов, наследование и полиморфизм

Помимо парадигмы обобщенного программирования, C++ поддерживает объектно-ориентированное проектирование и программирование. В этом разделе мы обратимся к этой более традиционной области, уделив основное внимание объектно-ориентированным возможностям C++.

Мы начнем с примера реального кода, в котором есть один тонкий изъян, и используем его в качестве трамплина для обзора порядка конструирования и деструкции объектов.

Затем мы обратимся к вопросам создания надежного кода, которые пересекаются с вопросами безопасности. Какая часть класса доступна из другого кода? Как осуществить "утечку" закрытой части класса, причем не столь важно, непреднамеренно или специально? Что такое инкапсуляция и как она соотносится с выбором прав доступа к членам? Наконец, как сделать наши классы более удобными с точки зрения управления версиями, а также с легко поддерживаемым интерфейсом, который не может быть случайно или преднамеренно разрушен в порожденных классах, что могло бы привести к неработоспособности и брешам в системе безопасности?

Итак, погрузимся в море классов и объектов...

Задача 14. К порядку! Сложность: 2

У программистов, изучающих C++, часто возникают неправильные представления о том, что можно и чего нельзя делать в C++. В приведенном ниже примере, представленном Яном Кристианом ван Винклем, студенты допускают фундаментальную ошибку — но многие компиляторы пропускают ее даже без предупреждений.

Вопрос для новичка

1. Приведенный далее код был действительно написан студентом, изучающим C++, и компилятор, которым он пользовался, не выдал никакого предупреждения (более того, так поступает целый ряд популярных компиляторов). Так что же не верно в приведенном коде и почему?

```
#include <string>
using namespace std;
class A {
public:
    A( const string& s ) { /* ... */ }
    string f() { return "hello, world"; }
};
class B : public A {
public:
    B() : A( s = f() ) {}
private:
    string s;
};
int main() {
    B b;
}
```

Вопрос для профессионала

2. В каком порядке выполняется инициализация различных частей создаваемого объекта класса в C++? Будьте предельно точны в своем ответе. Укажите порядок инициализации различных частей объекта класса X в следующем примере.

```
class B1 { };
class V1 : public B1 { };
class D1 : virtual public V1 { };
class B2 { };
class B3 { };
class V2 : public B1, public B2 { };
class D2 : public B3, virtual public V2 { };
class M1 { };
class M2 { };
class X : public D1, public D2 {
    M1 m1_;
    M2 m2_;
};
```

Решение

1. **...Так что же не верно в приведенном коде и почему?**

```
// Пример 14-1
//
// ...
    B() : A( s = f() ) {}
// ...
```

В указанной строке проявляются две взаимосвязанные проблемы, относящиеся ко времени жизни объекта и его использованию до создания. Обратите внимание, что выражение `s = f()` является аргументом конструктора подобъекта базового класса `A` и, следовательно, выполняется до того, как будет построен базовый подобъект `A` (или любая часть объекта `B`).

Во-первых, эта строка кода пытается использовать еще не существующий базовый подобъект `A`. Компилятор студента, написавшего этот код, не замечал некорректного использования `A::f`, состоящего в том, что функция `f` вызывается для подобъекта `A`, который еще не сконструирован. Компилятор не обязан диагностировать такие ошибки; однако это то, что называется "качеством реализации", и достаточно хороший компилятор вполне мог бы заметить данную ошибку.

Во-вторых, здесь же выполняется попытка использовать член `s` подобъекта, который еще не существует, т.е. применить оператор присваивания к строке-члену объекта, конструирование которого еще не завершено.

2. В каком порядке выполняется инициализация различных частей создаваемого объекта класса в C++? Будьте предельно точны в своем ответе.

Порядок инициализации определяется рекурсивным применением следующего набора правил.

- Сначала конструктор последнего производного класса вызывает конструкторы подобъектов виртуальных базовых классов. Инициализация виртуальных базовых классов выполняется в глубину, в порядке слева направо.
- Затем конструируются подобъекты непосредственных базовых классов в порядке их объявления в определении класса.
- После этого конструируются (нестатические) подобъекты-члены в порядке их объявления в определении класса.
- И наконец, выполняется тело конструктора.

В качестве примера рассмотрим приведенный в задаче код. Вид наследования (открытое, закрытое или защищенное) не влияет на порядок инициализации, так что все наследование показано мной как открытое.

Укажите порядок инициализации различных частей объекта класса X в следующем примере.

```
// Пример 14-2
//
class B1 { };
class V1 : public B1 { };
class D1 : virtual public V1 { };
class B2 { };
class B3 { };
class V2 : public B1, public B2 { };
class D2 : virtual public V2, public B3 { };
class M1 { };
class M2 { };
class X : public D1, public D2 {
    M1 m1_;
    M2 m2_;
};
```

Иерархия наследования имеет структуру, показанную на рис. 14.1.

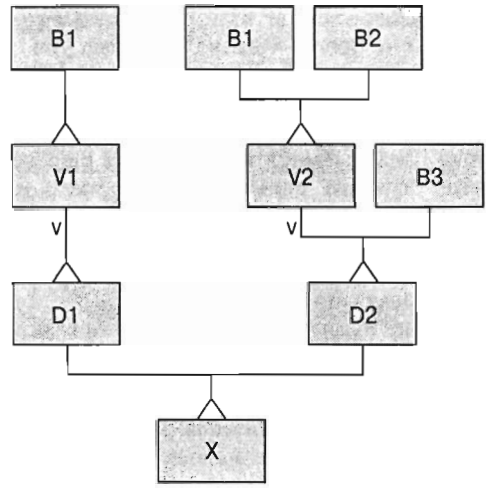

Рис. 14.1. Иерархия наследования в примере 14-2

Порядок инициализации объекта `X` в примере 14-2 имеет следующий вид (каждый показанный ниже вызов конструктора представляет выполнение его тела):

- Сначала конструируются виртуальные базовые классы:
  конструирование `V1`: `B1::B1()` `V1::V1()`
  конструирование `V2`: `B1::B1()` `B2::B2()` `V2::V2()`
- Затем конструируются невиртуальные базовые классы:
  конструирование `D1`: `D1::D1()`
  конструирование `D2`: `B3::B3()` `D2::D2()`
- После этого конструируются члены: `M1::M1()` `M2::M2()`
- И наконец, выполняется конструктор `X`: `X::X()`

Резюме

Хотя основная цель данной задачи — достичь лучшего понимания порядка конструирования объектов (и их деструкции — в обратном порядке), это не мешает нам повторить одно правило, имеющее некоторое отношение к данной теме.

➢ Рекомендация

Не злоупотребляйте наследованием.

Если не учитывать отношения дружбы, наследование представляет собой наиболее сильную взаимосвязь, имеющуюся в C++, и использовать ее следует только там, где это действительно необходимо. Более подробно этот вопрос рассмотрен в книгах [Sutter00] и [Sutter02].

Задача 15. Потребление и злоупотребление правами доступа

Сложность: 6

Кто в действительности имеет право доступа к внутренней организации вашего класса? Эта задача — о фальсификаторах, мошенниках, карманниках и ворах, а также о том, как их распознать и спастись от них.

Вопрос для новичка

1. Какой код может обращаться к следующим частям класса:
   а) `public`
   б) `protected`
   в) `private`

Вопрос для профессионала

2. Рассмотрим следующий заголовочный файл.

```
// Файл x.h
//
class X {
public:
    X() : private_(1) { /*...*/ }
    template<class T>
    void f( const T& t ) { /*...*/ }
    int Value() { return private_; }
    // ...
private:
    int private_;
};
```

   Покажите, как произвольный вызывающий код может получить непосредственный доступ к закрытому члену класса `private_`:
   а) при помощи нестандартного и непереносимого "хака";
   б) при помощи переносимой и соответствующей стандарту методики.
3. Подумайте, не является ли это "дырой" в механизме управления правами доступа в C++ (а следовательно, дырой в инкапсуляции C++)?

Решение

Эта задача — о фальсификаторах, обманщиках, мошенниках и ворах...

1. **Какой код имеет право доступа к следующим частям класса:**

   а) `public`

   Возможность обращения к открытым членам имеет любой код.

   б) `protected`

   Возможность обращения к защищенным членам имеют только функции-члены того же класса и его друзья, а также функции-члены и друзья производных классов.

   в) `private`

   Возможность обращения к закрытым членам имеют только функции-члены того же класса и его друзья.

Это обычный ответ, и обычно это правильно. Но в этой задаче мы рассмотрим специальный случай, когда этот ответ не полон, поскольку иногда C++ предоставляет возможность законного (пусть и аморального) нарушения прав доступа к закрытым членам.

2. Рассмотрим следующий заголовочный файл ... Покажите, как произвольный вызывающий код может получить непосредственный доступ к закрытому члену класса `private_`:

а) **при помощи нестандартного и непереносимого "хака";**

б) **при помощи переносимой и соответствующей стандарту методики.**

Есть странная притягательность в самых трагических сценах, когда мы смотрим не отрываясь на автомобильные аварии, падающие небоскребы и... взлом кода.

При попытке ответить на вопрос о том, как обратиться к закрытому члену нестандартными и непереносимыми методами, в голову приходит целый ряд идей. Вот три из них.

Преступник №1: фальсификатор

Метод фальсификатора состоит в том, чтобы продублировать определение фальсифицируемого класса, в которое внесены все необходимые нам изменения, например:

```
// Пример 15-1: ложь и подделка
//
class X {
    // Вместо включения файла x.h, вручную (и незаконно)
    // дублируем определение X и добавляем строку наподобие
    // этой:
    friend ::Hijack( X& );
};
void Hijack( X& x ) {
    x.private_ = 2;       // Злобный смех
}
```

Этот человек — фальсификатор. Запомните его хорошенько, ему нельзя доверять...

Конечно, то, что сделано, — не законно. Это не законно хотя бы потому, что нарушает правило одного определения, которое гласит, что если тип (в нашем случае — `X`) определен более одного раза, то все его определения должны быть идентичны. Используемый же в приведенном примере объект хотя и называется `X` и выглядит похожим на `X`, но это не тот же `X`, который используется остальным кодом программы. Тем не менее, такой "хак" будет работать на большинстве компиляторов, поскольку расположение данных объекта останется неизменным.

Преступник №2: карманник

Карманник сработает тихо — подменив смысл определения класса, например, следующим образом.

```
// Пример 15-2: использование макромагии
//
#define private public          // Не законно!
#include "x.h"
void Hijack( X& x ) {
    x.private_ = 2;             // Злобный смех
}
```

Этот человек — карманник. Запомните его умелые пальцы!

Конечно, то, что делает карманник, — не законно. Код из примера 15-2 непереносим по двум причинам.

- Не законно переопределять при помощи директивы `#define` зарезервированные слова.

- При этом происходит такое же нарушение правила одного определения, как и у фальсификатора. Однако если расположение данных объекта остается неизменным, этот способ сработает.

Преступник №3: мошенник

Принцип работы мошенника — в подмене понятий.

```
// Пример 15-3: попытка имитации размещения данных объекта
//
class BaitAndSwitch {// В надежде, что размещение данных в
public:              // этом классе будет тем же, что и в X
    int notSoPrivate;
};
void f( X& x ) {
    (reinterpret_cast<BaitAndSwitch&>(x)).notSoPrivate = 2;
}                                              // Злобный смех
```

Этот человек — мошенник. Хорошенько его запомните! Его реклама рассчитана только на то, чтоб затащить вас в магазин, а уж там он обязательно ухитриться продать вам совершенно не ту вещь, о которой шла речь в рекламе, да еще и в несколько раз дороже, чем в любом другом месте.

Конечно, то, что делает мошенник, — не законно. Код из примера 15-3 не законен по двум причинам.

- Нет гарантии, что размещение объектов `X` и `BaitAndSwitch` в памяти будет одинаковым — хотя на практике это обычно так и есть.
- Результат `reinterpret_cast` не определен, хотя большинство компиляторов сделают именно то, чего от них хочет мошенник. В конце концов, говоря волшебное слово `reinterpret_cast`, вы заставляете компилятор поверить вам и закрыть глаза на подготавливаемое вами мошенничество.

Но вопрос не исчерпывался только обманными способами. Если помните, нас спрашивали и о том, как добиться интересующего результата совершенно корректно и переносимо с точки зрения стандарта. Увы, крупные преступники имеют весьма респектабельный вид и большие счета в банках, так что их очень трудно схватить за руку.

Персона грата №4: адвокат

Многие из нас недаром больше боятся хорошо одетых и улыбающихся адвокатов, чем (других) преступников.

Рассмотрим следующий код.

```
// Пример 15-4: проныра-законник
//
namespace {
    struct Y {};
}
template<>
void X::f( const Y& ) {
    private_ = 2;                  // Злобный смех
}
void Test() {
    X x;
    cout << x.Value() << endl;  // Выводит 1
    x.f( Y() );
    cout << x.Value() << endl;  // Выводит 2
}
```

Этот человек — адвокат, который знает все лазейки. Его невозможно поймать, поскольку он слишком осторожен, чтобы нарушить букву закона, при этом нарушая его дух. Запомните и избегайте таких неджентльменов.

Как бы мне ни хотелось сказать "Конечно, то, что делает адвокат, — не законно", увы, я не могу этого сделать, поскольку все сделанное в последнем примере законно. Почему? В примере 15-4 использован тот факт, что у X есть шаблон функции-члена. Приведенный код соответствует стандарту, так что последний гарантирует, что он будет работать так, как ожидается. Причины здесь две.

- Можно специализировать шаблон-член для любого типа.

Единственное место, где могла бы проявиться ошибка, — это две разные специализации для одного и того же типа, что нарушало бы правило одного определения. Но и тут все оказывается в порядке:

- Код использует гарантированно уникальный тип, поскольку он находится в безымянном пространстве имен. Следовательно, гарантируется, что приведенный код корректен и не будет конфликтовать ни с какой другой специализацией.

Не нарушай

Остается только один вопрос.

3. Подумайте, не является ли это "дырой" в механизме управления правами доступа в C++ (а следовательно, дырой в инкапсуляции C++)?

Данный пример демонстрирует интересное взаимодействие двух возможностей C++: модель управления правами доступа и модель шаблонов. В результате оказывается, что шаблоны-члены неявно нарушают инкапсуляцию в том смысле, что они предоставляют переносимую возможность обхода механизма управления доступом к классу.

В действительности не это является проблемой. Проблема в "защите от дурака", т.е. от непреднамеренного случайного использования (с чем язык справляется очень хорошо) и в защите от намеренного злоупотребления (против чего эффективная защита невозможна). В конечном итоге, если программист намерен нарушить систему защиты, он найдет способы это сделать, как продемонстрировано в примерах с 15-1 по 15-3.

Правильный ответ на вопрос — не делайте этого! По общему признанию, бывают ситуации, когда хочется иметь быстрый способ временного обхода механизма контроля прав доступа, например, при отладке... но это не просто плохая привычка, которая может привести к использованию таких методов в окончательной версии кода. В конечном итоге это приведет к более серьезным неприятностям.

➢ Рекомендация

Никогда не "коверкайте" язык программирования. Например, не пытайтесь нарушить инкапсуляцию путем копирования определения класса с добавлением друга или при помощи локального инстанцирования шаблона функции-члена.

Задача 16. Крепко закрыт? Сложность: 5

В какой степени закрыты закрытые части класса в C++? Благодаря этой задаче мы увидим, что закрытые имена определенно недоступны для внешнего кода, не являющегося дружеским, но, тем не менее, имеются некоторые пути, по которым до них можно дотянуться — как хорошо известные, так и не очень.

Вопрос для профессионала

1. Ответьте быстро. Предположим, что функции `Twice` определены в другой единице трансляции, подключаемой при компоновке. Будет ли приведенная далее программа на C++ компилироваться и корректно выполняться? Если нет, то почему? Если да, то что она даст на выходе?

```
// Twice(x) возвращает 2*x
//
class Calc {
public:
    double Twice( double d );
private:
    int    Twice( int i );
    std::complex<float> Twice( std::complex<float> c );
};
int main() {
    Calc c;
    return c.Twice( 21 );
}
```

Решение

В основе решения лежит упомянутый вопрос — до какой степени в действительности закрыты закрытые части класса, такие как функции `Twice` в данной задаче?

Доступность

Главное, что требуется осознать, — это то, что `private` так же, как `public` и `protected`, представляют собой спецификаторы доступа, т.е. они управляют тем, в какой степени другой код может обращаться к именам членов — и не более того. Процитируем стандарт [C++03].

Член класса может быть:

- *закрытым (`private`), т.е. его имя может использоваться только членами и друзьями класса, в котором он объявлен;*
- *защищенным (`protected`), т.е. его имя может использоваться только членами и друзьями класса, в котором он объявлен, а также членами и друзьями классов, производных от данного;*
- *открытым (`public`), т.е. его имя может использоваться везде, без каких-либо ограничений доступа.*

Это базовый материал, но давайте для полноты рассмотрим простой пример, демонстрирующий, как обеспечивается проверка прав доступа, и убедимся, что нет стандартных путей, позволяющих обойти этот механизм. В примере 16-1 показано, что код вне класса, не являющийся его другом, не в состоянии обратиться к закрытой функции по имени ни непосредственно (путем явного вызова), ни косвенно (через

указатель на функцию), поскольку имя функции не может быть использовано никаким образом, включая получение адреса функции[25]:

```
// Пример 16-1: доступ к No::Satisfaction невозможен
//
class No {
private:
    virtual void Satisfaction() { }
};
int main() {
    No no;
    no.Satisfaction();                // Ошибка
    typedef void (No::*PMember)();
    PMember p = &No::Satisfaction;    // Ошибка
    return (no.*p)();                  // Ну-ну...
}
```

Обратите внимание на то, что в этом примере рассматривается виртуальная функция, и права доступа распространяются и на нее. Закрытый член, представляющий собой виртуальную функцию, может быть перекрыт в любом производном классе, но доступ к нему из производного класса невозможен. Точнее, производный класс может перекрыть любую виртуальную функцию своей собственной функцией с тем же именем, но не может вызвать или каким-либо иным образом использовать имя закрытой виртуальной функции из базового класса, например:

```
// Пример 16-1, продолжение: производный класс может
//              перекрыть закрытый виртуальный член, но не
//              может к нему обратиться
//
class Derived : public No {
    virtual void Satisfaction() {    // Корректно
        No::Satisfaction();          // Ошибка
    }
};
```

Нет никакого способа, которым бы внешний код (не являющийся членом или другом) мог воспользоваться именем этой функции. Итак, на вопрос о том, насколько закрыты закрытые члены, у нас готова первая часть ответа.

- Закрытое (`private`) имя члена *доступно* только для других членов и друзей.

Если бы на этом все и заканчивалось, это была бы самая короткая (и бессмысленная) задача в книге. Но, как вы понимаете, на доступности имени наш рассказ не заканчивается.

Видимость

Ключевое слово `private` управляет *доступностью* членов, но есть еще одна связанная с ней концепция, которую часто путают с доступностью, а именно *видимость*. Вернемся к коду, приведенному в вопросе задачи: будет ли он компилироваться и корректно выполняться?

Краткий ответ — нет. В приведенном виде программа некорректна и компилироваться не будет по двум причинам. Первая причина — довольно очевидная ошибка.

```
// Пример 16-2 (код из условия задачи)
//
// Twice(x) возвращает 2*x
//
class Calc {
public:
```

[25] Мошеннические способы наподобие подмены ключевого слова `private` словом `public` (см. задачу 15) мы не рассматриваем.

```
    double Twice( double d );
private:
    int    Twice( int i );
    std::complex<float> Twice( std::complex<float> c );
};
int main() {
    Calc c;
    return c.Twice( 21 );
}
```

Каждый программист на C++ знает, что несмотря на то, что версия `Twice`, работающая с комплексным объектом, не *доступна* коду в функции `main`, она остается *видима* и создает зависимость на уровне исходного текста. В частности, несмотря на то, что код функции `main`, вероятно, ничего не знает о типе `complex`, он не в состоянии использовать имя `Twice(complex<float>)` (ни вызвать эту функцию, ни получить ее адрес), кроме того, использование `complex` никоим образом не может повлиять на размер `Calc` или его размещение в памяти, но все равно для компиляции этого кода требуется, как минимум, предварительное объявление `complex`. (Если бы функция `Twice(complex<float>)` была определена как встраиваемая, то требовалось бы также полное определение типа `complex`, несмотря на невозможность вызова этой функции.)

Итак, мы получили следующую часть ответа на вопрос о закрытых членах.

- Закрытые члены *видимы* всему коду, которому видно определение класса. Это означает, что типы параметров закрытых функций должны быть объявлены, даже если они никогда не используются в данной единице трансляции.

Все знают, что исправить первую ошибку очень просто — добавив `#include <complex>`. Теперь у нас останется только одна, но менее очевидная проблема.

```
// Пример 16-3: частично исправленный пример 16-2
//
#include <complex>
class Calc {
public:
    double  Twice( double d );
private:
    int     Twice( int i );
    std::complex<float> Twice( std::complex<float> c );
};
int main() {
    Calc c;
    return c.Twice( 21 ); // Ошибка, Twice недоступна
}
```

Этот результат оказывается неожиданным для большого количества программистов на C++. Некоторые программисты ожидают, что, поскольку доступна перегрузка функции `Twice`, которая принимает параметр типа `double`, а число 21 можно привести к этому типу, то именно эта функция должна быть вызвана. На самом деле это не так по очень простой причине: разрешение перегрузки выполняется до проверки доступности.

Когда компилятор должен разрешить вызов функции `Twice`, он выполняет следующие три вещи в указанном порядке.

1. *Поиск имен.* Перед тем как приступить к другим задачам, компилятор находит область действия, в которой имеется как минимум один объект с именем `Twice`, и составляет список возможных кандидатов для вызова. В нашем случае поиск имен производится сначала в области действия `Calc`, чтобы выяснить, имеется ли хотя бы один член с таким именем. Если такого члена нет, будут по одному рассматриваться базовые классы и охватывающие пространства имен, до тех пор, пока не будет найдена область действия, содержащая как минимум одного кандидата. В нашем случае, однако, уже первая просмотренная компилятором область действия

содержит объект по имени Twice — и даже не один, а три, и все они попадают в список кандидатов. (Дополнительная информация о поиске имен в C++ и о его влиянии на разработку классов и их интерфейсов есть в [Sutter00].)

2. *Разрешение перегрузки.* Затем компилятор выполняет разрешение перегрузки для того, чтобы выбрать единственного кандидата из списка, наилучшим образом соответствующего типам аргументов вызова. В нашем случае передаваемый функции аргумент — 21, тип которого int, а функции из списка кандидатов принимают параметры типов double, int и complex<float>. Очевидно, что реально передаваемый параметр наилучшим образом соответствует аргументу типа int (точное соответствие, не требующее преобразования типов), так что для вызова выбирается функция Twice(int).
3. *Проверка доступности.* И наконец, компилятор выполняет проверку доступности для определения того, может ли быть вызвана выбранная функция. В нашем случае... ну, вы сами понимаете, что происходит в нашем случае.

Не играет никакой роли, что есть функция Twice(double), которая могла бы быть вызвана, поскольку имеется лучшее, чем у нее, соответствие типа параметра, а степень соответствия всегда превалирует над доступностью.

Интересно, что неоднозначное соответствие типам параметров играет большую роль, чем доступность. Рассмотрим небольшое изменение примера 16-3.

```
// Пример 16-4(а): внесение неоднозначности
//
#include <complex>
class Calc {
public:
    double    Twice( double d );
private:
    unsigned  Twice( unsigned i );
    std::complex<float> Twice( std::complex<float> c );
};
int main() {
    Calc c;
    return c.Twice( 21 ); // Ошибка - вызов Twice
                          // неоднозначен
}
```

В этом случае мы не сможем пройти второй шаг. Разрешение перегрузки не может найти наилучшее соответствие в списке кандидатов, поскольку аргумент может быть преобразован как в unsigned, так и в double, и, в соответствии с правилами языка, эти два преобразования одинаково хороши. Поскольку эти две функции имеют одинаковую степень соответствия, компилятор не в состоянии выбрать одну из них и сообщает о неоднозначности вызова. В результате в этом случае компилятор так и не доберется до проверки доступности.

Пожалуй, еще интереснее ситуация, когда невозможность соответствия играет большую роль, чем доступность. Рассмотрим еще один вариант примера 16-3.

```
// Пример 16-4(б): сокрытие имени глобальной функции
//
#include <string>
int Twice( int i );           // Глобальная функция
class Calc {
private:
    std::string Twice( std::string s );
public:
    int Test() {
        return Twice( 21 ); // Ошибка, Twice(string) не
                            // подходит
    }
};
int main() {
```

```
    return Calc().Test();
}
```

Здесь мы также не можем пройти второй шаг: разрешение перегрузки не сможет найти ни одной соответствующей функции в списке кандидатов (в котором теперь находится единственная функция `Calc::Twice(string)`), поскольку аргумент типа `int` не может быть преобразован в `string`. Компилятор просто не доберется до проверки доступности. Запомните, как только найдена область действия, в которой имеется хотя бы один объект с заданным именем, поиск завершается, даже если кандидат(ы) оказываются не соответствующими типам аргументов и не могут быть вызваны и/или оказываются недоступны. Поиск других кандидатов в охватывающей области действия проводиться не будет[26].

Итак, мы получили еще одну часть ответа на вопрос о закрытых членах.

- Закрытый член *видим* всему коду, которому видно определение класса. Это означает, что он участвует в поиске имен и разрешении перегрузки и, таким образом, может сделать вызов некорректным или неоднозначным несмотря на то, что сам он, в любом случае, не может быть вызван.

Бьярн Страуструп так писал об этом в своей книге [Stroustrup94]:

"Если бы спецификации `public/private` в первую очередь управляли не доступностью, а видимостью, изменение доступности члена с `public` на `private` могло бы привести к незаметному изменению смысла программы — от одной корректной интерпретации [в нашем примере вызов `Calc::Twice(int)`] *к другой* [в нашем примере вызов `Calc::Twice(double)`]. *Это не решающий аргумент, но принятое решение позволяет программистам в процессе отладки добавлять и удалять спецификаторы `public` и `private`, не опасаясь, что программа незаметно изменит свой смысл. Я до сих пор гадаю, не явилось ли это решение неким озарением".*

И снова доступность

Я уже говорил о том, что закрытое имя члена *доступно* (может использоваться) только для других членов и друзей. Обратите внимание, что я преднамеренно не сказал "может быть вызван только другими членами или друзьями", поскольку на самом деле это не так. Доступность определяет, вправе ли код использовать имя. Позвольте мне подчеркнуть это цитатой из стандарта.

Член класса может быть

- *закрытым (`private`); т.е. **его имя** может использоваться только членами и друзьями класса, в котором он объявлен.*

Если код, который имеет право непосредственного использования имени (в данном случае член класса или друг) использует это имя для создания указателя на функцию, а затем передает его другому коду, то получивший его код может воспользоваться этим указателем независимо от того, имеет ли он право на использование имени — имя ему больше не нужно, так как у него есть указатель. Пример 16-5 иллюстрирует применение этого способа на практике — функция-член, имеющая доступ к имени `Twice(int)`, использует этот доступ для передачи указателя на данный член.

```
// Пример 16-5: передача доступа
//
class Calc;
typedef int (Calc::*PMember)(int);
class Calc {
public:
    PMember CoughItUp() { return &Calc::Twice; }
```

[26] Поиск прекращается даже в тех случаях, когда найденное во время поиска имя не является именем функции. — *Прим. ред.*

```
private:
    int Twice( int i );
};
int main() {
    Calc c;
    PMember p = c.CoughItUp(); // Дает доступ к Twice(int)
    return (c.*p)( 21 );       // ok
}
```

См. также [Newkirk97], где описывается полезное применение преднамеренной утечки деталей закрытой реализации.

Итак, мы получили еще одну часть ответа на вопрос о закрытых членах.

- Код, который имеет доступ к члену, может передать этот доступ любому другому коду, передав ему указатель на этот член.

И наконец, имеется еще один способ, которым некоторые классы могут переносимо и в полном соответствии со стандартом дать всем права доступа к закрытому члену: это шаблоны членов. В задаче 15 мы рассмотрели несколько способов получения доступа к закрытым частям класса. Большинство из них не законны , но один из них стопроцентно соответствует стандарту.

```
// Пример 16-6: адаптированный пример 15-4
//
// В заголовочном файле
//
class X {
public:
    template<class T>
    void f( const T& t ) {
        // ...
    }
    // ...
private:
    int private_;
};

// В пользовательском коде
//
namespace {
    struct Y {};
}
template<>
void X::f( const Y& ) {
    private_ = 2;  // Злобный смех
}
```

Что здесь произошло? Любой шаблон члена может быть специализирован для любого типа. Специализируя его для какого-то своего уникального типа (уникальность обеспечивается безымянным пространством имен), вы создаете собственную функцию-член, а член класса имеет доступ ко всем частям класса.

Итак, вот последняя часть ответа на вопрос о закрытых членах:

- Имя закрытого члена *доступно* только другим членам (включая явные инстанцирования шаблонов членов) и друзьям.

Резюме

Итак, насколько же закрыты закрытые члены? Вот что мы выяснили.

Имя закрытого члена *доступно* только другим членам (включая явные инстанцирования шаблонов членов) и друзьям. Но код, который имеет доступ к члену, может передать эти права доступа любому другому коду посредством указателя на данный член.

Закрытый член *видим* любому коду, которому видно определение класса. Это означает, что типы его параметров должны быть объявлены, даже если они не используются в единице трансляции, и что он может сделать вызов некорректным или неоднозначным, несмотря на то, что сам вызван быть не может.

Задача 17. Инкапсуляция Сложность: 4

Что именно представляет собой инкапсуляция в применении к программированию на C++? В чем именно смысл инкапсуляции и управления доступом для членов-данных — должны ли они иногда быть открытыми или защищенными? В этой задаче мы получим ответы на приведенные вопросы и узнаем, как эти ответы могут повлиять на надежность кода.

Вопрос для новичка

1. Что такое инкапсуляция? Насколько она важна в объектно-ориентированном проектировании и программировании?

Вопрос для профессионала

2. В каких случаях (если таковые имеются) нестатические члены данных должны быть сделаны открытыми (`public`), защищенными (`protected`) и закрытыми (`private`)? Выразите ваш ответ в виде рекомендации программисту.
3. Шаблон класса `std::pair` использует открытые члены-данные, поскольку он не инкапсулирует никаких данных, а просто позволяет их группировать. Представим шаблон класса наподобие `std::pair`, но с тем отличием, что у него дополнительно имеется флаг `deleted`, который может быть установлен и опрошен (но не сброшен). Ясно, что такой флаг должен быть закрытым для защиты от непосредственного изменения пользователем. Если остальные члены-данные будут открытыми, как в `std::pair`, в конечном итоге мы получим примерно следующий код.

```
template<class T, class U>
class Couple {
public:
    // Основные члены-данные открыты...
    T first;
    U second;
    // ...но есть еще нечто, делающее его более
    // "классоподобным", а также закрытая реализация
    Couple()           : deleted_(false) {}
    void MarkDeleted() { deleted_ = true; }
    bool IsDeleted()   { return deleted_; }
private:
    bool deleted_;
};
```

Должны ли в этом случае остальные члены-данные быть, как показано в коде, открытыми? Почему? Если да, то является ли этот код хорошим примером того, почему иногда одновременное наличие открытых и закрытых данных в одном классе может быть удачным решением?

Решение

1. Что такое инкапсуляция?

В словаре Вебстера приведено следующее определение инкапсуляции:
окружать, упаковывать или защищать, или находиться в капсуле, оболочке.

Инкапсуляция в программировании имеет тот же смысл — защита внутренней реализации класса путем ее сокрытия за внешним интерфейсом, видимым внешнему миру.

Определение слова "капсула", в свою очередь, дает нам указания о том, каким должен быть хороший интерфейс класса (здесь приведены не все возможные значения этого слова):

капсула —

1. структура наподобие мембраны или мешка, окружающая часть органа или орган в целом;

2. закрытый контейнер, содержащий споры или семена; ...

4. желатиновая оболочка вокруг лекарственного препарата; ...

металлическая пломба; ...

6. оболочка вокруг некоторых микроскопических организмов; ...

9. небольшое герметизированное помещение для летчика или космонавта. ...

Обратите внимание на однотипность определений — охватывает, окружает, упаковывает, запечатывает...

Хороший интерфейс класса скрывает внутренние детали его реализации, представляя внешнему миру только "лицо", которое отделено от "внутренностей". Поскольку капсула охватывает одну связанную группу подобъектов, ее интерфейс также должен быть связанным — все его части должны иметь непосредственное отношение друг к другу.

Хороший интерфейс класса должен быть полным и не показывать вовне никаких деталей внутренней реализации класса. Он работает как герметичная оболочка или как брандмауэр (времени компиляции, времени выполнения или и того, и другого одновременно), так что внешний код не может зависеть от внутренней реализации класса, и любые изменения внутреннего строения и реализации класса никак не влияют на внешний использующий этот класс код. Бактерия, оболочка которой не замкнута, проживет недолго...

Хороший интерфейс класса защищает его "внутренности" от неавторизованного доступа и манипуляций. В частности, главная задача интерфейса — гарантировать, что любое обращение к внутренним структурам класса и работа с ними сохраняют инварианты класса.

Основной способ убить бактерию (как и, скажем, человека) — использование устройств, которые нарушают целостность ее внешней и/или внутренней капсул. На микроуровне это могут быть химические реактивы, ферменты или организмы (возможно, даже наномеханизмы), способные проделать в оболочке соответствующие отверстия. На макроуровне предпочтение отдается ножам и автоматам...

Место инкапсуляции в объектно-ориентированном программировании

Насколько она важна в объектно-ориентированном проектировании и программировании?

Инкапсуляция — основная концепция объектно-ориентированного программирования. Точка.

Прочие объектно-ориентированные методы — такие как сокрытие данных, наследование и полиморфизм — важны в первую очередь потому, что они выражают частные случаи инкапсуляции.

- Инкапсуляция практически всегда влечет за собой *сокрытие данных.*
- *Полиморфизм* времени выполнения, использующий виртуальные функции, более полно отделяет интерфейс (представленный базовым классом) от реализации (которая представляется производным классом, который может даже не существовать в тот момент, когда разрабатывается код, который будет его использовать).
- *Полиморфизм* времени компиляции, использующий шаблоны, полностью отделяет интерфейс от реализации, так как любой класс, удовлетворяющий некоторым

требованиям, может быть использован шаблоном. Используемые классы не обязаны быть связаны наследованием или каким-либо иным отношением.

Инкапсуляция — не всегда сокрытие данных, но сокрытие данных — частный случай инкапсуляции. Инкапсуляция — не всегда полиморфизм, но полиморфизм — всегда форма инкапсуляции.

Объектная ориентированность часто определяется следующим образом:

связывание в единое целое данных и функций, которые оперируют с этими данными.

Такое определение справедливо с определенными допущениями — поскольку оно исключает свободные функции (не являющиеся членами), которые являются логической частью класса (такие как оператор << C++). Кроме того, данное определение не подчеркивает другой существенный аспект объектной ориентированности, а именно

одновременное отделение данных от вызывающего кода посредством интерфейса, представляющего собой набор функций, которые работают с этими данными.

Этот дополнительный аспект подчеркивает слабое связывание внешнего кода и данных, а также то, что предназначение собранных функций — формирование интерфейса, защищающего данные.

Кратко можно сказать, что объектно-ориентированное программирование состоит в отделении интерфейсов от реализации, что обеспечивает, с одной стороны, высокую степень единства кода и данных, и низкую степень связывания — с другой. Разработчики программного обеспечения сталкивались с задачей обеспечения такой связи функций и данных задолго до "открытия" объектов. Эти концепции относятся к управлению зависимостями, которое представляет собой одно из ключевых понятий в современном проектировании программного обеспечения, в особенности для больших систем (см. [Martin95] и другие статьи Мартина (Martin), датированные 1996 годом в [ObjectMentor], в особенности те из них, в заголовке которых есть слово "Principle").

Открытые, закрытые или защищенные данные?

2. В каких случаях (если таковые имеются) нестатические члены данных должны быть открытыми (`public`), защищенными (`protected`) и закрытыми (`private`)? Выразите ваш ответ в виде рекомендации программисту.

Обычно мы начинаем с того, что рассматриваем правило, а уже затем — исключения из него. Давайте в этот раз нарушим порядок и сначала рассмотрим исключения, а потом перейдем к правилу.

Единственное исключение из общего правила (которое будет приведено далее) — это когда все члены класса (как функции, так и данные) открытые, как в случае структуры в C. В этом случае класс — не вполне полноценный класс со своим интерфейсом, поведением и инвариантами — словом, это не класс, а набор данных. Такой "класс" — просто удобное объединение объектов в единое целое, и это вполне нормальное явление, в особенности с точки зрения обратной совместимости с программами C, которые работают со структурами.

За исключением этого частного случая, все члены-данные должны быть закрытыми.

Открытые данные — нарушение инкапсуляции, поскольку они позволяют вызывающему коду непосредственно работать со внутренними объектами. Это требует высокой степени доверия! В конце концов, в реальной жизни я вряд ли разрешу кому-либо иметь дело непосредственно с моими внутренностями (скажем, позволив копаться внутри живота), поскольку при этом очень легко, пусть даже непреднамеренно, допустить ошибку, которая может, мягко говоря, повредить моему организму. Другое дело — косвенная работа через открытый интерфейс, когда я могу сам решить, что и как делать (например, получив бутылку с этикеткой "Выпей меня", я сам, опираясь на собственные ощущения от содержимого бутылки и свой здравый смысл, решу, пить ли содержимое бутылки или все же с его помощью вымыть машину). Конечно, есть люди, достаточно квалифицированные, чтобы копаться в моих внутренностях

непосредственно (например, хирург), но даже в этом случае а) это бывает редко, б) я сам решаю, нужна ли мне помощь хирурга и в) я сам решаю, какому хирургу я могу доверить такие драгоценные для меня внутренности.

Аналогично, в большинстве случаев вызывающий код не должен работать со внутренней организацией класса непосредственно (например, просматривая или изменяя члены-данные), поскольку при этом очень легко непреднамеренно совершить неверные действия; в лучшем случае внешний код должен работать с внутренними данными косвенно, при помощи открытого интерфейса класса, когда класс может сам решить, что следует делать с переданными ему параметрами (рассмотренный пример `Бутылка("выпей меня")`) на основании знаний и суждений автора данного класса. Конечно, определенный код может быть специально предназначен для непосредственной работы с внутренней организацией класса (обычно такой код должен быть функцией-членом класса, но, например, `operator<<` делать членом не рекомендуется), но даже в таком случае: а) это бывает редко, б) класс сам решает, объявлять ли кого-либо другом класса или нет, и в) класс сам решает, какой именно код заслуживает достаточного доверия, чтобы его можно было объявить другом и дать ему доступ к внутренней организации класса.

Словом, открытые данные есть зло (кроме данных в структурах в стиле C). Аналогично, защищенные данные — такое же зло, но на этот раз без всяких исключений.

"Подождите минутку, — может возразить кто-то из читателей, — я согласен с вами, когда вы говорите об открытых данных, но почему вы считаете таким же злом защищенные данные?" Потому что те же аргументы, которые были перечислены для открытых данных, применимы и к защищенным данным, которые тоже являются частью интерфейса: защищенный интерфейс также является интерфейсом ко внешнему коду, только к меньшему его подмножеству — а именно коду производных классов. Почему в этом случае нет исключений? Потому что защищенные данные не могут быть просто сгруппированными данными; если бы это было так, то ими могли бы воспользоваться только производные классы, а какой в этом смысл?

Об истории этого вопроса и о том, почему человек, ратовавший за наличие защищенных данных в языке, теперь считает это плохой идеей, можно прочесть в книге [Stroustrup94].

➢ Рекомендация

Всегда делайте все члены-данные закрытыми. Единственное исключение — структура в стиле C, которая не предназначена для инкапсуляции чего бы то ни было, и все члены которой открыты.

Преобразование в общем случае

Давайте теперь докажем, правило "все члены-данные всегда должны быть закрытыми" от противного — предположим, что справедливо обратное утверждение (что имеются ситуации, когда `public/protected` члены-данные могут быть подходящим решением) и покажем, что в каждом таком случае данные не должны быть ни открытыми, ни защищенными.

```
// Пример 17-2(a): не закрытые данные (плохо)
//
class X {
    // ...
public:
    T1 t1_;
protected:
    T2 t2_;
};
```

Для начала заметим, что этот код всегда можно преобразовать без потери общности и эффективности в следующий.

```
// Пример 17-2(б): инкапсулированные данные (хорошо)
//
class X {
    // ...
public:
    T1& UseT1() { return t1_; }
protected:
    T2& UseT2() { return t2_; }
private:
    T1 t1_;
    T2 t2_;
};
```

Таким образом, даже если есть причины для непосредственного доступа к `t1_` или `t2_`, возможно простое преобразование, в результате которого он предоставляется посредством (встраиваемых) функций. Примеры 17-2(а) и 17-2(б) эквивалентны. Однако нет ли каких-то особых преимуществ для использования класса в том виде, как он приведен в примере 17-2(а)?

Для того чтобы обосновать, что метод из примера 17-2(а) никогда не должен использоваться, мы должны показать, что:

1. пример 17-2(а) не имеет никаких преимуществ, которых нет в примере 17-2(б);
2. пример 17-2(б) обладает конкретными преимуществами; и
3. пример 17-2(б) не приводит к дополнительным затратам.

Рассмотрим эти пункты в обратном порядке.

Выполнение пункта 3 показывается тривиально: встраиваемая функция, возвращающая ссылку и, следовательно, не выполняющая копирование, должна оптимизироваться компилятором вплоть до полного устранения ее кода.

Пункт 2 также прост: давайте проанализируем зависимость исходного текста. В примере 17-2(а) весь вызывающий код, который использует `t1_` и/или `t2_`, должен обращаться к ним с явным указанием имени; в примере 17-2(б) вызывающий код использует имена `UseT1` и `UseT2`. Пример 17-2(а) недостаточно гибкий, поскольку любые изменения `t1_` или `t2_` (например, удаление их и замена чем-то другим) требует изменения всего использующего класс вызывающего кода. В примере 17-2(б) можно, например, полностью удалить `t1_` и/или `t2_`, никак не изменяя вызывающий код, поскольку функции-члены, образующие интерфейс класса, скрывают его внутреннюю организацию.

И наконец, в пункте 1 заметим, что все, что пользователь мог сделать с `t1_` или `t2_` непосредственно в примере 17-2(а), он может точно так же сделать и при наличии функции для доступа к данным в примере 17-2(б). В вызывающем коде придется дописать пару скобок, и это все отличия в использовании двух примеров.

Давайте рассмотрим конкретный пример. Пусть, например, мы хотим добавить определенную функциональность, скажем, простой счетчик количества обращений к `t1_` или `t2_`. Если это члены данных, как в примере 17-2(а), то вот что мы должны для этого сделать.

1. Следует создать функцию для доступа к тому члену, который нас интересует, и сделать данные закрытыми (другими словами, выполнить преобразование к коду примера 17-2(б)).
2. Все пользователи вашего класса испытывают сомнительное удовольствие от поиска и замены всех обращений к `t1_` и `t2_` в своем коде на их функциональные эквиваленты. Особенно большую радость это вызывает у тех, кто уже давно занят со-

всем другой работой... Если после этого вам придут подарки от ваших пользователей — прислушайтесь, не тикает ли что-то в полученном вами пакете...

3. Весь код ваших пользователей перекомпилируется.
4. Если что-то оказалось пропущено, код не будет корректно скомпилирован, и надо будет повторить шаги 2 и 3 до тех пор, пока не будут устранены все обращения к членам `t1_` и `t2_`.

Если же у нас уже имеются функции доступа, как в примере 17-2(б), то вот что нам остается сделать.

1. Вносим изменения в существующие функции доступа.
2. Все ваши пользователи перекомпонуют (если функции находятся в отдельном `.cpp`-файле и не являются встраиваемыми) свои программы или, в худшем случае (если функции определены в заголовочных файлах), перекомпилируют их.

Самое неприятное в реальной ситуации то, что если вы начали с примера 17-2(а), то можете уже никогда не перейти к примеру 17-2(б). Чем больше пользователей зависят от вашего интерфейса, тем труднее оказывается его изменить. Все это приводит к следующему правилу, которое можно назвать Законом Второго Шанса.

➢ Рекомендация

Самая важная задача — разработка правильного интерфейса. Все остальное можно исправить позже. У вас может не оказаться возможности исправить неверный интерфейс.

Как только интерфейс начинает широко использоваться, от него может зависеть такое большое количество людей, что изменить его будет просто нереально. Да, интерфейс может быть расширен (путем добавления новых функций вместо изменения старых) без неприятных последствий для пользователей, но это не поможет исправить существующие части класса, если позже выяснится, что они были спроектированы не верно. В лучшем случае добавление функций дает возможность выполнить задачу другим способом, что обычно только запутывает ваших пользователей, которые вполне обоснованно начинают выяснять, почему имеется два (три, *N*) способа достижения некоторой цели, и какой именно из них следует использовать.

Коротко говоря, плохой интерфейс может быть очень трудно, а то и просто невозможно исправить впоследствии. Старайтесь разработать правильный интерфейс с первого раза и спрячьте за ним все детали внутренней организации класса.

Актуальный момент

3. **Шаблон класса `std::pair` использует открытые члены-данные, поскольку он не инкапсулирует никаких данных, а просто позволяет их группировать.**

Заметим, что здесь мы имеем дело с рассматривавшимся исключением, когда допустимо использование открытых данных. Но даже в этом случае использование функций доступа ни в чем не хуже применения открытых членов-данных.

Представим шаблон класса наподобие `std::pair`, но с тем отличием, что у него дополнительно имеется флаг `deleted`, который может быть установлен и опрошен (но не сброшен). Ясно, что такой флаг должен быть закрытым для защиты от непосредственного изменения пользователем. Если остальные члены-данные будут открытыми, как в `std::pair`, в конечном итоге мы получим примерно следующий код.

```
// Пример 17-3(а): одновременное использование открытых и
// закрытых данных?
```

```
//
template<class T, class U>
class Couple {
public:
    // Основные члены-данные открыты...
    T first;
    U second;
    // ...но есть еще нечто, делающее его более
    // "классоподобным", а также закрытая реализация
    Couple()           : deleted_(false) {}
    void MarkDeleted() { deleted_ = true; }
    bool IsDeleted()   { return deleted_; }
private:
    bool deleted_;
};
```

Должны ли в этом случае остальные члены-данные быть, как показано в коде, открытыми? Почему? Если да, то является ли этот код хорошим примером того, почему иногда одновременное наличие открытых и закрытых данных в одном классе может быть удачным решением?

Рассматриваемый класс `Couple` предложен в качестве контрпримера к обычной рекомендации по кодированию. Здесь приведен класс, который является "почти структурой", но имеет некоторые закрытые служебные данные. Эти данные (в приведенном примере — простой атрибут) имеют инвариант. Утверждается, что обновление атрибута происходит абсолютно независимо от обновления открытых членов класса `Couple`.

Начнем с утверждения об абсолютной независимости. Я не согласен! Обновление может быть независимым, но понятно, что сам атрибут не является независимым от этих значений; иначе он бы не был сгруппирован с ними в один объект. Атрибут `deleted_` — приложение к сопутствующим объектам.

Вот как мы можем решить поставленную задачу с использованием функций доступа.

```
// Пример 17-3(б): корректная инкапсуляция, изначально
// использующая встраиваемые функции доступа. Позже при
// необходимости эти функции могут превратиться в
// нетривиальный код (или остаться такими, как есть).
//
template<class T, class U>
class Couple {
    Couple()            : deleted_(false) { }
    T& First()          { return first_; }
    U& Second()         { return second_; }
    void MarkDeleted()  { deleted_ = true; }
    bool IsDeleted()    { return deleted_; }
private:
    T    first_;
    U    second_;
    bool deleted_;
};
```

"Ну и зачем беспокоиться о ничего не делающих функциях доступа?" — мог бы спросить, не подумав, кто-то из читателей. Отвечаю. Как уже упоминалось в обсуждении примера 17-2(б), если сегодня вызывающему коду потребовалось изменение некоторого аспекта данного объекта (в данном примере атрибут `deleted_`), то завтра может потребоваться добавление в него новых возможностей — даже если это будут всего лишь средства для отладки. Пример 17-3(б) обеспечивает вас необходимой гибкостью, которая не вызывает лишних затрат из-за использования встраиваемых функций.

Пусть, например, месяц спустя вам потребуется фиксировать все обращения к объектам, помеченным как удаленные.

- В примере 17-3(а) вы не сможете сделать этого без изменения дизайна класса и всего использующего его кода.
- В примере 17-3(б) вы просто помещаете необходимую функциональность в функции-члены `First` и `Second`. Такое изменение прозрачно для всех пользователей класса `Couple`. Максимум, что потребуется от них, — перекомпиляция приложения; им не придется вносить никаких изменений в свой код.

Оказывается, что пример 17-3(б) имеет и другие практические преимущества. Например, как отметил Ник Мейн (Nick Mein): "Вы можете поместить точку останова в функцию доступа и выяснить, где и когда происходит изменение значения, что очень помогает в отслеживании ошибок". Такое случается, и довольно часто.

Резюме

За исключением случая структуры в стиле C (в которой все члены открыты), все члены-данные всегда должны быть закрыты. Поступая иначе, вы нарушаете все принципы инкапсуляции, о которых шла речь в начале этой задачи, и создаете зависимости от имен членов, которые впоследствии затруднят выполнение корректной инкапсуляции. Не существует никаких причин для использования открытых и защищенных членов-данных; они всегда могут быть тривиально обернуты в (изначально) встраиваемые функции доступа, которые не приводят ни к каким дополнительным расходам, так что лучше всегда делать все правильно с самого начала. (Правда, имеются примеры защищенных членов-данных в стандартной библиотеке. Эти примеры не могут служить образцом.)

Начинайте с правильного проектирования интерфейса. Внутренние детали легко можно исправить и позже, но возможности исправить интерфейс у вас может больше не оказаться.

Задача 18. Виртуальность — Сложность: 7

В этой задаче мы возвращаемся к старым вопросам, чтобы дать на них новые и/или улучшенные ответы. В этой мы сформулируем четыре рекомендации по проектированию классов, которые отвечают на ряд вопросов. Почему интерфейсы должны быть невиртуальными? Почему виртуальные функции должны быть закрытыми? Остается ли в силе старый совет по поводу деструкторов?

Вопрос для новичка

1. В чем заключается "обычный совет" по поводу деструкторов базовых классов?

Вопрос для профессионала

2. Когда виртуальные функции должны быть открытыми, защищенными, закрытыми? Поясните ваш ответ.

Решение

В этой задаче я хочу представить современный взгляд на два часто повторяющихся вопроса о виртуальных функциях. Отвечая на эти вопросы, мы сформулируем четыре рекомендации по проектированию классов.

Не будучи новыми, эти вопросы остаются актуальными, хотя отвечаем мы на них сегодня по-другому, с учетом опыта работы с современным C++.

Обычный совет о деструкторах базовых классов

1. В чем заключается "обычный совет" по поводу деструкторов базовых классов?

Я знаю, что вы уже знакомы с вопросом: должны ли деструкторы базовых классов быть виртуальными?

Это не только часто задаваемый, но и весьма жарко обсуждаемый вопрос. Обычный ответ на него: "Конечно, деструкторы базовых классов должны быть виртуальными!" Это неправильный ответ, и даже стандартная библиотека C++ содержит массу контрпримеров. Тем не менее, деструкторы базовых классов достаточно часто виртуальны, чтобы создать иллюзию правильности процитированного ответа.

Мы вскоре вернемся к этому вопросу. Это второй из двух вопросов о доступности виртуальных функций. Начнем с более общего вопроса.

Виртуальный вопрос №1: открытость или закрытость?

Общий вопрос, который мы должны рассмотреть, формулируется следующим образом.

2. Когда виртуальные функции должны быть открытыми, защищенными, закрытыми? Поясните ваш ответ.

Краткий ответ звучит так: редко (если вообще должны); иногда; по умолчанию, соответственно. Словом, тот же ответ, что и для членов класса других типов.

Большинство из нас на собственном горьком опыте научились делать все члены класса закрытыми по умолчанию, кроме тех, которые мы действительно хотим предоставить для всеобщего пользования. Это стиль хорошей инкапсуляции. Конечно, мы давно знаем, что члены-данные должны всегда быть закрытыми (кроме случая структур в стиле C, которые представляют собой просто удобный способ группирования данных; см. задачу 17). То же самое правило применимо и для функций-членов, так что я предлагаю следующие рекомендации, выражающие преимущества "приватизации" кода.

➢ Рекомендация

Предпочтительно делать интерфейс невиртуальным.

Обратите внимание — я сказал "предпочтительно", а не "всегда".

Интересно, что стандартная библиотека C++ следует этой рекомендации. Не считая деструкторов (которые рассматриваются отдельно в рекомендации №4) и не учитывая дублирования виртуальных функций, которые участвуют в специализациях шаблонов классов, в стандартной библиотеке есть:

- 6 открытых виртуальных функций; все они представляют собой `std::exception::what` и ее перекрытия; и
- **142** неоткрытых виртуальных функций.

Недавно мне встретился еще один пример. Когда я писал эту книгу, я работал на Microsoft над C++-аспектами платформы .NET и .NET Frameworks (FX), которые выросли в WinFX, который является объектно-ориентированным наследником Win32 API и программной моделью Longhorn, следующего поколения операционной системы Windows. WinFX даже в текущем состоянии разработки представляет собой *огромный* API — уже сейчас в нем более 14000 классов и около 100000 функций-членов (включающих в себя текущее состояние .NET Frameworks и многое другое). Это действительно *много*. Вы не ошибетесь, если спросите — не слишком ли это монстрообразно, но сейчас дело не в этом.

Вот почему я упомянул .NET Frameworks и его эволюцию в WinFX. Для такого монстра, как эта библиотека классов, единственная надежда на работоспособность заключается в повсеместном строго соблюдающемся хорошем дизайне классов. Я счастлив сообщить, что так оно и есть. Вот одна из рекомендаций проектирования WinFX, которая кажется удивительно знакомой, и хотя я согласен с ней, я не имею никакого отношения к ее принятию — она принята по не зависящим от меня обстоятельствам.

> *Рекомендуется обеспечивать настройку посредством защищенных методов. Открытый интерфейс базового класса должен обеспечивать богатый набор функциональных возможностей для потребителя этого класса. Однако настройка класса для предоставления богатых функциональных возможностей потребителю должна обеспечиваться реализацией минимально возможного количества методов. Для достижения этой цели следует обеспечить набор невиртуальных или финальных[27] открытых методов, каждый из которых вызывает единственный защищенный метод (член семейства методов) с суффиксом 'Core', который и реализует данный метод. Эта технология известна как 'метод шаблона'* (.NET Framework (WinFX) Design Guidelines, январь 2004).

Давайте рассмотрим эту практическую рекомендацию подробнее.

Традиционно многие программисты используют базовые классы с открытыми виртуальными функциями. Например, мы можем написать следующий код.

```
// Пример 18-1: традиционный базовый класс
//
class Widget {
public:
    // Каждая из этих функций может (не обязательно) быть
    // чисто виртуальной, и в этом случае она может быть
    // реализована в Widget, а может и не быть - см.
    // [Sutter02].
    //
    virtual int Process( Gadget& );
```

[27] Имеются в виду функции Java/C#. — *Прим. перев.*

```
    virtual bool IsDone();
    // ...
};
```

Эти открытые виртуальные функции, как и все открытые виртуальные функции, одновременно определяют и интерфейс, и настраиваемое поведение. Проблема заключается в слове "одновременно", поскольку каждая открытая виртуальная функция вынуждена обслуживать двух потребителей с разными потребностями и разными целями.

- Одна группа пользователей — внешний вызывающий код, которому для работы с классом требуется его открытый интерфейс.
- Другая группа — производные классы, которым требуется "интерфейс настройки", представляющий собой набор виртуальных функций, посредством которых производные классы расширяют и уточняют функциональные возможности базовых классов.

Открытая виртуальная функция вынуждена выполнять две работы. Одна — *определять интерфейс*, поскольку она открытая и, соответственно, является непосредственной частью интерфейса. Вторая — *определить детали реализации*, а именно, настроить внутреннее поведение, поскольку эта функция виртуальная и, таким образом, обеспечивает возможность замещения в производном классе реализации этой функции в базовом классе. То, что перед виртуальной функцией стоят две существенно различные задачи и имеется два разных класса пользователей, — признак недостаточного разделения задач и того, что неплохо было бы пересмотреть сам подход к дизайну класса.

А что, если мы захотим отделить спецификацию интерфейса от спецификации настраиваемого поведения реализации? Тогда мы будем вынуждены в конечном итоге перейти к чему-то наподобие шаблона проектирования "метод шаблона" (Template Method pattern [Gamma95]), поскольку то, чего мы хотим добиться, очень напоминает данный шаблон. Однако наша задача существенно уже, и поэтому заслуживает более точного имени. Назовем этот шаблон проектирования шаблоном невиртуального интерфейса (Nonvirtual Interface (NVI) pattern). Вот пример данного шаблона проектирования в действии.

```
// Пример 18-2: более современный базовый класс,
//              использующий Невиртуальный Интерфейс (NVI)
//              для отделения интерфейса от внутренней
//              реализации класса
//
class Widget {
public:
    // Стабильный невиртуальный интерфейс
    //
    int Process( Gadget& ); // Использует DoProcess...()
    bool IsDone();          // Использует DoIsDone()
    // ...
private:
    // Настройка - деталь реализации, которая может как
    // соответствовать интерфейсу, так и не соответствовать
    // ему. Каждая из этих функций может (не обязательно)
    // быть чисто виртуальной и, если это так, иметь (или не
    // иметь) реализацию в классе Widget (см. [Sutter02])
    //
    virtual int DoProcessPhase1( Gadget& );
    virtual int DoProcessPhase2( Gadget& );
    virtual bool DoIsDone();
    // ...
};
```

Использование шаблона NVI дает возможность получения устойчивого невиртуального интерфейса при делегировании всей работы по настройке закрытым виртуальным

функциям. В конце концов, виртуальные функции разработаны для того, чтобы позволить производным классам настроить свое поведение. Поскольку интерфейс класса предполагается стабильным и непротиворечивым, лучше всего не позволять производным классам каким-либо образом изменять или настраивать его.

Подход NVI обладает рядом преимуществ и не имеет существенных недостатков.

Во-первых, обратите внимание, что теперь базовый класс полностью контролирует свой интерфейс и стратегию и может диктовать предусловия и постусловия его работы, выполнять добавление функциональности и другие подобные действия в одном удобном для повторного использования месте — функциях невиртуального интерфейса. Это способствует хорошему дизайну класса, поскольку позволяет базовому классу обеспечить согласованность производных классов при подстановке в соответствии с принципом подстановки Лисков (Liskov) [Liskov88]. В случае, когда особое значение приобретают вопросы производительности программы, базовый класс может выполнять проверку ряда предусловий и постусловий только в отладочном режиме, отказываясь от таких проверок либо в процессе компиляции окончательной версии программы, либо подавляя эти проверки во время выполнения программы в соответствии с настройками ее запуска.

Во-вторых, при лучшем разделении интерфейса и реализации, мы можем обеспечить большую свободу в настройке поведения класса, не влияя на его вид для внешних пользователей. Так, в примере 18-2 мы решили, что имеет смысл предоставить пользователю одну функцию `Process` и в то же время обеспечить более гибкую настройку, разбив реализацию на две части — `DoProcessPhase1` и `DoProcessPhase2`. Это оказалось очень просто. Мы бы не смогли добиться этого при использовании версии с открытыми виртуальными функциями без того, чтобы такое разделение стало видимо в интерфейсе, тем самым добавляя сложности для пользователей, которым в этой ситуации пришлось бы вызывать две функции (см. также задачу 19 в [Sutter00]).

В-третьих, теперь базовый класс лучше приспособлен к будущим изменениям. Мы можем позже изменить наши замыслы и добавить проверку выполнения пред- и постусловий или разделить работу на несколько этапов, или, например, реализовать полное разделение интерфейса и реализации с использованием идиомы указателя на реализацию (Pimpl, см. [Sutter00]), или внести другие изменения в интерфейс для настройки класса, при этом никак не влияя на код, который использует этот класс. Например, существенно труднее начать с открытой виртуальной функции и позже пытаться обернуть ее в другую для проверки пред- и постусловий, чем изначально предоставить невиртуальную функцию-оболочку (даже если никакой дополнительной проверки или иной работы в настоящий момент не требуется) и вставить в нее необходимые проверки позже. (Дополнительная информация о том, как сделать класс более приспособленным для будущих изменений, имеется в [Hyslop00].)

"Но, — могут возразить некоторые, — все, что делает такая открытая виртуальная функция — это вызов закрытой виртуальной функции. Это — всего лишь одна строка. Насколько нужны такие однострочные функции, если они практически бесполезны, да и к тому же приводят к снижению эффективности (за счет лишнего вызова функции) и повышению сложности (за счет добавления лишней функции)?" Сначала пара слов об эффективности: на практике снижения эффективности не будет, так как если такая однострочная передающая вызов функция объявлена как встраиваемая, то все известные мне компиляторы выполняют оптимизацию такого вызова, полностью убирая его, т.е. в результате при вызове нет никаких накладных расходов[28]. Теперь поговорим о сложности. Единственное, в чем проявляется сложность, — это дополнительное время, необходимое для написания тривиальных однострочных функций-оболочек. Все. На интер-

[28] На самом деле некоторые компиляторы всегда делают такую функцию встраиваемой и убирают ее, независимо от того, хотите вы этого или нет; впрочем, это уже другая история — см. задачу 25.

фейс это не оказывает никакого влияния: класс имеет все то же количество открытых функций для пользователей класса, так что им не надо ничего изучать дополнительно, и то же количество виртуальных функций, что и ранее, так что программисту производного класса тоже не прибавляется работы. Как видите, ни интерфейс для внешних пользователей, ни интерфейс наследования для производных классов не становятся сложнее, зато теперь они явным образом разделены, а это — хорошо.

Итак, изложенный материал оправдывает невиртуальные интерфейсы и доказывает, что виртуальные функции только выигрывают от закрытия. Но мы еще не ответили на вопрос, должны ли виртуальные функции быть закрытыми или защищенными. Ответ:

> **➢ Рекомендация**
>
> Лучше делать виртуальные функции закрытыми (`private`).

Это просто. Такой подход позволяет производному классу переопределять функции для необходимой настройки поведения класса, при этом не делая их доступными для непосредственного вызова производным классам (что было бы возможно при объявлении функций защищенными). Дело в том, что виртуальные функции существуют для того, чтобы обеспечить возможность настройки поведения класса; если они при этом не должны непосредственно вызываться из кода производных классов, нет никакой необходимости в том, чтобы делать их не закрытыми. Однако иногда нам надо вызывать базовые версии виртуальных функций (см., например, [Hyslop00]), и только в этом случае имеет смысл делать эти виртуальные функции защищенными, т.е. мы можем сформулировать очередное правило.

> **➢ Рекомендация**
>
> Виртуальную функцию можно делать защищенной только в том случае, когда производному классу требуется вызов реализации виртуальной функции в базовом классе.

Основной вывод заключается в том, что применение шаблона NVI к виртуальным функциям помогает нам отделить интерфейс от реализации. Можно добиться еще более полного разделения, если воспользоваться шаблоном типа Bridge [Gamma95], идиомой наподобие Pimpl (преимущественно для управления зависимостями во время компиляции и гарантий безопасности исключений) [Sutter00, Sutter02] или более общими handle/body или envelope/letter [Coplien92], а также других подходов. Если же только вам не нужна большая степень разделения интерфейса и реализации, то NVI зачастую оказывается *достаточно* для ваших нужд. С другой стороны, применение NVI — хорошая идея, которую стоит принять по умолчанию в своей практической деятельности при создании нового кода и рассматривать как минимально *необходимое* разделение. В конце концов, она не приводит к дополнительным расходам (не считая написания дополнительной строки кода на функцию), но зато существенно уменьшает количество проблем впоследствии.

Дополнительные примеры использования шаблона NVI для "приватизации" виртуального поведения можно найти в [Hyslop00].

Кстати о [Hyslop00] — вы не обратили внимания на то, что в представленном там коде имеется открытый виртуальный деструктор? Это приводит нас ко второй теме нашей задачи.

Виртуальный вопрос №2: деструкторы базовых классов

Теперь мы готовы заняться вторым классическим вопросом — должны ли деструкторы базовых классов быть виртуальными?

Как уже упоминалось, типичный ответ на такой вопрос: "Конечно же, деструкторы базовых классов должны быть виртуальными!" Этот ответ неправильный, и стандартная библиотека C++ содержит контрпримеры, опровергающие это мнение. Однако деструкторы базовых классов очень часто должны быть виртуальными, что и создает иллюзию корректности приведенного ответа.

Немного менее распространенный и несколько более правильный ответ: "Конечно, деструкторы базового класса должны быть виртуальными, если вы собираетесь удалять объекты полиморфно, т.е. через указатель на базовый класс." Этот ответ технически корректен, но не совсем полон.

Я пришел к выводу, что полный корректный ответ должен звучать следующим образом.

➢ Рекомендация

Деструктор базового класса должен быть либо открытым и виртуальным, либо защищенным и невиртуальным.

Давайте посмотрим, почему.

Понятно, что любая операция, которая выполняется посредством интерфейса базового класса и должна вести себя виртуально, должна быть виртуальна. Это справедливо даже при использовании NVI, поскольку хотя открытая функция и невиртуальна, она делегирует работу закрытой виртуальной функции, и таким образом мы получаем требуемое виртуальное поведение.

Если уничтожение может быть выполнено полиморфно посредством интерфейса базового класса, то оно должно вести себя виртуально и, соответственно, быть виртуальным. В действительности, это требование языка — если вы выполняете полиморфное удаление без виртуального деструктора, то получаете весь спектр "неопределенного поведения", с которым лично я предпочел бы никогда не встречаться. Следовательно:

```
// Пример 18-3: необходимость виртуального деструктора
//
class Base { /* ... */ };
class Derived : public Base { /* ... */ };
Base* b = new Derived;
delete b;  // Лучше бы Base::~Base быть виртуальным!
```

Заметим, что деструктор — единственный случай, когда шаблон NVI *не может* быть применен к виртуальной функции. Почему? Потому что когда выполнение достигло тела деструктора базового класса, все части производных объектов уже уничтожены и более не существуют. Если в теле деструктора базового класса будет вызвана виртуальная функция, то выбор виртуальных функций не сможет пройти по иерархии классов дальше базового класса. В теле деструктора (конструктора) порожденные классы уже (или еще) не существуют.

Но базовые классы не всегда должны допускать полиморфное удаление. Рассмотрим, например, шаблоны классов, такие как `std::unary_function` и `std::binary_function` из стандартной библиотеки C++ [C++03]. Эти два шаблона классов выглядят следующим образом.

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg       argument_type;
    typedef Result    result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1    first_argument_type;
    typedef Arg2    second_argument_type;
    typedef Result  result_type;
};
```

Оба эти шаблона предназначены, в частности, для инстанцирования в качестве базовых классов (для ввода стандартных typedef-имен в производные классы) и не имеют виртуальных деструкторов, поскольку они не предназначены для полиморфного удаления. То есть, код наподобие следующего — не просто неразрешенный, но и просто незаконный. Поэтому вы можете с полным основанием считать, что такой код никогда не будет существовать.

```
// Пример 18-4: проблематичный код, который никогда не будет
//              существовать в реальности.
//
void f( std::unary_function* f ) {
    delete f;          // Ошибка, не корректно
}
```

Заметим, что стандарт не одобряет такие фокусы и объявляет пример 18-4 попадающим непосредственно в ловушку неопределенного поведения, если вы передадите указатель на объект, производный от `std::unary_function`, но при этом не требует от компилятора запретить вам написать такой код (а жаль). Хотя это легко сделать — при этом ни в чем не нарушая стандарт — просто дать `std::unary_function` (и другим классам наподобие него) пустой, но *защищенный* деструктор; в этом случае компилятор будет вынужден диагностировать ошибку и обвинить в ней нерадивого программиста. Может, мы увидим такое изменение в очередной версии стандарта, может — нет, но было бы неплохо, чтобы компилятор мог отвергнуть такой код. (Да, сделав деструктор защищенным, мы тем самым делаем невозможным непосредственное инстанцирование `unary_function`, но это не имеет значения, поскольку этот шаблон полезен только в качестве базового класса.)

А что если базовый класс конкретный (может быть создан объект данного класса), но вы хотите, чтобы он поддерживал полиморфное удаление? Должен ли его деструктор быть открытым — ведь иначе вы не сможете создать объект данного типа? Это возможно, но только если вы нарушили другое правило, а именно — ничего *не порождайте из конкретных классов*. Или, как сказал Скотт Мейерс (Scott Meyers) в разделе [Meyers96], "делайте классы-не листья[29] абстрактными". (Конечно, на практике можно столкнуться с нарушением этого правила — понятно, в чьем-то коде, не в вашем — и в этом единственном случае вам придется иметь открытый виртуальный деструктор просто для того, чтобы приспособиться к уже имеющемуся скверному дизайну. Конечно, лучше — если это возможно — изменить сам дизайн.)

Коротко говоря, вы оказываетесь в одной из двух ситуаций. Либо а) вы хотите обеспечить возможность полиморфного удаления через указатель на базовый класс, и тогда деструктор должен быть открытым и виртуальным, либо б) вам этого не нужно, и тогда деструктор должен быть невиртуальным и защищенным — чтобы предотвратить нежелательное использование вашего класса.

Резюме

Итак, лучше делать виртуальные функции базового класса закрытыми (или, при наличии веских оснований, защищенными). Это разделяет интерфейс и реализацию, что позволяет стабилизировать интерфейс и упростить дальнейшие изменения и переработки реализации. Для функций обычного базового класса:

- рекомендация №1: лучше делать интерфейс невиртуальным, с использованием шаблона проектирования невиртуального интерфейса (NVI);
- рекомендация №2: лучше делать виртуальные функции закрытыми (`private`);

[29] Иначе говоря, классы, соответствующие внутренним узлам дерева наследования. — *Прим. перев.*

- рекомендация №3: виртуальную функцию можно делать защищенной только в том случае, когда производному классу требуется вызов реализации виртуальной функции в базовом классе.

Единственное исключение делается для деструктора:

- рекомендация №4: деструктор базового класса должен быть либо открытым и виртуальным, либо защищенным и невиртуальным.

Правда, сама стандартная библиотека не всегда следует всем этим рекомендациям. Частично это отражение процесса накопления знаний сообществом программистов C++.

Задача 19. Не можешь — научим, не хочешь — заставим! (или как заставить потомков вести себя прилично) Сложность: 5

Оказывается, иногда вы можете уберечь программистов производных классов от некоторых простых ошибок. Эта задача — о безопасном дизайне базовых классов, который не позволяет разработчикам производных классов пойти неверным путем.

Вопрос для новичка

1. В каком случае происходит неявное объявление и определение перечисленных ниже функций? С какой семантикой? Будьте точны и опишите условия, при которых их неявно определенные версии делают программы некорректными:
   а) конструктор по умолчанию;
   б) копирующий конструктор;
   в) копирующий оператор присваивания;
   г) деструктор.
2. Какие функции неявно объявлены и созданы компилятором в следующем классе X? С какими сигнатурами?

```
class X {
    auto_ptr<int> i_;
};
```

Вопрос для профессионала

3. Пусть у вас есть базовый класс, который требует, чтобы все производные классы не использовали ни одной неявно объявленной и созданной компилятором функции. Например:

```
class Count {
public:
    // Этим комментарием автор класса Count документирует,
    // что производные классы должны наследоваться
    // виртуально, и что их конструкторы должны вызывать
    // конструктор класса Count специального назначения.
    //
    Count( /* Специальные параметры */ );
    Count& operator=( const Count& );  // Обычный
    virtual ~Count();                  // Обычный
};
```

К сожалению, программистам, как и всем остальным людям, свойственно ошибаться, так что иногда они будут забывать, что они обязаны явно написать две функции.

```
class BadDerived : private virtual Count {
    int i_;
    // Конструктор по умолчанию: должен вызывать специальный
    // конструктор, но делает ли он это?

    // Копирующий конструктор: должен вызывать специальный
    // конструктор, но делает ли он это?

    // Копирующее присваивание: ok?
    // Деструктор: ok?
};
```

Имеется ли способ в контексте этого примера, при помощи которого автор класса `Count` мог бы заставить автора производного класса программировать в соответствии с указанными правилами — т.е. чтобы сообщение об ошибке выдавалось во время компиляции (предпочтительно) или хотя бы во время выполнения программы, если автор производного класса нарушил указанное в комментарии к классу `Count` требование?

Вопрос в общем виде: имеется ли способ, при помощи которого автор базового класса может заставить автора производного класса явным образом написать каждую из четырех перечисленных выше базовых операций? Если да, то как? Если нет, то почему?

Решение

Неявно генерируемые функции

В C++ компилятор может неявно генерировать четыре функции-члена класса: конструктор по умолчанию, копирующий конструктор, оператор копирующего присваивания и деструктор.

Причина этого заключается в сочетании удобства и обратной совместимости с C. Вспомним, что структуры в стиле C `struct` — это просто классы, содержащие только открытые члены-данные; в частности, они не должен иметь (явно определенных) функций-членов, и все же должен существовать способ создавать, копировать и уничтожать их. Для этого C++ автоматически генерирует соответствующие функции (или некоторое их подмножество) для выполнения указанных действий, если вы не определили их самостоятельно.

В задаче спрашивается о том, что именно означает слово "соответствующие".

1. **В каком случае происходит неявное объявление и определение перечисленных ниже функций? С какой семантикой? Будьте точны и опишите условия, при которых их неявно определенные версии делают программы некорректными.**

Говоря кратко, неявно объявленная функция становится неявно определенной только при попытке ее использовать. Например, неявно объявленный конструктор по умолчанию неявно создается компилятором только тогда, когда вы пытаетесь создать объект без указания параметров конструктора.

Почему следует различать случаи неявного объявления и неявного определения функции? Потому что вполне реальна ситуация, когда функция никогда не вызывается, и если это так, то программа остается корректной даже если неявное определение функции некорректно.

Для удобства в этой задаче, если явно не оговорено иное, "член" означает "нестатический член-данные класса". Кроме того, я буду говорить "неявно сгенерированные", подразумевая "неявно объявленные и определенные".

Спецификации исключений неявно определенных функций

Во всех четырех случаях неявного объявления функций компилятор делает их спецификации исключений достаточно свободными, допускающими генерацию любых исключений неявно определенными функциями. Например:

```
// Пример 19-1(a)
//
class C // ...
{
    // ...
};
```

Поскольку явно объявленных конструкторов нет, семантика неявно сгенерированного конструктора по умолчанию заключается в вызове конструкторов по умолчанию базовых классов и членов. Следовательно, спецификация исключений неявно сгенерированного конструктора класса C должна позволять генерацию любых исключений, которые могут быть сгенерированы конструкторами по умолчанию базовых классов или членов. Если *хоть один* базовый класс C или его член имеет конструктор по умолчанию без явной спецификации исключений, то неявно объявленный конструктор по умолчанию C может генерировать любое исключение:

```
// public:
inline C::C();          // Может генерировать что угодно
```

Если *все* базовые классы и члены C имеют конструкторы по умолчанию с явно указанными спецификациями исключений, то неявно объявленный конструктор C может генерировать исключения любого из типов, упомянутых в этих спецификациях исключений:

```
// public:
inline C::C() throw    (
    // Все, что могут генерировать конструкторы по умолчанию
    // базовых классов или членов; т.е. здесь находится
    // объединение всех типов, упомянутых в спецификациях
    // исключений конструкторов по умолчанию базовых классов
    // и членов C
    );
```

Оказывается, здесь есть потенциальная ловушка. Что, если одна из неявно сгенерированных функций перекрывает унаследованную виртуальную функцию? Этого не может произойти для конструкторов (поскольку конструкторы не бывают виртуальными), но вполне возможно для оператора копирующего присваивания (если вы сделаете базовую версию соответствующей сигнатуре версии, неявно сгенерированной в производном классе), и то же может произойти и для деструктора.

```
// Пример 19-1(б): опасный момент!
//
class Derived;
class Base {
public:
    // Несколько искусственный пример, но технически вполне
    // возможно использовать этот метод для объявления
    // оператора присваивания Base, который получает в
    // качестве аргумента Derived. Перед тем как даже просто
    // подумать применить такой метод, обязательно прочтите
    // подраздел 33 в [Meyers96].
    //
    virtual Base& /* или Derived& */
    operator=( const Derived& ) throw( B1 );
    virtual ~Base() throw( B2 );
};

class Member {
public:
    Member& operator=( const Member& ) throw( M1 );
    ~Member() throw( M2 );
};

class Derived : public Base {
    Member m_;
    // Неявно объявлены четыре функции:
    //   Derived::Derived();                    // ok
    //   Derived::Derived( const Derived& );    // ok
    //   Derived& Derived::operator =
    //          (const Derived&) throw(B1,M1);  // Ошибка
```

```
    //   Derived::~Derived() throw(B2,M2);      // Ошибка
};
```

В чем здесь проблема? Две функции созданы неверно, поскольку когда вы перекрываете любую унаследованную виртуальную функцию, спецификация исключений вашей производной функции должна представлять собой ограниченную версию спецификации исключений в базовом классе. В конце концов, только это и имеет смысл: если бы это было не так, то это означало бы, что код, который вызывает функцию посредством указателя на базовый класс, может получить исключение, которое, как обещает базовый класс, не может быть сгенерировано. Например, считая допустимым контекст примера 19-1(б), рассмотрим следующий код:

```
Base* p = new Derived;

// Здесь может быть сгенерировано исключение B2 или M2,
// несмотря на то, что Base::~Base обещает не генерировать
// никаких исключений, кроме B2:
delete p;
```

Это еще одна существенная причина для того, чтобы деструкторы имели спецификации исключений `throw()` или не имели их вообще. Кроме того, деструкторы никогда не должны генерировать исключений и всегда должны разрабатываться так, как если бы они имели спецификацию исключений `throw()`, даже если она не указана явно (см. [Sutter00], где есть подраздел "Деструкторы, генерирующие исключения, и почему они неприемлемы").

➢ Рекомендация

Никогда не позволяйте деструкторам генерировать исключения. Всегда разрабатывайте деструкторы так, как если бы они имели пустые спецификации исключений.

Никогда не используйте спецификации исключений, кроме, возможно, пустых, но я советую избегать и их (см. задачу 13).

Это еще одна причина быть осторожными с виртуальными операторами присваивания. См. раздел 29 в [Meyers96] и раздел 76 в [Dewhurst03], где подробнее рассказано об опасностях виртуального присваивания и о том, как их избежать.

➢ Рекомендация

Не делайте операторы присваивания виртуальными.

Теперь рассмотрим последовательно все четыре неявно генерируемые функции.

Неявный конструктор по умолчанию

а) **конструктор по умолчанию**

Конструктор по умолчанию неявно объявляется в случае, когда вы не объявили ни одного собственного конструктора. Такой неявно объявленный конструктор является открытым и встраиваемым.

Неявно объявленный конструктор неявно определяется только в том случае, если вы действительно пытаетесь вызвать его, причем это имеет тот же вид, как если бы вы написали пустой конструктор по умолчанию самостоятельно, причем такой конструктор может генерировать все исключения, которые могут генерировать конструкторы по умолчанию базовых классов и членов.

Такой пустой конструктор по умолчанию некорректен, если бы был некорректен такой конструктор по умолчанию, который бы вы написали самостоятельно

(например, если какой-либо из базовых классов или членов не имеет конструктора по умолчанию).

Неявный копирующий конструктор

б) **копирующий конструктор**

Копирующий конструктор неявно объявляется в том случае, если вы не объявили его самостоятельно. Неявно объявленный копирующий конструктор открытый и встраиваемый, он принимает в качестве параметра ссылку, по возможности на константный объект (это возможно тогда и только тогда, когда все базовые классы и члены имеют копирующие конструкторы, принимающие в качестве параметров ссылку на `const` или `const volatile`), и на неконстантный, если это невозможно.

Стандарт в большинстве случаев игнорирует ключевое слово `volatile`. Компилятор изо всех сил будет стараться добавить квалификатор `const` к параметру неявно объявленного копирующего конструктора (и копирующего оператора присваивания), когда это возможно, чего нельзя сказать о `volatile`.

Неявно объявленный копирующий конструктор становится неявно определенным при попытке его реального вызова для копирования объекта данного типа, выполняет почленное копирование базовых подобъектов и членов и может генерировать любое из исключений, которые могут генерировать копирующие конструкторы базовых классов или членов. Такой копирующий конструктор некорректен, если недоступен хоть один из копирующих конструкторов базовых классов или членов (или возникает неоднозначность при выборе копирующего конструктора).

Неявный копирующий оператор присваивания

в) **копирующий оператор присваивания**

Копирующий оператор присваивания неявно объявляется, если вы не объявили его самостоятельно. Неявно объявленный копирующий оператор присваивания является открытым и встраиваемым и возвращает ссылку на неконстантный объект, которому выполнено присваивание. Оператор по возможности получает ссылку на константный объект (это возможно тогда и только тогда, когда все базовые классы и члены имеют операторы копирующего присваивания, которые получают в качестве параметра константную ссылку), или ссылку на неконстантный объект в противном случае. Как и в случае копирующего конструктора, квалификатор `volatile` не используется.

Неявно объявленный оператор копирующего присваивания неявно определен только в случае реальной попытки присваивания объекта данного типа, и выполняет почленное присваивание подобъектов базового класса и членов (включая возможные множественные присваивания подобъектов виртуальных базовых классов), и может генерировать исключения всех типов, которые могут генерировать копирующие присваивания базовых классов и членов. Копирующее присваивание некорректно, если любой из базовых классов или членов является константным, ссылкой, либо имеет недоступный или неоднозначный оператор копирующего присваивания[30].

Неявный деструктор

г) **деструктор**

Деструктор неявно объявляется в случае, если вы не объявили его самостоятельно. Неявно объявленный деструктор открытый и встраиваемый.

Неявно объявленный деструктор неявно определяется только в том случае, если вы действительно пытаетесь вызвать его, причем он имеет тот же вид, как если бы вы

[30] Оператор копирующего присваивания может копировать массивы, являющиеся членами классов, что дает единственный способ неявно [поэлементно] копировать массив. — *Прим. ред.*

написали пустой деструктор самостоятельно, причем такой деструктор может генерировать все исключения, которые могут генерировать деструкторы базовых классов и членов. Неявный деструктор недействителен, если хотя бы один из базовых классов или членов имеет недоступный деструктор, или один из деструкторов базовых классов виртуален и не все деструкторы базовых классов или членов имеют идентичные спецификации исключений (см. раздел "Спецификации исключений неявно определенных функций", стр. 127).

Член auto_ptr

2. Какие функции неявно объявлены и созданы компилятором в следующем классе X? С какими сигнатурами?

```
// Пример 19-2
class X {
    auto_ptr<int> i_;
};
```

Небольшое отступление. Этот пример просто иллюстрирует правила, которым подчиняется неявное объявление и определение функций, так что не стоит считать его образцом хорошего стиля. Вообще говоря, применения auto_ptr в качестве членов класса (да и в других ситуациях) следует избегать; предпочтительно использовать класс shared_ptr, которого еще нет в стандартной библиотеке, но он уже включен в предварительную редакцию следующей версии стандарта C++.

В классе X неявно объявлены в качестве открытых членов следующие функции (каждая из которых становится неявно определенной, когда написанный вами код пытается ее использовать).

```
inline X::X() throw()             : i_() { }
inline X::X( X& other ) throw()   : i_( other.i_ ) { }
inline X& X::operator=( X& other ) throw()
                          {i_=other.i_; return*this;}
inline X::~X() throw()            { }
```

Копирующий конструктор и оператор копирующего присваивания получают в качестве параметров ссылки на неконстантные объекты. Это обусловлено тем, что так же поступают копирующий конструктор и оператор копирующего присваивания auto_ptr. Аналогично, все эти функции имеют пустые спецификации исключений, так как в состоянии их обеспечить — ни одна аналогичная операция auto_ptr не генерирует исключений (как, впрочем, вообще все операции auto_ptr).

Заметим, что копирующий конструктор и оператор копирующего присваивания класса auto_ptr передают владение. Это может оказаться не тем действием, на которое рассчитывает автор X, так что класс X, скорее всего, должен обеспечивать собственные версии копирующего конструктора и оператора копирующего присваивания (Дополнительную информацию по этому вопросу можно найти в [Sutter02].)

Семейные проблемы

3. Пусть у вас есть базовый класс, который требует, чтобы все производные классы не использовали ни одной неявно объявленной и созданной компилятором функции. Например:

```
// Пример 19-3
class Count {
public:
    // Этим комментарием автор класса  Count документирует,
    // что производные классы должны наследоваться
    // виртуально, и что их конструкторы должны вызывать
    // конструктор класса  Count специального назначения.
    //
    Count( /* Специальные параметры */ );
```

```
    Count& operator=( const Count& );  // Обычный
    virtual ~Count();                  // Обычный
```

Итак, мы имеем класс, производные классы которого должны вызывать специальный конструктор Count, например, чтобы отслеживать количество объектов производных классов в системе. Это серьезная причина для использования виртуального наследования, которое позволит избежать двойного учета при множественном наследовании, когда может случиться так, что у некоего производного класса окажется больше одного базового класса Count[31].

Интересно, заметили ли вы, что в классе Count есть ошибка проектирования? У него есть неявно генерируемый копирующий конструктор, что, вероятно, является нежелательным для корректной работы счетчика. Для того чтобы исправить ситуацию, надо просто объявить закрытый копирующий конструктор без его определения:

```
private:
    // Не определен; копирующего конструктора нет
    Count( const Count& );
};
```

Итак, мы хотим, чтобы класс Count определял поведение дочерних производных классов. Но дети не всегда слушаются родителей, правда? :-)

К сожалению, программистам, как и всем остальным людям, свойственно ошибаться, так что иногда они будут забывать, что они обязаны явно написать две функции.

```
class BadDerived : private virtual Count {
    int i_;
    // Конструктор по умолчанию: должен вызывать специальный
    // конструктор, но делает ли он это?
```

Вкратце — нет, конструктор по умолчанию не вызывает специализированный конструктор. Более того, существует ли вообще конструктор по умолчанию BadDerived? Ответ, который вряд ли покажется вам обнадеживающим, — отчасти. Имеется неявно объявленный конструктор по умолчанию (хорошо), но если вы попытаетесь его вызвать, у вас ничего не получится (плохо).

Рассмотрим, почему так получается. Начнем с того, что BadDerived не определяет ни одного собственного конструктора, так что конструктор по умолчанию будет объявлен неявно. Но в тот момент, когда вы попытаетесь его вызвать (например, при создании объекта BadDerived), этот конструктор по умолчанию становится неявно определенным или, как минимум, должен стать таковым. Однако поскольку неявно определенный конструктор, как предполагается, вызывает конструктор по умолчанию базового класса, который не существует, мы получаем неработоспособную программу. Отсюда можно заключить, что любая программа, которая попытается создать объект BadDerived, не соответствует правилам языка, и класс BadDerived совершенно справедливо назван некорректным.

Так есть ли у данного класса конструктор по умолчанию? Отчасти. Он объявлен, но при попытке вызвать его выясняется, что он ни на что не годен. Если дети так себя ведут, такую семью трудно назвать счастливой.

```
    // Копирующий конструктор: должен вызывать специальный
    // конструктор, но делает ли он это?
```

По тем же причинам неявно сгенерированный копирующий конструктор будет объявлен, но при определении не будет вызывать специальный конструктор Count. Как видно из исходного определения класса Count, этот копирующий кон-

[31] Этот пример адаптирован из кода, приведенного в неопубликованной статье Марко Далла Гасперина (Marco Dalla Gasperina) "Подсчет объектов и виртуальное наследование". Его код не имеет ошибок проектирования, о которых пойдет речь дальше. Тема этой статьи несколько отличается от рассматриваемой в данной задаче, но этот пример вполне применим для нее.

структор будет просто вызывать неявно сгенерированный копирующий конструктор класса Count.

Если нам надо подавить неявно генерируемый копирующий конструктор Count, как показано ранее, то класс BadDerived будет иметь неявно объявленный копирующий конструктор, но поскольку он не может быть неявно определен (т.к. копирующий конструктор Count оказывается недоступен), то все попытки его использования делают программу неработоспособной.

К счастью, с этого момента начинаются хорошие новости.

```
    // Копирующее присваивание:  ok?
    // Деструктор:  ok?
};
```

Да, неявно сгенерированный оператор копирующего присваивания и деструктор будут работоспособны, а именно — будут вызывать (а в случае деструктора перекрывать) соответствующие функции базового класса. Так что хорошая новость в том, что хоть что-то работает правильно.

Однако не все еще хорошо в этой семье. В конце концов, в каждом домашнем хозяйстве должен быть минимальный порядок. Можем ли мы найти способ поддержания мира в этой семье?

Не хочешь — заставим!

Имеется ли способ в контексте этого примера, при помощи которого автор класса Count мог бы заставить автора производного класса программировать в соответствии с указанными правилами — т.е. чтобы сообщение об ошибке выдавалось во время компиляции (предпочтительно) или хотя бы во время выполнения программы, если автор производного класса нарушил указанное в комментарии к классу Count требование?

Идея состоит не в том, чтобы запретить неявное объявление (мы не в состоянии это сделать), а в том, чтобы сделать неявное определение некорректным, чтобы компилятор мог вразумительно сообщить о возникшей ошибке.

Вопрос в общем виде: имеется ли способ, при помощи которого автор базового класса может заставить автора производного класса явным образом написать каждую из четырех перечисленных выше базовых операций? Если да, то как? Если нет, то почему?

Все время, пока мы знакомились с неявным объявлением и определением четырех базовых операций, мы наталкивались на слова "недоступный" и "неоднозначный". Оказывается, что добавление неоднозначных перегрузок, даже с различными спецификаторами доступа, нам не сильно поможет. Трудно добиться чего-то лучшего, чем то, что мы получили, сделав функции базового класса выборочно недоступными, объявляя их закрытыми (определены ли они реально — вопрос второй) — и этот подход работает для всех функций, кроме одной.

```
// Пример 19-4: попытка заставить производный класс не
//              использовать неявно сгенерированные функции,
//              делая функции базового класса недоступными
//
class Base {
public:
    virtual ~Base();
private:
    Base( const Base& );             // Не определена
    Base& operator=( const Base& ); // Не определена
};
```

Этот класс Base не имеет конструктора по умолчанию (поскольку объявлен, хотя и не определен, пользовательский конструктор), и имеет скрытые копирующий конструктор и копирующий оператор присваивания. Нет никакого способа скрыть деструктор,

который должен всегда быть доступен для производных классов и который обычно должен быть открытым и виртуальным, как в примере 19-4, или защищенным и невиртуальным (см. задачу 18).

Идея заключается в том, что если мы даже захотим обеспечить поддержку данной операции (например, копирующего присваивания), то если мы не в состоянии сделать это при помощи обычной функции, то мы делаем эту обычную функцию недоступной и предоставляем другую функцию, которая делает необходимую нам работу.

Что это нам дает?

```
class Derived : private Base {
int i_;
    // Конструктор по умолчанию: объявлен, но определение
    // некорректно (нет конструктора Base по умолчанию)

    // Копирующий конструктор: объявлен, но определение
    // некорректно (копирующий конструктор Base недоступен)

    // Копирующее присваивание: объявлено, но определение
    // некорректно (Копирующее присваивание Base недоступно)

    // Деструктор: все в порядке, будет компилироваться
};
```

Не так плохо — мы получили три ошибки времени компиляции из четырех возможных, и оказывается, что это все, чего мы можем добиться.

Это простое решение не в состоянии справиться с деструктором, но это не страшно, поскольку деструкторы в меньшей степени подвержены замене для специальных случаев. Базовый деструктор всегда должен быть вызван, тут двух мнений быть не может; кроме того, в конечном счете, может существовать только один деструктор. Трудность обычно представляет вызов необычного конструктора для корректной инициализации базового класса; после этого базовый класс может сохранить всю необходимую информацию для корректного выполнения деструктором всех стоящих перед ним задач.

Итак, все оказалось не плохо — впрочем, простые решения, как правило, наилучшие. В нашем случае, впрочем, есть несколько более сложных альтернатив. Давайте бегло с ними познакомимся, чтобы убедиться, что ни одна из них не в состоянии предложить более полное решение поставленной задачи.

Альтернатива №1: сделать функции базового класса неоднозначными. Этот метод ничуть не лучше: он так же не влияет на неявно сгенерированный деструктор и требует большего количества работы.

Альтернатива №2: предоставить базовые версии функций, аварийно завершающие работу программы. Например, мы можем заставить их генерировать исключение `std::logic_error`. Это также не приводит к решению вопроса о неявно генерируемом деструкторе (не нарушает работу всех возможных деструкторов), а также превращает ошибку времени компиляции в ошибку времени выполнения, что существенно хуже.

➢ Рекомендация

Предпочитайте ошибки времени компиляции ошибкам времени выполнения.

Альтернатива №3: обеспечить чисто виртуальные базовые версии. Это бесполезно: метод не применим к конструкторам (как к конструктору по умолчанию, так и к копирующему конструктору); он не в состоянии помочь нам в случае копирующего присваивания, поскольку производные версии имеют отличающиеся сигнатуры; и он не в состоянии справиться с деструкторами, так как неявно генерируемая версия будет удовлетворять требованию определения деструктора.

Альтернатива №4: использование виртуального базового класса без конструктора по умолчанию. Этот метод заставляет каждый производный класс явным образом вызывать

конструктор виртуального базового класса. Этот подход обеспечивает решение для двух конструкторов и имеет дополнительное преимущество, заключающееся в том, что он работает даже для классов, которые являются опосредованно производными от `Base`, так что это действительно единственная альтернатива, которая может использоваться в сочетании с решением примера 19-4. Правда, в этом случае корректными окажутся неявно сгенерированные оператор копирующего присваивания и деструктор производного класса.

Резюме

Самый простой способ воспрепятствовать неявной генерации производными классами конструктора по умолчанию, копирующего конструктора, или оператора копирующего присваивания заключается в том, чтобы сделать версии этих функций в базовом классе закрытыми (или без определения).

Управление памятью и ресурсами

Если и есть какая-то проблема, дорогая сердцу программистов на C и C++, то это — управление памятью и другими ресурсами. Одна из самых сильных сторон C++ по сравнению с другими языками заключается в той мощи, которую C++ предоставляет программисту для управления памятью и другими ресурсами, в частности, для выборочного автоматического управления памятью при использовании стандартных контейнеров.

Насколько хорошо вы понимаете, как используется память различными стандартными контейнерами? Можете ли вы с уверенностью утверждать, что контейнер `list`, содержащий 1 000 объектов, будет требовать меньший объем памяти, чем, например, контейнер `set` с 1 000 объектов того же типа? Или, возвращаясь к вопросам безопасности исключений: поможет ли использование версии оператора `new`, не генерирующей исключений, сделать код более безопасным? И наконец, почему на многих современных платформах не имеет смысла беспокоиться о возможных отказах при выполнении оператора `new`?

Задача 20. Контейнеры в памяти. Часть 1: уровни управления памятью — Сложность: 3

Управление памятью в современных операционных системах может быть очень сложным, но это — только один из уровней управления памятью, имеющий значение для программ на C++. Стандартная библиотека предоставляет несколько других уровней, каждый из которых (и все вместе) может оказать большое влияние на вашу программу.

Вопрос для новичка

1. Что такое диспетчеры памяти (известные также как распределители памяти), и в чем заключается их основная функция? Вкратце опишите две основные стратегии управления динамической памятью в C++.

Вопрос для профессионала

2. В чем состоит отличие различных уровней управления памятью в контексте стандартной библиотеки C++ и типичных средах, в которых используются реализации этой библиотеки? Что можно сказать об их взаимоотношениях, как они взаимодействуют друг с другом и как между ними распределяются обязанности?

Решение

Главный вопрос рассматриваемой пары задач будет задан в задаче 21 — сколько памяти используют разные стандартные контейнеры для хранения одинакового количества объектов одного и того же типа T?

Для того чтобы подойти к этому вопросу, мы сначала должны совершить небольшой экскурс в структуры данных, но перед этим поближе познакомиться с управлением динамической памятью. В частности, мы должны рассмотреть две основные темы:

- внутренние структуры данных, используемые контейнерами, такими как `vector`, `deque`, `list`, `set/multiset` и `map/multimap`; и
- как работает распределение динамической памяти.

Давайте начнем с распределения[32] динамической памяти, а затем разберемся, что это означает для стандартной библиотеки.

Диспетчеры памяти и их стратегии: краткий обзор

1. **Что такое диспетчеры памяти (известные также как распределители памяти), и в чем заключается их основная функция? Вкратце опишите две основные стратегии управления динамической памятью в C++.**

Для того чтобы разобраться в вопросе об объеме памяти, используемой различными контейнерами, надо сначала понять, как работают лежащие в их основе распределители динамической памяти. В конечном итоге, контейнер должен получить память от некото-

[32] Строго говоря, здесь рассматривается только частная задача распределения памяти, а именно задача *выделения* памяти (allocation). Однако в силу того, что выделение памяти тесно связано с ее *освобождением* (deallocation) в одну задачу *распределения* (которая, в свою очередь, является частью глобальной задачи управления памятью (memory management)), и распространенностью термина *распределение* в русскоязычной литературе, в дальнейшем в книге будет использован именно этот термин, а из контекста его использования будет понятно, о какой именно подзадаче идет речь. — *Прим. перев.*

рого *диспетчера памяти*, а этот диспетчер, в свою очередь, должен решить, как разделить доступную память, опираясь на некоторую *стратегию управления памятью*.

Вот как выглядит краткое описание двух распространенных стратегий управления памятью в C++. Более подробное рассмотрение данного вопроса выходит за рамки нашей книги; дополнительную информацию вы найдете в описании вашей операционной системы.

- *Распределение общего назначения*, или *универсальное распределение* может обеспечить блок памяти любого размера, который может запросить вызывающая программа (*размер запроса*, или *размер блока*). Такое распределение очень гибкое, но имеет ряд недостатков, основными из которых являются пониженная из-за необходимости выполнения большего количества работы производительность и *фрагментация памяти*, вызванная тем, что при постоянном выделении и освобождении блоков памяти разного размера образуется большое количество небольших по размеру несмежных участков свободной памяти.
- *Распределение фиксированного размера* всегда выделяет блоки памяти одного и того же фиксированного размера. Очевидно, что такая стратегия менее гибкая, чем универсальная, но зато она работает существенно быстрее и не приводит к фрагментации памяти.

Третья важная стратегия, *распределение со сборкой мусора*, не полностью совместима с указателями C и C++, функциями типа `malloc`, `new` и потому не имеет прямого отношения к рассматриваемому нами вопросу. Сборка мусора становится все более популярна и постепенно приходит и в C++ (но не для работы с указателями и оператором `new`). Я планирую рассмотреть этот вопрос в своей будущей книге, а сейчас вы можете обратиться к [C++CLI04] и [Jones96][33].

На практике мы часто сталкиваемся с комбинацией этих стратегий. Например, возможно, ваш диспетчер памяти использует схему общего назначения для всех запросов размером больше некоторого значения S, а в качестве оптимизации для всех запросов размером меньше S используется выделение блоков памяти фиксированного размера. Обычно достаточно неудобно иметь отдельные области памяти для запросов размером 1 байт, 2 байта и так далее, так что большинство диспетчеров используют отдельные области для выделения блоков, размер которых кратен некоторому числу, скажем, 16 байтам. Если вы запрашиваете блок размером 16 байтов, все отлично; но если вы запросите 17 байтов, то память будет выделена из области для 32-байтовых блоков, и 15 байтов памяти пропадут впустую. Это источник дополнительных расходов памяти, но об этом мы поговорим чуть позже.

Очевидный вопрос звучит следующим образом: кто выбирает используемую стратегию управления памятью?

Выбор стратегии

2. В чем состоит отличие различных уровней управления памятью в контексте стандартной библиотеки C++ и типичных средах, в которых используются реализации этой библиотеки? Что можно сказать об их взаимоотношениях, как они взаимодействуют друг с другом и как между ними распределяются обязанности?

Имеется ряд возможных уровней управления памятью, каждый из которых может скрывать предыдущий уровень.

---

[33] Кроме того, за дополнительной информацией о распределении памяти можно порекомендовать обратиться, например, к разделу 2.5 книги Д. Кнут *Искусство программирования, том 1. Основные алгоритмы, 3-е изд.* — М.: Издательский дом "Вильямс", 2000. — *Прим. ред.*

- Ядро *операционной системы* предоставляет базовые услуги по распределению памяти. Эта базовая стратегия распределения памяти и ее свойства могут изменяться от одной операционной системы к другой, и на этот уровень в наибольшей степени влияет используемое аппаратное обеспечение.
- *Библиотека времени выполнения компилятора*, используемая по умолчанию, содержит собственные средства работы с памятью, такие как оператор `new` в C++ или функция `malloc` в C, которые работают с использованием собственных служб распределения памяти. Эти службы, предоставляемые компилятором, могут представлять собой всего лишь небольшую обертку вокруг соответствующих служб операционной системы и наследовать их свойства. Возможно и другое решение, когда система управления памятью, предоставляемая компилятором, перекрывает стратегию операционной системы, получая от нее блоки большого размера, которые затем перераспределяет в соответствии с собственной стратегией.
- *Стандартные контейнеры и распределители* используют сервисы, предоставляемые компилятором и, в свою очередь, могут перекрывать их путем реализации собственных стратегий и оптимизаций.
- И наконец, *пользовательские контейнеры* и/или *пользовательские распределители* могут использовать любой из сервисов более низкого уровня (например, они могут обращаться непосредственно к сервисам операционной системы, если для данной программы не имеет значения переносимость) и работать независимо от них, т.е. так, как того хочет автор.

Все эти уровни показаны на рис. 20.1.

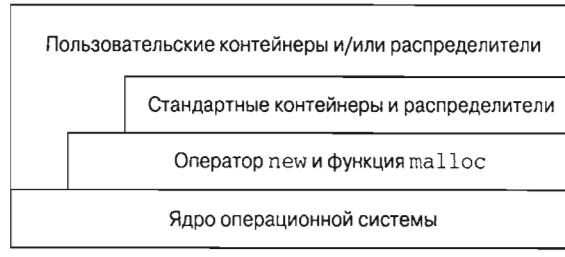

Рис. 20.1. Основные уровни управления памятью. Обычно каждый уровень реализуется посредством более низкого уровня(ей)

Таким образом, распределение памяти осуществляется различными способами и может варьироваться от операционной системы к операционной системе, от компилятора к компилятору в пределах одной операционной системы, от контейнера к контейнеру — и даже от объекта к объекту. Так, в случае использования объекта `vector<int>` применяется стратегия, реализованная в `allocator<int>`, а в случае объекта `vector<int, MyAllocator>` может использоваться совершенно иная стратегия выделения памяти.

➢ Рекомендация

Никогда не мешает знать, кто и за что отвечает. Потратьте немного времени, чтобы точно выяснить, какие стратегии используются на каждом уровне в используемой вами среде программирования.

Резюме

Управление памятью в современных операционных системах может быть очень сложным, но это — только один из уровней управления памятью, имеющий значение для программ на C++. Стандартная библиотека предоставляет несколько других уровней, в первую очередь при помощи собственных примитивов для выделения и освобождения памяти, посредством стандартных контейнеров и распределителей, а также контейнеров и распределителей памяти, которые вы можете написать самостоятельно.

Но когда вы запрашиваете память, знаете ли вы о том, что вы получите на самом деле, и во что это вам обойдется? Сколько памяти требуется стандартным контейнерам — в теории, и на практике? Именно об этом мы и поговорим в следующей задаче.

Задача 21. Контейнеры в памяти. Часть 2: какие они на самом деле? Сложность: 3

Когда вы запрашиваете память — что вы знаете о том, что вы получите и во что в действительности это вам обойдется? Сколько памяти используют стандартные контейнеры — теоретически, практически и в коде, который будет написан вами сегодня вечером?

Вопрос для новичка

1. Когда вы запрашиваете *n* байтов памяти с использованием `new` или `malloc`, в самом ли деле вы используете *n* байтов памяти? Поясните, почему.

Вопрос для профессионала

2. Сколько памяти используют различные стандартные контейнеры для хранения одинакового количества объектов одного и того же типа `T`?

Решение

Что попросишь, то получишь?

1. **Когда вы запрашиваете n байтов памяти с использованием new или malloc, в самом ли деле вы используете n байтов памяти? Поясните, почему.**

Когда вы запрашиваете *n* байтов памяти с использованием `new` или `malloc`, в действительности вы используете *как минимум n* байтов памяти, поскольку обычно диспетчер памяти должен добавить определенное количество памяти сверх запрошенной вами. Обычно это накладные расходы, связанные с внутренними структурами диспетчера памяти, размером и выравниванием объектов в памяти.

Рассмотрим накладные расходы, связанные с внутренними структурами диспетчера памяти. В универсальной схеме распределения памяти (т.е. с не фиксированными размерами блоков) диспетчер памяти должен помнить о том, какой размер имеет каждый выделенный блок, чтобы знать, какое количество памяти должно быть освобождено при вызове оператора `delete` или функции `free`. Обычно диспетчер памяти хранит это значение в начале реально выделяемого блока памяти, возвращая вам указатель на "вашу" область памяти, которая располагается непосредственно после необходимой области памяти, зарезервированной для служебной информации (см. рис. 21.1). Конечно, это означает, что должна быть выделена дополнительная память для сохранения размера блока, т.е. для числа, достаточного для хранения значения, равного максимально возможному корректному запросу памяти; обычно для этой цели достаточно числа, размер которого равен размеру указателя. При освобождении блока диспетчер памяти получает переданный ему вами указатель, вычитает из него количество байтов системной информации, считывает размер блока и выполняет его освобождение.

Конечно, в схеме с фиксированным размером блока (которая возвращает блоки памяти данного заранее известного размера) хранение дополнительной информации о размере блока не требуется, поскольку размер блока и так всегда известен.

Теперь рассмотрим расходы, связанные с размером и выравниванием объекта. Даже если не требуется хранение дополнительной информации, диспетчер памяти часто резервирует большее количество памяти, чем было запрошено, потому что память часто выделяется блоками определенного размера.

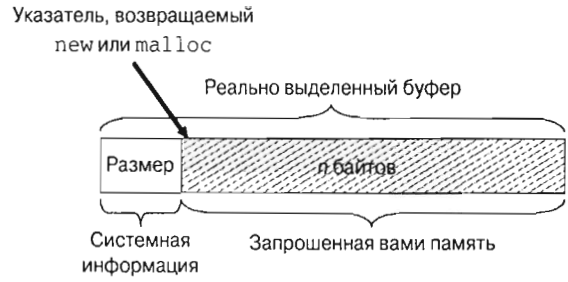

Рис. 21.1. Типичное выделение n *байтов памяти*

С одной стороны, на некоторых платформах выдвигается требование того или иного расположения объектов определенных типов данных в памяти (например, некоторые платформы требуют размещения указателей по адресам, кратным 4), и в случае несоблюдения этих требований программа оказывается либо неработоспособной, либо скорость ее работы существенно снижается. Такое требование к размещению данных называется *выравниванием* (alignment) и приводит к необходимости затрат дополнительной памяти для заполнения промежутков внутри объекта и, возможно, за концом данных объекта. Выравнивание затрагивает даже обычные старые встроенные массивы C, поскольку вносит свой вклад в значение, возвращаемое `sizeof(struct)`. На рис. 21.2 показана разница между внутренним заполнением и заполнением после конца объекта (хотя оба дают свой вклад в значение `sizeof(struct)`).

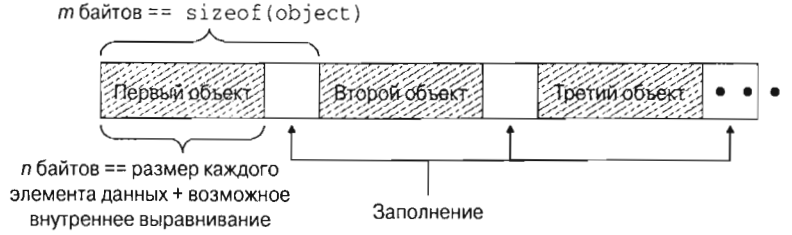

Рис. 21.2. Размещение в памяти массива n*-байтовых объектов с* m*-байтовым выравниванием (обратите внимание, что* `sizeof(object)==m`*)*

Например:

```
// Пример 21-1: предполагается, что sizeof(long) == 4, и все
//              значения long требуют выравнивания по
// 4-байтовой границе
//
struct X1 {
    char c1;  // Смещение 0, размер - 1 байт
              // Байты 1-3: 3 заполняющих байта
    long l;   // Байты 4-7: 4 байта на 4-байтовой границе
    char c2;  // Байт 8: 1 байт
              // Байты 9-11: заполняющих байтов (см. текст)
};            // sizeof(X1) == 12
```

В обозначениях рис. 21.2, в данном примере `n == 1 + 3 + 4 + 1 == 9` и `m == sizeof(X1) == 12`[34]. Заметим, что в значение `sizeof(X1)` вносят вклад все заполнители — как внутренние, так и внешние. Может показаться, что внешнее заполнение за

[34] Только никуда негодная реализация может использовать заполняющие байты сверх минимально необходимого количества.

концом объекта излишне, но оно необходимо, например, когда вы работаете с массивом объектов X1, располагающихся в памяти один за другим, чтобы обеспечить выравнивание данных типа long по 4-байтовой границе. Такое выравнивание за концом данных зачастую удивляет людей, впервые сталкивающихся с размещением данных в памяти. Особенно может удивить следующий результат перестановки полей структуры:

```
// Пример 21-2: измененная структура из примера 21-1
//
struct X2 {
    long l;      // Байты 0-3
    char c1;     // Байт 4
    char c2;     // Байт 5
                 // Байты 6-7: 2 заполняющих байта
};               // sizeof(X2) == 8
```

Теперь члены-данные действительно располагаются в памяти непрерывно (n == 6)[35], но все еще имеется дополнительное пространство за концом объекта. Это делает размер объекта равным m == sizeof(X2) == 8. Это заполнение за концом объекта оказывается наиболее заметным при создании массива объектов X2. Заполняющие шестой и седьмой байты показаны на рис. 21.2 незаштрихованными квадратами.

Кстати, именно поэтому при написании стандарта было достаточно трудно сформулировать требование "последовательного расположения" элементов vector в том же смысле, что и массивы. На рис. 21.2 память рассматривается как непрерывная, несмотря на наличие областей неиспользуемой памяти. Так что же в действительности означает "последовательное расположение"? По сути, последовательно расположены отдельные блоки памяти размером sizeof(struct), и такое определение вполне работоспособно, поскольку sizeof(struct) включает дополнительную заполняющую память.

Стандарт C++ гарантирует, что вся память, выделенная оператором new или функцией malloc, будет надлежащим образом выровнена для всех возможных объектов, которые вы можете захотеть в ней сохранить, а это означает, что оператор new и функция malloc должны удовлетворять самому строгому типу выравнивания на данной платформе.

Альтернативная схема выделения памяти блоками фиксированного размера может использовать области памяти для блоков определенных размеров, кратных некоторому базовому размеру *m*, и при запросе *n* байтов возвращать блок с размером, округленным до ближайшего большего кратного *m*.

Память и стандартные контейнеры: теория

Теперь мы перейдем к главному вопросу данной задачи.

2. Сколько памяти используют различные стандартные контейнеры для хранения одинакового количества объектов одного и того же типа T?

Каждый стандартный контейнер использует собственную структуру памяти, что приводит к различным накладным расходам памяти на один хранимый объект.

- Внутреннее представление, используемое vector<T> для хранения данных, представляет собой непрерывный C-массив объектов типа T, так что никаких дополнительных расходов памяти на хранение элементов у этого контейнера нет

[35] Компилятор не может самостоятельно выполнить перестановку данных из примера 21-1 к виду из примера 21-2. Стандарт требует, чтобы все данные, располагающиеся в одной и той же группе public, protected или private располагались компилятором в указанном порядке. Если же вы предваряете ваши данные спецификаторами доступа, то компилятор *может* выполнить перестановку в пределах групп данных, разделенных спецификаторами доступа для улучшения размещения. Это является одной из причин, по которой некоторые программисты предпочитают предварять каждый член-данные спецификатором доступа.

(конечно, не считая заполнения для выравнивания; заметим, что в случае вектора "последовательное размещение" имеет тот же смысл, что и у С-массива, как показано на рис. 21.2).

- `deque<T>` можно рассматривать как `vector<T>`, чья внутренняя память разделена на части. `deque<T>` хранит блоки, или "страницы" объектов; реальный размер страниц стандартом не определяется и зависит в первую очередь от размера объекта `T` и от выбора, сделанного разработчиком вашей стандартной библиотеки. Такое разбиение на страницы требует хранения одного дополнительного указателя на страницу, что приводит к дополнительным расходам, составляющим доли бита на один объект. Например, в системе с 8-битовыми байтами и 4-битовыми целыми числами и указателями `deque<int>` с 4-Кбайтовой страницей приводит к расходам памяти, равным 1/32=0.03125 бита на один `int`. Других накладных расходов не имеется, поскольку `deque<T>` не хранит никаких дополнительных указателей или другой информации для отдельных объектов `T`. В стандарте нет требования, чтобы страницы `deque<T>` представляли собой С-массивы, однако это — общепринятый способ реализации.
- `list<T>` представляет собой двухсвязный список узлов, хранящих элементы типа `T`. Это означает, что для каждого элемента `T` в `list<T>` хранятся также два указателя на предыдущий и последующий узлы в списке. Каждый раз при вставке нового элемента `T` мы также создаем два указателя, так что `list<T>` требует как минимум двух указателей накладных расходов памяти на один элемент.
- `set<T>` (а также аналогичные в этом отношении `multiset<T>`, `map<Key,T>` и `multimap<Key,T>`) тоже хранят узлы, в которых содержатся объекты типа `T` (или `pair<const Key,T>`). Обычная реализация `set` представляет собой дерево с тремя дополнительными указателями на один узел дерева. Зачастую, услышав об этом, многие спрашивают: "Почему три указателя? Разве недостаточно двух — на левый и правый дочерние узлы?" Дело в том, что требуется еще один указатель на родительский узел, иначе задача определения "следующего" узла по отношению к некоторому произвольно взятому не может быть решена достаточно эффективно. (Возможны и другие способы реализации этих контейнеров; например, можно использовать т.н. alternating skip list — но и в этом случае требуется как минимум три указателя на каждый элемент (см. [Marrie00]).)

В табл. 21.1 приведены дополнительные расходы памяти на хранение одного элемента в различных стандартных контейнерах. Заметим, что иногда можно снизить расходы на хранение объектов, перенеся их на итераторы — т.е. часть работы может быть перенесена в итераторы, что даст нам "толстые" итераторы в обмен на "худые" контейнеры. Впрочем, я не знаком ни с одной коммерческой реализацией стандартной библиотеки, которая бы придерживалась этой методики.

Таблица 21.1. Дополнительные расходы памяти на хранение одного объекта у разных контейнеров

Контейнер	Типичные накладные расходы памяти на один хранимый объект
`vector`	Нет дополнительных расходов
`deque`	Пренебрежимо малые расходы — обычно доли битов
`list`	Два указателя
`set`, `multiset`	Три указателя
`map`, `multimap`	Три указателя на `pair<const Key, T>`

Память и стандартные контейнеры: практика

Теперь мы перейдем к самой интересной части. Не будем спешить делать выводы из табл. 21.1. Например, исходя из приведенных данных, вы можете сделать вывод о том, что `list` требует меньших расходов памяти, чем `set` — ведь первому требуется только два дополнительных указателя, а последнему — три. Интересно, что это может оказаться неверным, если принять во внимание стратегию распределения памяти.

Рассмотрим вопрос более детально, для чего обратимся к табл. 21.2, в которой приведены типичные структуры узлов, используемые реализациями `list`, `set`/`multiset` и `map`/`multimap`.

Таблица 21.2. Блоки динамической памяти, используемые для хранения объектов в разных контейнерах

Контейнер	Типичные блоки памяти для хранения объектов
`vector`	Отсутствуют; объекты индивидуально не хранятся
`deque`	Отсутствуют; объекты хранятся в страницах; почти все страницы хранят большое количество объектов

`list`	`struct LNode {`
	`  LNode* prev;`
	`  LNode* next;`
	`  T object;`
	`};`

`set`, `multiset`	`struct SNode {`
	`  SNode* prev;`
	`  SNode* next;`
	`  SNode* parent;`
	`  T object;`
	`}; // Или эквивалентная структура`

`map`, `multimap`	`struct MNode {`
	`  MNode* prev;`
	`  MNode* next;`
	`  MNode* parent;`
	`  std::pair<const Key, T> data;`
	`}; // Или эквивалентная структура`

Теперь рассмотрим, что происходит в реальной ситуации при приведенных далее предположениях, которые справедливы для большинства распространенных в настоящее время платформ.

- Указатели и целые числа имеют размер 4 байта (типично для 32-битовых платформ).
- `sizeof(string)` равно 16. Заметим, что это просто размер непосредственно объекта `string`; здесь не учитываются буферы данных, которые могут быть выделены строке; количество и размер внутренних буферов `string` варьируется от реализации к реализации, но это не влияет на приведенные далее сравнительные результаты. (Рассматриваемое значение `sizeof(string)` представляет собой реальную величину, взятую из одной распространенной реализации стандартной библиотеки.)
- Стратегия распределения памяти по умолчанию использует выделение блоков фиксированного размера; размеры блоков кратны 16 байтам (типичное распределение памяти в Microsoft Visual C++).

Таблица 21.3. Реальные расходы памяти на хранение объекта, рассматриваемые в предположении, что `sizeof(string) == 16`, указатели и `int` имеют размер 4 байта, и выделение памяти происходит 16-байтовыми блоками

Контейнер	Теоретический размер данных узла	Реальный размер блока выделенной для узла памяти (с учетом выравнивания и накладных расходов при выделении памяти)
`list<char>`	9 байтов	16 байтов
`set<char>`, `multiset<char>`	13 байтов	16 байтов
`list<int>`	12 байтов	16 байтов
`set<int>`, `multiset<int>`	16 байтов	16 байтов
`list<string>`	24 байта	32 байта
`set<string>`, `multiset<string>`	28 байтов	32 байта

В табл. 21.3 приведены результаты простого анализа с использованием рассмотренных выше предположений. Вы можете повторить эксперимент на своей платформе со своим компилятором — просто подставив собственные значения. Чтобы написать программу, которая определяет реальные накладные расходы при выделении блока определенного размера на вашей платформе, обратитесь к приложению 3 в книге Джона Бентли (Jon Bentley) [Bentley00].

Благодаря табл. 21.3 мы немедленно обнаруживаем один интересный результат: во многих случаях — примерно для 75% всех возможных размеров типа `T` — накладные расходы при использовании `list` и `set`/`multiset` в рассматриваемой среде оказываются одинаковы. Более того, вот результат, который еще интереснее, — `list<char>` и `set<int>` имеют в данной среде одни и те же реальные накладные расходы, несмотря на то, что последний контейнер хранит данные большего размера *и* требует большего количества дополнительной информации для каждого узла.

Если используемое количество памяти является важным фактором при выборе вами структуры данных в конкретной ситуации, потратьте несколько минут на проведение подобного анализа для вашей системы и посмотрите на реальные различия в потреблении памяти разными контейнерами — иногда эти различия гораздо меньше, чем вы думаете!

Резюме

Для каждого вида контейнера определяются свои компромиссы между расходами памяти и производительностью. При использовании `vector` и `set` можно быстро решить те задачи, которые невозможно столь же эффективно решить при использовании `list` — например, поиск за время O(log N)[36]; при использовании `vector` можно сделать вещи, невозможные при применении `list` или `set` — например, произвольный доступ. Вставка элемента в середину легко выполняется в `list`, менее эффективно — в `set`, и очень медленно — в `vector`. Словом, примеров таких по-разному выполняющихся задач очень много. Большая гибкость зачастую требует больших накладных расходов памяти, но если учесть выравнивание данных и возможные стратегии распределения памяти, то различие может оказаться куда меньшим, чем вы думаете! Вопросы выравнивания данных и оптимизации использования памяти рассматривались также в книге [Sutter00].

[36] Если содержимое вектора отсортировано.

➢ Рекомендация

Следует четко представлять, к каким реальным расходам памяти приводит использование различных видов контейнеров и стратегии распределения динамической памяти.

Задача 22. Новый взгляд на new. Часть 1: многоликий оператор new

Сложность: 4

Любой класс, предоставляющий собственный оператор new или new[], должен также обеспечить соответствующие версии обычного оператора new, размещающего new и new, не генерирующего исключений. В противном случае у пользователей вашего класса могут возникнуть ненужные проблемы.

Вопрос для новичка

1. Какие три варианта оператора new описаны в стандарте C++?

Вопрос для профессионала

2. Что такое оператор new, специфичный для класса, и как им следует пользоваться? Опишите, в каких случаях вы должны быть особо осторожны при предоставлении собственных, специфичных для класса операторов new и delete.

3. Какой именно оператор new вызывается в приведенном далее коде в строках, пронумерованных от 1 до 4?

```
class Base {
public:
    static void* operator new(std::size_t, const FastMemory&);
};
class Derived : public Base {
  //...
};
Derived* p1 = new Derived;                                  // 1

Derived* p2 = new (std::nothrow) Derived;                   // 2

void* p3 = /* Некоторая область памяти, достаточная
              для размещения Derived */ ;
new (p3) Derived;                                           // 3

FastMemory f;
Derived* p4 = new (f) Derived;                              // 4
```

Решение

В этой и следующей задачах я хочу сформулировать и обосновать два основных совета.

- Любой класс, предоставляющий собственный оператор new или new[], должен также обеспечить соответствующие версии обычного оператора new, размещающего new и new, не генерирующего исключений. В противном случае у пользователей вашего класса могут возникнуть ненужные проблемы.
- Избегайте использования new(nothrow) и убедитесь, что, когда вы проверяете отказ new, вы действительно проверяете именно то, что хотите проверить.

Советы могут показаться неожиданными, поэтому давайте детально их проанализируем. Для простоты я не упоминаю отдельно оператор new для массивов, но все сказанное об операторе new для одного объекта, относится и к нему.

Размещающий, обычный и не генерирующий исключений оператор new

1. Какие три варианта оператора new описаны в стандарте C++?

В стандарте C++ описаны три варианта оператора new и разрешено произвольное количество дополнительных перегрузок.

Одним полезным видом этого оператора является размещающий new, который строит объект в имеющейся области памяти, без выделения нового пространства. Например:

```
// Пример 22-1(a): использование размещающего new
//

// Выделение достаточного количества памяти
void* p = ::operator new( sizeof(T) );
new(p)T;  // Создание объекта T по адресу p, вероятно, вызовом
          // ::operator new(std::size_t, void*) throw()
```

Стандарт также предоставляет "обычный старый new", который не получает никаких дополнительных параметров и не генерирует исключений (nothrow new). Ниже представлен полный список перегруженных операторов new из стандарта C++.

```
// Перегрузки оператора new в стандарте языка
// (имеются соответствующие версии и для new[]):
//
void* ::operator new(std::size_t size) throw(std::bad_alloc);
        // Обычный старый оператор new
        // Использование: new T

void* ::operator
    new(std::size_t size, const std::nothrow_t&) throw();
        // Оператор new, не генерирующий исключений
        // Использование: new (std::nothrow) T

void* ::operator new(std::size_t size, void* ptr) throw();
        // Размещающий оператор new
        // Использование: new (ptr) T
```

Программа может заменить все, кроме последнего, операторы new на свои собственные версии. Все эти стандартные функции находятся в глобальной области действия, а не в пространстве имен std. Табл. 22.1 дает краткие характеристики стандартных версий new.

Таблица 22.1. Сравнение стандартных версий оператора new

Версия new	Дополнительный параметр	Выделение памяти	Возможность сбоя[37]	Генерация исключений	Замещаемость
Обычный	Нет	Да	Да (генерирует исключение)	std::bad_alloc	Да
Не генерирующий исключений	std::nothrow_t	Да	Да (возвращает 0)	Нет	Да
Размещающий	void*	Нет	Нет	Нет	Нет

[37] После выполнения оператора new вызывается конструктор объекта, и, конечно, этот конструктор может вызвать сбой — но в данном случае нас это не волнует. Сейчас нас интересует только вопрос о сбое в самом операторе new.

Рассмотрим пример, показывающий применение описанных версий new.

```
// Пример 22-1(б): использование перегрузок оператора new
//
FastMemory f;
new (f) T;                      // Вызов некоторого
    // пользовательского оператора new(std::size_t,
    // FastMemory&) (или аналогичного, с преобразованием
    // типов), по-видимому, для выбора пользовательской
    // области памяти

new (42, 3.14159, "xyzzy") T;  // Вызов некоторого
    // пользовательского оператора
    // new(std::size_t, int, double, const char*)
    // (или аналогичного, с преобразованием типов)

new (std::nothrow) T;           // Вероятно, вызов
    // стандартного или пользовательского оператора
    // ::new(std::size_t, const std::nothrow_t&) throw()
```

В каждом из случаев в примерах 22-1*а* и 22-1*б* параметры в скобках в выражении с оператором new превращаются в дополнительные параметры, передаваемые оператору new при вызове. Конечно, в отличие от случая в примере 22-1*а*, все выражения в примере 22-1*б*, вероятно, используют тот или иной механизм выделения памяти вместо размещения объекта в некоторой заданной области памяти.

Оператор new, специфичный для класса

2. Что такое оператор new, специфичный для класса, и как им следует пользоваться? Опишите, в каких случаях вы должны быть особо осторожны при предоставлении собственных, специфичных для класса операторов new и delete.

Помимо разрешения программе переопределить некоторые из глобальных операторов new, C++ также позволяет классу определить собственные версии операторов, специфичные для данного класса. При рассмотрении примеров 22-1*а* и 22-1*б* вы, наверное, обратили внимание на слово "вероятно" в двух комментариях?

```
new(p)T;   // Создание объекта T по адресу p, вероятно, вызовом
           // ::operator new(std::size_t, void*) throw()

new (std::nothrow) T;           // Вероятно, вызов
    // стандартного или пользовательского оператора
    // ::new(std::size_t, const std::nothrow_t&) throw()
```

Почему только "вероятно"? Потому что вызываемые операторы — не обязательно операторы из глобальной области видимости, а могут быть и специфичными для данного класса T. Для ясного понимания этого момента обратите внимание на два интересных взаимодействия между оператором new класса и глобальным оператором new.

- Хотя вы не можете непосредственно заменить стандартный размещающий оператор new, вы *можете* написать собственный размещающий оператор new для данного класса, который будет использован для размещения объектов этого класса.
- Вы можете добавить специфичный для класса оператор new, не генерирующий исключений, как с заменой соответствующего глобального оператора, так и без таковой.

Таким образом, в приведенных строках кода возможна ситуация, когда T (или один из его базовых классов) предоставляет свои собственные версии одного (или обоих) из этих операторов, и в таком случае будут использованы именно эти операторы.

Рассмотрим простой пример, в котором класс предоставляет все три варианта оператора new.

```
// Пример 22-2: специфичные для класса версии new
//
class X {
public:
    static void* operator new(std::size_t ) throw();     // 1
    static void* operator new(std::size_t,
                        const std::nothrow_t& ) throw(); // 2
    static void* operator new(std::size_t,
                        void* ) throw();                 // 3
};

X* p1 = new X;                                    // Вызов 1
X* p2 = new (std::nothrow) X;                     // Вызов 2
void* p3 = /* Некоторая область памяти, достаточная
              для размещения X */ ; ;
new (p3) X;                                       // Вызов 3(!)
```

Я поместил восклицательный знак после третьего вызова для того, чтобы еще раз привлечь ваше внимание к тому факту, что вы можете обеспечить свой класс собственной версией размещающего оператора new, несмотря на то, что не можете заменить глобальную версию этого оператора.

Сюрприз сокрытия имен

Весь изложенный материал приводит нас к основной причине появления этого материала в книге, а именно — проблеме сокрытия имен.

3. Какой именно оператор new вызывается в приведенном далее коде в строках, пронумерованных от 1 до 4?

```
// Пример 22-3: сокрытие имени new
//
class Base {
public:
    static void* operator new(std::size_t, const FastMemory&);
};
class Derived : public Base {
  //...
};
Derived* p1 = new Derived;                                  // 1

Derived* p2 = new (std::nothrow) Derived;                   // 2
void* p3 = /* Некоторая область памяти, достаточная
              для размещения  Derived */ ;

new (p3) Derived;                                           // 3

FastMemory f;
Derived* p4 = new (f) Derived;                              // 4
```

Большинство из нас знакомы с проблемой сокрытия имен в других контекстах, когда имя в производном классе скрывает имя в базовом классе, но следует помнить, что сокрытие имен может проявиться и при работе с оператором new.

Вкратце вспомним, как работает поиск имен: компилятор начинает поиск в текущей области видимости (в нашем случае — в области видимости Derived) и ищет требуемое имя (в нашем случае — operator new); если ни одного экземпляра искомого имени не найдено, он переходит к охватывающей области видимости (области видимости Base, а затем глобальной области видимости) и повторяет поиск. Как только найдена область видимости хотя бы с одним именем (в нашем случае — область видимости Base), поиск прекращается и работа продолжается только в плане

выбора наилучшего соответствия среди найденных имен; это означает, что остальные охватывающие области видимости (в данном случае — глобальная область видимости) более не рассматриваются и все функции в них оказываются скрыты. Далее компилятор просматривает все обнаруженные имена, выбирает функцию при помощи разрешения перегрузки и проверяет права доступа к ней, чтобы выяснить, может ли найденная функция быть вызвана в данном контексте. Внешние области видимости игнорируются, даже если ни одна из найденных перегруженных функций не имеет подходящей сигнатуры, т.е. ни одна из них не может быть вызвана. Внешние области видимости также игнорируются, если функция с соответствующей вызову сигнатурой не может быть вызвана из-за недоступности. (Более детально вопросы поиска и сокрытия имен рассмотрены в [Sutter00].)

Это означает, что если класс (или любой из его базовых классов) содержит собственный оператор new с любой сигнатурой, то эта функция скрывает все глобальные операторы new и вы не имеете возможности записать обычное выражение с оператором new, в котором будет использоваться глобальная версия оператора. Вот что это обозначает в контексте примера 22-3.

```
Derived* p1 = new Derived;       // Ошибка: соответствия нет

Derived* p2 =
    new(std::nothrow) Derived;   // Ошибка: соответствия нет

void* p3 = /* Некоторая область памяти, достаточная
              для размещения Derived */ ;
new (p3) Derived;                // Ошибка: соответствия нет

FastMemory f;
Derived* p4 =
    new(f) Derived;              // Вызов Base::operator new()
```

Но что если мы хотим использовать глобальные версии операторов, как минимум в первых двух случаях, там, где перегрузка в базовом классе приводит к ошибке? Единственный разумный способ обеспечить возможность классу Derived использовать глобальные операторы new — это предоставить соответствующие функции в классе, которые просто передают вызов глобальным операторам (в противном случае в коде надо использовать квалифицированные имена ::new, чтобы компилятор использовал глобальные версии операторов).

Это приводит к интересному выводу, который лучше всего выразить в виде рекомендации (см. также [Meyers97]).

➤ Рекомендация

Если вы предоставляете хотя бы один специфичный для данного класса оператор new, следует также обеспечить класс обычным (без дополнительных параметров) оператором new.

Специфичная для класса версия оператора почти всегда должна сохранять семантику глобальной версии, поэтому следует объявлять ее со спецификацией исключений throw(std::bad_alloc), а реализовывать ее желательно с использованием глобальной версии — если только вам действительно не нужны некоторые специфические действия оператора new.

```
// Предпочтительная реализация оператора new, специфичного
// для класса
//
void* C::operator new(std::size_t s) throw(std::bad_alloc) {
    return ::operator new( s );
}
```

Заметим, что вы можете вызвать замещенную стандартную версию оператора new вместо используемой по умолчанию, но обычно это именно то, что требуется: в большинстве случаев замещенный глобальный оператор new используется для отладки или контроля за использованием памяти, и такое поведение желательно отразить и в специфичных для классов версиях new.

Если же вы не хотите поступать таким образом, то ваш класс будет невозможно использовать кодом, который будет пытаться динамически распределять память для объектов обычным способом.

➢ Рекомендация

Если вы предоставляете хотя бы один специфичный для данного класса оператор new, следует также обеспечить класс размещающим оператором new.

Этот оператор должен сохранять семантику глобальной версии, поэтому его следует объявлять как не генерирующий исключений и реализовать посредством глобальной версии.

```
// Предпочтительная реализация размещающего оператора new,
// специфичного для класса
//
void* C::operator new( std::size_t s, void* p ) throw() {
    return ::operator new( s, p );
}
```

Если вы этого не сделаете, то не сможете работать с кодом, который использует размещающий оператор new с вашим классом. В частности, реализации контейнеров стандартной библиотеки часто используют размещающие конструирование объектов и ожидают, что оно будет работать, как обычно; в конце концов, это единственный способ явного вызова конструктора в C++. Если вы не напишете такой оператор new, то, скорее всего, вы не сможете воспользоваться даже std::vector<C>.

➢ Рекомендация

Если вы предоставляете хотя бы один специфичный для данного класса оператор new, то следует также предоставить оператор new, не генерирующий исключений, на тот случай, если пользователи вашего класса захотят им воспользоваться; в противном случае этот оператор окажется скрытым другими перегрузками оператора new в данном классе.

Реализовать этот оператор можно с использованием глобального не генерирующего исключений new.

```
// Вариант А реализации не генерирующего исключений
// оператора new класса
//
void* C::operator new(std::size_t s, const std::nothrow_t& n)
                    throw() { return ::operator new( s, n ); }
```

Другой вариант реализации использует обычный оператор new класса (выполняется то же, что глобальным не генерирующим исключений оператором new; однако при этом мы застрахованы от изменения глобального оператора другим программистом):

```
// Вариант Б реализации не генерирующего исключений
// оператора new класса
//
void* C::operator
    new(std::size_t s, const std::nothrow_t&) throw()
{
```

```
    try {
        return C::operator new( s );
    }
    catch( ... ) {
        return 0;
    }
}
```

Заметим, что рассмотренные варианты вызовов глобальных операторов невозможно реализовать при помощи using-объявления, такого как `using ::operator new;`. Единственное место, где такое объявление может оказаться полезным — в описании класса C. В пределах класса можно использовать только using-объявления, которые вносят имена из базовых классов, но не такие, как имена из глобальной области видимости или имена из других классов. Требование, чтобы вызывающий код сам добавлял using-объявления, не только обременительно, но и бесполезно, поскольку мы можем быть не в состоянии его изменять. Часть кода может находиться в модулях, доступных только для чтения, как, например, библиотеки сторонних производителей, или даже внутри стандартной библиотеки C++, если мы попытаемся обеспечить стандартные контейнеры доступом к специфичному для класса размещающему оператору new.

Резюме

Если вы предоставляете хотя бы один специфичный для данного класса оператор new, то:

- следует также обеспечить класс обычным (без дополнительных параметров) оператором new;
- следует также обеспечить класс размещающим оператором new;
- следует также обеспечить класс оператором new, не генерирующим исключений, на тот случай, если пользователи вашего класса захотят им воспользоваться; в противном случае этот оператор окажется скрытым другими перегрузками оператора new в данном классе.

В следующей задаче мы более подробно разберемся, что же означает сбой оператора new и как лучше всего обнаружить и обработать его. Мы также увидим, что в своих программах желательно избегать применения new(nothrow), но что еще более неожиданно, мы обнаружим, что у ряда популярных платформ отказы при распределении памяти не сообщают о себе так, как того требует стандарт.

Задача 23. Новый взгляд на new. Часть 2: прагматизм в управлении памятью Сложность: 5

Избегайте использования new(nothrow) и убедитесь, что при проверке отказа new вы действительно проверяете именно то, что хотите проверить.

Вопрос для новичка

1. Опишите два основных стандартных способа сообщения об ошибке оператором new в случае нехватки памяти.

Вопрос для профессионала

2. Помогает ли использование не генерирующей исключения версии оператора new сделать код более безопасным с точки зрения исключений? Обоснуйте ваш ответ.
3. Опишите реальные ситуации — в пределах стандарта C++ или вне его — когда проверка исчерпания памяти невозможна или бесполезна.

Решение

В предыдущей задаче я проиллюстрировал и обосновал следующую рекомендацию.

- Любой класс, предоставляющий собственный оператор new или new[], должен также предоставлять соответствующие специфичные для данного класса версии обычного оператора new, размещающего оператора new и оператора new, не генерирующего исключений. В противном случае у пользователей вашего класса могут возникнуть лишние проблемы.

Сейчас мы уделим наше внимание вопросу о том, что же означает отказ оператора new и как лучше всего обнаружить его и обработать.

- Избегайте использования new(nothrow), и убедитесь, что при проверке отказа new вы действительно проверяете именно то, что хотите проверить.

Первая часть рекомендации может показаться немного неожиданной, но последняя — просто удивительной, потому что на некоторых распространенных платформах отказы выделения памяти обычно даже не проявляются так, как того требует стандарт.

Здесь я опять для простоты не буду отдельно рассматривать оператор new для массивов. Все, что будет сказано об операторе new для одного объекта, относится и к оператору для массивов.

Исключения, ошибки и new(nothrow)

1. Опишите два основных стандартных способа сообщения об ошибке оператором new в случае нехватки памяти.

В то время как остальные разновидности оператора new сообщают об ошибках, генерируя исключение bad_alloc, не генерирующий исключений оператор new сообщает о них проверенным временем способом — как и функция malloc, он возвращает нулевой указатель. Этим гарантируется, что такой оператор, как следует из его названия, никогда не генерирует исключений.

Главный вопрос — дает ли это нам что-нибудь или нет? Ряд программистов ошибочно полагали, что использование не генерирующего исключений оператора new увеличивает безопасность программы, поскольку уменьшает количество возможных исключений. Но так ли это на самом деле?

2. Помогает ли использование не генерирующей исключения версии оператора new сделать код более безопасным с точки зрения исключений? Обоснуйте ваш ответ.

Ответ (возможно, неожиданный): нет, на самом деле не помогает. Сообщения об ошибках и их обработка — "две большие разницы".

Выбор между генерацией исключения bad_alloc и возвратом нулевого указателя — это выбор между двумя эквивалентными способами сообщения об ошибке. Таким образом, обнаружение и обработка ошибки представляет собой выбор между проверкой наличия исключения и проверкой значения указателя, не нулевой ли он.

Для вызывающей оператор new программы отличие этих способов не более чем синтаксическое Это означает, что можно написать две совершенно одинаковые с точки зрения безопасности вообще и безопасности исключений в частности программы — для каждого из способов обнаружения ошибки, поскольку это всего лишь синтаксис, который приводит к минимальным изменениям в структуре вызывающей функции — например, вместо чего-то наподобие

```
if (null) { HandleError(); throw MyOwnException(); }
```

будет использоваться что-то вроде

```
catch(bad_alloc) { HandleError(); throw MyOwnException(); }
```

Ни один из способов, которыми оператор new сообщает о происшедшей ошибке, не дает никакой дополнительной информации и не обеспечивает дополнительной безопасности, так что ни один из них не делает программу безопаснее или менее подверженной ошибкам — конечно, при аккуратном написании обработки ошибок.

Но в чем тогда проявляется отличие для вызывающей программы, которая не проверяет наличие ошибок? В этом случае единственное различие будет в том, как именно проявится ошибка, но конечный результат будет одинаково печален. Либо неперехваченное исключение bad_alloc без всяких церемоний завершит выполнение программы (со сверткой стека или без нее), либо непроверенный нулевой указатель приведет при разыменовании к нарушению защиты памяти и немедленному аварийному завершению программы. Оба варианта достаточно катастрофичны для программы, но все же не перехваченное исключение имеет некоторые плюсы: по крайней мере, в этом случае будут сделаны попытки уничтожения некоторых из объектов и тем самым — освобождения ресурсов; кроме того, ряд аккуратно написанных объектов типа TextEditor при этом постараются сохранить свое состояние, чтобы впоследствии восстановить его. (Предупреждение: если память действительно исчерпана, то написать код, который бы корректно свернул стек и сохранил состояние объектов без использования дополнительной памяти, оказывается сложнее, чем кажется. Но, с другой стороны, вряд ли аварийный останов из-за обращения к памяти по некорректному указателю будет в чем-то лучше.)

Отсюда мы выводим первую мораль:

➢ Рекомендация

Мораль №1: избегайте использования оператора new, не генерирующего исключения.

Не генерирующий исключений оператор new не добавляет программе ни корректности, ни безопасности исключений. Для некоторых ошибок, а именно ошибок, которые игнорируются программой, — этот вариант оказывается хуже, чем вариант new с генерацией исключения, поскольку последний дает хоть какой-то шанс для сохранения состояния во время свертки стека. Как указывалось в предыдущей задаче, если классы предоставляют собственные операторы new, но при этом забывают об операторах new, не генерирующих исключений, то последние будут скрыты и не бу-

дут работать. В большинстве случаев не генерирующие исключений операторы new не дают никаких преимуществ, так что в силу всего сказанного выше их следует избегать.

Мне представляется, что есть только две ситуации, когда применение оператора new, не генерирующего исключения, может дать определенный выигрыш. Первый случай постепенно становится все менее значимым: это перенос большого количества старых приложений C++, в которых предполагалось, что ошибка в операторе new приведет к возврату нулевого указателя, и выполнялись соответствующие проверки. В таком случае может оказаться проще глобально заменить все "new" на "new(nothrow)" в старых файлах; однако таких файлов осталось не так уж и много, так как генерацию оператором new исключения bad_alloc трудно назвать новшеством.

Второй случай, когда оправданно применение new, не генерирующего исключения — его применение в функции, критичной ко времени выполнения, или во внутреннем цикле, а сама функция компилируется слабым компилятором, генерирующим неэффективный код обработки исключений и приводит к заметной разнице во времени работы функции с использованием разных версий оператора new — генерирующей и не генерирующей исключений. Заметим, что, когда я говорю "заметной", я говорю о времени работы всей функции в целом, а не только об игрушечном тесте самого оператора new. Только после того, как будет доказано наличие заметной разницы во времени работы такой функции, в ней можно рассмотреть возможность использования оператора new, не генерирующего исключений, но при этом обязательно рассмотреть также другие возможные методы повышения производительности выделения памяти, включая разработку собственного распределителя, работающего с блоками фиксированного размера и т.п. (только перед тем как приступить к этой работе, прочтите задачу 21).

Итак, мы пришли к морали №2:

> ➢ **Рекомендация**
>
> Мораль №2: очень часто проверка отказа в операторе new бесполезна.

Эта рекомендация может ужаснуть многих программистов. "Как вы можете предлагать такое — *не* проверять результат работы оператора new или, по крайней мере, утверждать, что такая проверка не важна? — могут спросить такие справедливо возмущенные люди. — Проверка ошибок — это краеугольный камень надежного программирования!" Да, это так — в общем случае, но — увы! — по причинам, свойственным исключительно распределению памяти, для него это не настолько важно, в отличие от других сбоев, которые должны быть проверены в обязательном порядке. Отсюда вытекает следующий вопрос.

Теория и практика

3. Опишите реальные ситуации — в пределах стандарта C++ или вне его — когда проверка исчерпания памяти невозможна или бесполезна.

Вот несколько причин, по которым проверка отказа при выполнении оператора new оказывается не столь важной, как можно было бы предположить.

Проверка результата работы оператора new может оказаться бесполезной в операционной системе, которая реально не выделяет память (commit) до тех пор, пока к ней не осуществляется обращение. В некоторых операционных системах[38] системные функции выделения памяти завершаются всегда успешно. Точка.

"Минутку, минутку, — можете удивиться вы. — Как же так — выделение памяти успешно даже в том случае, когда этой памяти в действительности нет?" Причина в

[38] В некоторых версиях Linux, AIX и других операционных системах (например, в OS/2) такое отложенное выделение памяти является поведением по умолчанию конфигурируемого свойства операционной системы.

том, что операция выделения памяти просто *записывает запрос* на выделение определенного количества памяти, но при этом реального выделения памяти для запрашивающего процесса не происходит до тех пор, пока процесс не обратится к ней. Даже когда выделенная память используется процессом, часто реальная (физическая или виртуальная) память выделяется постранично, при обращении к конкретной странице, так что может оказаться, что в действительности вместо большого блока памяти процессу выделяется только его часть — с которой процесс реально работает.

Заметим, что если оператор new использует возможности операционной системы непосредственно, то он всегда завершается успешно, но последующий за ним невинный код наподобие `buf[100] = 'c';` может привести к генерации исключения или аварийному останову. С точки зрения стандарта C++ оба действия некорректны, поскольку, с одной стороны, стандарт C++ требует, чтобы в случае, когда оператор new не может реально выделить запрошенную память, он генерировал исключение (этого не происходит), и чтобы код наподобие `buf[100] = 'c'` не приводил к генерации исключений или другим сбоям (что может произойти).

Почему некоторые операционные системы выполняют такое отложенное выделение памяти? В основе такой схемы лежит прагматичный довод: процессу, который запросил данную память, сразу в полном объеме она может не понадобиться, более того — процесс может никогда не использовать ее полностью или не использовать ее всю одновременно — а тем временем ею мог бы воспользоваться другой процесс, которому эта память нужна ненадолго. Зачем выделять процессу всю память немедленно при запросе, если реально она ему может и не понадобиться? Поэтому описанная схема имеет ряд преимуществ.

Основная проблема при таком подходе связана со сложностью обеспечения соответствия требованиям стандарта, что в свою очередь делает разработку корректной программы сложной задачей: ведь теперь *любое* обращение к успешно выделенной динамической памяти может привести к аварийному останову программы. Это явно нехорошо. Если выделение памяти завершилось неудачей, программа знает, что памяти для завершения операции недостаточно, и может выполнить какие-то соответствующие ситуации действия — уменьшить запрашиваемый блок памяти, использовать менее критичный к памяти, но более медленный алгоритм, да, наконец, просто корректно завершиться со сверткой стека. Но если у программы нет способа узнать, доступна ли в действительности выделенная память, то любая попытка ее чтения или записи может привести к аварийному останову программы, причем предсказать его невозможно, поскольку такая неприятность может произойти как при первом обращении к некоторой части буфера, так и после миллионов успешных обращений к другим частям буфера.

На первый взгляд кажется, что единственный способ избежать этого заключается в немедленной записи (или чтении) всего блока памяти, чтобы убедиться в его существовании, например:

```
// Пример 23-1: инициализация вручную с обращением к каждому
//              байту выделенной памяти.
//
char* p = new char[1000000000];
memset( p, 0, 1000000000 );
```

Если память выделяется для типа, который является типом класса, а не обычным старым типом (POD[39]), то обращение к памяти выполняется автоматически.

---

[39] POD означает "plain old data" — "простые старые данные". Неформально POD означает любой тип, представляющий собой набор простых данных, возможно, с пользовательскими функциями-членами для удобства. Говоря более строго, POD представляет собой класс или объединение, у которого нет пользовательского конструктора, копирующего присваивания, и деструктора, а также нет (нестатических) членов-данных, являющихся ссылками, указателями на члены или не являющимися POD.

```
// Пример 23-2: Инициализация по умолчанию: если T – не
//              POD-тип, то этот код инициализирует все
//              объекты T немедленно по выделении памяти и
//              обращается к каждому (значащему, не
//              заполняющему) байту.
//
T* p = new T[1000000000];
```

Если T — не POD-тип, то каждый объект инициализируется по умолчанию, что означает, что записываются все значащие байты каждого объекта, т.е. выполняется обращение ко всей выделенной памяти[40].

Вы можете решить, что это полезно, но это не так. Да, если вызов функции memset в примере 23-1 или оператора new в примере 23-2 завершится успешно, значит, память действительно выделена и фиксирована. Но если произойдет описанный ранее сбой при обращении к памяти, то мы не получим ни нулевого указателя, ни исключения bad_alloc — нет, произойдет все та же ошибка доступа и программа аварийно завершится, и с этим ничего не поделаешь (если только нет возможности перехватить и обработать этот сбой некоторыми платформо-зависимыми способами). Этот способ ничуть не лучше и не безопаснее выделения памяти без обращения к ней, в надежде, что память окажется "на месте" в тот момент, когда она будет нам нужна.

Это возвращает нас к вопросу о соответствии стандарту, которого все же могли бы достичь разработчики компиляторов, например, они могли бы использовать знания об операционной системе для перехвата ошибок доступа к памяти и тем самым предотвратить аварийное завершение программы. Т.е. они могли бы разработать оператор new так, как мы только что рассматривали, — выделяющим память и выполняющим запись каждого ее байта (или, по крайней мере, каждой страницы) с использованием перехвата обращения к несуществующей памяти средствами операционной системы и преобразования его в стандартное исключение bad_alloc (или нулевой указатель в случае не генерирующего исключений оператора new). Тем не менее, я сомневаюсь, чтобы разработчики компиляторов пошли на это, по двум причинам: во-первых, это существенно снижает производительность, и, во-вторых, ошибка оператора new — слишком большая редкость в реальной жизни. И это подводит нас к следующему пункту.

На практике ошибки выделения памяти встречаются очень редко. Действительно, многие современные серверные программы крайне редко встречаются с исчерпанием памяти.

В системе виртуальной памяти каждая программа работает в своем собственном адресном пространстве. Это приводит к активному использованию страничной организации памяти и, при больших запросах к памяти, к активному использованию диска для подкачки памяти. В результате задолго до того, как память будет исчерпана, активная работа с диском приведет к существенному падению производительности системы в целом, и процессы так и не дождутся отказа оператора new просто потому, что вследствие падения производительности из-за постоянного свопинга в дело придется вмешаться системному администратору, так что программы не умрут непосредственно из-за нехватки памяти, а будут убиты его рукой.

Конечно, вполне возможно создать программу, которая будет требовать все больше и больше памяти, при этом не используя большую ее часть. Это возможно, хотя и необычно (во всяком случае, для меня). Сказанное также не относится к системам без виртуальной памяти, например, ко встроенным системам или системам, работающим в реальном времени. Некоторые из них настолько критичны ко всякого рода сбоям, что не используют динамическую память в принципе, не говоря уж о виртуальной памяти.

Когда вы обнаруживаете ошибку оператора new, вы можете сделать не так уж много. Как указывал Эндрю Кёниг в своей статье "Когда памяти не хватает" ("When Memory Runs Low" [Koenig96]), поведение по умолчанию при ошибке оператора new,

[40] Мы не рассматриваем случай патологического типа T, конструктор которого не инициализирует данные объекта.

состоящее в завершении работы программы (обычно, как минимум, с попыткой свертки стека) представляет собой наилучший выбор в большинстве ситуаций, в особенности в процессе тестирования.

Конечно, когда оператор new выполняется неуспешно, иногда можно сделать и другие вещи. Если вы хотите вывести диагностическую информацию, то подключаемая функция-обработчик ошибок оператора new — вполне подходящее для этого место. Иногда можно воспользоваться стратегией с использованием резервного "неприкосновенного" буфера памяти для чрезвычайных ситуаций. Однако любой, кто захочет воспользоваться одним из таких способов, должен четко понимать, что именно он делает, и тщательно тестировать обработку ошибки на целевой платформе, поскольку зачастую на самом деле все работает не совсем так, как вы себе это представляете. И наконец, если память действительно исчерпана, вам может не удаться сгенерировать нетривиальное (т.е. невстроенное) исключение. Даже такая тривиальная инструкция как `throw string("failed");` скорее всего, будет пытаться выделить динамическую память с использованием оператора new (в зависимости от степени оптимизации вашей реализации класса string).

Да, бывают ситуации, когда можно сделать что-то полезное в ответ на неуспешное завершение оператора new, но, как правило, не стоит делать что-то большее, чем свертка стека или использование обработчика ошибки, выполняющего журнальную запись о ней.

Что надо проверять

Бывают отдельные случаи, когда проверка исчерпания памяти и попытка восстановления после него имеют смысл. Некоторые из них перечислены в статье [Koenig96]. Например, можно выполнить выделение всей памяти и ее инициализацию в начале программы, а после самостоятельно ее распределять. В таком случае, если ваша программа аварийно завершится из-за нехватки памяти (обращения к некорректному блоку), то, по крайней мере, это произойдет сразу же, т.е. до того, как программа приступит к реальной работе. Такой подход требует приложения дополнительных усилий и годится лишь в ситуации, когда вы заранее знаете требуемое программе количество памяти.

Основной тип восстанавливаемой ошибки оператора new, который я видел в промышленных системах — это создание буферов, размер которых поступает в программу извне, с некоторого устройства ввода. Рассмотрим, например, коммуникационное приложение, в котором каждый передаваемый пакет предваряется значением длины пакета, и первое, что должен сделать получатель — это прочесть длину получаемого пакета и выделить для него буфер достаточного размера. В этой ситуации я видел попытки выделения "монстрообразных" блоков памяти, в первую очередь из-за искажения потока передаваемых данных (или ошибки в программе обработки). В этом случае приложение *должно* проверять наличие искажения данных (а еще лучше использовать протокол, позволяющий избежать такого рода искажений данных) и отбрасывать неверные данные или заведомо некорректные размеры буфера, поскольку при такой стратегии программа остается в состоянии продолжать выполнение разумных действий, в частности, использовать повторную передачу информации с меньшим размером пакета или даже просто отбросив пакет с некорректным размером и продолжать обработку других пакетов — вместо того, чтобы просто "рухнуть" под тяжестью запроса блока памяти непомерного размера.

В конце концов, смешно говорить об "исчерпании памяти" при попытке выделения блока размером 2 Гбайт, в то время, как в системе остается доступным 1 Гбайт памяти![41]

Еще один случай, когда восстановление после сбоя оператора new имеет смысл, — это когда ваша программа оптимистично пытается выделить огромный рабочий буфер, но при невозможности снижает свои требования до тех пор, пока не сможет получить требуемое. В этом случае программа должна быть создана таким образом, чтобы уметь подстраиваться под имеющийся буфер памяти, а при необходимости и работать с несколькими не последовательными блоками памяти.

Резюме

Избегайте использования не генерирующего исключения оператора new, поскольку он не дает никаких дополнительных преимуществ, зато обычно приводит к худшему поведению программы при ошибке выделения памяти, чем обычный оператор new, генерирующий исключения.

Помните, что проверка неуспешного выполнения оператора new зачастую бесполезна по ряду причин.

Если же вы действительно беспокоитесь о возможном исчерпании памяти, то убедитесь, что вы проверяете именно то, что намерены, поскольку:

- проверка отказов оператора new обычно бесполезна в операционной системе, в которой реальное предоставление памяти процессу не происходит до тех пор, пока память не начинает реально использоваться;
- в системе виртуальной памяти задолго до исчерпания памяти резко снижается производительность программ, что приводит к вмешательству со стороны системного администратора;
- за исключением некоторых специальных случаев, даже обнаружив отказ оператора new, вы можете мало что сделать для обработки этой ситуации — если памяти в системе действительно не осталось.

[41] Интересно, что выделение буфера памяти, размер которого определяется извне, — классический пример уязвимости системы защиты. Атака злонамеренными пользователями или программами, пытающимися вызвать проблемы с буфером динамической памяти, — классика жанра, которая до сих пор остается любимым средством для взлома (или слома) систем. Заметим, что крах программы, вызванный отказом в выделении блока памяти требуемого размера, — разновидность атаки DOS (Denial-Of-Service — отказ в обслуживании), но не взлома системы в целях получения несанкционированного доступа к ней. Близкая к этому, но отличная методика, связанная с переполнением буфера, также остается излюбленным методом у хакеров, и остается только удивляться, как много программистов все еще продолжают использовать функции типа `strcpy` и другие, которые не проверяют размеры буфера и оставляют широкий простор для взломщиков.

Оптимизация и эффективность

Зачастую нам требуется сделать более эффективной ту или иную часть нашей программы, и имеется масса вариантов оптимизаций, которые могут помочь нам в этом.

Время от времени появляются предположения, что использование ключевого слова `const` помогает компилятору при выполнении оптимизации кода. Так ли это на самом деле? Почему? Помимо `const`, множество программистов для повышения эффективности кода часто используют другое ключевое слово — `inline`. Влияет ли это слово на производительность программ? Если да, то когда и каким образом? Когда может быть выполнено встраивание кода функции, как под контролем программиста, так и без него?

И наконец, мы завершим этот раздел примером того, как знания предметной области помогают при разработке приложения с высокой производительностью, которой невозможно достичь никакими низкоуровневыми оптимизациями, играми с битами и другими подобными способами улучшения кода.

Задача 24. Константная оптимизация Сложность: 3

Помогает ли корректное применение ключевого слова `const` компилятору при оптимизации кода? Обычная реакция программиста на этот вопрос: "Да, конечно!" Не будем спешить...

Вопрос для новичка

1. Рассмотрим следующий исходный текст.

```
const Y& f( const X& x ) {
    // ... некоторые действия с x и поиск объекта Y ...
    return someY;
}
```

Помогает ли объявление параметра и/или возвращаемого значения с использованием `const` компилятору сгенерировать более оптимальный код или улучшить его каким-то иным образом? Обоснуйте ваш ответ.

Вопрос для профессионала

2. Поясните, может ли в общем случае наличие или отсутствие ключевого слова `const` помочь компилятору улучшить генерируемый им код и почему.

3. Рассмотрим следующий исходный текст.

```
void f( const Z z ) {
    // ...
}
```

Ответьте на следующие вопросы.

а) При каких условиях и для каких видов классов `Z` это конкретное указание `const` может помочь сгенерировать другой, лучший код?

б) Говорим ли мы об оптимизации компилятором или о некотором другом типе оптимизации при условиях, рассмотренных в пункте *а*)? Поясните свой ответ.

в) В чем состоит лучший путь достижения того же эффекта?

Решение

const: ненавязчивый сервис

1. **Рассмотрим следующий исходный текст.**

```
// Пример 24-1
const Y& f( const X& x ) {
    // ... некоторые действия с  x и поиск объекта  Y ...
    return someY;
}
```

Помогает ли объявление параметра и/или возвращаемого значения с использованием `const` компилятору сгенерировать более оптимальный код или улучшить его каким-то иным образом?

Коротко говоря — нет, вряд ли.

Обоснуйте ваш ответ.

Что же именно компилятор может улучшить? Может ли он избежать копирования аргумента или возвращаемого значения? Нет, поскольку аргумент уже передается по

ссылке, и возвращаемое значение также уже является ссылкой. Может ли он разместить копию x или someY в памяти, доступной только для чтения? Нет, поскольку и x, и someY располагаются вне пределов функции и приходят в нее из "внешнего мира" и туда же возвращаются. Даже если someY динамически выделяется "на лету" в самой функции f, и сам объект, и владение им передается за пределы функции вызывающему коду.

А что можно сказать о коде внутри функции f? Может ли компилятор как-то улучшить генерируемый для тела функции f код на основе указаний const? Это приводит нас ко второму, более общему вопросу.

2. Поясните, может ли в общем случае наличие или отсутствие ключевого слова const помочь компилятору улучшить генерируемый им код и почему.

Обращаясь вновь к примеру 24-1, становится очевидно, что здесь остается актуальной та же причина, по которой константность параметров не может привести к улучшению кода. Даже если вы вызываете константную функцию-член, компилятор не может полагаться на то, что не будут изменены биты объектов x или someY. Кроме того, имеются дополнительные проблемы (если только компилятор не выполняет глобальную оптимизацию). Компилятор может не знать, нет ли другого кода, в котором имеется неконстантная ссылка, являющаяся псевдонимом объектов x и/или someY, и не могут ли они быть случайно изменены через эту ссылку в процессе работы функции f. Компилятор может также не знать, объявлены ли реальные объекты, для которых x и someY являются простыми ссылками, как константные.

Только того факта, что x и someY объявлены как const, не достаточно, чтобы их биты были физически константны. Почему? Поскольку любой класс может иметь члены, объявленные как mutable, или внутри функций-членов класса может использоваться приведение const_cast. Даже код внутри самой функции f может выполнить приведение const_cast либо преобразование типов в стиле C, что сведет на нет все объявления const.

Есть один случай, когда ключевое слово const может действительно что-то значить, — это происходит тогда, когда объекты сделаны константными в точке их описания. В этом случае компилятор часто может поместить такие "действительно константные" объекты в память "только для чтения", в особенности если это объекты POD[42], для которых их образ в памяти может быть создан в процессе компиляции и размещен в выполнимой программе. Такие объекты пригодны для размещения в ПЗУ.

Как const может оптимизировать

3. Рассмотрим следующий исходный текст.

```
// Пример 24-2
void f( const Z z ) {
    // ...
}
```

Ответьте на следующие вопросы.

а) **При каких условиях и для каких видов классов Z это конкретное указание const может помочь сгенерировать другой, лучший код?**

Если компилятор знает, что z — действительно константный объект, он в состоянии выполнить некоторую оптимизацию даже без глобального анализа. Например, если тело f содержит вызов наподобие g(&z), то компилятор может быть уверен, что все части z, не являющиеся mutable, не изменятся в процессе вызова g.

[42] См. пояснения о том, что такое POD, в предыдущем разделе, на стр. 159. — *Прим. перев.*

Однако кроме этого случая, запись const в примере 24-2 не является оптимизацией для большинства классов Z, а там, где это все же приводит к оптимизации, — это уже не оптимизация генерации объектного кода компилятором.

С точки зрения генерации кода компилятором вопрос в основном сводится к тому, не может ли компилятор опустить создание копии или разместить z в памяти только для чтения. То есть было бы неплохо, если бы мы знали, что z действительно не изменяется в процессе работы функции f, поскольку теоретически это означает, что мы могли бы использовать непосредственно внешний объект, передаваемый функции в качестве аргумента, без создания его копии, или, если мы все же делаем его копию, то ее можно было бы разместить в памяти только для чтения, если это дает выигрыш в производительности или желательно по каким-либо иным соображениям.

В общем случае компилятор не может использовать константность параметров для устранения создания копии аргумента или предполагать побитовую константность. Как уже упоминалось, имеется слишком много ситуаций, когда данные предположения оказываются неверными. В частности, Z может иметь члены, описанные как mutable, или где-то (в самой функции f, в некоторой другой функции или в непосредственном или косвенном вызове функции-члена Z) может быть выполнено приведение типов const_cast или использованы какие-то другие трюки.

Имеется один случай, когда компилятор способен сгенерировать улучшенный код, если:

- определения копирующего конструктора Z и всех функций Z, прямо или косвенно использующихся в теле функции f, видимы в данной точке;
- эти функции достаточно просты и не имеют побочного действия;
- компилятор имеет агрессивный оптимизатор.

В этом случае компилятор может быть уверен в корректности своих действий и может не создавать копию объекта в соответствии с правилом, которое гласит, что компилятор может выполнять любые оптимизации, при условии, что соответствующая стандарту программа дает те же результаты, что и без них.

В качестве отступления стоит упомянуть об одной детали. Некоторые программисты утверждают, что есть еще один случай, когда компилятор может генерировать лучший код на основании const, а именно — при выполнении глобальной оптимизации. Дело в том, что это утверждение остается верным и в том случае, если убрать из него упоминание const. Не важно, что глобальная оптимизация все еще весьма редка и дорога; настоящая проблема заключается в том, что глобальная оптимизация использует всю имеющуюся информацию об объекте, в том числе о его реальном использовании, так что такая оптимизация работает одинаково независимо от того, объявлен ли объект как const или нет — решение принимается на основании того, что в действительности происходит с объектом, а не того, что обещает программист. Так что и в этом случае применение ключевого слова const ничего не дает в плане оптимизации генерируемого кода.

Заметим, что несколькими абзацами ранее я сказал, что "запись const в примере 24-2 не является оптимизацией для большинства классов Z" и для "генерации объектного кода компилятором". В оптимизаторе компилятора имеется гораздо больше возможностей, чем кажется, и в некоторых случаях const может быть полезен для некоторой реальной оптимизации.

б) **Говорим ли мы об оптимизации компилятором или о некотором другом типе оптимизации при условиях, рассмотренных в пункте *а*)? Поясните свой ответ.**

Например, автор Z может выполнять некоторые действия с константным объектом по-другому, написав более эффективную версию функции для случая с константным объектом.

Пусть в примере 24-2 Z — класс "handle/body", такой как класс String с использованием подсчета ссылок для выполнения отложенного копирования.

```
// Пример 24-3
//
void f( const String s ) {
    // ...
    s[4];    // или использование итераторов
    // ...
}
```

Для данного класса String известно, что вызов оператора operator[] для константного объекта String не должен приводить к изменению содержимого строки, так что можно предоставить константную перегрузку оператора operator[], которая возвращает значение типа char вместо ссылки char&:

```
class String {
    // ...
public:
    const char operator[]( size_t ) const;
    char&      operator[]( size_t );
    // ...
}
```

Аналогично, класс String может предоставить вариант итератора const_iterator, оператор operator* для которого возвращает значение типа char, а не char&.

Если выполнить описанные действия и после этого воспользоваться оператором operator[] или итераторами, а переданный по значению аргумент объявлен как const, то тогда — чудо из чудес! — класс String без какой-либо дополнительной помощи, автомагически :) оптимизирует ваш код, устранив глубокое копирование...

в) **В чем состоит лучший путь достижения того же эффекта?**

...но того же эффекта можно достичь, просто передавая аргумент по ссылке.

```
// Пример 24-4: более простой способ
//
void f( const Z& z ) {
  // ...
}
```

Этот способ работает независимо от того, использует ли объект идиому handle/body или подсчет ссылок или нет, так что просто воспользуйтесь им!

➤ Рекомендация

Избегайте передачи параметров как константных значений. Предпочтительно передавать их как ссылки на константные объекты, кроме тех случаев, когда это простые объекты наподобие int, с мизерными затратами на копирование.

Резюме

Вера в то, что ключевое слово const помогает компилятору генерировать более качественный код, очень распространена. Да, const действительно хорошая вещь, но основная цель данной задачи — показать, что предназначено это ключевое слово в первую очередь для человека, а не для компиляторов или оптимизаторов..

Когда речь идет о написании безопасного кода, const — отличный инструмент, который позволяет программистам писать более безопасный код с дополнительными проверками компилятором. Но когда речь идет об оптимизации, то const остается в принципе полезным инструментом, поскольку позволяет проектировщикам классов лучше выполнять оптимизацию вручную; но генерировать лучший код компиляторам оно помогает в гораздо меньшей степени.

Задача 25. inline Сложность: 7

Быстро ответьте: когда выполняется встраивание? И можно ли написать функцию, которая гарантированно никогда не окажется встраиваемой? В этой задаче мы рассмотрим много различных случаев встраивания, причем некоторые из них вас удивят.

Вопрос для новичка

1. Что такое встраивание (inlining)?

Вопрос для профессионала

2. Когда выполняется встраивание? Может ли оно выполняться:
   а) во время написания исходного текста?
   б) во время компиляции?
   в) во время компоновки?
   г) при инсталляции приложения?
   д) в процессе работы?
   е) в некоторое другое время?
3. Дополнительный вопрос: какого рода функции гарантированно не будут встраиваемыми?

Решение

Какой ответ выберете вы на второй вопрос? Если вы выберете ответы *а*) или *б*), то вы не одиноки. Это наиболее распространенные ответы на данный вопрос, и в книге [Sutter02] я уже вкратце обращался к этой теме. Но если бы тема этим и исчерпывалась, то мы могли бы на этом завершить рассмотрение задачи и отправиться на пикник. Но на этот раз пикника не будет. У настоящего мужчины всегда найдется что сказать. Причем сказать можно достаточно много.

Причина появления данной задачи в книге — стремление показать, почему наиболее точный ответ на главный вопрос задачи — любой из перечисленных, а на последний — "никакие". Непонятно, почему? Тогда читайте дальше.

Краткий обзор

1. Что такое встраивание (inlining)?

Если вы читали [Sutter02][43], то вы можете пропустить этот и несколько следующих подразделов и перейти сразу к разделу "Ответ В: во время компоновки" на стр. 171.

Коротко говоря, встраиваемость означает замену вызова функции подстановкой копии ее тела. Рассмотрим, например, следующий исходный текст.

```
//  Пример 25-1
//
double Square( double x ) { return x * x; }
int main() {
    double d = Square( 3.14159 * 2.71828 );
}
```

Идея встраиваемости вызова функции (как минимум, концептуально) заключается в том, что программа преобразуется так, как если бы она была написана следующим образом.

[43] Задача 7.1 в русском издании книги. — *Прим. ред.*

```
int main() {
    const double __temp = 3.14159 * 2.71828;
    double d = __temp * __temp;
}
```

Такое встраивание устраняет излишние расходы на выполнение вызова функции, а именно — на внесение параметров в стек, переход процессором в другое место памяти для выполнения кода функции, что, помимо затрат на переход, может привести к полному или частичному сбросу кэша инструкций процессора. Встраивание — это далеко не то же, что и трактовка `Square` как макроса, поскольку вызов встраиваемой функции остается вызовом функции, и ее аргументы вычисляются только один раз. В случае же макросов вычисление аргументов может выполняться неоднократно; так, макрос `#define SquareMacro(x) ((x)*(x))` при вызове `SquareMacro(3.14159*2.71828)` будет раскрыт до `(3.14159*2.71828)*(3.14159*2.71828)` (т.е. умножение π на e будет выполнено не один раз, а два).

Заслуживает внимания еще один частный случай — рекурсивный вызов, когда функция вызывается из самой себя, непосредственно или опосредованно. Хотя такие вызовы зачастую не могут быть встраиваемыми, в некоторых случаях компилятор может сделать рекурсию встраиваемой так же, как может частично разворачивать некоторые циклы.

Между прочим, вы заметили, что в примере 25-1, который иллюстрирует встраиваемость, не использовано ключевое слово `inline`? Это сделано преднамеренно. Мы вернемся к этому моменту еще не раз в процессе рассмотрения основного вопроса задачи.

2. **Когда выполняется встраивание? Может ли оно выполняться:**
   а) **во время написания исходного текста?**
   б) **во время компиляции?**
   в) **во время компоновки?**
   г) **при инсталляции приложения?**
   д) **в процессе работы?**
   е) **в некоторое другое время?**
3. **Дополнительный вопрос: какого рода функции гарантированно не будут встраиваемыми?**

Ответ А: во время написания исходного текста

В процессе написания исходного текста разработчики могут использовать ключевое слово `inline` в своих программах. Это не является реальным выполнением встраивания в смысле перемещения кода для устранения вызова функции, но это попытка выбрать и явно выделить подходящие места для встраивания функции, так что мы будем рассматривать ее как наиболее раннюю возможность принятия решения о встраивании[44].

Когда вы намереваетесь написать ключевое слово `inline` в вашем исходном тексте, вы не должны забывать о трех важных вещах.

- *По умолчанию не делайте этого.* Преждевременная оптимизация — зло, и вы не должны использовать `inline` до тех пор, пока профилирование не покажет необходимость этого в определенных случаях. Более полную информацию по этому вопросу можно найти в [Sutter02] или запросив на поисковом сервере типа Google что-то вроде "преждевременная оптимизация" или "premature

[44] Еще одна интерпретация "во время написания исходного текста" может заключаться в буквальном встраивании функций некоторыми разработчиками путем физического перемещения блоков исходного текста. Это действие еще больше отдаляет нас от обычного понимания термина "встраивание", так что я не буду его здесь рассматривать.

site:www.gotw.ca" — вы получите массу страшных предупреждений о преждевременной оптимизации вообще и встраивании в частности.

- *Это означает всего лишь "попробуйте, пожалуйста".* Как описано в [Sutter02], ключевое слово `inline` — всего лишь подсказка для компилятора, возможность попытаться мило поговорить с ним, предоставляемая языком программирования (далее будут описаны недостатки таких "милых" разговоров). Ключевое слово `inline` вообще не имеет никакого семантического действия в программе на C++. Оно не влияет на другие конструкции языка, на использование функции, объявленной `inline` (например, вы можете получить адрес такой функции), и нет никакой стандартной возможности программно определить, объявлена ли данная функция как встраиваемая или нет.
- *Это часто делается не на требуемом уровне детализации.* Мы пишем ключевое слово `inline` для *функции*, но когда выполняется встраивание, то в действительности оно происходит при *вызове* функции. Это отличие очень важно, поскольку одна и та же функция может (и зачастую должна) быть встраиваемой в одном месте вызова, но не в некотором другом. Ключевое слово `inline` не дает вам никакой возможности выразить этот факт, поскольку мы можем указать, что встраиваемой является функция сама по себе, что эквивалентно неявному указанию делать эту функцию встраиваемой везде, во всех возможных местах вызова. Такое предвидение редко бывает точным. Так что хотя в разговоре мы часто говорим о "встраиваемой функции", более точно было бы говорить о встраиваемом вызове функции.

➢ Рекомендация

Избегайте использования ключевого слова `inline` или других попыток оптимизации до тех пор, пока на их необходимость не укажут измерения производительности программы.

Ответ Б: во время компиляции

Обычно во время работы компиляторы сами, без внешней подсказки, выполняют описанный в примере 25-1 вид встраивания.

Что делает компилятор, когда мы мило разговариваем с ним путем объявления некоторых функций встраиваемыми? Это зависит от ситуации. Не все компиляторы хорошо поддерживают такие "милые" разговоры, даже если задаривать их шоколадом и цветами. Ваш компилятор запросто может проигнорировать вашу просьбу, и даже не одним, а тремя способами.

- Не делая встроенными вызовы функций, которые вы объявили как `inline`.
- Делая встроенными вызовы функций, которые вы *не* объявляли встраиваемыми.
- Встраивая некоторые из вызовов, оставляя другие вызовы той же функции обычными невстраиваемыми (независимо от того, объявлена ли функция как `inline`).

Вернемся к примеру 25-1 и обратим внимание на то, что в нем нигде не говорится, что функция должна быть встраиваемой. Это сделано преднамеренно, потому что этим я хотел проиллюстрировать тот факт, что встраивание все равно может произойти, даже без объявления функции `inline`. Не удивляйтесь, если ваш сегодняшний компилятор поступит именно так. Поскольку вы не можете написать соответствующую стандарту программу, которая выявит отличия при встраивании функции компилятором, этот способ оптимизации оказывается совершенно законным, и компилятор может (а часто и должен) выполнять его за вас.

Обычно современные компиляторы в состоянии лучше программиста решить, какие вызовы функций следует сделать встраиваемыми, а какие — нет, включая ситуации, когда одна и та же функция в разных местах может быть обработана по-разному. Почему? Простейшая причина в том, что компилятор знает больше о контексте вызова, поскольку ему известна "реальная" структура точки вызова — машинный код, сгенерированный для данной точки после применения других оптимизаций, таких как разворачивание циклов или удаление недостижимых ветвей программы. Например, компилятор может быть способен определить, что встраивание функции в некотором внутреннем цикле может сделать цикл слишком большим для размещения в кэше процессора, что приведет только к падению производительности, так что такой вызов не будет сделан встроенным, в то время как другие вызовы той же функции в других местах программы могут остаться встроенными.

Ответ B: во время компоновки

Теперь мы переходим к более интересным и более современным аспектам встраивания.

Вопрос: может ли функция быть встроена во время компиляции? Ответ: да. Этот ответ является основой дополнительного вопроса о функциях, которые не могут быть встраиваемыми ни при каких условиях. Этот вопрос добавлен мною в задачу, потому что обычно все верят, что такие виды функций существуют. В частности, обычно считается, что невозможно встроить функции, описания которых размещено в отдельном модуле, а не в заголовочном файле.

Давайте попытаемся выполнять встраивание максимально интенсивно, насколько это возможно. Рассмотрим пример 25-1 с небольшим изменением.

```
// Пример 25-2: затрудним работу оптимизатора, поместив
//              функцию в отдельный модуль, и сделав
//              недоступным ее исходное описание.
//

//--- Файл main.cpp ---
//
double Square( double x );
int main() {
    double d = Square( 3.14159 * 2.71828 );
}

//--- Файл square.obj (или .o) ---
//
// Содержит скомпилированное описание функции
// double Square( double x ) { return x * x; }
```

Здесь идея заключается в том, что реализация функции `Square` вынесена из единицы трансляции `main.cpp`. На самом деле сделано даже больше: недоступны даже исходные тексты функции `Square` — только ее объектный код. "Теперь вызов `Square` гарантированно не будет встраиваемым!" — скажет множество людей. Пока идет компиляция — они правы. В процессе компиляции `main.cpp` никакой самый мощный и продвинутый компилятор не в состоянии получить доступ к исходным текстам определения функции `Square`.

На вот продвинутый компоновщик с такой задачей справиться в состоянии, и некоторые популярные реализации так и поступают. Ряд компиляторов, в частности, Hewlett-Packard, поддерживают такое *межмодульное встраивание* (cross-module inlining); в Microsoft Visual C++ 7.0 (известном как ".NET") и более поздних версий имеется опция `/LTCG`, которая означает "link time code generation" (генерация кода в процессе компиляции). Реальное преимущество такой технологии "позднего встраивания" заключается в знании реального контекста каждой точки вызова, что

позволяет более интеллектуально подойти к вопросу о том, где и когда стоит выполнить встраивание.

Вот еще пища для размышлений: вы заметили где-нибудь в описании примера 25-2, что функция `Square` должна обязательно быть написана на C++? В этом и заключается второе преимущество технологии "позднего встраивания", которая нейтральна по отношению к языку. Функция `Square` может быть написана на Fortran или Pascal. Приятно. Все, о чем должен позаботиться компоновщик, — выяснить используемые функцией соглашения о передаче параметров и удалить код размещения аргументов в стеке и снятия с него, вместе с машинной командой вызова функции.

Но это далеко не все — настоящему мужчине всегда есть что сказать! Ведь мы только приступаем к серьезному разговору, так что не спешите...

Ответ Г: при инсталляции приложения

Теперь перемотаем пленку нашего разговора прямо к тому радостному моменту, когда мы наконец-то скомпилировали и скомпоновали наше приложение, собрали его в `tar`-файл, какой-нибудь `setup.exe` или `.wpi`, и гордо отправили упакованный компакт своему первому покупателю. Наступило самое время для того, чтобы:

а) сорвать наклейку с предупреждением о лицензии;

б) оплатить почтовые расходы;

в) объявить, что уж теперь-то все встраивание далеко позади.

Да?

Да, да, нет.

С середины 1990-х годов постоянно увеличивается количество продаж приложений, предназначенных для работы под управлением специализированных виртуальных машин. Это означает, что вместо того, чтобы скомпилировать программу в машинный код для конкретного процессора, операционной системы и API, приложение компилируется в поток байт-кода, который интерпретируется или компилируется на машине пользователя средой времени выполнения, что позволяет абстрагироваться от конкретных возможностей процессора или операционной системы. Наиболее распространенные примеры включают (но не ограничиваются) Java Virtual Machine (JVM) и .NET Common Language Run-time (CLR)[45]. В случае использования таких целевых сред компилятор транслирует исходный текст C++ в упомянутый поток байт-кода (известный также как язык команд виртуальной машины (virtual machine's instruction language, IL)), который представляет собой программу, созданную с использованием кодов операций из системы команд среды времени выполнения.

Отклоняясь от основной темы — некоторые из этих сред имеют очень богатые средства поддержки конструкций объектно-ориентированных языков на уровне системы команд, так что классы, наследование и виртуальные функции поддерживаются ими непосредственно. Компилятор для такой платформы может (и многие так и поступают) транслировать исходную программу в промежуточный язык команд виртуальной машины последовательно, класс за классом, функция за функцией, возможно, после выполнения некоторой собственной оптимизации, включающей возможность встраивания еще на уровне компиляции. Если компилятор работает таким образом, то исходный текст функции на C++ может быть более или менее полно восстановлен по коду функции с той же сигнатурой, представленной в целевой системе команд. Конечно, компилятор не обязан следовать описанной методике, но даже если он поступает как-то иначе, следующие далее замечания о встраивании остаются в силе.

---

[45] А также такие родственники CLR, как Mono, DotGNU и Rotor, которые реализуют также стандарт ISO Common Language Infrastructure (CLI), определяющий подмножество CLR.

Вернемся к нашему вопросу. Какое все это имеет отношение к встраиванию в процессе инсталляции приложения? Даже при использовании описанных сред времени выполнения в конечном счете процессор работает со своей собственной системой команд. Следовательно, среда отвечает за трансляцию языка команд виртуальной машины в код, который в состоянии понять процессор целевого компьютера. Очень часто это делается при первой инсталляции приложения, и именно в этот момент, как и в других процессах компиляции, могут быть выполнены (и часто выполняются) дополнительные оптимизации. В частности, компилятор .NET — NGEN IL — может выполнять встраивание в процессе инсталляции приложения, когда программа на языке IL транслируется в команды процессора, готовые к выполнению.

Здесь также стоит обратить внимание на нейтральность среды по отношению к языку программирования, так что оптимизация (включая встраивание), выполняемая в процессе инсталляции приложения, легко преодолевает границы применения отдельных языков программирования. Вас не должно удивлять, что если ваша программа на C# делает вызов небольшой функции на C++, то эта функция может в конечном итоге оказаться встроенной.

Так когда же выполнять встраивание слишком поздно? Никогда не говори никогда, потому что наш разговор еще не окончен...

Ответ Д: в процессе работы

Ну хорошо, но когда мы запускаем программу на выполнение — то уж здесь-то все возможности для встраивания должны быть позади?

Это может показаться совершенно невозможным, но встраивание вполне реально и в процессе выполнения программы, причем несколькими способами. В частности, я хочу упомянуть *оптимизацию, управляемую профилированием* (profile-directed optimization), и *защищенное встраивание* (guarded inlining). Так же, как и в случае среды, компилирующей программу при инсталляции, для этого нам потребуется наличие соответствующей поддержки времени выполнения на машине пользователя.

Идея, лежащая в основе оптимизации, управляемой профилированием, заключается в том, что когда приложение выполняется, вставленные в программу обработчики могут собирать информацию о том, как реально используется программа, в частности, какие функции вызываются чаще других и при каких условиях (например, размер рабочего множества по сравнению с общим размером кэша в момент вызова функции). Собранные данные могут быть использованы для модификации выполнимого образа программы, так что избранные вызовы функций могут стать встраиваемыми при настройке приложения на работу в целевой среде, основанной на измерениях производительности в процессе реального выполнения программы.

Защищенное встраивание представляет собой другой пример того, насколько агрессивной может оказаться оптимизация встраивания во время выполнения программы. В частности, в [Arnold00] и [JikesRVM] документирована Jikes Research Virtual Machine (RVM), динамический оптимизирующий компилятор née the Jalapeño для JVM. Помимо прочего, этот компилятор способен встраивать вызовы виртуальных функций, полагая, что получатель виртуального вызова будет иметь данный объявленный тип (чтобы избежать не только затрат на вызов функции, но и дополнительных затрат на диспетчеризацию виртуальности). На сегодняшний день компиляторы в состоянии сделать определенные вызовы виртуальных функций невиртуальными (и, таким образом, имеют возможность встраивать их), если целевой тип оказывается статически известен. Новое в среде Jikes/Jalapeño то, что теоретически она может делать вызовы невиртуальными и встраивать их, даже если статический целевой тип *не* известен. Однако поскольку такое предположение может оказаться неверным, компилятор вставляет дополнительную защиту, которая выполняет проверку типа целевого объекта в процессе выполнения программы, и если он не соответствует ожидаемому, то программа возвращается к механизму обычного вызова виртуальной функции.

Ответ E: в некоторое другое время

Вспомним, что в ответе *г*) мы рассматривали встраивание в процессе инсталляции в некоторой среде времени выполнения, такой как JVM или .NET CLR. Конечно, проницательные читатели уже заметили, что ранее я упоминал только трансляцию из байт-кода в машинные команды в процессе инсталляции приложения, но есть и другое более распространенное время такой трансляции, а именно — JIT, где аббревиатура JIT означает компиляцию "just-in-time", т.е. при необходимости.

Идея, лежащая в основе такого подхода, заключается в том, чтобы выполнять компиляцию функций только при необходимости, перед их явным использованием. Преимущество такого подхода в снижении стоимости компиляции программы в систему команд процессора, поскольку вместо одного большого этапа компиляции вы получаете много маленьких компиляций отдельных частей кода непосредственно перед их выполнением. Соответствующий недостаток такой технологии — в замедлении первых запусков программы и снижении качества оптимизации, так как такая компиляция должна выполняться очень быстро и не может затрачивать много времени на анализ встраивания и других возможных оптимизаций. Но такой компилятор все еще в состоянии выполнять оптимизации наподобие встраивания, и многие из них так и поступают, но вообще говоря, лучшие результаты можно получить при выполнении той же работы раньше, например, в процессе инсталляции (см. ответ *г*)), когда расходы времени не так критичны, и оптимизатор может позволить себе выполнить работу более тщательно и качественно.

> ➢ **Рекомендация**
>
> Встраивание может быть выполнено в любой момент времени.

Резюме

Подобно всем оптимизациям, встраивание зачастую более эффективно, если оно выполняется инструментарием, который осведомлен о генерируемом коде и/или среде выполнения, а не программистом. Чем позже выполняется встраивание, тем более точным и соответствующим целевой среде работы программы оно оказывается.

Мы говорим о встраивании функций, но гораздо более точно говорить о встраивании вызовов функций. В конце концов, одна и та же функция может оказаться встроенной в каком-то одном месте, но не в некоторых других. И, поскольку имеется масса возможностей для встраивания даже после завершения начальной компиляции, одна и та же функция может оказаться встроенной не только в разных местах, но и при помощи разного инструментария в каждом из этих мест.

Встраивание — это нечто гораздо большее, чем одно ключевое слово `inline`.

Задача 26. Форматы данных и эффективность. Часть 1: игры в сжатие. Сложность: 4

Умеете ли вы выбирать высококомпактные форматы данных, эффективно использующие память? Насколько хорошо вы справляетесь с кодом, работающим с отдельными битами? Эта и следующая задачи дают вам вполне достаточно возможностей развить оба этих навыка в процессе рассмотрения эффективного представления шахматных партий и битового буфера для их хранения.

Дополнительные требования: предполагается, что вы знакомы с азами шахмат.

Вопрос для новичка

1. Какой из перечисленных стандартных контейнеров использует меньше памяти для хранения одного и того же количества объектов одинакового типа `T`: `deque`, `list`, `set` или `vector`? Поясните свой ответ.

Вопрос для профессионала

2. Вы создаете всемирный шахматный сервер, который хранит все когда-либо сыгранные на нем партии. Поскольку база данных может оказаться очень большой, вы хотите представить каждую партию с минимально возможными затратами памяти. Здесь мы рассматриваем только ходы игры, игнорируя всю остальную информацию, например, имена игроков и комментарии.
   Для каждого из приведенных далее размеров данных приведите формат представления партии, требующий указанное количество памяти для представления полухода (под полуходом мы подразумеваем ход одного игрока). Считается, что в одном байте — 8 битов.
   а) 128 байтов на полуход
   б) 32 байта на полуход
   в) от 4 до 8 байтов на полуход
   г) от 2 до 4 байтов на полуход
   д) 12 битов на полуход

Решение

1. **Какой из перечисленных стандартных контейнеров использует меньше памяти для хранения одного и того же количества объектов одинакового типа `T`: `deque`, `list`, `set` или `vector`? Поясните свой ответ.**

Вспомним задачи 20 и 21, в которых говорилось об использовании памяти и структурах стандартных контейнеров. Каждый тип контейнера представляет собой тот или иной компромисс между используемой памятью и производительностью.

- `vector<T>` хранит данные в виде непрерывного массива объектов `T` и, таким образом, не имеет никаких дополнительных расходов на хранение одного элемента.
- `deque<T>` может рассматриваться как `vector<T>`, внутреннее представление которого разбито на отдельные блоки. `deque<T>` хранит блоки, или "страницы" объектов. Для этого требуется по одному дополнительному указателю на страницу, что обычно приводит к дополнительным расходам памяти, составляющим доли бита на один элемент. Других дополнительных затрат памяти нет, по-

скольку у deque<T> нет никаких дополнительных указателей или другой информации для отдельных объектов T.

- list<T> представляет собой двухсвязный список узлов, которые хранят элементы T. Это означает, что для каждого элемента T в list<T> хранятся также два указателя, которые указывают на предыдущий и последующий узлы списка. При вставке нового элемента в список мы создаем два дополнительных указателя, так что дополнительные затраты контейнера list<T> составляют два указателя на один хранимый элемент.
- И наконец, set<T> (и аналогичные в данном отношении multiset<T>, map<Key,T> и multimap<Key,T>) также гранят объекты T (или pair<const Key,T>) в узлах. Обычная реализация set<T> включает три дополнительные указателя на каждый узел. Зачастую, услышав об этом, многие спрашивают: "Откуда три указателя? Разве недостаточно двух — на левый и правый дочерние узлы?" Дело в том, что требуется еще один указатель на родительский узел, иначе задача определения "следующего" узла по отношению к некоторому произвольно взятому не может быть решена достаточно эффективно. (Возможны и другие способы реализации этих контейнеров; например, можно воспользоваться структурой списков с чередующимися пропусками (alternating skip list) — но и в этом случае требуется как минимум три указателя на каждый элемент (см. [Marrie00]).)

Частью решения об эффективном представлении данных в памяти является выбор правильного (как в смысле памяти, так и в смысле производительности) контейнера, поддерживающего необходимую вам функциональность. Но это еще не все: не менее важной является задача выбора эффективного представления данных, которые будут размещаться в этих контейнерах. Именно в этом и заключается суть рассматриваемой нами задачи.

Различные способы представления данных

Цель второго вопроса задачи — продемонстрировать, что может быть множество способов представления одной и той же информации.

2. Вы создаете всемирный шахматный сервер, который хранит все когда-либо сыгранные на нем партии. Поскольку база данных может оказаться очень большой, вы хотите представить каждую партию с минимально возможными затратами памяти. Здесь мы рассматриваем только ходы игры, игнорируя всю остальную информацию, например, имена игроков и комментарии.

В оставшейся части задачи мы используем следующие стандартные термины и аббревиатуры:

K King (Король)
Q Queen (Ферзь)
R Rook (Ладья)
B Bishop (Слон)
N Knight (Конь)
P Pawn (Пешка)

Поля на шахматной доске определяются горизонталью и вертикалью, к которым они относятся. Вертикали обозначаются слева направо (с точки зрения игрока белыми фигурами) буквами от *a* до *h*, а горизонтали — цифрами от 1 (горизонталь, ближайшая к игроку белыми фигурами) до 8.

Для каждого из приведенных далее размеров данных приведите формат представления партии, требующий указанное количество памяти для представления полухода (под полуходом мы подразумеваем ход одного игрока). Считается, что в одном байте — 8 битов.

а) **128 байтов на полуход**

Одно из представлений, требующее такого количества памяти, основано на предположении, что программа знает текущее размещение фигур на доске (которое выводится из предыдущих ходов) и сохраняет новую позицию на доске целиком, используя для этого по два байта для каждой клетки доски. В этом случае мы можем принять правило, что первый байт указывает цвет фигуры — 'W' или 'B' (или '.' для указания отсутствия фигуры в клетке), а во втором байте хранится тип фигуры — 'K', 'Q', 'R', 'B', 'N', 'P' или '.'.

Используя эту схему и сохраняя доску по горизонталям от 1 до 8, и по вертикалям от *a* до *h* в пределах каждой горизонтали, возможное представление полухода может быть таким:

```
WRWNWBWQWKWBWNWR
WPWPWP..WPWPWPWP
................
......WP........
................
................
BPBPBPBPBPBPBPBP
BRBNBBBQBKBBBNBR
```

Здесь представлен полуход "1. d4", с которого я обычно начинаю партию.

б) **32 байта на полуход**

Представление *а*) явно чрезмерно расточительно, поскольку представляет собой вполне удобочитаемый человеком текст, в то время как нам достаточно, чтобы формат могла прочесть машина. В конце концов, выводить позиции для пользователя базы данных будет специальное программное обеспечение.

Мы можем снизить количество необходимой памяти до 32 байт на полуход, храня всего лишь 4 бита информации для каждой клетки: 3 бита для указания фигуры (например, представление 0 для короля, 1 для ферзя, 2 для ладьи, 3 для слона, 4 для коня, 5 для пешки и 6 — для пустой клетки требуется 3 бита, при этом одно значение остается неиспользованным), и 1 бит для цвета (значение этого бита игнорируется для пустой клетки).

При использовании такой схемы для сохранения всей доски как и ранее, по горизонталям от 1 до 8, и по вертикалям от *a* до *h* в пределах каждой горизонтали, требуется всего лишь 32 байта:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

в) **от 4 до 8 байтов на полуход**

Этой величины можно достичь, используя представление полухода в виде текста в старой шахматной записи.

Старая "описательная" запись шахматной партии идентифицирует клетки с использованием дескрипторов переменной длины, наподобие K3 или QN8 вместо двухсимвольных дескрипторов вроде e3 или b8. Для записи полухода при этом требуется как минимум 4 символа (например, P-Q4) и не более 8 символов (например, RKN1-KB1, P-KB8(Q)). Заметим, что никакие завершающие нули или другие ограничители строк не требуются, поскольку записанные таким образом ходы дешифруются однозначно.

При этой схеме запись полухода может выглядеть как

```
P-KB8(Q)
```

г) **от 2 до 4 байтов на полуход**

Это достигается путем хранения полуходов как текста в современной записи шахматных партий.

Современная "алгебраическая" запись более компактна, и любой полуход можно записать с использованием от 2 символов (например, d4) до 4 символов (например,

Rgf1, gh=Q). В этом случае также не нужны никакие разделители в силу однозначности декодирования.[46]

При использовании этой схемы полуход может выглядеть следующим образом:

```
gh=Q
```

д) **12 битов на полуход**

Еще более компактную запись можно получить, применив иной подход. Полуход однозначно определяется исходной клеткой и клеткой назначения. Поскольку всего клеток 64, для представления одной клетки достаточно 6 битов, так что всего для записи полухода требуется 12 битов. Этого достаточно для обычных ходов; однако для записи рокировки потребуется большее количество памяти.

Этот способ оказывается существенно лучше всех описанных ранее. Давайте теперь ненадолго отложим вопрос упаковки информации в сторону и приступим к следующей задаче, в которой рассмотрим, как можно создать вспомогательные структуры данных для хранения таких "объектов нестандартного размера", с которыми не так-то легко работать и которые ухитряются пересекать границы байтов.

[46] Кстати, основное достоинство такого представления вне компьютерного мира в том, что такая запись может быть легко выполнена на бумаге человеком, даже в условиях цейтнота. Оказывается, уменьшение длины записи от максимум 8 символов до максимум 4 вместе с определенной концептуальной простотой играет большую роль для пользователей (которых в шахматном мире называют игроками :)).

Задача 27. Форматы данных и эффективность. Часть 2: игры с битами Сложность: 8

Пришло время рассмотреть более компактные и эффективно использующие память структуры данных, и поработать с кодом, оперирующим на битовом уровне.

Вопрос для профессионала

1. Для реализации решения *д*) второго вопроса задачи 26 вы решили создать класс, управляющий буфером битов. Реализуйте его максимально переносимо, с тем чтобы он корректно работал на любом компиляторе, соответствующем стандарту C++, независимо от платформы.

```
class BitBuffer {
public:
    // ... добавьте при необходимости другие функции...

    // Добавляет num битов, начиная с первого бита p.
    //
    void Append( unsigned char* p, size_t num );

    // Запрос количества используемых битов (изначально 0).
    //
    size_t Size() const;

    // Получает num битов, начиная с start-ого бита, и
    // сохраняет результат, начиная с первого бита dst.
    //
    void Get(size_t start, size_t num, unsigned char* dst)
                                                    const;
private:
    // ...дополнительные детали...
};
```

2. Нельзя ли хранить партию в шахматы с использованием менее чем 12 битов на полуход? Если можно — покажите, как. Если нет — поясните, почему.

Решение

BitBuffer, убийца битов

1. **Для реализации решения *д*) второго вопроса задачи 26 вы решили создать класс, управляющий буфером битов. Реализуйте его максимально переносимо, с тем чтобы он корректно работал на любом компиляторе, соответствующем стандарту C++, независимо от платформы.**

Для начала обратите внимание, что здесь нет упоминания о том, что байт состоит из 8 битов, которое было в предыдущей задаче — здесь это условие попросту неприменимо. Нам нужно решение, компилирующееся и корректно работающее в любой реализации C++, соответствующей стандарту, независимо от того, на какой платформе она работает.

Требуемый условием задачи интерфейс выглядит следующим образом.

```
class BitBuffer {
public:
    void Append( unsigned char* p, size_t num );
    size_t Size() const;
    void Get(size_t start, size_t num, unsigned char* dst)
                                                    const;
```

```
    // ...
};
```

Вы можете удивиться, почему интерфейс `BitBuffer` определен с использованием указателей на `unsigned char`. Во-первых, в стандарте C++ нет такой вещи, как указатель на бит. Во-вторых, стандарт C++ гарантирует, что операции над беззнаковыми типами (включая `unsigned char`, `unsigned short`, `unsigned int` и `unsigned long`) не будут вызывать сообщения компилятора типа "Вы не инициализировали этот байт!" или "Это некорректное значение!". Бьярн Страуструп пишет в [Stroustrup00]:

Беззнаковые целые типы идеально подходят для использования в качестве хранилища для битового массива.

От компиляторов требуется предоставить возможность рассматривать `unsigned char` (как и другие беззнаковые типы) как просто хранилище набора битов — именно то, что нам и требуется. Имеются и другие подходы, но данный подход позволит нам получить навыки программирования работы с отдельными битами, что и является основной целью данной задачи.

Главный вопрос при реализации `BitBuffer` — какое внутреннее представление следует использовать? Я рассмотрю две основные альтернативы.

Попытка №1: использование `unsigned char`

Первая мысль — реализовать `BitBuffer` с использованием большого внутреннего блока `unsigned char`, и самостоятельно работать с отдельными битами при их размещении и выборке. Мы можем позволить классу `BitBuffer` иметь член типа `unsigned char*`, который указывал бы на буфер, но, тем не менее, давайте воспользуемся вектором `vector<unsigned char>`, чтобы не заботиться о вопросах распределения памяти.

Вам кажется, что все это звучит очень просто? Если да — значит, вы не пытались реализовать (и протестировать!) сказанное. Потратьте на это два-три часа, и вновь вернитесь к данной задаче? Я готов держать пари, что больше вы так не скажете.

Мне не стыдно признаться, что эта версия класса отняла у меня массу времени и усилий при ее написании. Даже черновые наброски оказалось сделать труднее, чем мне казалось сначала, не говоря уже об отладке и устранении всех ошибок, когда мне пришлось не раз воспользоваться отладочной печатью для вывода промежуточных результатов и как минимум полдюжины раз пройти код пошагово.

И вот результат. Я не говорю, что он идеален, но он прошел все мои тесты, включая добавление одного и нескольких битов и некоторые граничные случаи. Обратите внимание, что эта версия исходного текста работает одновременно с блоками байтов — например, если мы используем 8-битовые байты и имеем смещение, равное 3 битам, то мы копируем первые три бита как отдельную единицу, и так же поступаем с последними 5 битами, получая 2 операции на байт. Для простоты я также требую, чтобы пользователь предоставлял буферы на байт большие, чем необходимо. Таким образом, я могу упростить свой код, позволив ему работать за концом буфера.

```
// Пример 27-1: реализация BitBuffer с использованием
//              vector<unsigned char>. Трудная,
//              скрупулезная работа. Брр...//

class BitBuffer {
public:
    BitBuffer() : buf_(0), size_(0) { }

    // Добавление num битов, начиная с первого бита p.
    //
    void Append( unsigned char* p, size_t num ) {
        int bits = numeric_limits<unsigned char>::digits;

        // Первый байт назначения и смещение бита
```

```
        int dst = size_ / bits;
        int off = size_ % bits;

        while( buf_.size() < (size_+num) / bits + 1 )
            buf_.push_back( 0 );
        for( int i = 0; i < (num+bits-1)/bits; ++i ) {
            unsigned char mask = FirstBits(num - bits*i);
            buf_[dst+i] |= (*(p+i) & mask) >> off;
            if( off > 0 )
                buf_[dst+i+1] = (*(p+i) & mask)<<(bits-off);
        }
        size_ += num;
    }

    // Запрос количества используемых битов
    // (изначально ноль).
    //
    size_t Size() const {
        return size_;
    }

    // Получение num битов, начиная с start-го бита
    // (нумерация начинается с 0), и сохранение результата
    // начиная с первого бита dst. Буфер, на который
    // указывает dst, должен быть по крайней мере на один
    // байт больше, чем минимально необходимый для хранения
    // num битов.
    //
    void Get(size_t start, size_t num, unsigned char* dst)
        const {
        int bits = numeric_limits<unsigned char>::digits;

        // Первый исходный байт и смещение бита
        int src = start / bits;
        int off = start % bits;

        for( int i = 0; i < (num+bits-1)/bits; ++i ) {
            *(dst+i) = buf_[src+i] << off;
            if( off > 0 )
                *(dst+i) |= buf_[src+i+1] >> (bits - off);
        }
    }
private:
    vector<unsigned char> buf_;
    size_t size_; // in bits
    // Создание маски, в которой первые n битов равны 1, а
    // остальные - 0.
    //
    unsigned char FirstBits( size_t n ) {
        int num=min(n,numeric_limits<unsigned char>::digits);
        unsigned char b = 0;
        while( num-- > 0 )
            b = (b >> 1) |
              (1<<(numeric_limits<unsigned char>::digits-1));
        return b;
    }
};
```

Этот исходный текст нетривиален. Потратьте некоторое время на то, чтоб ознакомиться с ним внимательнее, понять, как он работает, и убедиться, что он корректно решает поставленные перед ним задачи[47]. (Если вам покажется, что вы нашли в исходном тексте ошибку, пожалуйста, сначала напишите тестовую программу, которая бы демонстрировала

[47] Вероятно, разобраться в приведенном исходном тексте (а кое-где даже улучшить его) вам поможет знакомство с книгой [Warren03]. — *Прим. ред.*

ее наличие. Если наличие ошибки подтвердится — прошу вас, отправьте мне сообщение об ошибке вместе с вашей тестовой программой, демонстрирующей наличие ошибки.)

Попытка №2: использование стандартного контейнера упакованных битов

Вторая идея заключается в том, чтобы обратить внимание на то, что стандартная библиотека уже содержит два контейнера для хранения битов: `bitset` и `vector<bool>`. Для наших целей `bitset` — плохой вариант, поскольку `bitset<N>` имеет фиксированную длину `N`, а мы должны работать с потоками битов переменной длины. Не получается... Зато `vector<bool>`, несмотря на все его недостатки в данном случае оказывается тем, "что доктор прописал"[48]. (Конечно, стандарт не требует, чтобы реализация `vector<bool>` использовала упакованные биты, а только поощряет это. Тем не менее, большинство реализаций именно так и поступают.)

Самое главное, что я могу сказать о приведенном далее исходном тексте, — это то, что исходный текст примера 27-2 был совершенно корректен *при первом его написании.*

Да, именно так. Все, что я сделал между первой компиляцией и окончательной версией исходного текста — это исправление несколько мелких синтаксических опечаток, в частности, добавление пропущенной точки с запятой и пары скобок там, где я упустил из виду, что приоритет оператора % выше приоритета оператора +. Вот этот исходный текст.

```
// Пример 27-2: Реализация BitBuffer с использованием
//              vector<bool>.
//
class BitBuffer {
public:
    // Добавление num битов, начиная с первого бита p.
    //
    void Append( unsigned char* p, size_t num ) {
        int bits = numeric_limits<unsigned char>::digits;
        for( int i = 0; i < num; ++i ) {
            buf_.push_back( *p & (1 << (bits-1 - i%bits)) );
            if( (i+1) % bits == 0 )
                ++p;
        }
    }

    // Запрос количества используемых битов
    // (изначально ноль).
    size_t Size() const {
        return buf_.size();
    }

    // Получение num битов, начиная с start-го бита
    // (нумерация начинается с 0), и сохранение результата
    // начиная с первого бита dst.
    //
    void Get( size_t start, size_t num, unsigned char* dst )
        const {
        int bits = numeric_limits<unsigned char>::digits;
        *dst = 0;
        for( int i = 0; i < num; ++i ) {
            *dst |= unsigned char(buf_[start+i])
                 << (bits-1 - i%bits);
            if( (i+1) % bits == 0 )
                *++dst = 0;
        }
    }
private:
```

[48] Более подробно вопрос о `vector<bool>` рассмотрен в [Sutter02] (задача 1.13 в русском издании). — *Прим. ред.*

```
    vector<bool> buf_;
};
```

Вас не должно удивлять, что написать эту версию было существенно проще, чем пример 27-1. Здесь вместо разработки собственного кода для работы с битами используется уже готовый; размер текста оказывается примерно в два раза меньше, чем в предыдущей версии, и как результат — непропорционально малое количество ошибок. Такой код, кроме того, яснее и понятнее; в частности, теперь мне не требуется, чтобы вызывающая программа выделяла дополнительную память только для того, чтобы мой код был проще, как это было в первой версии исходного текста.

Я подозреваю, что оба решения (в особенности первое) можно было бы улучшить — в частности, в исходном тексте могут быть не замеченные мною ошибки, текст может быть упрощен способом, о котором я не подумал, — но я думаю, что оба решения близки к идеалу как в плане корректности, так и в плане стиля.

Плотная упаковка

Давайте теперь еще раз обратимся к упакованному представлению шахматной партии и посмотрим, нельзя ли упаковать ее еще плотнее.

2. Нельзя ли хранить партию в шахматы с использованием менее чем 12 битов на полуход? Если можно — покажите, как. Если нет — поясните, почему.

Да, но если вы захотите использовать это представление в коде, вам потребуется специальный битовый контейнер вроде `Bitbuffer`.

Например, имеется три способа.

Достичь упаковки полухода в 10 битов достаточно просто. Нам достаточно указать, какая именно фигура (а для каждого полухода их не более 16) и в какую клетку (которых на доске 64) ходит. Цвет фигуры определяется номером полухода. Перенумеровав фигуры (например, в порядке их размещения по горизонталям, а в пределах горизонтали — по вертикалям) и клетки доски, мы получаем, что для указания фигуры нам надо 4 бита, а для указания клетки, в которую она ходит, — 6 битов; итого достаточно 10 битов, чтобы однозначно определить полуход.

Можно ли улучшить этот результат? Судите сами: мы можем закодировать все клетки доски в качестве целевых клеток полухода, в то время как правила игры позволяют быть таковыми только малому количеству клеток. Таким образом, в нашем представлении явно имеется избыточность. Аналогично, наше представление позволяет записать ход любой фигуры в заданную клетку, в то время как такой ход могут сделать далеко не все фигуры, причем некоторые фигуры в принципе не могут попасть в некоторые клетки. Так что, например, можно использовать для кодирования клетки 6 битов, а затем выяснить, какие фигуры могут пойти в данную клетку, и использовать от 0 до 4 битов для указания одной из них. Если таких фигур мало, нам не потребуются все 4 бита. При декодировании из анализа текущей позиции мы знаем, какое количество фигур может попасть в целевую клетку, а значит, нам известно, сколько именно битов из входного потока требуется запросить, чтобы декодировать полуход.

Мы можем закодировать полуход с использованием не менее 0 и не более 8 битов следующим образом: сначала надо разработать способ упорядочения разрешенных шагов; например, мы можем упорядочить фигуры описанным выше способом, в соответствии с занимаемыми ими позициями, а для каждой фигуры — упорядочить возможные ходы с использованием того же принципа, что и для фигур. Затем номер фактического хода записывается с использованием минимально необходимого количества битов. Например, начальная позиция имеет 20 возможных ходов; для представления их в бинарном виде требуется $\text{ceil}(\log_2(20)) = 5$ битов.

В результате минимальное количество битов, необходимое для записи полухода, равно 0 — если имеется только один возможный, вынужденный ход. Но сколько би-

тов требуется в наихудшем случае? Этот вопрос непосредственно связан с вопросом о том, каково максимальное количество различных ходов может быть сделано в корректной шахматной позиции? Насколько мне известно, текущий известный максимум — 218 ходов в следующей позиции:

```
-  -  -  Q  -  -  -  -    3Q4
-  Q  -  -  -  -  Q  -    1Q4Q1
-  -  -  -  Q  -  -  -    4Q3
-  -  Q  -  -  -  -  R    2Q4R
Q  -  -  -  -  Q  -  -    Q4Q2
-  -  -  Q  -  -  -  -    3Q4
-  Q  -  -  -  -  R  p    1Q4Rp
-  K  -  B  B  N  N  k    1K1BBNNk
```

В этом наихудшем случае для кодирования хода в виде обычного двоичного числа достаточно 8 битов. В среднем, по-видимому, 5 битов будет достаточно для того, чтобы хранить типичный ход. Начальная позиция имеет 20 возможных ходов; типичный эндшпиль, например, король, ладья и две пешки на пустой доске, имеет около 30 допустимых ходов — что также приводит к 5 битам при использовании описанного метода.

Если задуматься, то становится понятным, что описанный метод близок к оптимальному, поскольку он представляет точный и непосредственный ответ на вопрос: Какой разрешенный ход был сделан? Мы используем минимальное количество битов для представления всех возможных ходов в виде двоичного числа, располагая полной информацией о том, что происходило до этого хода.

Можно ли улучшить и этот результат? Да, но теперь результаты будут не столь впечатляющими, так как нам потребуются новые знания о шахматах, а речь идет об экономии долей битов. Для того чтобы проиллюстрировать возможность улучшения достигнутых нами результатов, обратимся еще раз к начальной позиции. В ней имеется 20 возможных ходов, что в нашей схеме требует $\text{ceiling}(\log_2(20)) = 5$ битов. Теоретически же в этой ситуации имеется только $\log_2(20) = 4.3$ битов информации, если считать все ходы равновероятными, так что в среднем нам должно хватить еще меньшего количества битов, если вспомнить о том, что бо́льшая часть всех партий начинается с одного из двух популярных ходов белых. Короче говоря, если мы располагаем дополнительной информацией об относительных вероятностях каждого хода (например, встраивая в механизм сжатия детерминистскую шахматную программу, которая может предположить, какие ходы в любой заданной позиции будут наиболее вероятны), то мы можем использовать кодирование с переменной длиной, такое как код Хаффмана или арифметическое сжатие, чтобы использовать меньшее количество битов для хранения более вероятных ходов. Цена такой высокой степени сжатия информации — повышенное время вычислений, использующих знания о предметной области.

Резюме

Эта задача проиллюстрировала, как знания о предметной области могут быть применены для получения существенно лучших решений поставленной задачи.

В результате даже без знаний о том, какие ходы наиболее вероятны в данной позиции, мы можем сохранить типичную 40-ходовую партию (80 полуходов) примерно в 50 байтах. Это очень немного, и достичь этого можно только путем применения знаний о предмете задачи для ее решения.

➢ Рекомендация

Оптимизация должна опираться на точные знания о том, *что* вы должны оптимизировать, и *как* вы должны это делать. Знания предметной области ничем невозможно заменить.

Ловушки, ошибки и головоломки

Перед тем как мы перейдем к заключительной части книги, содержащей конкретные примеры стиля программирования, давайте рассмотрим несколько других тем, связанных с языком C++, или просто головоломных ситуаций.

Почему C++, как и большинство других языков программирования, резервирует ключевые слова, так что вы не можете, например, использовать переменную с именем `class`? Почему строка кода, которая выглядит, как вполне правильная, выполняющая определенную работу, в результате не приводит ни к одной команде процессора, как будто она никогда и не существовала? И почему, казалось бы, вовсе некорректный текст нормально компилируется и работает? Почему при работе с числами с плавающей точкой желательно пользоваться `double`?

И наконец — мы когда-нибудь сможем разобраться с макросами?..

Задача 28. Ключевые слова, не являющиеся таковыми

Сложность: 3

Все ключевые слова равноценны для синтаксического анализатора, но некоторые оказываются равноценнее других. Здесь мы увидим, почему так важно резервирование ключевых слов, а также познакомимся с двумя ключевыми словами, которые вовсе не влияют на семантику программы на C++.

Вопрос для новичка

1. Почему большинство языков программирования имеют зарезервированные ключевые слова, которые не могут произвольно использоваться в программе?

Вопрос для профессионала

2. Как добавление ключевого слова `auto` изменяет семантику программы на C++?
3. Как добавление ключевого слова `register` изменяет семантику программы на C++?

Решение

Зачем нужны ключевые слова

Для C++ очень важно наличие ключевых слов, зарезервированных языком, которые не могут использоваться в качестве имен, например, типов, функций или переменных.

1. Почему большинство языков программирования имеют зарезервированные ключевые слова, которые не могут произвольно использоваться в программе?

В случае отсутствия таких зарезервированных слов было бы слишком легко написать программу, которую оказалось бы невозможно скомпилировать из-за ее неразрешимых неоднозначностей. Рассмотрим неправдоподобно простой исходный текст из примера 28-1*а*.

```
// Пример 28-1(а): корректная программа на C++.
//
int main() {
    if( true );         // 1: OK
    if(  42 );          // 2: OK
}
```

В строках 1 и 2 выполняется проверка условий; если условия истинны (а в приведенном примере так оно и есть), выполняется пустая инструкция.

Вряд ли это самый захватывающий исходный текст всех времен и народов. Я даже думаю, что это не самый захватывающий текст из тех, что написаны вами на прошлой неделе. Но, тем не менее, это корректный исходный текст на C++, и простейший пример для иллюстрации проблем, которые бы свалились нам на голову, если бы ключевые слова C++ могли использоваться в качестве идентификаторов.

Рассмотрим следующий умозрительный исходный текст, только что прибывший в мой кабинет (без тихого хлопка и не сопровождаясь запахом серы) из альтернативного мира, в котором C++ не резервирует ключевые слова, а программисты счастливы использовать их и в качестве идентификаторов. Пока что не будем разбираться в том, как воспримет текст компилятор, и рассмотрим более простой вопрос: как воспримет текст примера 28-1*б человек*?

```
// Пример 28-1(б): Не C++, но если бы это был он?
//
class if {              // Назовем класс "if" (не разрешено; но
public:                 // если бы это было можно сделать?)
    if( bool ) {}       // 3: Гм... конструктор?
};
```

Что означает строка 3? "Это просто, — может сказать кто-то из читателей. — Мы знаем, что условная инструкция в этом месте лишена смысла, причем имя типа не может быть проверяемым условием, так что понятно, что перед нами конструктор. Может, действительно есть определенный смысл в разрешении использовать ключевые слова в качестве имен — в конце концов, не так трудно предположить, что именно они должны означать!" Некоторые из разработчиков языков пошли по этой кривой дорожке... но это очень короткая дорожка! Вы споткнетесь на ней при первой же ситуации наподобие той, что демонстрируется с помощью строк 4 и 5 из следующего исходного текста.

```
// Пример 28-1(б), продолжение. Вернемся к
//                 первоначальному примеру...
//
int main() {
    if( true )      // 4: Гмм... и что это означает теперь?
    if( 42 );       // 5: Гмм... а это?
}
```

Что должны означать строки 4 и 5 в этом альтернативном мире? Имеют ли они тот же смысл, что и в рассмотренном нами ранее примере 28-1*а*? Или в них используется тип `if`, который имеет подходящий конструктор, и эти строки означают создание двух безымянных временных объектов? Если вы немного подумаете над этим вопросом, то быстро сообразите, что даже человек не в состоянии точно ответить на него. Чего же тогда ожидать от компилятора там, где пасует даже человек?

"Минутку, — может возразить человек, решивший пройти по кривой дорожке до конца, — но ведь можно ввести правило, которое сделает данный код рабочим! Создание временных объектов в строках 4 и 5 только для того, чтобы они тут же были уничтожены, — не слишком полезная вещь, так что мы можем считать, что это не то, к чему стремился программист, и рассматривать данные строки как обычные условные инструкции". Я надеюсь, что вы с негодованием отвергнете это решение как непригодный для употребления "хак". Тому есть множество причин. 1. Точно так же легко решить, что запись "`if(true);`" представляет собой по сути отсутствие операции, что вряд ли являлось целью программиста, так что следует трактовать обе инструкции как объявления временных объектов. 2. Какой бы из вариантов вы ни выбрали, он будет сильно зависеть от того, находится ли класс `if` в строках 4 и 5 в области видимости или нет. Коротко говоря, наличие таких (скажем прямо — весьма дурно пахнущих) правил в языке следует воспринимать как яркий красный свет, сигнализирующий о серьезных проблемах в дизайне языка.

Конечно, есть и другие способы получения неоднозначностей в случае, когда ключевые слова не резервируются. Вот еще один простой случай, который демонстрирует пример 28-1*в*.

```
// Пример 28-1в: Не C++, но если бы это был он?
//
class SomeFunctor {
public:
    int operator()( bool ) { return 42; }
};

SomeFunctor if;     // Назовем переменную "if"

// Вернемся опять к нашему коду...
```

```
int main() {
    if( true );      // 6: Гм... и что это означает?
    if( 42 );        // 7: Гм... а это?
}
```

Опять же, что должны означать строки 6 и 7? Являются ли они теми обычными условными конструкциями, которые мы так хорошо знаем и с которыми мы уже встречались в примере 28-1*a*? Или здесь используется переменная `if`, к которой применен оператор `operator()`, — и даже с совместимыми аргументами! — и в этом случае инструкции в строках 6 и 7 означают `if.operator()(true);` и `if.operator()(42);`? Я думаю, вполне очевидна невозможность определить, что именно имел в виду программист. Повторяю, это невозможно определить для человека; что уж тогда говорить о компиляторах.

Итак, становится ясно, что C++ (как и другие языки программирования) просто вынужден намертво, двухдюймовыми гвоздями прибить значения к некоторым именам. Он вынужден зарезервировать эти имена для собственного использования, поэтому они и называются *зарезервированными словами.*

Ключевые слова C++

Стандарт C++ резервирует в качестве ключевых слов 63 имени, которые перечислены в табл. 28.1. Большинство из этих имен хорошо вам знакомы и ежедневно используются вами.

Таблица 28.1. Ключевые слова C++

`asm`	`do`	`if`	`return`	`typedef`
`auto`	`double`	`inline`	`short`	`typeid`
`bool`	`dynamic_cast`	`int`	`signed`	`typename`
`break`	`else`	`long`	`sizeof`	`union`
`case`	`enum`	`mutable`	`static`	`unsigned`
`catch`	`explicit`	`namespace`	`static_cast`	`using`
`char`	`export`	`new`	`struct`	`virtual`
`class`	`extern`	`operator`	`switch`	`void`
`const`	`false`	`private`	`template`	`volatile`
`const_cast`	`float`	`protected`	`this`	`wchar_t`
`continue`	`for`	`public`	`throw`	`while`
`default`	`friend`	`register`	`true`	
`delete`	`goto`	`reinterpret_cast`	`try`	

Кроме того, вместо обычной формы записи, 11 операторов могут использовать запись при помощи слов, например, в условном выражении можно записать `and` вместо `&&`. Стандарт резервирует эти слова, так что вы не можете распоряжаться ими для использования в качестве собственных имен. Список этих имен приведен в табл. 28.2.

Таблица 28.2. Дополнительные зарезервированные слова

`and`	`and_eq`	`bitand`	`bitor`	`compl`	`not`
`not_eq`	`or`	`or_eq`	`xor`	`xor_eq`	

Итого, 74 — посчитайте сами! — определенных имени в ваших программах не могут быть использованы вами произвольным образом, в частности, в качестве имен типов, функций или переменных (см. [C++03, §2.11]).

Зарезервированные комментарии

Большинство ключевых слов выполняют какие-то действия. И это правильно — иначе зачем они нужны?

Однако некоторые ключевые слова и близко не делают того, на что вы могли бы надеяться. Некоторые из них не имеют никакого семантического влияния на программу вообще, т.е. семантически они эквивалентны пробельным символам, эдаким приукрашенным комментариям. Я имею в виду три ключевых слова — `auto`, `register` и, во многих отношениях, `inline` (см. задачу 25). (Тем, кто удивлен, почему здесь не указано ключевое слово `export`, могу посоветовать обратиться к задачам 9 и 10.)

auto

Рассмотрим сначала ключевое слово `auto`.

2. Как добавление ключевого слова `auto` изменяет семантику программы на C++?

Если говорить коротко: никак. `auto` — совершенно излишний спецификатор класса памяти. Он появляется только в описаниях локальных объектов (объявленных в блоке кода) и указывает, что эти объекты автоматически уничтожаются при завершении их функции или блока; однако во всех случаях, где допустимо применение `auto`, предполагается именно такое поведение, если это ключевое слово *не* указано, что и делает спецификатор `auto` излишним. То есть, `auto` означает то же, что и любой пробельный символ.

Сейчас некоторые подкованные читатели стандарта C++ могут громко возразить: "Это не совсем то, что говорит стандарт! В нем всего лишь сказано, что спецификатор `auto` *почти* всегда излишен!" И это так:

> *...спецификатор `auto` почти всегда излишен и редко используется; одно из предназначений `auto` — для явного отличия объявления от выражения.* — [C++03] §7.1.1

Да, это сказано в стандарте, но это не так (и я уже подал соответствующее сообщение в комитет). Почему? Правило C++, которое применяется к таким неоднозначностям, гласит, что все, что может быть объявлением, должно быть объявлением — и добавление `auto` ничего не меняет. Например:

```
// Пример 28-2: auto не разрешает неоднозначности.
//
int i;
int j;
int main() {
    int(i);       // Объявление i; не ссылка на ::i
    auto int(j);  // Объявление j, а не ссылка на ::j

    int f();      // Объявление функции, а не объект типа
                  // int, инициализированный по умолчанию
    auto int f(); // Такое же объявление функции, хотя и
                  // приведет к ошибке на строгом
}                 // компиляторе
```

Более подробно неоднозначность объявлений в C++ рассматривается в задаче 29 (см. также раздел 39 в [Meyers01]). В итоге `auto` не может использоваться для разрешения описанных неоднозначностей.

➢ Рекомендация

Никогда не используйте `auto`. Это ключевое слово имеет тот же смысл, что и пробельный символ.

Кстати, возможно, в будущем `auto` будет играть гораздо более важную роль. При работе над очередным стандартом C++ комитет рассматривает вариант применения

auto в качестве ключевого слова для автоматического выведения типа, так что в будущем, быть может, мы будем заменять объявления наподобие

```
vector<SomeNamepace::SomeType>::const_iterator i = v.begin();
```

простым и выразительным

```
auto i = v.begin();  // Возможное будущее расширение C++
```

register

Но достаточно об auto. Обратимся к register.

3. Как добавление ключевого слова register изменяет семантику программы на C++?

Говоря коротко: для большинства современных компиляторов — никак.

Чтобы понять, почему — посмотрим, что говорит стандарт сразу после процитированного примечания об auto... Начинается он так:

Спецификатор register имеет ту же семантику, что и спецификатор auto...

Гм... Как мы только что выяснили, это означает "не имеет семантики". Грустное начало... Но прочтем дальше:

...и, кроме того, подсказывает компилятору, что объявленный таким образом объект будет активно использоваться. [Примечание: подсказка может быть проигнорирована и в большинстве реализаций она будет проигнорирована, если запрашивается адрес объекта.] — [C++03] §7.1.1

Идея спецификатора register в том, что если некоторые переменные активно используются, то имеет смысл по возможности разместить их в регистрах процессора, что позволит работать с ними гораздо быстрее, чем если они будут выбираться из (относительно) медленного кэша или, что еще хуже, из основной памяти.

Все это было бы хорошо, но сегодня идея о том, что программист должен сам указывать компилятору, какую переменную следует разместить в регистре, а какую нет, — выглядит анахронизмом. Дело в том, что практически нереально, чтобы даже программист высочайшего уровня смог выбрать оптимальное распределение переменных по регистрам, которое обеспечит наивысшую производительность кода. Даже если программист точно знает, на каком процессоре будет выполняться его код (что бывает очень редко), и знает этот процессор на уровне разработчиков кодогенератора для этого процессора (что тоже маловероятно), он никогда не сможет распределить объекты по регистрам так же хорошо, как это сделал бы хороший компилятор, хотя бы потому, что программист ничего не знает о других преобразованиях, например, о встраивании, разворачивании циклов, устранении неиспользуемого кода и других — которые были выполнены компилятором. Вы не только не сможете выполнить работу так же хорошо, как ваш компилятор, но и не должны стремиться к этому — для этой работы имеется специальный инструментарий.

> ➢ **Рекомендация**
>
> Никогда не используйте ключевое слово register, кроме тех случаев, когда вы досконально знаете компилятор и особенности применения в нем этого спецификатора. Для большинства компиляторов это ключевое слово означает то же, что и пробельный символ.

Резюме

Итак, вы уже знаете, почему C++ резервирует ключевые слова, и выяснили, что два из них — auto и register — не привносят никаких семантических изменений в программу на C++. Не используйте их — по сути это всего лишь разновидность

пробельных символов, а внести в исходный текст пробельный символ можно с гораздо меньшими усилиями.

Никогда не пишите в программе `auto`. Это ключевое слово не имеет никакого значения.

Никогда не пишите в программе `register` (если только у вас нет абсолютной уверенности в том, что это слово имеет значение при использовании вашего компилятора в конкретном исходном тексте). Для большинства компиляторов этот спецификатор ничем не отличается от пробельного символа.

Задача 29. Инициализация ли это? Сложность: 3

В этой задаче мы рассмотрим такой вопрос: "Что если мы инициализировали объект, и ничего не произошло?"

В задаче предполагается, что все соответствующие стандартные заголовочные файлы включены и сделаны все необходимые using-объявления.

Вопрос для новичка

1. Что делает следующий исходный текст?

```
deque<string> coll1;
copy(istream_iterator<string>( cin ),
     istream_iterator<string>(),
     back_inserter( coll1 ) );
```

Вопрос для профессионала

2. Что делает следующий исходный текст?

```
deque<string> coll2( coll1.begin(), coll1.end() );
deque<string> coll3( istream_iterator<string>( cin ),
                     istream_iterator<string>() );
```

3. Что следует изменить в программе, чтобы она работала так, как, вероятно, ожидает программист?

Решение

Базовый механизм заполнения

1. **Что делает следующий исходный текст?**

```
// Пример 29-1(а)
//
deque<string> coll1;
copy(istream_iterator<string>( cin ),
     istream_iterator<string>(),
     back_inserter( coll1 ) );
```

В приведенном фрагменте объявляется изначально пустой дек строк с именем coll1. Затем выполняется заполнение путем копирования строк, разделенных пробельными символами, из стандартного потока ввода cin в дек с использованием функции deque::push_back, пока не закончится весь входной поток.

Исходный текст примера 29-1(*а*) эквивалентен следующему.

```
// Пример 29-1(б): Эквивалентная запись 29-1(а)
//
deque<string> coll1;
istream_iterator<string> first( cin ), last;
copy( first, last, back_inserter( coll1 ) );
```

Единственное отличие от исходного текста 29-1(*а*) состоит в том, что в нем объекты istream_iterator создавались "на лету", как временные неименованные, так что они уничтожались по завершении вызова copy. В примере 29-1(*б*) объекты istream_iterator являются именованными переменными и благополучно "переживают" вызов copy; они не будут уничтожены до тех пор, пока не будет достигнут конец области действия, в которой находится исходный текст примера 29-1(*б*).

Не инициализация

2. Что делает следующий исходный текст?

```
// Пример 29-2(а)
//
deque<string> coll2( coll1.begin(), coll1.end() );
```

В этом коде объявлен второй дек строк с именем coll2, и выполняется его заполнение с помощью подходящего конструктора. Здесь использован конструктор deque, который принимает пару итераторов, соответствующих диапазону, из которого должно выполняться копирование. В представленном исходном тексте coll2 инициализируется диапазоном итераторов, который соответствует "всему содержимому coll1".

Код примера 29-2(*а*) практически эквивалентен следующему.

```
// Пример 29-2(б): Практически то же, что и в 29-2(а)
//

// Дополнительный шаг: вызов конструктора по умолчанию
deque<string> coll2;

// Добавление элементов с использованием push_back
copy( coll1.begin(), coll1.end(), back_inserter( coll2 ) );
```

Единственное (небольшое) отличие заключается в том, что сначала вызывается конструктор по умолчанию coll2, а затем вставка элементов в контейнер с использованием функции push_back выполняется как отдельный шаг программы. Первоначальная версия кода выполняет все это при помощи конструктора, в который передается пара итераторов, которая, вероятно (хотя и не обязательно) означает то же самое, что и ранее.

Возможно, вы удивитесь, что я растолковываю очевидные вещи. Причина станет понятной, когда мы рассмотрим последнюю часть исходного текста. К сожалению, в нашем случае код ведет себя не так, как от него ожидают.

```
// Пример 29-2(в): Объявление еще одного дека?
//
deque<string> coll3( istream_iterator<string>( cin ),
                     istream_iterator<string>() );
```

Код выглядит и ведет себя на первый взгляд так же, как и в примере 29-1(*а*), а именно — создает дек строк, который заполняется из стандартного потока ввода, с тем отличием, что он пытается использовать синтаксис из примера 29-2(*а*), а именно — воспользоваться конструктором с диапазоном итераторов, передаваемым при помощи параметров.

В этом исходном тексте есть одна потенциальная и одна реальная проблемы. Потенциальная проблема заключается в том, что стандартный ввод cin полностью поступает в контейнер, так что входные данные, необходимые для ввода в другой части программы, могут оказаться считанными в контейнер, что может вызвать определенные логические проблемы.

Однако самая большая проблема в том, что на самом деле приведенный исходный текст ничего этого не делает. Почему? Потому что это — не объявление объекта coll3 типа deque<string>. На самом деле это объявление (вдохните побольше воздуха) функции с именем coll3,

которая возвращает deque<string> по значению
и получает два параметра:
  istream_iterator<string> с именем формального параметра cin,
  и функцию без имени формального параметра[49],

[49] Само собой разумеется, при этом происходит преобразование имени функции в указатель на нее. — *Прим. ред.*

которая не имеет параметров
и возвращает `istream_iterator<string>`.

(Попробуйте-ка произнести это как скороговорку.)

Что же здесь происходит? Мы имеем дело с тяжким наследием C, правилом, которое оставлено для обеспечения обратной совместимости: если часть кода может быть интерпретирована как объявление, она и является объявлением. Процитируем стандарт C++:

В грамматике имеется неоднозначность, когда инструкция может быть как выражением, так и объявлением. Если выражение с явным преобразованием типов в стиле вызова функции (_expr.type.conv_) является крайним слева, то оно может быть неотличимо от объявления, в котором первый оператор объявления начинается с открывающей круглой скобки "(". В этом случае инструкция рассматривается как объявление. — [C++03] §6.8

Не вдаваясь в детали, скажем, что причина возникновения этого правила — стремление помочь компиляторам при работе с жутким синтаксисом объявлений в C, который может оказаться неоднозначным. Чтобы компилятор мог справиться с этими неоднозначностями, введено универсальное положение — "если ты в сомнении — это объявление".

Если вы еще не убеждены, то взгляните на задачу 42 из [Sutter00] (задача 10.1 в русском издании), где есть похожий, но более простой пример. Давайте разберем объявление шаг за шагом, чтобы понять, что оно означает.

```
// Пример 29-2(г): Идентичен примеру 29-2(в), с удалением
//                 излишних скобок и добавлением typedef
//
typedef istream_iterator<string> (Func)();
deque<string> coll3( istream_iterator<string> cin, Func );
```

Это более похоже на объявление функции? Может, да, может, нет — так что давайте сделаем следующий шаг и уберем имя формального параметра `cin`, которое все равно игнорируется, и изменим имя `coll3` на нечто более похожее на обычное имя функции:

```
// Пример 29-2(д): идентичен примеру 29-2(в), с небольшими
//                 изменениями имен
//
typedef istream_iterator<string> (Func)();
deque<string> f( istream_iterator<string>, Func );
```

Теперь все становится понятно. Это "может быть" объявлением функции, так что в соответствии с синтаксическими правилами C и C++, оно таковым и является. С толку сбивает то, что оно выглядит очень похоже на синтаксис конструктора, а имя формального параметра `cin` (которое идентично имени переменной, находящейся в области видимости, и даже определено стандартом) и вовсе запутывает дело. Но в данном случае это не имеет значения, так как имя формального параметра и `std::cin` не имеют ничего общего, кроме одинакового написания.

Программисты время от времени сталкиваются с этой проблемой в своей работе, поэтому мы и отвели для ее рассмотрения отдельную задачу. Поскольку рассмотренный исходный текст, как это ни удивительно, представляет собой всего лишь объявление функции, реально он ничего не делает — для него компилятором не генерируется никакого кода, не выполняются никакие действия — не вызываются конструкторы дека, не создаются объекты.

Корректное заполнение

Было бы нечестно бросить вас на полпути, указав на проблему и не показав ее решения. А потому — последняя часть задачи:

3. Что следует изменить в программе, чтобы она работала так, как, вероятно, ожидает программист?

Все, что нам надо, — внести в исходный текст такие изменения, чтобы компилятор не мог рассматривать его как объявление функции. Есть два способа сделать это. Вот более привлекательный способ.

```
// Пример 29-3(а): устранение неоднозначности при помощи
//                 дополнительной пары скобок (неплохое
//                 решение; оценка 7/10)
//
deque<string> coll3((istream_iterator<string>(cin)),
                    istream_iterator<string>() );
```

Здесь мы просто добавили дополнительные скобки вокруг параметра, давая понять компилятору, что мы хотим, чтобы это был параметр конструктора, а не объявление параметра функции. Этот способ срабатывает, поскольку хотя `istream_iterator<string>(cin)` может быть объявлением переменной (или, как уже упоминалось, объявлением параметра), `(istream_iterator<string>(cin))` объявлением параметра быть не может. Соответственно, исходный текст примера 29-3*а* не может быть объявлением функции по той же причине, по которой не может быть объявлением функции `void f( (int i) )`, а именно — из-за наличия дополнительных скобок, которые не могут окружать параметр в объявлении функции.

Имеются и другие способы разрешения неоднозначности, которые делают невозможной интерпретацию указанного выражения как объявления функции, но я не буду приводить их здесь по очень простой причине: чтобы эти способы сработали, и вы, и ваш компилятор должны очень хорошо понимать этот "темный угол" стандарта языка.

➢ Рекомендация

Избегайте "темных углов" языка программирования, включая вполне корректные конструкции, которые, тем не менее, способны сбивать с толку не только программистов, но даже компиляторы.

Описанная неоднозначность синтаксиса по своей природе похожа на острие ножа, по которому лучше не ходить вовсе, а потому лучше полностью избегать использования такого синтаксиса. Я предпочитаю сам и настойчиво рекомендую вам другой способ, существенно более простой и понятный даже не слишком умным компиляторам и, кроме того, облегчающий чтение и понимание текста человеком.

```
// Пример 29-3(б): использование именованных переменных
//                 (рекомендуемое решение; оценка 10/10)
//
istream_iterator<string> first( cin ), last;
deque<string> coll3( first, last );
```

Как в примере 29-3*а*, так и в примере 29-3*б* достаточно внести предлагаемые изменения только в один параметр, но чтобы быть последовательным, я сделал именованными оба параметра.

➢ Рекомендация

В качестве аргументов конструктора лучше использовать именованные переменные. Это позволяет избежать возможной неоднозначности между вызовом конструктора и объявлением функции. Кроме того, зачастую это делает более понятным назначение вашего кода и облегчает тем самым его поддержку.

Резюме

Избегайте пыльных, темных, заросших паутиной углов языка программирования. Программисты обычно знают столько, сколько им достаточно для успешной работы.

При написании исходного текста помните о необходимости максимальной ясности и однозначности, всегда говорите именно то, что вы хотите сказать. Используйте в качестве аргументов конструктора именованные переменные, чтобы избежать описанных здесь неприятностей и сделать ваш код более ясным, поддерживаемым и сопровождаемым.

Задача 30. Двойная точность — вежливость программистов

Сложность: 4

Нет, эта задача не об этике. А о том, что есть разные виды "плавающей точки". Давайте проверим, насколько хорошо вы разбираетесь в основных операциях с плавающей точкой в C и C++.

Вопрос для новичка

1. В чем заключается различие между `float` и `double`?

Вопрос для профессионала

2. Пусть приведенная далее программа выполняется одну секунду (что достаточно необычно для современных настольных компьютеров).

```
int main() {
    double x = 1e8;
    while( x > 0 ) {
        --x;
    }
}
```

Как вы думаете, сколько времени будет выполняться этот код, если изменить `double` на `float`? Почему?

Решение

Два слова о float и double

1. **В чем заключается различие между `float` и `double`?**

Процитируем отрывок из стандарта C++:

Имеется три типа с плавающей точкой: `float`, `double` и `long double`. Тип `double` обеспечивает, как минимум, ту же точность, что и `float`, а тип `long double` — как минимум, ту же точность, что и `double`. Множество значений, которые принимает тип `float`, представляет собой подмножество значений типа `double`; а множество значений типа `double` является подмножеством значений типа `long double`. — [C++03, §3.9.1/8]

Как это определение, в особенности его последнее предложение, могут повлиять на ваши исходные тексты?

Колесо времени

2. **Пусть приведенная далее программа выполняется одну секунду (что достаточно необычно для современных настольных компьютеров).**

```
int main() {
    double x = 1e8;
    while( x > 0 ) {
        --x;
    }
}
```

Как вы думаете, сколько времени будет выполняться этот код, если изменить `double` на `float`? Почему?

Вероятно, новое время выполнения составит около одной секунды (в конкретных реализациях работа с `float` может быть немного быстрее, такой же по скорости, или немного медленнее, чем работа с `double`), либо выполнение никогда не прекратится — в зависимости от того, может ли тип `float` точно представить все целые значения от 0 до `1e8` включительно или нет.

Приведенная цитата из стандарта означает, что вполне возможны значения, представимые типом `double`, которые не в состоянии представить тип `float`. В частности, на некоторых распространенных платформах и компиляторах `double` может точно представить все целые числа из диапазона от 0 до `1e8`, но тип `float` на это не способен.

Что происходит, если `float` не в состоянии точно представить все целые числа от 0 до `1e8`? Тогда измененная программа, приступив к уменьшению счетчика, в конечном счете достигнет значения *N*, которое не может быть точно представлено и для которого выполняется равенство *N-1* == *N* (из-за недостаточной точности представления чисел с плавающей точкой)... и произойдет зацикливание на этом значении, пока не закончится электроэнергия, не произойдет сбой операционной системы (что для ряда операционных систем — обычное дело), Солнце не превратится в пульсар и не сожжет все внутренние планеты, Вселенная не погибнет от тепловой смерти... — словом, что произойдет раньше.[50]

О суживающем преобразовании типов

Некоторые читатели могут удивиться: "Какие могут быть проблемы — если, конечно, не считать приближение конца света? Константа `1e8` имеет тип `double`. При замене `double` на `float` программа не должна компилироваться из-за суживающего преобразования типов, не так ли?" Ну что ж, давайте привычно процитируем еще одно место из стандарта:

> *rvalue типа с плавающей точкой может быть преобразовано в rvalue другого типа с плавающей точкой. Если исходное значение точно представимо целевым типом, то результат преобразования является точным представлением. Если исходное значение оказывается между двумя соседними целевыми значениями, то результат преобразования является одним из этих значений и зависит от конкретной реализации. В противном случае поведение не определено.* — [C++03] §4.8/1

Это означает, что константа типа `double` может неявно быть преобразована в константу типа `float`, даже если при этом происходит потеря точности (т.е. части данных, так что результат обратного преобразования к типу `double` не будет равен исходному значению). Такое поведение в C++ разрешено для совместимости с C и для удобства работы, но о нем не следует забывать при работе с числами с плавающей точкой.

Резюме

Вычисления с плавающей точкой сложны, трудны и почти всегда — неочевидны. Я бы сказал — с известной долей самомнения — что все программисты делятся на три группы: те, кто знают, что они не понимают вычисления с плавающей точкой (и они правы); те, кто думают, что они понимают их (но они ошибаются); и те немногие эксперты, которые совсем не уверены в том, что когда-либо полностью сумеют разобраться в вычислениях с плавающей точкой (и это мудро).

---

[50] Вообще-то толку от такой программы — ноль, так что она просто увеличивает энтропию Вселенной, приближая ее тепловую смерть и ничего не давая взамен. Так что никогда не пишите такие экологически опасные программы!

Кстати, любая работа человека или машины приводит к тому же эффекту и ускоряет наступление конца света. Этот аргумент может вам пригодиться, когда ваш работодатель предложит вам поработать сверхурочно...

➢ Рекомендация

Запомните, что вычисления с плавающей точкой — таинственная и странная вещь. Будьте внимательны при использовании чисел с плавающей точкой и избегайте преобразования типов с плавающей точкой. Почти все, что люди (как они думают) знают об арифметике, оказывается не верно применительно к вычислениям с плавающей точкой.

Качественный компилятор предупредит вас, если вы попытаетесь сделать что-то с неопределенным поведением, а именно — поместить в переменную `float` величину типа `double`, которая меньше минимального или больше максимального представимого типом `float` значения. По-настоящему хороший компилятор позволяет включить предупреждения о попытках сделать что-то, что разрешено и определено в стандарте, но может вызвать потерю информации, а именно — поместить величину типа `double` в переменную типа `float`, находящуюся между минимальным и максимальным представимыми типом `float` значениями, но не имеющую точного представления в этом типе.

Словом, будьте внимательны, избегайте полагаться на преобразования типов с плавающей точкой и не забудьте включить всю диагностику, на которую только способен ваш компилятор, — тогда у вас есть хоть какой-то шанс перебраться вброд через мутные воды арифметики с плавающей точкой.

Задача 31. Сумеречное состояние... кода Сложность: 4

Иногда жизнь заставляет нас заниматься отладкой в ситуациях, которые иначе как таинственными, назвать сложно. Попробуйте свои силы и в этом. Сможете ли вы объяснить причину возникающих проблем?

Вопрос для профессионала

1. Один программист написал следующий код.

```
//--- file biology.h ---
//
// ... необходимые заголовочные файлы и прочее ...

class Animal {
public:
  // Функции, работающие с данным объектом:
  //
  virtual int Eat     ( int ) { /*…*/ }
  virtual int Excrete ( int ) { /*…*/ }
  virtual int Sleep   ( int ) { /*…*/ }
  virtual int Wake    ( int ) { /*…*/ }

  // Специально для животных, которые уже были в браке, и
  // иногда не терпят своих бывших супругов:
  //
  int EatEx     ( Animal* a ) { /*…*/ }
  int ExcreteEx ( Animal* a ) { /*…*/ }
  int SleepEx   ( Animal* a ) { /*…*/ }
  int WakeEx    ( Animal* a ) { /*…*/ }
  // …
};
// Конкретные классы
//
class Cat    : public Animal { /*…*/ };
class Dog    : public Animal { /*…*/ };
class Weevil : public Animal { /*…*/ };
// ... прочие животные ...

// Удобные, хотя и излишние, вспомогательные функции
//
int Eat    ( Animal* a ) { return a->Eat( 1 );     }
int Excrete( Animal* a ) { return a->Excrete( 1 ); }
int Sleep  ( Animal* a ) { return a->Sleep( 1 );   }
int Wake   ( Animal* a ) { return a->Wake( 1 );    }
```

К сожалению, этот исходный текст не компилируется. Компилятор отклоняет определение как минимум одной из ...Ex-функций с сообщением о том, что функция уже определена.

Чтобы обойти эту ошибку компиляции, программист закомментировал ...Ex-функции, после чего скомпилировал программу и начал тестирование. К сожалению, функция-член `Animal::Sleep` не всегда работала корректно; однако когда программист вызывал ее непосредственно, все было в порядке. Однако при попытке вызвать ее из функции-оболочки `Sleep`, в которой не выполняется никаких иных действий, кроме вызова функции-члена, иногда не происходило ничего... не всегда, но такое случалось. И наконец, когда программист попытался разобраться в проблеме при помощи отладчика (или карты символов, сгенерированной компоновщиком), то он вообще не нашел и следа кода `Animal::Sleep`.

Это что, ошибка компилятора? Следует ли программисту писать гневное письмо разработчику компилятора? Искать хитрый вирус, заползший из недр Интернета? Списать все на проблему 2000 года? Выписать шамана из дебрей Амазонки для изгнания злых духов из системного блока?

Что же все-таки произошло?

Решение

1. **Один программист написал следующий код.**

```
[...]
```

Что же все-таки произошло?

Если говорить коротко — такие симптомы могут быть вызваны разными заболеваниями, но их совокупность с высокой степенью вероятности говорит о том, что программист наступил на грабли непродуманного использования макросов.

Мотивация

Некоторые распространенные среды программирования C++ предоставляют макросы, преднамеренно созданные для изменения имен функций. Обычно это делается исходя из "хороших" или "невинных" побуждений, в частности — для обратной или прямой совместимости API. Например, если функция `Sleep` в некоторой версии операционной системы заменена функцией `SleepEx`, производитель, поставляющий заголовочный файл, в котором объявлены эти функции, может решить "любезно" предоставить макроопределение, которое автоматически заменит имя `Sleep` на `SleepEx`:

```
#define Sleep SleepEx
```

Это — определенно Плохая Мысль. Макросы представляют собой антитезу инкапсуляции, поскольку их область действия невозможно проконтролировать — даже автору макроса.

Макросам наплевать...

Макросы по многим причинам — весьма неприятная вещь, которая может стать попросту опасной. В первую очередь это связано с тем, что макросы — средство замены текста, действующее во время обработки исходного текста препроцессором, до того как начнется какая-либо проверка синтаксиса и семантики. Далее перечислены некоторые из неприятностей, связанных с макросами.

1. Макросы изменяют имена — слишком часто, чтобы это не приносило особого вреда.

Будет преуменьшением сказать, что проблема только в том, что это мешает при отладке приложения. Такое переименование при помощи макросов означает, что когда вы думаете, что вызываете некоторую функцию, на самом деле ее вызов не происходит.

Рассмотрим, например, нашу функцию `Sleep`, не являющуюся членом.

```
int Sleep( Animal* a ) { return a->Sleep( 1 ); }
```

Вы не в состоянии найти `Sleep` ни в объектном коде, ни в карте, созданной компоновщиком, — просто потому что в действительности она называется `SleepEx`. Когда вы столкнетесь с отсутствием функции `Sleep`, вы можете решить, что компилятор встроил ее в код — это может объяснить, куда подевалась ваша функция и почему ее не видно в объектном коде. Если вы придете к такому заключению и напишете гневное письмо по поводу сверхагрессивной оптимизации разработчику компилятора, окажется, что вы обратились не по адресу, не в ту компанию (или, про крайней мере, не в тот ее отдел).

Некоторые из читателей книги вполне могли сталкиваться с такими неприятностями в своей практике. Если вы, как и я, не удовлетворитесь первым попавшимся

объяснением таинственного исчезновения функции и начнете искать ее при помощи пошагового прохождения программы, то обнаружите, что в той строке, где должна вызываться ваша функция, действительно имеет место вызов функции, только какой-то другой, несмотря на то, что в вашем исходном тексте указано верное ее имя. Обычно такое поведение при пошаговом прохождении быстро направляет мысли в правильное русло, и вы понимаете, что же именно произошло, и не всегда тихим, но всегда недобрым словом поминаете автора этого замечательного макроса.

Но это еще не все. Дело в том, что:

1(б). C++ имеет собственные средства для работы с именами. То, что макросы предоставляют другой способ для выполнения той же работы, приводит к эффекту, который лучше всего определить как "нездоровое взаимодействие".

Вы можете решить, что не такое уж это и большое дело — изменение имени функции. Ну что ж, зачастую это действительно так. Но что если вы изменили имя функции на другое, принадлежащее существующей функции? Что делает C++, когда обнаруживает две функции с одинаковыми именами? Он перегружает их. А это совсем не так хорошо, тем более если это происходит так, что вы об этом и не подозреваете.

Именно это, увы, и происходит в случае с нашей функцией `Sleep`. Вся причина того, что разработчик библиотеки любезно предоставил макрос для автоматического переименования `Sleep` в `SleepEx` заключается в наличии обоих функций в поставляемой библиотеке. Рассмотрим случай, когда эти функции могут иметь различные сигнатуры. Когда мы пишем собственную функцию `Sleep`, мы знаем о наличии библиотечной функции `Sleep` и можем принять меры для того, чтобы избежать неоднозначностей или других ошибок, которые могут привести к проблемам. Более того, мы можем умышленно использовать перегрузку, чтобы получить функцию с поведением, похожим на поведение библиотечной функции `Sleep`. Но если при этом окажется, что имя функции совсем иное, то перегрузка не только поведет себя не так, как ожидалось, но ее может не быть вообще.

В контексте нашего вопроса такой изменяющий имя функции `Sleep` макрос может частично объяснить, почему в разных ситуациях вызываются разные функции; то, какая именно функция будет вызвана, может зависеть от того, как именно сработает разрешение перегрузки для конкретных типов, использованных в различных местах вызова. Иногда вызывается наша функция, иногда — библиотечная; а то, какая именно функция будет вызвана, зависит от многих обстоятельств, причем, возможно, не самым очевидным образом.

Если бы даже на этом наш рассказ о последствиях применения макросов и закончился, рассказанное было бы достаточно неприятно. Но, к сожалению, осколки от взрыва макроса разлетаются в разных направлениях...

2. Макросы не заботятся о типах.

Исходное предназначение макроса `Sleep`, о котором шла речь выше, — изменение имени глобальной функции. К сожалению, макрос изменяет имя `Sleep` везде, где только находит его. Если ему попадется глобальная переменная с именем `Sleep`, то это имя будет молча заменено на `SleepEx`. Это еще одна очень Плохая Вещь.

3. Макросы не заботятся об области видимости.

Еще хуже то, что макрос, созданный для замены имени глобальной функции, не являющейся членом класса, будет с тем же успехом заменять как имена функций, так и другие такие же имена, являющиеся членами класса или инкапсулированные в ваше собственное пространство имен. В рассматриваемом в задаче случае у нас имеется класс, в котором есть функции `Sleep` и `SleepEx`; многие из описанных в условии задачи проблем могут быть, по крайней мере, частично объяснены наличием макроса, переименовывающего `Sleep`, что приводит к невидимой перегрузке наших собственных функций друг с другом. Такая перегрузка функций-членов (как и глобальная, о которой рассказывалось в п. 1) может объяснить, почему иногда вызывается не та

функция-член, которая ожидается. Это зависит от того, как именно будет выполнено разрешение перегрузки для конкретных типов, использованных в точке вызова.

Если вы решите, что это еще одна Плохая Вещь — вы окажетесь совершенно правы. Больше всего это напоминает доктора без перчаток и диплома (неразумный разработчик заголовочного файла библиотеки) с грязными руками (макросами), который вскрывает брюшную полость пациента (класс или пространство имен) и начинает там копаться, переставляя местами внутренности (члены и прочий код)... во время приступа лунатизма (даже не осознавая, что же он делает).

Резюме

Если говорить коротко — макросы никогда не заботятся ни о чем.

➢ Рекомендация

Избегайте макросов. Никогда, никогда, никогда даже не думайте о том, чтобы написать макрос, который представляет собой обычное слово или аббревиатуру.

Ваша стандартная реакция на макрос должна быть однозначной — "Изыди!" (или, на худой конец, "Vade retro!") — пока вы полностью не уясните себе причины его использования в конкретном случае, где его применение не является "хаком". Макросы небезопасны с точки зрения типов, с точки зрения областей видимости... да они просто небезопасны и точка! Если вы вынуждены писать макрос — избегайте размещения его в заголовочном файле и попытайтесь дать ему длинное и персонифицированное имя, которое вряд ли кому-нибудь придет в голову использовать в качестве имени в программе (и вообще вряд ли придет в голову кому-либо).

➢ Рекомендация

Для инкапсуляции имен предпочтительно использовать пространства имен.

Коротко говоря — пользуйтесь инкапсуляцией. Хорошая инкапсуляция — не только признак хорошего дизайна; это еще и возможная защита от неожиданных угроз, о которых вы можете даже не подозревать. Макросы представляют собой антитезу инкапсуляции в силу того, что область действия макроса не может контролироваться даже его автором. Классы и пространства имен входят в число полезных инструментов C++, которые помогают управлять и минимизировать взаимозависимости между различными частями программы, которые по дизайну не должны быть связаны друг с другом. Благоразумное использование этих и других возможностей C++ для обеспечения инкапсуляции не только обеспечивает лучший дизайн, но и служит в то же время защитой от непродуманного кодирования коллег-программистов.

Задача 32. Небольшие очепятки и прочие курьезы

Сложность: 5

Иногда даже маленькая и почти незаметная опечатка может стать источником больших проблем. Приведенные ниже примеры показывают, как трудно бывает найти допущенную опечатку и как легко увидеть ее там, где ее никогда и не было.

Попытайтесь ответить на вопросы, не прибегая к помощи компилятора.

Вопрос для профессионала

1. Каким будет вывод следующей программы, скомпилированной соответствующим стандарту компилятором?

```
// Пример 32-1
//
#include <iostream>
#include <iomanip>

int main() {
    int x = 1;
    for( int i = 0; i < 100; ++i );
        // Следующая строка - инкремент???????????/
        ++x;
    std::cout << x << std::endl;
}
```

2. О скольких различных ошибках сообщит соответствующий стандарту компилятор при работе с приведенным исходным текстом?

```
// Пример 32-2
//
struct X {
    static bool f( int* p ) {
        return p && 0[p] and not p[1:>>p[2];
    };
};
```

Решение

1. Каким будет вывод следующей программы, скомпилированной соответствующим стандарту компилятором?

Вывод примера 32-1, если предположить, что в конце строки с комментарием нет невидимых пробельных символов, будет представлять собой 1.

Здесь есть две хитрости — одна очевидная и одна более тонкая.

Начнем с цикла `for`.

```
for( int i = 0; i < 100; ++i );
```

Наличие точки с запятой в конце — стандартная опечатка, которая (обычно непредумышленно) делает тело цикла `for` состоящим из пустой инструкции. Несмотря на отступы (и даже охватывающие фигурные скобки) следующие за таким циклом строки телом цикла не являются.

Здесь же это не более чем преднамеренный отвлекающий маневр — поскольку вовсе неважно, выполнялись бы следующие строки в цикле или нет. Дело в том, что в любом случае инкремент `x` не выполняется ни разу.

Давайте внимательно рассмотрим строку с комментарием. Вы заметили, как странно она заканчивается — символом `'/'`?

```
// Следующая строка - инкремент???????????/
```

Николай Смирнов пишет:

Вероятно то, что случилось с программой, очевидно для вас, но я потерял несколько дней на отладку большой программы, в которой допустил подобную ошибку. Я внес строку с комментарием, который заканчивался большим количеством вопросительных знаков, и случайно отпустил клавишу <Shift> раньше времени. В результате случайно получилась последовательность '??/', представляющая собой триграф, который в первой фазе оказался преобразован в символ '\', а он в свою очередь во второй фазе обработки исходного текста аннулировал следующий за ним символ '\n' (Н. Смирнов, частное сообщение).

Последовательность ??/ преобразуется в символ '\', который, будучи последним символом в строке, представляет собой директиву стыковки строк! В нашем случае такой стыкуемой строкой оказалась строка с инструкцией ++x;, которая теперь оказалась частью комментария, так что инкремент в приведенной программе ни разу не выполняется.

Интересно, что если вы посмотрите документацию по Gnu g++ в части, где говорится об опции командной строки -Wtrigraphs, то встретите следующее некорректное обобщение:

Предупреждение не выдается для триграфов в комментариях, поскольку они не влияют на смысл программ[51].

Это может быть вполне справедливо в подавляющем большинстве случаев, но вы только что познакомились с контрпримером — из реальной программы! — когда приведенное утверждение оказывается неверным.

2. О скольких различных ошибках сообщит соответствующий стандарту компилятор при работе с приведенным исходным текстом?

```
// Пример 32-2
//
struct X {
    static bool f( int* p ) {
        return p && 0[p] and not p[1:>>p[2];
    };
};
```

Краткий ответ: ноль. Этот код совершенно корректен и соответствует стандарту (хочет ли того автор данного исходного текста или нет).

Рассмотрим по порядку каждое из выражений, которые могут вызвать вопросы, и покажем, почему в действительности они корректны.

- Выражение 0[p] совершенно корректно и имеет тот же смысл, что и выражение p[0]. В C (и C++) выражение вида x[y], где либо x, либо y имеет тип указателя, а другая величина представляет собой целочисленное значение, всегда означает *(x+y). В нашем случае 0[p] и p[0] имеют один и тот же смысл, поскольку означают *(0+p) и *(p+0) соответственно, что совершенно одинаково. Дополнительную информацию можно найти в [C99] §6.5.2.1.
- and и not представляют собой корректные ключевые слова, представляющие собой альтернативную запись операторов && и ! соответственно.
- :> представляет собой совершенно корректный диграф символа], а не "смайлик". Этот диграф превращает последнюю часть выражения в простое неравенство p[1]>p[2].

[51] Поиск в Google по запросу *"trigraphs within comments"* дает как этот, так и несколько других интересных и/или забавных трюков.

- "Лишняя" точка с запятой после описания функции-члена разрешена и не привносит никаких неприятностей. Синтаксис определения класса C++ разрешает объявление пустых членов (одинокие точки с запятой) где угодно в пределах класса, так часто, как только вы пожелаете. Например, следующая строка — совершенно корректное определение класса без членов:

```
class X { ;;;;;;;;;; };
```

Для большинства программистов наибольший сюрприз из представленных здесь преподносит диграф :>. Конечно, возможно, здесь вкралась опечатка и автор на самом деле имел в виду нечто иное, например, p[1]>>p[2], но даже если это и опечатка, то исходный текст и с ней (в таком случае — к сожалению) остается совершенно корректным.

Задача 33. Ооооператоры

Сложность: 4

Сколько операторов можно поместить подряд так, чтобы это имело смысл? В определенном смысле эту задачу можно считать шуткой.

Вопрос для новичка

1. Какое наибольшее количество плюсов (+) могут быть расположены подряд, без включения пробельных символов, в корректной программе на C++?
   Примечание: конечно, плюсы в комментариях, директивах препроцессора, макросах и литералах не учитываются. Это было бы неспортивно.

Вопрос для профессионала

2. Аналогично, чему равно наибольшее количество каждых из приведенных далее символов, которые могут располагаться в программе подряд, без вставки пробельных символов, вне комментариев в корректной программе на C++?

   а) &

   б) <

   в) |

   Например, для пункта *а*) код `if(a&&b)` демонстрирует тривиальный случай двух последовательных символов & в корректной программе на C++. Попробуйте найти примеры с большим количеством символов.

Решение

Правило "максимального глотка"

Правило "максимального глотка" гласит, что при разбивке символов исходного текста на лексемы компилятор должен поступать в соответствии с жадным алгоритмом — т.е. выбирать наиболее длинную из возможных лексем. Таким образом, >> всегда интерпретируется как единая лексема — оператор извлечения из потока (или битового сдвига вправо), и никогда — как две отдельные лексемы >, даже в ситуации наподобие следующей:

```
template<class T = X<Y>> ...
```

Вот почему такой код следует писать с дополнительным пробелом:

```
template<class T = X<Y> > ...
```

Аналогично, >>> всегда интерпретируется как >> с последующим за ним >, но не как > с последующим >>.

Операторные шутки

1. **Какое наибольшее количество плюсов (+) могут быть расположены подряд, без включения пробельных символов, в корректной программе на C++?**

   ***Примечание*: конечно, плюсы в комментариях, директивах препроцессора, макросах и литералах не учитываются. Это было бы неспортивно.**

   Можно создать исходный файл, содержащий последовательность символов + произвольной длины (ограниченной возможностями компилятора по обработке длинной строки).

   Если последовательность состоит из четного количества символов +, она интерпретируется как `++ ++ ++ ++ ... ++`, т.е. как последовательность двухсимвольных

лексем ++. Для того чтобы такой код был работоспособен и имел однозначно определенную семантику, все, что нам надо, — это класс, в котором определен пользовательский префиксный оператор ++, обеспечивающий возможность цепочечного вызова. Например:

```
// Пример 33-1(a)
//
class A {
public:
    A& operator++() { return *this; }
};
```

Теперь мы можем записать код

```
A a;
++++++a;          // Означает: ++ ++ ++ a;
```

который работает следующим образом

```
a.operator++().operator++().operator++()
```

А что, если последовательность имеет нечетное число символов +? Тогда она интерпретируется как ++ ++ ++ ++ ... ++ +, последовательность двухсимвольных лексем ++, за которой следует заключающий символ +. Для того чтобы такой код был работоспособен, надо дополнительно определить унарный оператор +.

```
// Пример 33-1(б)
//
class A {
public:
    A& operator+ () { return *this; }
    A& operator++() { return *this; }
};
```

Теперь мы можем записать код

```
A a;
+++++++a;     // Означает: ++ ++ ++ + a;
```

который работает следующим образом.

```
a.operator+().operator++().operator++().operator++()
```

Это очень простой трюк. Создание необычно длинных последовательностей других символов может оказаться немного сложнее, но все равно возможным.

Злоупотребление операторами

Код в примерах 33-1*а* и 33-1*б* не слишком злоупотребляет обычной семантикой операторов ++ и +. В дальнейшей работе, однако, нам придется далеко отойти от общепринятых правил кодирования. То, что вы увидите дальше, — не для промышленного использования, а всего лишь для собственного удовольствия.

2. Аналогично, чему равно наибольшее количество каждых из приведенных далее символов, которые могут располагаться в программе подряд, без вставки пробельных символов, вне комментариев в корректной программе на C++?

Для ответа на этот вопрос мы создадим вспомогательный класс

```
// Пример 33-2
//
class A {
public:
    void operator&&( int ) { }
    void operator<<( int ) { }
    void operator||( int ) { }
```

```
};
typedef void (A::*F)(int);
```

Теперь начнем с рассмотрения символа

а) &

Ответ: пять.

Ну, && — это просто; не сложно и &&&. Так что пойдем сразу дальше. Можем ли мы создать последовательность из четырех символов &&&&? Если да, то она должна интерпретироваться как && &&, но выражение наподобие a && && b синтаксически некорректно; мы не можем разместить два бинарных оператора один за другим.

Трюк заключается в том, что мы можем использовать вторую пару && как оператор, сделав первую пару && окончанием чего-то, что оператором не является. В таком случае требуется не так много времени, чтобы увидеть, что первая пара && может быть окончанием имени, а именно — имени функции, так что все, что нам нужно, — оператор operator&&(), который может получать указатель на некоторый другой оператор operator&&() в качестве первого параметра:

```
void operator&&( F, A ) { }
```

Это позволяет нам записать

```
&A::operator&&&&a;     // && &&
```

что означает

```
operator&&( &A::operator&&, a );
```

Это самая длинная последовательность из четного количества символов &, поскольку &&&&&& — некорректная запись. Почему? Потому что она означает && && &&, но даже делая первую пару && частью имени, мы не можем сделать последнюю пару && началом другого имени; остается только разместить два бинарных оператора && подряд, что невозможно.

Но не можем ли мы добавить еще один символ &, чтобы получить нечетное количество символов & в последовательности? Конечно, можем. Конкретно — &&&&& означает && && &; для первой части решение у нас уже есть, так что не надо долго думать, чтобы суметь прицепить к нему унарный оператор &.

```
&A::operator&&&&&a;     // && && &
```

что означает

```
operator&&( &A::operator&&, &a );
```

Теперь разберемся с оставшимися символами:

а) <

б) |

В обоих случаях ответ один — четыре.

Имея решение вопроса 2*а*, не сложно расправиться и с оставшимися двумя. Чтобы получить последовательность из четырех символов, воспользуемся тем же трюком, что и ранее, и определим

```
void operator<<( F, A ) { }
void operator||( F, A ) { }
```

Это позволит нам написать

```
&A::operator<<<<a;  // << <<
&A::operator||||a;  // || ||
```

что означает

```
operator<<( &A::operator<<, a );
operator||( &A::operator||, a );
```

Это самые длинные последовательности, которые мы можем получить, потому что `<<<<<<` и `||||||` должны быть некорректны исходя из тех же рассуждений, которые мы применяли к последовательности `&&&&&&`. Но в этот раз мы не можем добавить дополнительные символы `<` или `|`, чтобы получить последовательности из пяти символов, поскольку унарные операторы `<` и `|` не существуют.

Дополнительный вопрос

А сколько вопросительных знаков (?) вы сможете разместить подряд в строке программы на C++?

Подумайте немного над этим вопросом, прежде чем читать дальше. Этот вопрос сложнее, чем кажется на первый взгляд.

• • • • •

Есть ли у вас ответ?

Вы можете решить, что ответ — один, поскольку в C++ имеется только одна допустимая лексема, включающая вопросительный знак — тернарный оператор `?:`. Да, есть только одно средство в языке, которое использует вопросительный знак, но "есть много, друг Горацио, вещей в трансляторах и препроцессорах, которые и не снились синтаксическим правилам..." В частности, помимо прочего, C++ это еще и препроцессор, а не только язык.

Для символа ? правильный ответ — три. Например:

```
1???-0:0;
```

Этот вопрос труднее в первую очередь потому, что он нарушает правило максимального глотка. Три вопросительных знака `???` *не* интерпретируются как лексемы `??` и `?`. Почему? Потому что "`??-`" — это триграф, а триграфы обрабатываются до лексического разбора, и даже до обработки директив препроцессора. Если до этого вы не слышали о триграфах — не волнуйтесь, это всего лишь означает, что вы никогда не имели дело с экзотическими клавиатурами и не читали задачу 32. Триграф — это альтернативный способ ввода некоторых символов в исходном тексте (в частности, #, \, ^, [,], |, {, } и ~) на клавиатурах, которые не имеют клавиш для данных символов.

В нашем случае задолго до лексического анализа триграф `??-` заменяется символом ~, оператором побитового отрицания. Таким образом, исходный текст превращается в следующий:

```
1?~0:0;
```

который при лексическом анализе разбивается на лексемы

```
1 ? ~ 0 : 0 ;
```

и означает

```
1 ? (~0) : 0 ;
```

Резюме

Триграфы — средство, унаследованное из C и редко применяемое на практике, но оно оказалось принципиально полезным с политической точки зрения в процессе стандартизации (не спрашивайте, почему). Просто примите к сведению наличие такого редко используемого средства. Замечу, что в некоторых компиляторах поддержка триграфов по умолчанию выключена, и, как было сказано в одной документации по поводу опции, включающей поддержку триграфов, она "включает эту неудобную и редко используемую возможность стандарта C". К этому комментарию мало что можно добавить.

Изучение конкретных примеров

Копаться в чужом коде, наверное, некрасиво. Но зато очень интересно.

Этот завершающий раздел посвящен новой теме. Мы рассмотрим фрагменты реального опубликованного кода, обсудим его дизайн и стиль кодирования, его слабые и сильные стороны, и подумаем над тем, как разработать более совершенную версию. Можно только удивляться, как много можно сделать с кодом, который был написан, проверен и откорректирован экспертами.

Наслаждаясь чужими исходными текстами, не забывайте: то, что мы будем делать с чужим текстом, вполне применимо и к вашим программам.

Задача 34. Индексные таблицы — Сложность: 5

Индексные таблицы представляют собой полезную идиому, так что стоит поближе познакомиться с этой технологией. Но насколько эффективно мы сумеем ее реализовать?

Вопрос для новичка

1. Кто выигрывает от ясного, понятного исходного текста?

Вопрос для профессионала

2. Приведенный далее исходный текст представляет интересную и, несомненно, полезную идиому создания индексных таблиц над существующими контейнерами. Более детальную информацию вы можете почерпнуть из статьи [Hicks00].

   Оцените приведенный далее исходный текст и найдите:

   а) механические ошибки, такие как неверный синтаксис или непереносимые соглашения;

   б) стилистические улучшения, которые могут повысить ясность, степень повторного использования и сопровождаемости исходного текста.

```
// Программа sort_idxtbl(…) выполняет перестановку
// индексов в таблице

#include <vector>
#include <algorith>

template <class RAIter>

struct sort_idxtbl_pair
{
    RAIter it;
    int i;

    bool operator<( const sort_idxtbl_pair& s )
        { return (*it) < (*(s.it)); }

    void set( const RAIter& _it, int _i ) { it=_it; i=_i; }

    sort_idxtbl_pair() {}
};

template <class RAIter>
void sort_idxtbl( RAIter first, RAIter last, int* pidxtbl )
{
    int iDst = last-first;
    typedef std::vector< sort_idxtbl_pair<RAIter> > V;
    V v( iDst );
    int i=0;
    RAIter it = first;
    V::iterator vit = v.begin();
    for( i=0; it<last; it++, vit++, i++ )
        (*vit).set(it,i);

    std::sort(v.begin(), v.end());

    int *pi = pidxtbl;
    vit = v.begin();
    for( ; vit<v.end(); pi++, vit++ )
        *pi = (*vit).i;
}
```

```
main()
{
    int ai[10] = { 15,12,13,14,18,11,10,17,16,19 };

    cout << "#################" << endl;
    std::vector<int> vecai(ai, ai+10);
    int aidxtbl[10];
    sort_idxtbl(vecai.begin(), vecai.end(), aidxtbl);

    for (int i=0; i<10; i++)
    cout << "i=" << i
         << ", aidxtbl[i]=" << aidxtbl[i]
         << ", ai[aidxtbl[i]]=" << ai[aidxtbl[i]]
         << endl;
    cout << "#################" << endl;
}
```

Решение

Небольшая проповедь о ясности

1. Кто выигрывает от ясного, понятного исходного текста?

Если говорить коротко — практически все.

Для начала, с ясным исходным текстом проще выполнять отладку, и по этой причине менее вероятно появление в нем большого количества ошибок. Таким образом, написание ясного исходного текста в первую очередь сразу же облегчает вашу жизнь. Далее, когда вы возвращаетесь к коду через месяц или год, то становится гораздо проще вспомнить, что в нем к чему, и понять, как он работает. Большинство программистов знают, что удержать в голове все детали той или иной программы трудно даже несколько недель, в особенности если за это время переключиться на другую работу; после нескольких месяцев или даже лет легко решить, что этот код написан кем-то посторонним, который удивительным образом следовал вашему стилю.

Но хватит заботиться о себе. Будем альтруистами. Те, кому придется сопровождать ваш код, тоже выигрывают, если он будет ясным и удобочитаемым. В конце концов, для сопровождения кода надо очень хорошо в нем разобраться, понять, как он работает и как связаны между собой и с внешним окружением его части, какие побочные эффекты при его выполнении могут возникнуть. Не понимая полностью исходный текст, в него очень легко внести новые ошибки вместо исправления старых. В ясном и понятном коде легче разобраться, и внесение в него изменений становится менее рискованным делом, с меньшей вероятностью внесения ошибок или нежелательных побочных эффектов.

Однако самое важное — выигрыш по всем перечисленным причинам конечных пользователей, которые будут пользоваться программами с малым количеством ошибок и высоким уровнем сопровождения, при котором не привносятся новые ошибки взамен старых.

> ➢ **Рекомендация**
>
> По умолчанию в первую очередь должна быть обеспечена корректность и ясность исходного текста программ.

Разбор индексных таблиц

2. Приведенный далее исходный текст представляет интересную и, несомненно, полезную идиому создания индексных таблиц над существующими контейнерами. Более детальную информацию вы можете почерпнуть из статьи [Hicks00].

Оцените приведенный далее исходный текст и найдите:

a) **механические ошибки, такие как неверный синтаксис или непереносимые соглашения;**

b) **стилистические улучшения, которые могут повысить ясность, степень повторного использования и сопровождаемости исходного текста.**

Позвольте мне повториться и напомнить, что этот код содержит интересную и весьма полезную идиому. Мне часто требуется доступ к одному и тому же контейнеру разными способами, например, в различном порядке сортировки. По этой причине было бы неплохо иметь один основной контейнер, который хранит данные (например, `vector<Employee>`) и вторичные контейнеры *итераторов*, указывающих на элементы основного контейнера и реализующие различные способы доступа (например, `set<vector<Employee>::iterator,Funct>`, где `Funct` — функтор, который косвенно сравнивает объекты `Employee`, выполняя упорядочение объектов, отличающееся от того, в котором они физически хранятся в контейнере (в данном случае — в `vector`)).

Кроме сказанного, имеет значение и стиль. Автор приведенного фрагмента любезно позволил мне воспользоваться им в качестве учебного примера, и я не пытаюсь дразнить его здесь. Я только адаптирую методику, давно открытую таким светилом, как Плоджер (P. J. Plauger)[52], пытаясь дать вам рекомендации по кодированию на основе опубликованных исходных текстов. Я критиковал чужие программы и раньше, причем я уверен, что мои программы также подлежат критике.

А теперь, после этого вступления, давайте посмотрим, как можно улучшить представленный в условии задачи фрагмент.

Исправление механических ошибок

a) **механические ошибки, такие как неверный синтаксис или непереносимые соглашения**

Первая область, открытая для конструктивной критики, — это механические ошибки в исходном тексте, которые на большинстве платформ не позволят скомпилировать данный текст.

```
#include <algorith>
```

1. Правильно указывайте стандартные заголовочные файлы. Здесь заголовочный файл `<algorithm>` ошибочно назван `<algorith>`. Мое первое предположение было, что это, вероятно, артефакт файловой системы с 8-символьными именами, использовавшейся для тестирования первоначальной версии кода, но даже моя старая версия VC++ на старой версии Windows (с использованием файловой системы с именами 8.3) отказалась компилировать этот код. В любом случае, это имя не является стандартным, и даже при работе со старыми файловыми системами компилятор должен поддерживать стандартные длинные имена (даже если он затем преобразует их в более короткие имена файлов, или даже если физически файлы не используются вообще).

Теперь обратимся к строке

```
main()
```

2. Корректно определяйте функцию `main`. Приведенная здесь сигнатура функции `main` никогда не была стандартом C++ [C++98], хотя и является согласующимся со стандартом расширением (если компилятор выдает соответствующее предупреждение). Такая сигнатура была корректна в языке C до принятия стандарта 1999 года,

---

[52] Плоджер Ф., Керниган Б. *Элементы стиля программирования.* — М.: Радио и связь, 1984. — *Прим. ред.*

в котором действовало правило о неявном int, но ни в C++, ни в C99 [C99] она не верна. Из стандарта C++:

- §3.6.1/2: переносимый код должен определять main либо как int main(), либо как int main(int, char*[]).
- §7/7, сноска 78, и §7.1.5/2 сноска 80: неявный int запрещен.
- Приложение C (совместимость), комментарий к 7.1.5/4: явно указано, что "голое" объявление main() не корректно в C++, функция main должна определяться как int main().

➢ Рекомендация

Не полагайтесь на неявный int; стандартом C++ это не предусматривается. В частности, "void main()" или "main()" никогда не были стандартными в C++, хотя многие компиляторы поддерживают их в качестве расширений.

```
cout << "################" << endl;
```

3. Всегда включайте заголовки для тех типов, определения которых вам требуются. Программа использует cout и endl, но не включает #include <iostream>. Почему, тем не менее, программа работала в системе ее разработчика? Потому что стандартные заголовочные файлы C++ могут включать друг друга, но в отличие от C, C++ не определяет, какие стандартные заголовочные файлы включают другие стандартные заголовочные файлы и какие именно.

В нашем случае программа включает <vector> и <algorithm>, и, видимо, в исходной системе один из этих заголовочных файлов включал также <iostream> — непосредственно или опосредованно. Такая программа могла работать в системе у ее автора, может быть, она даже заработает и у меня, но это непереносимый и нездоровый стиль.

4. Следуйте рекомендациям задачи 39 из книги More Exceptional C++ *[Sutter02] (задача 10.10 русского издания) о пространствах имен.* Говоря о cout и endl, программа должна квалифицировать их с использованием std::, либо применить директивы using std::cout; и using std::endl;. К сожалению, это распространено среди программистов — забывать о квалификаторах области видимости пространства имен. Спешу заметить, что автор фрагмента корректно указал пространство имен vector и sort, что является хорошим стилем программирования.

Улучшение стиля

б) **стилистические улучшения, которые могут повысить ясность, степень повторного использования и сопровождаемости исходного текста.**

Помимо механических ошибок, есть еще ряд вещей, которые лично я реализовал бы иначе. Начнем с пары комментариев по поводу вспомогательной структуры.

```
template <class RAIter>
struct sort_idxtbl_pair
{
    RAIter it;
    int i;
    bool operator<( const sort_idxtbl_pair& s )
        { return (*it) < (*(s.it)); }
    void set( const RAIter& _it, int _i ) { it=_it; i=_i; }
    sort_idxtbl_pair () {}
};
```

5. Корректно и последовательно используйте квалификатор const. В частности, sort_idxtbl_pair::operator< не модифицирует *this, так что этот оператор следует объявить как константную функцию-член.

➢ Рекомендация

Последовательно и корректно используйте квалификатор `const`.

6. Удаление избыточного кода. В программе явным образом определяется конструктор класса `sort_idxtbl_pair` по умолчанию, хотя он не имеет никаких отличий от неявно генерируемой версии, так что особого смысла в нем нет. Кроме того, поскольку `sort_idxbl_pair` является структурой с открытыми данными, наличие отдельной операции `set`, которая вызывается только в одном месте, приводит только к усложнению кода, ничего не давая взамен.

➢ Рекомендация

Избегайте избыточности и повторения кода.

Теперь перейдем к основной функции `sort_idxtbl`.

```
template <class RAIter>
void sort_idxtbl( RAIter first, RAIter last, int* pidxtbl )
{
    int iDst = last-first;
    typedef std::vector< sort_idxtbl_pair<RAIter> > V;
    V v( iDst );
    int i=0;
    RAIter it = first;
    V::iterator vit = v.begin();
    for( i=0; it<last; it++, vit++, i++ )
        (*vit).set(it,i);
    std::sort(v.begin(), v.end());
    int *pi = pidxtbl;
    vit = v.begin();
    for( ; vit<v.end(); pi++, vit++ )
        *pi = (*vit).i;
}
```

7. Используйте значащие и соответствующие по смыслу имена. В данном случае имя `sort_idxtbl` вводит в заблуждение, поскольку функция не сортирует индексную таблицу, а создает ее! С другой стороны, имя параметра шаблона `RAIter` выбрано удачно, так как указывает на использование итератора с произвольным доступом (random-access iterator) — именно то, что и требуется в данном исходном тексте, так что такое имя служит хорошим напоминанием.

➢ Рекомендация

Используйте понятные и выразительные имена.

8. Будьте последовательны. В функции `sort_idxtbl` переменные иногда инициализируются в инструкции инициализации цикла, а иногда — нет, что затрудняет чтение кода — по крайней мере, для меня.

9. Избавляйтесь от неоправданной сложности. Эта функция просто обожает излишние локальные переменные! Есть три подтверждения этому. Во-первых, переменная `iDst`, инициализированная значением `last-first` и использованная только один раз. Почему бы просто не записать в месте ее использования разность `last-first` и избавиться от нее? Во-вторых, итератор вектора `vit` создается там, где уже имеется и может использоваться индекс (код от этого станет только понятнее). В-третьих, локальная переменная `it` инициализируется значением параметра функции, который после этого никогда не используется; лично я предпочитаю в такой ситуации воспользоваться параметром функции (даже если вы измените его — ничего страшного!) вместо введения другого имени.

10. Повторное использование, часть 1: интенсивнее используйте стандартную библиотеку. В приведенной программе вполне правильно используется функция `std::sort`, и это хорошо. Но что такое последний цикл, как не копирование, которое вполне можно выполнить при помощи функции `std::copy`? Зачем вводить специальный класс `sort_idxtbl_pair`, если его единственное отличие от `std::pair` в наличии функции сравнения? Помимо простоты, повторное использование стандартной библиотеки делает код более удобочитаемым. Будьте скромнее и используйте результаты чужого труда!

➢ Рекомендация

Следует хорошо знать и использовать в своем коде возможности стандартной библиотеки вместо написания собственного кода.

---

11. Повторное использование, часть 2: убить двух зайцев, повышая степень повторного использования своего кода. В исходном фрагменте ничто, кроме самой функции, не может быть использовано повторно. Вспомогательный класс `sort_idxtbl_pair` написан сугубо для использования в данной функции и не может быть повторно использован независимо от функции.

12. Повторное использование, часть 3: улучшение сигнатуры. Исходная сигнатура

```
template <class RAIter>
void sort_idxtbl( RAIter first, RAIter last, int* pidxtbl )
```

получает простой указатель `int*` на выходную область, чего я обычно стараюсь избегать в пользу более управляемого представления (например, `vector`). В конце концов, пользователь должен иметь возможность вызвать `sort_idxtbl` и поместить выходные данные в обычный массив, вектор или куда-то еще. Требование "иметь возможность поместить данные в произвольный контейнер" просто кричит о том, что надо воспользоваться итератором, правда? (См. также задачи 5 и 6.)

```
template< class RAIn, class Out >
void sort_idxtbl( RAIn first, RAIn last, Out result )
```

➢ Рекомендация

Избегайте жесткого указания типов там, где без этого можно обойтись. Это позволит расширить возможности повторного использования вашего кода.

---

13. Повторное использование, часть 4, или Сравнивайте итераторы при помощи !=. При сравнении итераторов всегда используйте оператор `!=` (который работает для всех типов итераторов) вместо `<` (который работает только для итераторов с произвольным доступом), конечно, кроме тех случаев, когда вы действительно нуждаетесь в использовании `<` и будете работать исключительно с итераторами с произвольным доступом. В приведенной программе для сравнения итераторов применен оператор `<`, что подразумевает использование итераторов с произвольным доступом, для которых программа изначально и предназначена: для создания индексов в векторах и массивах (которые оба поддерживают итераторы с произвольным доступом). Но нет причин, по которым мы бы не могли захотеть сделать то же с контейнерами другого вида, например, `list` или `set` (которые не поддерживают итераторы с произвольным доступом), и единственная причина, по которой исходная программа не может с ними работать, — это использование оператора `<` для сравнения итераторов вместо оператора `!=`.

Вот как по этому поводу пишет Скотт Мейерс в 32 разделе своей книги [Meyers96]: *Хорошее программное обеспечение приспособлено к изменениям. Оно в состоянии включать новые возможности, его можно перенести на новые платформы, оно допускает "подгонку" под новые требования, может работать с новыми входными данными.*

Такие гибкие, интеллектуальные и надежные программы не получаются случайно — это результат конструирования и реализации специалистами, которые, помня о нуждах дня сегодняшнего, не забывают о дне завтрашнем и его возможных требованиях к программе. Такие программы, приспособленные к видоизменению, создаются программистами, которые закладывают в свои программы будущее.

➢ Рекомендация

Предпочтительно сравнивать итераторы при помощи оператора !=, а не <.

---

14. Если только вам не требуется прежнее значение итератора, используйте префиксный инкремент (декремент). При работе с итераторами использование префиксной формы инкремента (++i), а не постфиксной (i++) должно стать привычкой; см. [Sutter00] (задача 1.6 русского издания). Конечно, существенного различия в приведенном фрагменте может и не быть, поскольку `vector<T>::iterator` может представлять собой простой указатель `T*` (хотя это может быть и не так), но если мы реализуем пункт 13, то код не будет ограничен использованием одного лишь типа `vector<T>::iterator`, а большинство итераторов других типов представляют собой классы. Возможно, их копирование не требует много процессорного времени — но зачем напрасно допускать даже такое малое снижение производительности программы?

➢ Рекомендация

Если только вам не требуется прежнее значение итератора, используйте префиксный инкремент.

На этом заканчивается наше рассмотрение наиболее важных вопросов, связанных с приведенным исходным текстом. Есть и другие замечания, но я не считаю их столь важными. Например, код промышленного уровня должен содержать комментарии, документирующие предназначение каждого класса и каждой функции и их семантику; однако это требование не имеет смысла в статье, в которой все проблемы описаны гораздо лучше простым "человеческим" языком и существенно детальнее, чем это можно сделать в комментариях.

Я сознательно не рассматривал стиль написания данного фрагмента, поскольку, в конце концов, его основным предназначением была всего лишь демонстрация читателям журнальной статьи принципа работы индексных таблиц.

Резюме

Давайте попытаемся сохранить базовый интерфейс исходного кода[53]. Ограничимся только коррекцией механических ошибок и стиля, и рассмотрим три альтернативные улучшенные версии приведенного в условии исходного текста. Каждая из них имеет свои преимущества, свои недостатки и свои предпочтения в стиле. Общее у всех трех версий то, что они более ясны и понятны, и содержат более переносимый код — так что при желании вы можете даже использовать их в своей собственной работе.

```
// Улучшенная версия кода из [Hicks00].
//
#include <vector>
#include <map>
#include <algorithm>
```

---

[53] Автор исходного текста сообщил об отзывах некоторых читателей, которые пошли по другому, но не менее элегантному пути — создавая контейнерообразный объект, который представляет собой обертку вокруг исходного контейнера, включая его итераторы, и обеспечивает итерирование с использованием различных способов сортировки.

```
// Решение 1 является несколько "подчищенной" версией кода,
// сохраняющей общую структуру исходного варианта. Код
// сократился с 23 до 17 строк (даже если считать "public:"
// и "private:" отдельными строками).
namespace Solution1 {
    template<class Iter>
    class sort_idxtbl_pair {
    public:
        void set(const Iter& it, int i) {it_ = it; i_ = i;}
        bool operator<(const sort_idxtbl_pair& other) const
            { return *it_ < *other.it_; }
        operator int() const { return i_; }
    private:
        Iter it_;
        int i_;
    };

    // Эта функция претерпела самые существенные изменения,
    // сократившись с 13 строк до 5. Для сравнения я
    // сохранил весь старый код. Старайтесь писать программы
    // так, чтобы в них не было излишней сложности
    //
    template<class IterIn, class IterOut>
    void sort_idxtbl(IterIn first, IterIn last, IterOut out){
        std::vector<sort_idxtbl_pair<IterIn> > v(last-first);
        // int iDst = last-first;
        // typedef std::vector< sort_idxtbl_pair<RAIter> > V;
        // V v(iDst);
        for( int i=0; i < last-first; ++i )
            v[i].set( first+i, i );
        // int i=0;
        // RAIter it = first;
        // V::iterator vit = v.begin();
        // for (i=0; it<last; it++, vit++, i++)
        // (*vit).set(it,i);
        std::sort( v.begin(), v.end() );
        // std::sort(v.begin(), v.end());
        std::copy( v.begin(), v.end(), out );
        // int *pi = pidxtbl;
        // vit = v.begin();
        // for (; vit<v.end(); pi++, vit++)
        // *pi = (*vit).i;
    }
}


// Решение 2 использует класс pair вместо вспомогательного
// класса. Код уменьшился до 15 строк (из них 2 -
// продолжение предыдущих из-за ширины страницы). 8 строк
// специфичны для данного кода, а 4 строки в принципе можно
// непосредственно использовать и в других контекстах.
//
namespace Solution2 {
    template<class T, class U>
    struct ComparePair1stDeref {
        bool operator()(const std::pair<T,U>& a,
                        const std::pair<T,U>& b ) const
            { return *a.first < *b.first; }
    };
    template<class IterIn, class IterOut>
    void sort_idxtbl(IterIn first, IterIn last, IterOut out){
        std::vector< std::pair<IterIn,int> > s( last-first );
        for( int i=0; i < s.size(); ++i )
            s[i] = std::make_pair( first+i, i );
```

```
        std::sort(s.begin(), s.end(),
                  ComparePair1stDeref<IterIn,int>());
        for( int i=0; i < s.size(); ++i, ++out )
            *out = s[i].second;
    }
}

// Решение 3 демонстрирует пару альтернативных деталей - в
// нем используется map для того, чтобы избежать отдельного
// этапа сортировки, и std::transform() вместо вручную
// написанного цикла. Код сократился до 16 строк и стал
// существенно более повторно используемым. Этой версии
// требуется больше памяти и, вероятно, ее
// производительность немного ниже, так что мне больше
// нравится решение 2. Этот код - пример поиска
// альтернативного решения задачи.
//
namespace Solution3 {
    template<class T>
    struct CompareDeref {
        bool operator()( const T& a, const T& b ) const
            { return *a < *b; }
    };
    template<class T, class U>
    struct Pair2nd {
        const U& operator()( const std::pair<T,U>& a ) const
            { return a.second; }
    };
    template<class IterIn, class IterOut>
    void sort_idxtbl(IterIn first, IterIn last, IterOut out){
        std::multimap<IterIn, int, CompareDeref<IterIn> > v;
        for( int i=0; first != last; ++i, ++first )
            v.insert( std::make_pair( first, i ) );
        std::transform(v.begin(), v.end(), out,
                       Pair2nd<IterIn const,int>());
    }
}

// Тестовая часть кода осталась по сути неизменной, за
// исключением использования итератора для получения
// результата (вместо int*) и использования исходного
// массива непосредственно в качестве контейнера.
//
#include <iostream>
int main() {
    int ai[10] = { 15,12,13,14,18,11,10,17,16,19 };
    std::cout << "#################" << std::endl;
    std::vector<int> aidxtbl( 10 );
    // Для тестирования другого решения используйте
    // соответствующее имя пространства имен
    Solution3::sort_idxtbl( ai, ai+10, aidxtbl.begin() );
    for( int i=0; i<10; ++i )
    std::cout << "i=" << i
        << ", aidxtbl[i]=" << aidxtbl[i]
        << ", ai[aidxtbl[i]]=" << ai[aidxtbl[i]]
        << std::endl;
    std::cout << "#################" << std::endl;
}
```

Задача 35. Обобщенные обратные вызовы Сложность: 5

Отчасти привлекательность обобщенного кода состоит в возможности его использования во всех ситуациях, которые только можно представить. Каким образом можно стилистически улучшить представленное в цитируемой статье средство, и каким образом можно сделать его более полезным, чтобы это был действительно обобщенный широко используемый код?

Вопрос для новичка

1. Какие качества целесообразны при разработке и написании обобщенных средств? Поясните свой ответ.

Вопрос для профессионала

2. Показанный далее код представляет интересную и очень полезную идиому для оболочек функций обратного вызова. Более детальное описание вы можете найти в оригинальной статье [Kalev01].
   Отрецензируйте предложенный код и определите:
   а) возможные стилистические улучшения дизайна для лучшего идиоматического использования C++;
   б) механические ограничения полезности данного средства.

```
template < class T, void (T::*F)() >
class callback
{
public:
    // Присваивание члену object
    callback(T& t) : object(t) {}
    // Запуск функции обратного вызова
    void execute() {(object.*F)();}
private:
    T& object;
};
```

Решение

Качества обобщенности

1. Какие качества целесообразны при разработке и написании обобщенных средств? Поясните свой ответ.

Обобщенный код прежде всего должен быть практичен и удобен в использовании. Это не значит, что он должен включать все возможные варианты использования, включая варку кофе и мытье посуды. Это всего лишь означает, что в отношении обобщенного кода следует стараться избегать как минимум трех вещей.

1. Избегайте чрезмерного ограничения типов. Например, если вы пишете обобщенный контейнер, то вполне разумно потребовать, чтобы содержащиеся в нем элементы имели, скажем, копирующий конструктор и не генерирующий исключений деструктор. Но надо ли требовать конструктор по умолчанию или оператор присваивания? Многие полезные типы, которые пользователи могут захотеть хранить в вашем контейнере, не имеют конструкторов по умолчанию, и если наш контейнер их использует, то такой тип не может быть использован в качестве типа элементов контейнера. Согласитесь, это не слишком обобщенно... (В качестве примера можно привести задачу 11 и ее контекст из [Sutter00] (задача 2.4 в русском издании).)

2. Избегайте чрезмерного ограничения функциональности. Если вы пишете некоторое программное средство, которое делает X и Y, то что, если некий пользователь захочет сделать Z, и это Z не слишком отличается от Y? Иногда вы будете хотеть сделать ваш код достаточно гибким для поддержки Z, иногда нет. Частью хорошего обобщенного дизайна является выбор путей и средств для настройки или расширения вашего кода. То, что это важно в обобщенном дизайне, не должно оказаться сюрпризом, поскольку тот же принцип применим и к дизайну класса в объектно-ориентированном программировании.

Дизайн, основанный на стратегии, представляет собой одну из нескольких важных методик, которые обеспечивают "подключаемое" поведение обобщенного кода. Примеры такого дизайна вы можете найти в нескольких главах в [Alexandrescu01]; начать можно с глав, где описываются `SmartPtr` и `Singleton`.

Это приводит нас к следующему вопросу.

3. Избегайте чрезмерно монолитного дизайна. Этот важный вопрос не возникает непосредственно при рассмотрении приведенного примера, но сам по себе заслуживает столь пристального внимания, что ему посвящена даже не одна задача, а целая миниссерия — задачи с 37 по 40.

Во всех трех пунктах рефреном звучит слово "чрезмерно". Это означает именно то, что я хотел сказать, — хорошее решение лежит между чрезмерно малой (синдром "я уверен, что никто не будет использовать этот код с чем-то, кроме `char`") и чрезмерно большой обобщенностью (нездоровые фантазии "А если кто-то захочет применить эту программу дисплея тостера на межпланетной станции?"). Правильное решение, как и истина, всегда где-то посредине.

Разбор обобщенных обратных вызовов

2. Показанный далее код представляет интересную и очень полезную идиому для оболочек функций обратного вызова. Более детальное описание вы можете найти в оригинальной статье [Kalev01].

Этот код представлен ниже.

```
template < class T, void (T::*F)() >
class callback
{
public:
    // Присваивание члену  object
    callback(T& t) : object(t) {}
    // Запуск функции обратного вызова
    void execute() {(object.*F)();}
private:
    T& object;
};
```

Итак, сколько способов ошибиться есть в простом классе со всего двумя однострочными функциями-членами? Как оказывается, чрезмерная простота и является частью проблемы. Этот шаблон класса не должен быть тяжеловесным, но и такая легковесность — это тоже перебор.

Улучшение стиля

Отрецензируйте предложенный код и определите:

а) **возможные стилистические улучшения дизайна для лучшего идиоматического использования C++.**

Сколько возможных улучшений вы обнаружили? Вот что имел в виду я.

4. Конструктор должен быть описан как `explicit`. Автор, вероятно, не намеревался обеспечить неявное преобразование `T` в `callback<T>`. "Правильные" классы не

позволяют себе создавать такие потенциальные проблемы для своих пользователей. Так что на самом деле нам нужно следующее.

```
// Присваивание члену object
explicit callback(T& t) : object(t) {}
```

То, что мы обнаружили в рассматриваемой строке, — вопрос стилистики, который сам по себе не является вопросом дизайна, но заслуживает отдельной рекомендации.

➢ Рекомендация

Предпочтительно описывать конструкторы как `explicit`, кроме тех случаев, когда вы действительно хотите обеспечить возможность преобразования типов.

(Мелочь). Комментарий не верен. Слово "присваивание" в комментарии не верно и вводит в заблуждение. Более точно в конструкторе мы "привязываем" ссылку к объекту типа `T`. Словом, будет лучше, если комментарий будет таким, как показано ниже

```
// Привязка к реальному объекту
explicit callback(T& t) : object(t) {}
```

Но в этом случае все, что говорит комментарий, сказано кодом, который предельно прост и в комментариях не нуждается, так что лучше оставить его и вовсе без комментария.

```
explicit callback(T& t) : object(t) {}
```

5. Функция `execute` должна быть `const`. В конце концов, функция `execute` никак не влияет на состояние объекта `callback<T>`! Это возвращает нас к азам — рекомендация о корректном использовании `const`, как вино, с возрастом становится только лучше. Значение корректного использования `const` известно в C и C++ как минимум с начала 1980-х годов, и это значение никуда не делось и в новом тысячелетии, и на него не влияет массовое использование шаблонов.

```
// Запуск функции обратного вызова
void execute() const {(object.*F)();}
```

➢ Рекомендация

Не забывайте о корректном использовании `const`.

Теперь, когда мы разобрались с функцией `execute`, перейдем к более серьезной идиоматической проблеме.

6. (Идиома). Функция `execute` должна быть оператором `operator()`. В C++ использование оператора вызова функции для выполнения операций в стиле функции идиоматично. Кстати, тогда комментарий, и так несколько излишний, становится совершенно ненужным и может быть удален. Код теперь идиоматически комментирует сам себя.

```
void operator()() const { (object.*F)(); }
```

"Но, — можете удивиться вы, — если мы обеспечиваем оператор вызова функции, значит, перед нами разновидность функционального объекта?" Отличный вопрос, который приводит нас к наблюдению, что обратный вызов также можно рассматривать как функциональный объект.

➢ Рекомендация

Идиоматические функциональные объекты следует обеспечивать оператором `operator()` вместо именованной функции вызова.

Ловушка: (Идиома). Должен ли данный обратный вызов быть производным от `std::unary_function`? См. раздел 36 в [Meyers01], где более подробно изложено обсуждение адаптируемости и почему в общем случае это Хорошая Вещь. Увы, в данном примере имеются две отличные причины не порождать обратный вызов от `std::unary_function`, как минимум, не сейчас.

- Это не унарная функция. У нее нет параметров, в то время, как у унарной функции — один параметр (`void` не в счет!).
- Порождение от `std::unary_function`, в любом случае, не улучшает расширяемость. Позже мы увидим, что обратный вызов должен работать и с другими видами сигнатур функций, и в зависимости от количества параметров может не оказаться соответствующего стандартного базового класса. Например, если мы поддерживаем функции обратного вызова с тремя параметрами, то стандартного класса `std::ternary_function`, от которого мы могли бы породить собственный класс, не существует.

Порождение от `std::unary_function` или `std::binary_function` — удобный способ снабдить обратный вызов небольшим количеством важных определений `typedef`, которые могут использоваться различными программными средствами, такими как замыкания, но это имеет значение, только если вы используете их как настоящие функциональные объекты. Вряд ли это окажется необходимо в силу природы обратного вызова и его предназначения. (Если в будущем дело обернется так, что обратные вызовы должны будут использоваться таким образом с одним или двумя параметрами, то соответствующие версии можно будет породить от `std::unary_function` и `std::binary_function`.)

Исправление механических ошибок и ограничений

б) **механические ограничения полезности данного средства**

7. Рассмотрим возможность передачи функции обратного вызова в качестве обычного (не шаблонного) параметра. Параметры шаблонов, не являющиеся типами, редко дают столь значительные преимущества, чтобы фиксировать их во время компиляции. То есть мы можем изменить исходный текст следующим образом.

```
template < class T >
class callback {
public:
    typedef void (T::*Func)();
    // Связывание с реальным объектом
    callback(T& t, Func func) : object(t), f(func) {}
    // Выполнение функции обратного вызова
    void operator()() const { (object.*f)(); }
private:
    T& object;
    Func f;
};
```

Теперь используемая функция может изменяться в процессе выполнения программы; к коду легко добавить функцию-член, которая позволит пользователю изменить функцию, с которой связан существующий объект `callback`, что было невозможно в предыдущей версии кода.

➢ Рекомендация

Предпочтительно использовать параметры, не являющиеся типами, в качестве обычных параметров функций, за исключением случаев, когда они действительно должны быть параметрами шаблонов.

---

8. Обеспечение контейнеризации. Если программе нужен один объект обратного вызова для последующего использования, то, скорее всего, ей их понадобится несколько. А если у кого-то возникнет желание поместить такие объекты в контейнер, например, `vector` или `list`? Сейчас это невозможно, поскольку эти объекты нельзя присваивать — они не поддерживают оператор `operator=`. Почему? Потому что они содержат ссылку, которая, будучи связана с объектом в конструкторе, в дальнейшем не может быть связана с каким-то другим объектом.

Указатели не имеют такой "привязанности" к объектам и могут указывать на все, что вы только пожелаете. В рассматриваемом нами случае использование указателя вместо ссылки совершенно безопасно и не мешает компилятору сгенерировать конструктор по умолчанию и оператор копирующего присваивания.

```
template < class T >
class callback {
public:
    typedef void (T::*Func)();
    // Связывание с реальным объектом
    callback(T& t, Func func) : object(&t), f(func) {}
    // Выполнение функции обратного вызова
    void operator()() const { (object->*f)(); }
private:
    T* object;
    Func f;
};
```

Теперь можно написать, например,

```
list< callback< Widget > > lcw(w, &Widget::SomeFunc);
```

➢ Рекомендация

Будет гораздо лучше, если разрабатываемые вами объекты будут совместимы с контейнерами. В частности, чтобы быть помещенным в стандартный контейнер, объект должен поддерживать присваивание.

"Минутку, — можете удивиться вы, — если у меня может быть список такого вида, то почему я не могу иметь список объектов обратного вызова произвольных типов, чтобы я мог запомнить их все и вызывать по мере необходимости?" Почему бы и нет, если вы добавите базовый класс.

9. Разрешение полиморфизма: обеспечение общего базового класса для объектов обратного вызова. Если мы хотим позволить пользователю иметь `list<callbackbase*>` (или лучше `list<shared_ptr<callbackbase> >`), то мы можем сделать это, просто обеспечив наличие базового класса, который по умолчанию ничего не делает в своем операторе `operator()`.

```
class callbackbase {
public:
    virtual void operator()() const { };
    virtual ~callbackbase() = 0;
};

callbackbase::~callbackbase() { }

template < class T >
class callback: public callbackbase {
public:
    typedef void (T::*Func)();
    // Связывание с реальным объектом
    callback(T& t, Func func) : object(&t), f(func) {}
    // Выполнение функции обратного вызова
    void operator()() const { (object->*f)(); }
```

```
private:
    T* object;
    Func f;
};
```

Теперь любой, кто захочет, может иметь `list<callbackbase*>` и полиморфно вызывать оператор `operator()` элементов этого списка. Конечно, использование `list<shared_ptr<callback> >` еще предпочтительнее; см. [Sutter02b].

Заметим, что добавление базового класса — компромисс, хотя и очень небольшой: мы вносим накладные расходы, связанные с дополнительной косвенностью, а именно — вызов виртуальной функции при запуске обратного вызова посредством базового интерфейса. Однако эти накладные расходы включаются в игру только при использовании базового интерфейса; кода, которому базовый интерфейс не требуется, эти расходы не касаются.

➢ Рекомендация

Рассмотрите возможность использования полиморфизма, с тем чтобы различные инстанцирования вашего шаблона класса могли использоваться взаимозаменяемо (если это имеет смысл для вашего шаблона класса). Если это так, то можно легко добиться полиморфизма, обеспечив наличие базового класса, совместно используемого всеми инстанцированиями шаблона класса.

---

10. (Идиома, компромисс). Можно обеспечить вспомогательную функцию `make_callback` для облегчения вывода типов. Сейчас пользователь должен явно указывать аргументы шаблона для временных объектов.

```
list< callback< Widget > > l;
l.push_back( callback<Widget>( w, &Widget::SomeFunc ) );
```

Но зачем писать `Widget` дважды? Неужели компилятору это и так непонятно? Да, непонятно, но вы можете помочь ему в контексте, когда нужны только временные объекты наподобие рассматриваемого. Вы можете разработать вспомогательную функцию.

```
list< callback< Widget > > l;
l.push_back( make_callback( w, &Widget::SomeFunc ) );
```

Эта функция `make_callback` работает так же, как стандартная `std::make_pair`. Это должна быть шаблонная функция, поскольку компилятор выводит типы только для этого вида шаблонов. Вот как должна выглядеть эта функция.

```
template<typename T >
callback<T> make_callback( T& t, void (T::*f) () ) {
    return callback<T>( t, f );
}
```

11. (Компромисс). Добавление поддержки других сигнатур обратного вызова. Больше всего работы я оставил напоследок. Перефразируя, можно сказать: "Есть много, друг Горацио, на свете сигнатур, которые и не снились нашей `void (T::*F)()`!"

➢ Рекомендация

Избегайте ограниченности ваших шаблонов; избегайте жесткого кодирования конкретных или менее общих типов.

Если нам гарантированно достаточно только одной сигнатуры для функций обратного вызова, то не стоит преумножать сущности сверх необходимости и на этом можно остановиться. Но если мы хотим обеспечить возможность обратного вызова

функций с разными сигнатурами, то нам придется существенно усложнить наш исходный текст.

Я не хочу приводить здесь весь исходный текст, так как он достаточно скучный. (Если вы действительно хотите познакомиться с этим утомительно повторяющимся кодом, или если у вас бессонница — возьмите книгу [Alexandrescu01], там вы найдете подобные примеры.) Я же только вкратце приведу несколько замечаний по этому поводу.

Во-первых, следует подумать о константных функциях-членах. Простейший способ работы с ними — обеспечить параллельный обратный вызов, который использует соответствующую `const`-сигнатуру, и в этой версии не забывать получать и хранить `T` по ссылке или указателю на `const`.

Во-вторых, надо вспомнить о различных возвращаемых типах. Простейший способ обеспечить варьируемый возвращаемый тип — это добавить еще один параметр шаблона.

В-третьих, функции обратного вызова могут иметь параметры. Вновь добавляем параметры шаблона, не забываем о наборе операторов `operator()` с этими параметрами, а также посолить, поперчить и хорошенько перемешать это варево... Не забудьте добавить по новому шаблону для обработки каждого потенциального количества аргументов обратного вызова.

Увы, рост кода приобретает просто взрывной характер, и вам надо подумать и вовремя остановиться, искусственно ограничив количество поддерживаемых параметров функций обратного вызова. Возможно, в будущем стандарте C++0x мы и найдем что-то наподобие возможности работы с шаблонами с переменным числом параметров, но пока что это — пустые мечты.

Резюме

Обобщая все сказанное и добавив микроулучшения наподобие последовательного использования ключевого слова `typename` и соглашений об именовании, мы получим окончательную версию кода.

```
class CallbackBase {
public:
    virtual void operator()() const { };
    virtual ~CallbackBase() = 0;
};

CallbackBase::~CallbackBase() { }

template<typename T>
class Callback : public CallbackBase {
public:
    typedef void (T::*F)();

    Callback( T& t, F f )    : t_(&t), f_(f) { }
    void operator()() const    { (t_->*f_)(); }

private:
    T* t_;
    F  f_;
};

template<typename T>
Callback<T> make_callback( T& t, void (T::*f) () ) {
    return Callback<T>( t, f );
}
```

Задача 36. Объединения Сложность: 4

Речь в данной задаче пойдет не о профсоюзах или партиях, а об объединениях в C++, их сильных и слабых сторонах и о правилах использования конструируемых объектов в качестве членов объединений.

Вопрос для новичка

1. Что такое объединения и для чего они предназначены?
2. Какие типы не могут использоваться в качестве членов объединений? Почему существуют такие ограничения? Поясните свой ответ.

Вопрос для профессионала

3. В статье [Manley02] рассматривается написание языка сценариев, в котором используются объединения. Допустим, что вы хотите, чтобы ваш язык поддерживал единый тип для переменных, которые в разные моменты времени могут хранить целые числа, строки или списки. Создание `union {int i; list<int> l; string s;};` не работает по причинам, которые разбираются в вопросах 1 и 2. Приведенный далее код представляет собой обходной путь, который пытается обеспечить поддержку произвольного типа в качестве участника объединения (более подробную информацию по этому вопросу вы сможете найти в указанной статье.)

   Отрецензируйте предложенный код и найдите:

   а) механические ошибки, такие как неверный синтаксис или непереносимые соглашения;

   б) стилистические улучшения, которые могут повысить ясность исходного текста, его повторное использование и сопровождение.

```
#include <list>
#include <string>
#include <iostream>
using namespace std;

#define max(a,b) (a)>(b)?(a):(b)

typedef list<int> LIST;
typedef string STRING;

struct MYUNION {
    MYUNION() : currtype( NONE ) {}
    ~MYUNION() {cleanup();}

    enum uniontype {NONE,_INT,_LIST,_STRING};
    uniontype currtype;

    inline int& getint();
    inline LIST& getlist();
    inline STRING& getstring();

protected:
    union {
        int i;
        unsigned char buff[max(sizeof(LIST),sizeof(STRING))];
    } U;

    void cleanup();
};
```

```
inline int& MYUNION::getint()
{
    if( currtype==_INT ) {
        return U.i;
    } else {
        cleanup();
        currtype=_INT;
        return U.i;
    } // else
}

inline LIST& MYUNION::getlist()
{
    if( currtype==_LIST ) {
        return *(reinterpret_cast<LIST*>(U.buff));
    } else {
        cleanup();
        LIST* ptype = new(U.buff) LIST();
        currtype=_LIST;
        return *ptype;
    } // else
}

inline STRING& MYUNION::getstring()
{
    if( currtype==_STRING) {
        return *(reinterpret_cast<STRING*>(U.buff));
    } else {
        cleanup();
        STRING* ptype = new(U.buff) STRING();
        currtype=_STRING;
        return *ptype;
    } // else
}

void MYUNION::cleanup()
{
    switch( currtype ) {
        case _LIST: {
            LIST& ptype = getlist();
            ptype.~LIST();
            break;
        } // case
        case _STRING: {
            STRING& ptype = getstring();
            ptype.~STRING();
            break;
        } // case
        default: break;
    } // switch
    currtype=NONE;
}
```

4. Покажите лучший способ получения обобщенного вариантного типа, и расскажите, с какими сложностями вам пришлось столкнуться.

Решение

Основные сведения

1. Что такое объединения и для чего они предназначены?

Объединения позволяют нескольким объектам, как классам, так и встроенным типам, занимать одно и то же место в памяти. Например:

```
// Пример 36-1
//
union U {
    int i;
    float f;
};

U u;
u.i = 42;                       // Активно поле i
std::cout << u.i << std::endl;
u.f = 3.14f;                    // Активно поле f
std::cout << 2 * u.f << std::endl;
```

Однако в каждый момент времени может быть "активен" только один тип — в конце концов, память может хранить одновременно только одно значение. Кроме того, объединения поддерживают только некоторые виды типов, что приводит нас к следующему вопросу.

2. Какие типы не могут использоваться в качестве членов объединений? Почему существуют такие ограничения? Поясните свой ответ.

Из стандарта C++:

Объект класса с нетривиальным конструктором, нетривиальным копирующим конструктором, нетривиальным деструктором или нетривиальным оператором копирующего присваивания, или массив объектов такого класса, не может быть членом объединения.

Коротко говоря, чтобы класс мог использоваться в объединении, он должен удовлетворять следующим критериям.

- Конструкторы, деструкторы и операторы копирующего присваивания должны генерироваться компилятором.
- Класс не имеет виртуальных функций или виртуальных базовых классов.
- То же должно быть справедливо для всех его базовых классов и нестатических членов.

Вот и все, но это ограничивает применение огромного количества типов.

Объединения унаследованы от C. Традиции языка C включают эффективность и поддержку низкоуровневого, приближенного к аппаратному обеспечению программирования, и эти традиции сохранены и в C++ — вот почему в C++ имеются объединения. С другой стороны, у C нет никаких традиций объектной модели с поддержкой классов с конструкторами и деструкторами и пользовательским копированием, что определенно имеется в C++. Вот почему в C++ использование таких новых типов со старыми объединениями, если таковое имеет место, не должно нарушать объектную модель C++, включая гарантии времени жизни объектов.

Если бы в C++ не было указанных ограничений, могли бы происходить Ужасные Вещи. Рассмотрим, например, что могло бы произойти, если бы был разрешен следующий код.

```
// Пример 36-2: Этот код не соответствует стандарту C++, но
//              что произошло бы, если бы он был разрешен?
//
void f() {
    union IllegalImmoralAndFattening {
        std::string s;
        std::auto_ptr<int> p;
    };
    IllegalImmoralAndFattening iiaf;
    iiaf.s = "Hello, world";  // Вызывать ли конструктор s?
    iiaf.p = new int(4);      // Вызывать ли конструктор p?
```

```
}   // Будет ли объект s уничтожен? Должен ли он
    // уничтожаться? А объект p?
```

Как указывают приведенные комментарии, если бы такой код был разрешен, серьезных проблем не удалось бы избежать. Чтобы избежать ненужного усложнения языка, проблематичные операции были просто запрещены.

Но не думайте, что объединения — просто рудимент. Наиболее полезны объединения, пожалуй, для экономии памяти, поскольку позволяют данным перекрываться, и это остается немалым преимуществом C++ даже сегодня. Например, некоторые из наиболее продвинутых реализаций стандартной библиотеки C++ используют их для "оптимизации малых строк", которая позволяет повторно использовать память внутри самого объекта. Вот основная идея этой оптимизации: в случае строк большого размера объект хранит обычный указатель на динамически выделенную память и дополнительную информацию, такую как, например, размер буфера. Однако эту память, использующуюся для указателя и дополнительной информации, в случае малых строк можно использовать непосредственно для хранения строки, избегая таким образом динамического распределения памяти. Дополнительную информацию об оптимизации малых строк (и других способах оптимизации) можно найти в [Sutter02] (задачи 7.2–7.5 в русском издании); см. также обсуждение современных коммерческих реализаций `std::string` в [Meyers01].

Построение объединений

3. В статье [Manley02] рассматривается написание языка сценариев, в котором используются объединения. Допустим, что вы хотите, чтобы ваш язык поддерживал единый тип для переменных, которые в разные моменты времени могут хранить целые числа, строки или списки. Создание `union {int i; list<int> l; string s;};` не работает по причинам, которые разбираются в вопросах 1 и 2. Приведенный далее код представляет собой обходной путь, который пытается обеспечить поддержку произвольного типа в качестве участника объединения (более подробную информацию по этому вопросу вы сможете найти в указанной статье.)

С одной стороны, в цитированной статье автором успешно решена реальная проблема. К сожалению, с кодом дела обстоят не вполне благополучно. Проблемы, связанные с исходным текстом в статье, можно разбить на три основные категории: законность, безопасность и мораль.

Отрецензируйте предложенный код и найдите:

а) **механические ошибки, такие как неверный синтаксис или непереносимые соглашения;**

б) **стилистические улучшения, которые могут повысить ясность исходного текста, его повторное использование и сопровождение.**

Первый комментарий, который следует сделать по поводу приведенного фрагмента, заключается в том, что основная идея, лежащая в основе данного решения, не законна с точки зрения стандарта C++. Вот как описана основная идея в статье:

Идея заключается в том, что вместо объявления членов объекта мы объявляем просто буфер [не динамически, а в виде массива типа `char` в объекте, который должен выступать в роли объединения] и создаем необходимые объекты на лету [путем размещающего конструирования]. — [Manley02]

Идея распространенная, но, к сожалению, нездоровая.

Выделение буфера одного типа с последующим использованием его для размещения объекта другого типа не соответствует стандарту и не переносимо, поскольку для буфера, который не выделен динамически (т.е. не выделен при помощи функции `malloc` или оператора `new`), не гарантируется корректное выравнивание для любого другого типа, кроме изначально объявленного. Даже если эта методика случайно и

сработает для некоторых типов с данным компилятором, нет никакой гарантии, что этот метод будет продолжать работать для других типов или с теми же типами в другой версии компилятора. Более подробно этот вопрос освещен в [Sutter00] (задача 4.5 в русском издании), в особенности обратите внимание на врезку "Безответственная оптимизация и ее вред". Кроме того, вопросы выравнивания рассматриваются также в [Alexandrescu02].

При работе над стандартом C++0x комитет по стандартизации языка рассматривает возможность добавления средств для управления выравниванием в стандарт, в частности, чтобы обеспечить возможность использования методик, подобных описанной, которые опираются на выравнивание, но пока что это вопрос будущего. Пока же, чтобы этот способ работал более-менее надежно хотя бы некоторое время, надо сделать что-то из перечисленного:

- воспользоваться хакерским приемом `max_align` (см. примечание в статье [Manley02] или поищите `max_align` при помощи Google);
- воспользоваться нестандартными расширениями наподобие `__alignof__` из Gnu C++, для того чтобы код надежно работал с компилятором, поддерживающим такое расширение. (Хотя Gnu и предоставляет макрос `ALIGNOF`, предназначенный для надежной работы на других компиляторах, он также является хакерской уловкой.)

Обойти эти неприятности можно путем динамического выделения массива при помощи функции `malloc` или оператора `new`, которые гарантируют, что буфер `char` будет выровнен таким образом, что в нем может быть размещен объект любого типа, но это тоже не самая лучшая идея, поскольку такие действия небезопасны с точки зрения типов, а кроме того, это приводит к снижению эффективности — а именно вопросы эффективности и были основной мотивацией описанной в статье разработки. Альтернативным корректным решением могло бы быть использование `boost::any` (см. ниже), которое хотя и приводит к аналогичному снижению эффективности из-за динамического выделения памяти и косвенного обращения, но, по крайней мере, безопасно и корректно.

Попытки действовать против правил языка или заставить его работать не так, как он должен, а как того хочется нам, очень сомнительны и должны быть окружены красными флажками. В упомянутой врезке из книги [Sutter00] я писал, что всякие "необычности" до добра не доводят. Да, возможны ситуации, когда вполне разумно использовать некоторую непереносимую конструкцию, которая гарантированно работоспособна в данной конкретной среде (в нашем случае, вероятно, можно использовать хак `max_align`), но даже в этом случае следует явно указать нестандартность решения и не использовать его в коде, рекомендованном для широкого использования.

Разбор кода

Давайте теперь поближе познакомимся с кодом.

```
#include <list>
#include <string>
#include <iostream>
using namespace std;
```

Всегда включайте все необходимые заголовочные файлы. Поскольку ниже используется `new`, следует также включить `#include <new>`. (Примечание: заголовочный файл `<iostream>` включен совершенно правильно, так как в исходном авторском тексте выполнялась проверка работоспособности (не вошедшая в данную книгу) разработанного кода с использованием потоков ввода-вывода.)

```
#define max(a,b) (a)>(b)?(a):(b)
```

```
typedef list<int> LIST;
typedef string STRING;

struct MYUNION {
  MYUNION() : currtype( NONE ) {}
  ~MYUNION() {cleanup();}
```

Первая классическая механическая ошибка — MYUNION небезопасно копировать, поскольку программист позабыл предоставить копирующий конструктор и оператор копирующего присваивания.

MYUNION разработан таким образом, что в его конструкторе и деструкторе выполняются некоторые специальные действия, так что данные функции приходится писать самому (генерируемые компилятором функции не подходят). Это корректно, однако недостаточно, поскольку аналогичные действия должны выполняться и в копирующем конструкторе и операторе копирующего присваивания, которые автор явно *не* предоставил. Это плохо, поскольку операции копирования, сгенерированные компилятором по умолчанию, будут работать неправильно, а именно — они просто побитово скопируют содержимое массива символов, что, скорее всего, приведет к неудовлетворительным результатам, в большинстве случаев — просто к порче содержимого памяти. Рассмотрим следующий код.

```
// Пример 36-3: копировать MYUNION небезопасно
//

{
  MYUNION u1, u2;
  u1.getstring() = "Hello, world";
  u2 = u1;    // Побитовое копирование u1 в u2
}             // Неприятности: двойное удаление одной и той
              // же строки (если даже считать, что побитовое
              // копирование имеет смысл)
```

➢ Рекомендация

Не забывайте о правиле Большой Тройки [Cline99]: если классу требуется пользовательский копирующий конструктор, оператор копирующего присваивания или деструктор, то, скорее всего, все они нужны ему одновременно.

Заканчивая на этом с механическими ошибками, перейдем к паре классических стилистических ошибок.

```
enum uniontype {NONE,_INT,_LIST,_STRING};
uniontype currtype;
inline int& getint();
inline LIST& getlist();
inline STRING& getstring();
```

В этом фрагменте две стилистические ошибки. Во-первых, разрабатываемая структура лишена возможности повторного использования из-за жестко закодированных конкретных типов. На самом деле в исходной статье рекомендуется выполнять такое кодирования всякий раз, когда это потребуется. Во-вторых, даже при такой преднамеренно ограниченной полезности код не слишком хорошо поддается расширению или сопровождению. Мы вернемся к этому вопросу чуть позже.

➢ Рекомендация

Избегайте жесткого кодирования информации, что без нужды делает код более хрупким и ограничивает его гибкость.

Имеются также две механические проблемы. Первая заключается в том, что член currtype открыт без существенных на то причин; это нарушает принцип инкапсуляции и означает, что любой пользователь может внести путаницу во флаги, пусть даже ненамеренно. Вторая механическая проблема связана с именами, использованными в перечислении; об этом я скажу позже, в отдельном подразделе.

```
protected:
```

Здесь мы сталкиваемся со второй механической ошибкой: внутреннее устройство класса должно быть закрытым, а не защищенным. Единственная причина, по которой оно может быть защищенным, — это необходимость обеспечить доступ со стороны производных классов. Что касается данного случая, то лучше не иметь никаких производных классов, так как наследовать MYUNION небезопасно по целому ряду причин — хотя бы из-за странных игр с внутренним устройством класса и отсутствия виртуального деструктора.

➢ Рекомендация

Всегда делайте все члены-данные закрытыми. Единственным исключением является случай структур в стиле C, которые не предназначены для инкапсулирования и все члены которых являются открытыми.

```
union {
    int i;
    unsigned char buff[max(sizeof(LIST),sizeof(STRING))];
  } U;
  void cleanup();
};
```

На этом оканчивается определение главного класса. Пойдем дальше и рассмотрим три параллельные функции доступа.

```
inline int& MYUNION::getint()
{
    if( currtype==_INT ) {
        return U.i;
    } else {
        cleanup();
        currtype=_INT;
        return U.i;
    } // else
}

inline LIST& MYUNION::getlist()
{
    if( currtype==_LIST ) {
        return *(reinterpret_cast<LIST*>(U.buff));
    } else {
        cleanup();
        LIST* ptype = new(U.buff) LIST();
        currtype=_LIST;
        return *ptype;
    } // else
}

inline STRING& MYUNION::getstring()
{
    if( currtype==_STRING) {
        return *(reinterpret_cast<STRING*>(U.buff));
    } else {
        cleanup();
        STRING* ptype = new(U.buff) STRING();
```

```
        currtype=_STRING;
        return *ptype;
    } // else
}
```

Маленькое замечание: комментарий `// else` ничего не дает и совершенно бесполезен.

➢ Рекомендация

Пишите (только) полезные комментарии. Никогда не пишите комментарии, которые повторяют код. Комментарии должны пояснять код и причины, по которым вы написали его именно так.

Но здесь еще есть три более серьезные проблемы. Первая заключается в том, что функции не написаны симметрично, и в то время как первое использование списка или строки дает объект, построенный при помощи конструктора по умолчанию, первое использование `int` дает нам неинициализированный объект. Если это сделано преднамеренно, чтобы отразить обычную семантику неинициализируемых переменных типа `int`, то это следовало документировать; в противном случае `int` также должен быть инициализирован. Например, если вызывающая функция обращается к `getint` и пытается сделать копию (неинициализированного) значения, то в результате мы получаем неопределенное поведение — не все платформы поддерживают копирование произвольных некорректных значений `int` и могут отвергнуть такие инструкции в процессе работы программы.

Вторая серьезная проблема состоит в том, что данный код не является корректным с точки зрения использования `const`. Корректный исходный текст, как минимум, содержал бы `const`-перегрузки для каждой из рассматриваемых функций; все они, естественно, возвращают то же значение, что и их неконстантные аналоги, но как ссылки на `const`.

➢ Рекомендация

Последовательно и корректно используйте модификатор `const`.

Третья проблема заключается в хрупкости такого подхода при необходимости внесения изменений. Он основан на переключении типов, и поэтому очень легко случайно рассинхронизировать функции при удалении или добавлении нового типа.

Теперь остановитесь и задумайтесь: как бы вы поступили при необходимости добавить новый тип? Перечислите по возможности полно все необходимые для этого действия.

Вы готовы? Давайте сравним наши результаты. Чтобы добавить новый тип, вы должны помнить о том, что:

- надо добавить новое значение в перечисление;
- надо добавить новую функцию доступа;
- надо обновить функцию `cleanup` для удаления нового типа; и
- надо добавить этот тип в вычисление размера буфера, чтобы он был достаточно большим для хранения всех типов, включая новый.

Если вы ошибетесь и пропустите что-то — что ж, это будет отличной иллюстрацией того, насколько сложно поддерживать и расширять такой исходный текст.

Перейдем к заключительной части.

```
void MYUNION::cleanup()
{
    switch( currtype ) {
        case _LIST: {
            LIST& ptype = getlist();
```

```
            ptype.~LIST();
            break;
        } // case
        case _STRING: {
            STRING& ptype = getstring();
            ptype.~STRING();
            break;
        } // case
        default: break;
    } // switch
    currtype=NONE;
}
```

Здесь опять можно высказать уже знакомое замечание по поводу комментариев — `// case` и `// switch` не несут смысловой нагрузки. Лучше не иметь комментариев вовсе, чем такие комментарии, которые только отвлекают внимание.

Но здесь есть и более серьезный вопрос: вместо простого `default: break;` было бы лучше использовать полный список типов (включая `int`) и сигнализировать о логической ошибке в случае неизвестного типа — вероятно, посредством `assert` или `throw std::logic_error(...);`.

И вообще, переключение типов является злом само по себе. Поиск "*switch C++ Dewhurst*" в Google дает массу ссылок на эту тему, включая [Dewhurst02]. Если вас интересует более детальная информация по этому вопросу, чтобы убедить коллег не использовать эту методику, — обратитесь к найденным ссылкам.

➢ Рекомендация

Избегайте переключения типов; предпочитайте безопасность с точки зрения использования типов.

Эти хитрые имена

Есть еще одна механическая проблема, которую я еще не раскрыл в полном объеме. Впервые она проявляет себя во всей красе (вернее, уродстве) в строке

```
enum uniontype {NONE,_INT,_LIST,_STRING};
```

Никогда, никогда, вы слышите? — никогда! — не используйте имена, которые начинаются с подчеркивания или содержат двойное подчеркивание; эти имена зарезервированы для исключительного использования вашим компилятором и разработчиками стандартной библиотеки, чтобы избежать столкновения с такими же именами в вашем коде. Не трогайте их имен, и они не будут трогать ваши![54]

Не останавливайтесь! Читайте дальше! Вы могли встретить этот совет раньше. Вы могли даже услышать его от меня или прочесть в моих книгах. Это могло настолько вам надоесть, что, зевнув, вы решили перейти к следующему подразделу. Так вот, это не просто теоретические измышления!

Строка с определением `enum` компилируется большинством компиляторов, которыми я компилировал данную программу: Borland 5.5, Comeau 4.3.0.1, gcc 2.95.3 / 3.1.1 / 3.4, Intel 7.0 и Microsoft Visual C++ от 6.0 по 8.0 (2005) beta. Но два из них — Metrowerks CodeWarrior 8.2 и EDG 3.0.1 со стандартной библиотекой Dinkumware 4.0 — не смогли с ней справиться.

---

[54] Если быть более точным, то правило гласит, что любое имя с двойным подчеркиванием в любом месте, такое как `like__this`, или начинающееся с подчеркивания и прописной буквы наподобие `_LikeThis`, является зарезервированным. Если хотите — запомните это правило, но, на мой взгляд, проще просто полностью избегать двойных подчеркиваний и имен, начинающихся с подчеркивания.

Metrowerks CodeWarrior 8 останавливался, сообщив о 52 ошибках (это не опечатка). 225 строк (это тоже не опечатка) сообщений об ошибках начинались со следующей диагностики, прямо указывающей на одну из запятых.

```
### mwcc Compiler:
#    File: 36.cpp
# --------------
#      17:      enum uniontype {NONE,_INT,_LIST,_STRING};
#   Error:                                      ^
#   identifier expected
### mwcc Compiler:
#      18:      uniontype currtype;
#   Error:      ^^^^^^^^
#   declaration syntax error
```

После этого следуют 52 сообщения об ошибках на еще 215 строках. Понятно, что вторую и дальнейшие ошибки можно просто игнорировать, так как эта лавина вызвана первой ошибкой — поскольку тип `uniontype` не был успешно определен, остаток текста, в котором он интенсивно используется, не может оказаться корректным с точки зрения компилятора.

Но что же случилось с определением `uniontype`? Указанная запятая вроде бы находится на положенном месте? Перед ней находится нормальный идентификатор, все, как положено... Все становится понятным, если попросить компилятор Metrowerks вывести исходный текст после обработки препроцессором. Пропустив много-много строк, мы встретим следующую, которая и поступает на вход компилятора.

```
enum uniontype {NONE,_INT, , };
```

Понятно, что это не корректный код C++, и компилятор совершенно справедливо жалуется на третью запятую, поскольку перед ней нет никакого идентификатора.

Но что же произошло с `_LIST` и `_STRING`? Можно предположить, что ими перекусил голодный зверь по кличке Препроцессор. Это произошло просто потому, что реализация Metrowerks определяет макрос, который при обработке препроцессором удалил имена `_LIST` и `_STRING`, что вполне законно, поскольку стандарт позволяет этой реализации определять собственные имена `_Names` — как и `other__names`.

Итак, компилятор Metrowerks просто съел как `_LIST`, так и `_STRING`. Это объясняет первую часть чудес. А как насчет второй части — реализации EDG's/Dinkumware? Судите сами.

```
"1.cpp", line 17: error: trailing comma is nonstandard
      enum uniontype {NONE,_INT,_LIST,_STRING};
                                      ^
"1.cpp", line 58: error: expected an expression
      if( currtype==_STRING) {
                    ^
"1.cpp", line 63: error: expected an expression
          currtype=_STRING;
                   ^
"1.cpp", line 76: error: expected an expression
          case _STRING: {
                 ^
4 errors detected in the compilation of "36.cpp".
```

Что произошло в этот раз, можно понять даже без генерации исходного текста препроцессором и его проверки. Компилятор ведет себя так, как если бы слово `_STRING` просто отсутствовало. Это связано с тем, что, как не сложно догадаться, оно тоже съедено все еще голодным Препроцессором.

Я надеюсь, что эти примеры убедили вас, что использовать в своих программах `_Имена наподобие__этого` не стоит и что данная проблема вовсе не так надуманна, как может показаться на первый взгляд. Как видите, это сугубо практическая проблема, поскольку ограничения на использование имен непосредственно влияют на ваши

взаимоотношения с авторами компилятора и стандартной библиотеки. Не лезьте на их территорию, и вы останетесь невредимы.

C++ достаточно открытый язык, который позволяет вам писать код любой степени сложности и гибкости, используя любые имена вне пространства имен `std`. Но все же, когда речь идет об именах, то у C++ в этом отношении есть запретная зона, опутанная колючей проволокой и обставленная таблицами с надписями "`Вход__ воспрещен. _Предъявите _Пропуск`". Нарушителей — в зависимости от степени их невезения — могут ждать крупные неприятности.

> ➢ **Рекомендация**
>
> Никогда не используйте имена определенного вида, а именно — начинающиеся с подчеркивания и прописной буквы или содержащие двойное подчеркивание. Такие имена зарезервированы для использования компиляторами и реализациями стандартной библиотеки.

Использование boost::any

4. **Покажите лучший способ получения обобщенного вариантного типа, и расскажите, с какими сложностями вам пришлось столкнуться.**

В рассматриваемой статье сказано:

Вы можете захотеть реализовать язык сценариев с единым типом переменных, которые могут быть целыми числами, строками или списками. — [Manley02]

Все правильно и не вызывает никаких разногласий. Однако дальше сказано следующее:

Объединение идеально подходит для реализации такого составного типа. — [Manley02]

Вместо этого статья служит демонстрацией того, почему объединения вообще не подходят для такой цели.

Но если не объединения, то что же тогда следует использовать? Одним очень хорошим кандидатом для реализации такого вариантного типа является средство any из [Boost], вместе с `many` и `any_cast`[55]. Интересно, что полная реализация обобщенного any (работающего с любым количеством и любыми комбинациями типов, и даже с рядом платформозависимых `#ifdef`) имеет примерно такой же размер исходного текста, как и реализация `MYUNION` для рассмотренного частного случая трех типов `int`, `list<int>` и `string`, но при этом any представляет собой средство общего назначения, не зависящее от типов, расширяемое, безопасное с точки зрения типов, политкорректное и не содержащее холестерина.

Естественно, и тут не обошлось без компромисса, который в данном случае представляет собой динамическое выделение памяти. Средство `boost::any` не пытается повысить эффективность за счет отказа от динамического распределения памяти, что было одним из исходных положений рассматриваемой статьи. Заметим также, что накладные расходы на динамическое распределение памяти в `boost::any` оказываются большими, чем в случае использования динамического выделения памяти для буфера в рассматриваемом нами фрагменте исходного текста. Это связано с тем, что в случае `MYUNION` динамическое выделение памяти выполнялось бы однократно в течение жизни объекта, в то время как `boost::any` выполняет динамическое выделение памяти всякий раз при изменении типа своего содержимого.

Вот как должна выглядеть демонстрационная часть исходного текста из статьи при использовании `boost::any` вместо `MYUNION` (оригинальный код статьи приведен в качестве комментария).

[55] Более детальное рассмотрение этого вопроса имеется в [Hyslop01].

```
any u;      // Вместо: MYUNION u;
```

Вместо самодельной структуры, которая должна разрабатываться отдельно для каждого конкретного случая, здесь использован класс any. Заметим, что это обычный класс, а не шаблон.

```
// Обращение к объединению как к int
u = 12345;        // Вместо: u.getint() = 12345;
```

Это присваивание any имеет более естественный синтаксис.

```
cout << "int="
     << any_cast<int>(u) << endl; // или просто int(u)
     // Вместо: cout << "int=" << u.getint() << endl;
```

Преобразование типа any мне представляется предпочтительным в силу его большей обобщенности (включая то, что не используется синтаксис доступа к члену класса) и большего соответствия естественному стилю C++; можно также воспользоваться более кратким int(u), без any_cast, если тип вам известен. Кроме того, функции MYUNION get*type* менее надежны, сложны в написании, сопровождении и т.д.

```
// Обращение к объединению как к std::list
u = list<int>();
list<int>& l = *any_cast<list<int> >(&u);
                  // Вместо: LIST& list = u.getlist();
l.push_back(5);   // Аналогично: list.push_back(5);
l.push_back(10);  // Аналогично: list.push_back(10);
l.push_back(15);  // Аналогично: list.push_back(15);
```

Я думаю, что any_cast можно усовершенствовать, с тем чтобы ссылку можно было получить более простым способом. (Кстати: мне не нравится использование list в качестве имени переменной, когда в области видимости оказывается шаблон с тем же именем; это дает слишком много возможностей для неоднозначности.)

Итак, мы достигли большей удобочитаемости и типизируемости. Остальные отличия не так уж велики.

```
list<int>::iterator it = l.begin();
            // Вместо: LIST::iterator it = list.begin();
while( it != l.end() ) {
    cout << "list item=" << *(it) << endl;
    it++;
} // while
```

Почти так же, как и в оригинале. Продолжим сравнение.

```
// Обращение к объединению как к std::string
u = string("Hello world!");
            // Вместо: STRING& str = u.getstring();
            // str = "Hello world!";
```

Опять версия с any немного проще исходной, но всего лишь немного.

```
cout << "string='"
     << any_cast<string>(u)    // или просто "string(u)"
     << "'" << endl;
     // Вместо: cout<<"string='"<<str.c_str()<<"'"<<endl;
```

Размеченные объединения Александреску

Нельзя ли достичь обеих поставленных целей — безопасности и отказа от динамического распределения памяти — при полном соответствии стандарту? Это похоже на проблемы, которые так любит Андрей Александреску (Andrei Alexandrescu), особенно если при этом можно написать шаблоны посложнее. Как видно из [Alexandrescu02], где он описал свои подход к объединениям (Variant), это возможно, причем с использованием шаблонов (думали ли вы, что объединения могут быть шаблонами?), и это сделано.

Коротко говоря, героические усилия Александреску по втискиванию своего шаблона `variant` в узкие рамки стандарта привели к очень близкому к полностью переносимому решению. У него только один слабый момент, причем на практике достаточно переносимый, несмотря на его выход за рамки гарантий стандарта. Главная его проблема (даже если опустить вопросы, связанные с выравниванием) в том, что код `variant` настолько сложен и продвинут, что будет работать только на очень небольшом количестве компиляторов; по крайней мере, лично мне удалось скомпилировать его код только на одном компиляторе.

Ключевая часть подхода Александреску заключается в попытке обобщить идею `max_align`, чтобы сделать ее библиотечным средством, которое соответствует стандарту C++. Это делается, в частности, для того, чтобы иметь возможность решать проблемы относительно безопасного использования нединамического символьного буфера. Александреску приложил героические усилия по использованию метапрограммирования с использованием шаблонов для вычисления безопасного выравнивания. Будет ли переносимым его решение? Вот что он пишет по этому поводу:

> *...приведенная выше реализация не является 100% переносимой для всех типов. Теоретически возможна реализация компилятора, который соответствует стандарту, но будет некорректно работать с размеченными объединениями. Это связано с тем, что стандарт не гарантирует, что все пользовательские типы будут иметь такое же выравнивание, как один из обычных старых типов (POD-типов). Однако вряд ли такой компилятор появится на практике, а не только в воспаленном воображении знатоков стандарта.*
>
> *[...] Переносимое вычисление выравнивания — дело трудное, но выполнимое. Но оно не может быть на 100% переносимым.* — [Alexandrescu02]

У подхода Александреску есть и другие ключевые возможности, в частности, шаблон объединения, который получает в качестве параметра шаблон `typelist` со списком типов, объекты которых могут в нем содержаться, поддержка инспектирования, обеспечивающая расширяемость, и методика реализации, состоящая в "подделке vtable", что позволяет повысить эффективность за счет отказа от лишнего уровня косвенности при обращении к содержащемуся в объединении типу. Все эти части существенно более тяжеловесны по сравнению с `boost::any`, но теоретически переносимы. Примечание "теоретически" существенно, поскольку, например, в той же книге [Alexandrescu01] часто встречаются комментарии в шаблонах наподобие "Гарантированно вызовет внутреннюю ошибку компилятора у [список различных распространенных компиляторов]", и имеется даже закомментированный текст с пометкой "Эта конструкция не будет работать ни на одном из компиляторов".

В этом и заключается главная слабость `variant`: большинство реальных компиляторов и близко не подошли к тому уровню, чтобы скомпилировать этот код, который в результате представляет собой всего лишь экспериментальную разработку, пусть и чрезвычайно важную. Я пытался собрать код Александреску с помощью разных компиляторов: Borland 5.5; Comeau 4.3.0.1; EDG 3.0.1; gcc 2.95, 3.1.1, и 3.2; Intel 7.0; Metrowerks 8.2; Microsoft VC++ 6.0, 7.0 (2002), и 7.1 (2003). Ряд компиляторов в этом списке очень строго соответствует стандарту, но, тем не менее, ни один из них не смог успешно скомпилировать предложенный им код.

Я попытался "причесать" исходный текст Александреску так, чтобы его можно было скомпилировать хотя бы одним из компиляторов, но добился успеха только с VC++ 7.1 (2003). Большинство компиляторов просто не способны на такой "подвиг", поскольку они обладают недостаточно строгой поддержкой шаблонов, чтобы работать с кодом Александреску. (Интересно отметить, что некоторые из компиляторов оказываются ужасно многословными — так, Intel 7.0 в ответ на компиляцию `main.cpp` разразился отчетом об ошибках, объем которого оказался равен почти полумегабайту — 430 Кбайт диагностических сообщений.)

Я внес три изменения в код, чтобы скомпилировать его без ошибок (хотя не смог избежать ряда предупреждений о сужающих преобразованиях) компилятором Microsoft VC++ 7.1 (2003):

- добавил отсутствующее слово `typename` в класс `AlignedPOD`;
- добавил отсутствующую конструкцию `this->`, чтобы в `ConverterTo<>::Unit<>::DoVisit()` сделать имя зависимым;
- добавил символы новой строки в конце ряда заголовочных файлов, как того требует стандарт C++ (некоторые компиляторы недостаточно строги в этом отношении и допускают отсутствие таких символов в качестве расширения; VC++ строго относится к этому требованию)[56].

Вот как прокомментировал автор [Manley02] компромиссы дизайна Александреску: *Этот способ не использует динамическую память, а также избегает проблем, связанных с выравниванием и переключением типов. К сожалению, у меня нет компилятора, который бы мог скомпилировать этот код, так что я не могу сравнить его производительность с вариантами, использующими myunion и any. Подход Александреску требует наличия 9 заголовочных файлов общим объемом около 80 Кбайт, что влечет за собой множество проблем поддержки* (К. Мэнли, частное письмо).

Все это так.

Я не намерен резюмировать здесь три статьи Андрея, но если вам они интересны — то они доступны в Web и указаны в списке литературы.

➢ Рекомендация

Если вам надо использовать вариантный тип, то лучше всего воспользоваться для этого классом `boost::any` (или чем-то в той же степени простым).

Когда используемый вами компилятор станет поддерживать шаблоны в достаточном объеме, а в стандарт включат поддержку выравнивания, наступит время рассмотреть в качестве возможной замены объединений шаблон наподобие `Variant`.

Резюме

Хотя дизайн и реализация `MYUNION` полны недостатков, сама проблема вполне реальна и стоит того, чтобы ею заняться. Я хотел бы поблагодарить К. Мэнли за то, что он нашел время и написал свою статью, привлекшую внимание к проблеме вариантных типов, а также Кевлину Хенни (Kevlin Henney) и Андрею Александреску за их вклад в решение данной проблемы. Это очень сложная проблема, и подходы Мэнли и Александреску оказываются не строго переносимы, хотя `Variant` Александреску приближается к этому идеалу — он переносим настолько, что на практике его не способен скомпилировать ни один из распространенных компиляторов.

В настоящий момент наиболее предпочтительным оказывается использование такого класса, как `boost::any`. Если в конкретных местах измерения указывают, что вам действительно требуется повышенная эффективность или дополнительные возможности, предоставляемые классом наподобие `Variant` Александреску, и у вас есть время и знания, то вы можете поэкспериментировать в написании собственных версий `Variant` путем применения только тех идей из [Alexandrescu02], которые вы сможете объяснить компилятору.

---

[56] Я благодарен коллеге Джеффу Пейлу (Jeff Peil) за его замечание об этом требовании [C++03] § 2.1/1, которое гласит: "Если непустой исходный файл не завершается символом новой строки или завершается символом новой строки, непосредственно перед которым идет символ обратной косой черты, поведение является неопределенным".

Задача 37. Ослабленная монолитность. Часть 1: взгляд на `std::string` Сложность: 3

Я решил завершить раздел, посвященный изучению конкретных примеров, минисерией, где рассматривается часть стандартной библиотеки, а именно — `std::string`. Мы начнем наше рассмотрение с обзора важных правил, с учетом которых разрабатывался стандартный класс `string`.

Вопрос для новичка

1. Что такое монолитный класс и чем он плох? Поясните свой ответ.

Вопрос для профессионала

2. Перечислите поименно все функции-члены `std::basic_string`.

Решение

Избегайте чрезмерно монолитных конструкций

1. Что такое монолитный класс и чем он плох? Поясните свой ответ.

Слово "монолитный" используется для описания программного обеспечения, которое представляет собой единое, неделимое большое целое наподобие монолита. Слово "монолит" происходит от слов "моно" (один) и "лит" (камень); оно вызывает в памяти образ огромного булыжника, который наглядно демонстрирует массивность и неделимость такого кода.

Единая тяжеловесная программа, которая предназначена "для всего", — зачастую просто тупик, в который зашли ее создатели. В конечном счете часто оказывается, что чем сложнее приложение, тем уже область его применения. В частности, таким монолитом может оказаться класс, в функции-члены которого затолкали всю мыслимую функциональность, несмотря на то, что всю ее можно с тем же успехом реализовать в виде функций, не являющихся ни членами, ни друзьями этого класса. У такого подхода как минимум два недостатка.

- (Главный). *Потенциально независимая функциональность оказывается локализованной в одном классе.* Интересующая нас операция могла бы использоваться и с другими типами, но из-за жесткой закодированности в конкретном классе это оказывается невозможно (в то время как если бы эта операция была реализована как шаблон функции, не являющийся членом класса, она могла бы использоваться существенно более широко).
- (Второстепенный). *Этот подход может препятствовать расширению класса с использованием новой функциональности.* "Минутку! — может возразить кто-то из читателей. — Не имеет значения, реализован ли существующий интерфейс класса при помощи функций-членов или не членов, так как я в любом случае могу расширить его при помощи моих собственных функций-не членов". Технически это так, но если вся функциональность класса достигается посредством функций-членов, т.е. реализована как естественная идиома класса, то расширение при помощи функций-не членов противоречит используемой идиоме и всегда остается решением второго сорта. То, что класс предоставляет свою функциональность в виде функций-членов класса, является семантическим указанием для его пользователей, которые привыкнут к такому стилю его использования, но который невозможно будет использовать при его расширениях.

Практичнее по возможности разбивать обобщенные компоненты на части.

> **Рекомендация**
>
> Пользуйтесь идиомой "один класс (или функция) — одна задача".
>
> Где это возможно — предпочтительно использовать функции, которые не являются членами и друзьями.

Остальная часть данной задачи просто дает нам дополнительное подтверждение приведенной рекомендации.

Класс `string`

2. Перечислите поименно все функции-члены `std::basic_string`.

Это действительно длинный список...

С учетом конструкторов в классе `string` не менее *103* функций-членов. Честно! Если это не монолит, то тогда я уж и не знаю, как он должен выглядеть...

Представьте себе подземную пещеру с озером, в котором имеется подводный проход в следующую пещеру. Приготовьтесь погрузиться в эту черную воду и проплыть сквозь этот проход...

Вдохните поглубже и задержите дыхание на время чтения ISO/IEC 14882:2003(E)[57].

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
  class Allocator = allocator<charT> >
  class basic_string {
    // : некоторые typedef :
    explicit basic_string(const Allocator& a = Allocator());
    basic_string(const basic_string& str, size_type pos = 0,
                 size_type n = npos,
                 const Allocator& a = Allocator());
    basic_string(const charT* s, size_type n,
                 const Allocator& a = Allocator());
    basic_string(const charT* s,
                 const Allocator& a = Allocator());
    basic_string(size_type n, charT c,
                 const Allocator& a = Allocator());
    template<class InputIterator>
      basic_string(InputIterator begin, InputIterator end,
                   const Allocator& a = Allocator());
    ~basic_string();
    basic_string& operator=(const basic_string& str);
    basic_string& operator=(const charT* s);
    basic_string& operator=(charT c);
    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type length() const;
    size_type max_size() const;
    void resize(size_type n, charT c);
    void resize(size_type n);
    size_type capacity() const;
```

[57] Толстый том, известный также как стандарт ISO C++ [C++03].

```
void reserve(size_type res_arg = 0);
void clear();
bool empty() const;
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
const_reference at(size_type n) const;
reference at(size_type n);
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str,
                     size_type pos, size_type n);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template<class InputIterator>
  basic_string& append(InputIterator first,
                       InputIterator last);
void push_back(const charT);
basic_string& assign(const basic_string&);
basic_string& assign(const basic_string& str,
                     size_type pos, size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
template<class InputIterator>
  basic_string& assign(InputIterator first,
                       InputIterator last);
 basic_string& insert(size_type pos1,
                      const basic_string& str);
basic_string& insert(size_type pos1,
                     const basic_string& str,
                     size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s,
                     size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n,
                     charT c);
iterator insert(iterator p, charT c);
void insert(iterator p, size_type n, charT c);
template<class InputIterator>
  void insert(iterator p, InputIterator first,
              InputIterator last);
```

Кажется, по пути есть маленький карман с воздухом. Не всплывайте — это только полпути! Сделайте очередной вдох — и:

```
basic_string& erase(size_type pos = 0,
                    size_type n = npos);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
basic_string& replace(size_type pos1, size_type n1,
                      const basic_string& str);
basic_string& replace(size_type pos1, size_type n1,
                      const basic_string& str,
                      size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1,
                      const charT* s, size_type n2);
basic_string& replace(size_type pos, size_type n1,
                      const charT* s);
basic_string& replace(size_type pos, size_type n1,
                      size_type n2, charT c);
basic_string& replace(iterator i1, iterator i2,
                      const basic_string& str);
basic_string& replace(iterator i1, iterator i2,
```

```
                        const charT* s, size_type n);
basic_string& replace(iterator i1, iterator i2,
                        const charT* s);
basic_string& replace(iterator i1, iterator i2,
                        size_type n, charT c);
template<class InputIterator>
  basic_string& replace(iterator i1, iterator i2,
                          InputIterator j1,
                          InputIterator j2);
size_type copy(charT* s, size_type n,
               size_type pos = 0) const;
void swap(basic_string<charT,traits,Allocator>&);
const charT* c_str() const;        //  explicit
const charT* data() const;
allocator_type get_allocator() const;
size_type find (const basic_string& str,
                size_type pos = 0) const;
size_type find (const charT* s, size_type pos,
                size_type n) const;
size_type find (const charT* s,
                size_type pos = 0) const;
size_type find (charT c, size_type pos = 0) const;
size_type rfind(const basic_string& str,
                size_type pos = npos) const;
size_type rfind(const charT* s, size_type pos,
                size_type n) const;
size_type rfind(const charT* s,
                size_type pos = npos) const;
size_type rfind(charT c, size_type pos = npos) const;
size_type find_first_of(const basic_string& str,
                        size_type pos = 0) const;
size_type find_first_of(const charT* s, size_type pos,
                        size_type n) const;
size_type find_first_of(const charT* s,
                        size_type pos = 0) const;
size_type find_first_of(charT c,
                        size_type pos = 0) const;
size_type find_last_of (const basic_string& str,
                        size_type pos = npos) const;
size_type find_last_of (const charT* s, size_type pos,
                        size_type n) const;
size_type find_last_of (const charT* s,
                        size_type pos = npos) const;
size_type find_last_of (charT c,
                        size_type pos = npos) const;
size_type find_first_not_of(const basic_string& str,
                            size_type pos = 0) const;
size_type find_first_not_of(const charT* s,
                            size_type pos,
                            size_type n) const;
size_type find_first_not_of(const charT* s,
                            size_type pos = 0) const;
size_type find_first_not_of(charT c,
                            size_type pos = 0) const;
size_type find_last_not_of (const basic_string& str,
                            size_type pos = npos) const;
size_type find_last_not_of (const charT* s,
                            size_type pos,
                            size_type n) const;
size_type find_last_not_of (const charT* s,
                            size_type pos = npos) const;
size_type find_last_not_of (charT c,
                            size_type pos = npos) const;
basic_string substr(size_type pos = 0,
                    size_type n = npos) const;
```

```
    int compare(const basic_string& str) const;
    int compare(size_type pos1, size_type n1,
                const basic_string& str) const;
    int compare(size_type pos1, size_type n1,
                const basic_string& str,
                size_type pos2, size_type n2) const;
    int compare(const charT* s) const;
    int compare(size_type pos1, size_type n1,
                const charT* s, size_type n2 = npos) const;
  };
}
```

Фуу... *103* функций-членов и их шаблонов! Наконец мы можем вынырнуть и вздохнуть полной грудью. Но на этом наши подземные приключения не заканчиваются, и минисерия продолжается.

Резюме

На практике следует разбивать обобщенные компоненты на части.

- Пользуйтесь идиомой "один класс (или функция) — одна задача".
- Где это возможно — предпочтительно использовать функции, которые не являются членами и друзьями.

Стоило ли перечислять здесь все функции-члены `string`? Да, поскольку эта информация еще пригодится нам в следующей задаче.

Задача 38. Ослабленная монолитность. Часть 2: разбор `std::string` Сложность: 5

"Один за всех и все за одного!" — этот девиз годится для мушкетеров, но не вполне пригоден для разработчиков классов. Здесь приведен пример, может, и не совсем типичный, который иллюстрирует, как легко заблудиться, чрезмерно увлекшись конструированием. Самое печальное в том, что этот пример взят из стандартной библиотеки.

Вопрос для новичка

1. Какие из функций-членов `std::string` *обязаны* быть членами и почему?

Вопрос для профессионала

2. Каким из функций-членов `std::string` *следует* быть членами и почему?
3. Покажите, почему функции-члены `std::string` `at`, `clear`, `empty` и `length` могут быть реализованы как не члены и не друзья без потери обобщенности, удобства использования и без влияния на остальной интерфейс `std::string`.

Решение

Членство — быть или не быть

Вспомним совет из последней задачи: там, где это имеет смысл с практической точки зрения, следует разбивать обобщенные компоненты на отдельные части. Пользуйтесь идиомой "один класс (или функция) — одна задача". Где это возможно — предпочтительно использовать функции, которые не являются членами и друзьями.

Например, если вы пишете класс `string` и сделаете поиск, проверку соответствия шаблону и лексический анализ доступными в виде функций-членов, то эта функциональность окажется жестко привязанной к данному классу и не сможет быть использована с последовательностями других типов. Средство, которое служит для той же цели, но состоит из нескольких частей, которые можно использовать независимо друг от друга, зачастую представляет собой лучший дизайн. Применительно к нашему примеру, это означает, что лучше отделить алгоритмы от контейнеров, что в большинстве случаев и сделано в STL.

И я (в [Sutter00] (раздел 5 русского издания)), и Скотт Мейерс (Scott Meyers) (в [Meyers00]) уже писали ранее о том, почему некоторые функции-не члены являются вполне законной частью интерфейса типа и почему следует предпочитать функции, не являющиеся ни членами, ни друзьями, — помимо прочего, это способствует инкапсуляции. Например, вот как пишет Скотт во вступлении к рассматриваемой статье:

> *Я начну с главного: если вы пишете функцию, которая может быть реализована либо как член, либо как не друг и не член, то ее надо реализовать именно в виде функции, не являющейся членом. Такое решение увеличивает инкапсуляцию класса. Когда вы думаете об инкапсуляции, вы должны думать о функциях-не членах.* — [Meyers00]

Так что когда мы рассматриваем функции, которые будут работать с `basic_string` (или любым другим типом класса), мы хотим сделать их функциями-не членами и не друзьями, если это возможно в рамках разумного. Следовательно, имеется несколько вопросов по поводу функций-членов `basic_string`.

- *Всегда делайте их членами, если они обязаны быть таковыми.* Какие из операций обязаны быть членами — в соответствии с требованиями языка C++ (например, конструкторы) или по функциональным причинам (например, они

должны быть виртуальными)? Если функции обязаны быть членами, то они, конечно, так и должны быть реализованы.

- *Если функциям требуется доступ ко внутренним данным, лучше сделать их членами.* Какие операции, требующие доступа ко внутренним данным, должны получить его посредством дружбы? Обычно их следует делать функциями-членами класса. Отметим, что имеются редкие исключения, такие как операции, требующие преобразования типов левых аргументов, и операторы наподобие <<, сигнатуры которых не позволяют ссылке `*this` быть их первыми параметрами. Несмотря на то, что они могут быть реализованы как функции, не являющиеся друзьями, с использованием функций-членов (возможно, виртуальных), в результате зачастую получается настолько "кривое" решение, что лучше и естественнее использовать отношение дружбы.
- *Во всех остальных случаях лучше использовать функции-не члены и не друзья.* Операции, которые могут нормально работать, будучи функциями-не членами и не друзьями, должны быть реализованы именно таким образом.

Пару слов об эффективности. Для каждой функции следует рассмотреть вопрос, можно ли реализовать ее как функцию-не член и не являющуюся другом так же эффективно, как и в виде функции-члена. Но не является ли это той самой преждевременной оптимизацией, которая не приводит ни к чему хорошему? Нет, ни в коей мере. Основная причина, по которой я рассматриваю вопросы эффективности, — это желание продемонстрировать, какое большое количество функций-членов `basic_string` можно не хуже реализовать как обычные функции, не являющиеся друзьями класса. В частности, я хочу отбросить все обвинения в том, что такой подход приводит к потенциальной потере эффективности, например, лишая возможности выполнения операции за время, которое от нее требует стандарт. Вторая причина заключается в том, что преждевременность оптимизации библиотеки общего назначения отличается от таковой для прикладной программы. Все необходимые измерения производительности программы можно выполнить на завершающей стадии разработки, в то время как при разработке библиотеки мало что известно о том, как и где она будет использоваться, так что лучше делать все операции максимально эффективными, там, где это не вступает в противоречие с интерфейсом и не вызывает ненужных усложнений. Заметим также, что разработчики библиотек зачастую имеют конкретную информацию из прошлого опыта (например, отчеты пользователей или собственный опыт в предметной области применения библиотеки) о том, какие операции используются в критичных ко времени выполнения фрагментах программ, так что оптимизация выполняется не вслепую и не может считаться преждевременной.

Так что не волнуйтесь о преждевременной оптимизации — здесь нет никакой ловушки. Мы всего лишь хотим выяснить, сколько членов `basic_string` можно ничуть не хуже реализовать в виде обычных функций, не являющихся друзьями. Даже если вы и готовы к тому, что многие из них реализуемы как обычные функции, полученные результаты все равно могут вас удивить.

Операции, которые обязаны быть членами

1. Какие из функций-членов `std::string` *обязаны* быть членами и почему?

На "первом проходе" выделим те операции, которые обязаны быть членами класса. В начале этого списка располагаются совершенно очевидные

- конструкторы (6),
- деструктор,
- операторы присваивания (3),
- операторы `[]` (2).

Совершенно ясно, что эти функции обязаны быть членами — ведь просто невозможно каким-либо иным способом написать конструктор, деструктор, оператор присваивания или оператор `[]`!

Итак, с 12 функциями из общего списка мы определились, осталось 91...

Операции, которые следует сделать членами

2. Каким из функций-членов `std::string` *следует* быть членами и почему?

Каким операциям требуется доступ ко внутренним данным, так что если делать их не членами, то необходимо будет сделать их друзьями? Это достаточная причина, чтобы реализовать такие операции как члены класса. Их список приведен далее. Некоторые из перечисленных операций обеспечивают косвенный доступ (например, `begin`) или изменение (например, `reserve`) внутреннего состояния строки:

- `begin` (2)
- `end` (2)
- `rbegin` (2)
- `rend` (2)
- `size`
- `max_size`
- `capacity`
- `reserve`
- `swap`
- `c_str`
- `data`
- `get_allocator`

Эти функции должны быть членами не только потому, что они тесно связаны с типом `basic_string`, но еще и потому что они образуют открытый интерфейс, используемый обычными функциями, не являющимися друзьями класса. Конечно, эти функции можно реализовать и как друзья класса, но зачем? (В действительности есть одна причина, по которой вы можете предпочесть написать также и функции, не являющиеся ни членами, ни друзьями, которые просто вызывают соответствующие члены, а именно — для единообразия интерфейса; мы коснемся этого вопроса немного позже.)

Я бы добавил в этот список в качестве фундаментальных следующие строковые операции:

- `insert` (1 — версия с тремя параметрами);
- `erase` (1 — версия "iter, iter");
- `replace` (2 — версия "iter, iter, num, char" и шаблонная версия).

Мы еще вернемся к вопросу `insert`, `erase` и `replace` немного позже. Для `replace`, в частности, важно выбрать в качестве членов наиболее гибкие и фундаментальные версии этой операции.

Спорные операции, которые могут не быть ни членами, ни друзьями

3. Покажите, почему функции-члены `std::string` `at`, `clear`, `empty` и `length` могут быть реализованы как не члены и не друзья без потери обобщенности, удобства использования и без влияния на остальной интерфейс `std::string`.

Сначала позвольте мне указать, что все перечисленные функции имеют одно общее фундаментальное свойство — все они легко и просто (и эффективно) могут быть переделаны в виде обычных функций, не являющихся друзьями.

- `at` (2)
- `clear`
- `empty`
- `length`

Конечно, их легко сделать обычными функциями, не связанными дружественными отношениями с классом. Но у этих функций есть еще одно общее фундаментальное свойство, а именно то, что все они упоминаются в контейнерах стандартной библиотеки как функции-члены.

"Минутку! — слышу я некоторых поклонников стандарта. — Не так быстро! Вы что, не знаете, что `basic_string` разработан таким образом, чтобы соответствовать требованиям к контейнерам, предъявляемым стандартом C++, а эти требования гласят, что некоторые функции должны быть членами? Так что не вводите читателей в заблуждение! Эти функции являются членами, хотите вы того или нет, и с этим ничего нельзя поделать!" Да, конечно, это так, но давайте отвлечемся от этого замечания для того, чтобы беспрепятственно продолжить изучение поднятого в задаче вопроса, и будем считать, что этого требования просто нет.

Рассматриваемый мною вопрос не имеет отношения к тому, что говорят требования к контейнерам. Меня интересует другое — какие функции могут без потери эффективности быть сделаны обычными функциями, не являющимися друзьями класса, и дает ли это какие-то дополнительные преимущества. Если такие преимущества существуют, то почему бы не усовершенствовать сами требования?

Рассмотрим функцию `empty`. Можем ли мы реализовать ее как обычную функцию, не являющуюся другом класса? Конечно. Стандарт требует, чтобы функция `basic_string::empty` вела себя следующим образом [C++03, §21.3.3/14]:

Возвращаемое значение: `size() == 0`

Значит, легко записать эту функцию как обычную, не являющуюся другом класса, без потери эффективности.

```
template<class charT, class traits, class Allocator>
bool empty(const basic_string<charT, traits, Allocator>& s) {
    return s.size() == 0;
}
```

Конечно, если у нас нет функции `size`, то реализация `empty` как обычной функции, не являющейся другом класса, невозможна. Альтернативный (и в некоторых случаях более надежный) способ реализации функции `empty` выглядит следующим образом.

```
template<class charT, class traits, class Allocator>
bool empty(const basic_string<charT, traits, Allocator>& s) {
    return s.begin() == s.end ();
}
```

Достижима и большая степень обобщенности.

```
template<typename T>
bool empty( const T& t ) {
    return t.begin() == t.end();
}
```

Эта окончательная версия никак не связана со строками; все, что она требует от своего аргумента, — это наличие функций `begin` и `end`. Открытые функции-члены класса должны обеспечивать необходимую и достаточную функциональность. Как мы увидим в дальнейшем, этот вопрос еще не раз возникнет при рассмотрении других

функций. (Далее в этой задаче я больше не буду записывать шаблоны функций как полностью обобщенные; все они будут связаны с классом basic_string. Однако они вполне могут быть обобщены, и мы увидим, что для этого есть масса причин.)

Обратите внимание, что хотя мы можем сделать size членом и реализовать функцию-не член empty с ее помощью, мы не можем сделать обратное. В ряде рассматриваемых здесь случаев имеется группа взаимосвязанных функций, некоторые из которых являются членами, а остальные — обычными функциями, которые реализованы с использованием упомянутых функций-членов. Какие именно функции должны быть членами? Мой совет — выбрать наиболее гибкие функции, не приводящие к потере эффективности, которые обеспечат нам гибкий фундамент, на котором можно будет легко построить все остальные функции. В нашем случае мы выбрали size в качестве функции-члена, поскольку ее результат всегда может быть кэширован (стандарт поощряет применение кэширования при реализации, поскольку функция size "должна" выполняться за постоянное время), и в этом случае реализация empty посредством функции size не менее эффективна, чем любой способ ее реализации с использованием полного непосредственного доступа ко внутренним данным класса[58].

А что можно сказать о следующей функции из условия задачи — функции at? К ней применимы те же рассуждения. Как для const, так и для не-const версии стандарт требует следующего:

Генерация исключений: out_of_range, если pos >= size().

Возвращаемое значение: operator[](pos).

Эту функцию также легко реализовать в виде обычной функции, не являющейся другом класса. Каждая из версий представляет собой двухстрочный шаблон функции, хотя и синтаксически громоздкий из-за использования всех этих параметров шаблона и имен вложенных типов.

```
template<class charT, class traits, class Allocator>
typename basic_string<charT,traits,Allocator>::const_reference
at(const basic_string<charT,traits,Allocator>&s,
   typename basic_string<charT,traits,
                         Allocator>::size_type pos)
{
    if(pos >= s.size()) throw out_of_range("don't do that");
    return s[pos];
}

template<class charT, class traits, class Allocator>
typename basic_string<charT, traits, Allocator>::reference
at(basic_string<charT, traits, Allocator>& s,
   typename basic_string<charT, traits,
                         Allocator>::size_type pos)
{
    if (pos >= s.size())
        throw out_of_range("I said, don't do that");
    return s[pos];
}
```

Что касается clear, то это всего лишь erase(begin(),end()) — не больше, не меньше. Реализация в виде обычной функции, не являющейся другом класса, — не более чем простенькое упражнение для читателя.

Осталось рассмотреть функцию length. Здесь тоже все просто — так как эта функция просто возвращает тот же результат, что и size. Кроме того, обратите внимание, что другие контейнеры не имеют функции-члена length, и в интерфейсе basic_string она играет роль "чисто строковой функции". Если мы сделаем ее не членом, то такая функция будет применима к любому контейнеру. Эта функция не

[58] Обратите внимание, что это заключение не применимо для класса list, поскольку время работы list::size линейно зависит от количества элементов в списке.

слишком полезна, поскольку представляет собой просто синоним `size`, но зато она хорошо иллюстрирует принцип, который я хотел вам продемонстрировать, а именно — как только алгоритмы становятся функциями-не членами, тут же растет их полезность и расширяется область применения.

Резюмируя, давайте рассмотрим преимущества и недостатки, которые мы получаем, переделывая функции-члены наподобие `empty`, в обычные функции. Начнем с того, что даже если мы можем переписать функцию-член как обычную функцию, не являющуюся другом класса, без потери эффективности, то почему мы должны этим заниматься? Какие реальные или потенциальные преимущества сможем мы при этом получить?

1. Простота. Сделав функции не членами, мы сокращаем код, который должны писать и сопровождать. Мы можем написать функцию `empty` раз и навсегда. Зачем нам много раз писать разные варианты функции —`basic_string::empty`, `vector::empty`, `list::empty` и так далее, включая написание этой функции для всех новых STL-совместимых контейнеров, которые могут появиться в каких-то библиотеках сторонних производителей или даже в будущих версиях стандарта C++?

Заметим, что у этого преимущества имеются некоторые ограничения. Некоторые из функций, такие как `at`, не способны обеспечить одинаковое время работы, поскольку сложность функции будет варьироваться в зависимости от используемого контейнера. Так, для `map` функция имеет логарифмическое время работы, в то время как для `vector` время работы функции является константой. Далее, может оказаться так, что не все контейнеры, имеющиеся в настоящее время или те, которые будут разработаны в будущем, будут предоставлять необходимый для работы функции интерфейс. Как видно из приведенного выше примера, функция `empty`, не являющаяся членом класса, использует функцию-член `size` и не будет работать с контейнерами, которые эту функцию не поддерживают[59]; кроме того, для контейнеров, которые имеют достаточный, но отличный от требуемого интерфейс, потребуются специализированные или перегруженные версии функций.

2. Последовательность. Таким образом мы избегаем неоправданных несовместимостей между алгоритмами, используемыми в различных контейнерах, а также между алгоритмами, используемыми в функциях-членах и обычных функциях (некоторые реально существующие несовместимости такого рода между функциями-членами и функциями-не членами указаны в [Meyers01]). Если требуется настройка поведения функций, то ее можно выполнить с помощью специализации или перегрузки шаблона функции.

3. Инкапсуляция. Использование обычных функций повышает степень инкапсуляции (что доказано в [Meyers00]).

Итак, положительные стороны такого решения нами перечислены. А как насчет отрицательных сторон? Их две, хотя лично мне кажется, что достоинства в данном случае перевешивают.

4. Засорение пространств имен. Поскольку `empty` — достаточно распространенное имя, размещение его в области видимости пространства имен приводит к риску засорения последнего — разве будут все функции с именем `empty` иметь одну и ту же семантику? Однако, во-первых, последовательность семантики — Хорошая Вещь, а во-вторых, разрешение перегрузки — хорошее противоядие против неоднозначностей, так что засорение пространства имен не такая большая проблема, как многие считали ранее. Действительно, собирая все имена функций в одно место и предоставляя общую их реализацию, мы на самом деле не столько засоряем пространство имен, сколько, как отмечает Мейерс, очищаем сами функции, собирая их в одном месте и тем самым помогая избегать очевидных и необоснованных случаев их несовместимости.

---

[59] Конечно, такой контейнер не отвечает требованиям, предъявляемым к стандартным контейнерам STL. Однако, возможно, мы захотим работать и с такими, не вполне отвечающими стандарту контейнерами.

5. Последовательность. Если все функции наподобие `empty` являются функциями-членами, то такое решение наиболее последовательно, поскольку все похожие функции имеют один и тот же синтаксис вызова. Вряд ли приятно запоминать, что надо писать `length(str)`, но `str.size()`. Этот аргумент может показаться убедительным, но только до тех пор, пока мы не заметим, что достаточно написать для всех имеющихся функций-членов соответствующие версии обычных функций, как непоследовательность оказывается только кажущейся.

В частности, что получится, если мы добавим для функций-членов наподобие `size` их обычные версии, которые будут просто вызывать соответствующие функции-члены? Это приведет к унификации синтаксиса вызовов. Например, если мы напишем функцию `size`, которая не будет членом класса, будут одинаково корректны оба варианта вызовов — как `size(str)`, так и `str.size()`, что избавит от необходимости запоминать, какие именно функции являются членами класса. В таком случае везде будет использоваться только синтаксис вызова обычной функции, и мы получим все преимущества простоты, последовательности и инкапсуляции.

Кстати, имеются и другие технические и конструкторские причины для предпочтения синтаксиса обычных функций. Последовательное написание обычных функций, не являющихся членами, существенно облегчает написание шаблонов. Если шаблоны могут для всех типов использовать функции, не являющиеся членами (в отличие от ситуации, когда часть функций является членами, а часть — нет), то они могут избежать использования классов-характеристик (traits) для выяснения, какие именно функции являются членами, а какие — нет, чтобы код корректно работал в обоих случаях (см. более подробные примеры в [Sutter02]). Другая причина заключается в том, что предоставляется возможность перегружать (бывшие) функции-члены и не члены. Если вы в ужасе отпрянете и будете этому сопротивляться — учтите, что, как показывает опыт, зачастую это Хорошая Идея, и некоторые члены комитета по стандартизации C++ считают, что в новый стандарт C++ (C++0x) можно бы было добавить перегрузку функций-членов и не членов. (Эта возможность может быть отвергнута, но некоторые эксперты считают, что в целом это потенциально неплохая идея).

Так что все возражения по поводу последовательности в действительности отпадают, поскольку написание всех, где только это возможно, функций как обычных функций, не являющихся членами, приводит только к большей последовательности.

В следующей задаче мы рассмотрим еще несколько операций, и выясним, нельзя ли и их сделать обычными функциями, не являющимися друзьями класса. Некоторые из них в соответствии с требованиями стандарта должны быть членами классов контейнеров, но мы будем рассматривать не вопрос о том, что нам говорит стандарт, а скорее о том, как бы мы разрабатывали эти классы, имея возможность начать все с белого листа.

Задача 39. Ослабленная монолитность. Часть 3: уменьшение `std::string` Сложность: 5

Продолжаем разработку диеты для быстрого похудения `std::string`...

Вопрос для новичка

1. Может ли `string::resize` быть функцией-не членом? Обоснуйте свой ответ.

Вопрос для профессионала

2. Проанализируйте следующие функции `std::string` и покажите, могут ли они быть преобразованы в обычные функции-не члены. Обоснуйте ваш ответ.
   а) Присваивание, а также +=/append/push_back.
   б) `insert`.

Решение

Операции, которые могут не быть членами

В этой задаче мы увидим, что все перечисленные далее функции могут быть реализованы как обычные функции, не являющиеся друзьями класса.

- `resize` (2)
- `assign` (6)
- `+=` (3)
- `append` (6)
- `push_back`
- `insert` (7 — все, кроме версии с тремя параметрами)

Приступим.

resize

1. Может ли `string::resize` быть функцией-не членом? Обоснуйте свой ответ.

Давайте посмотрим.

```
void resize(size_type n, charT c);
void resize(size_type n);
```

Можно ли эти функции сделать обычными функциями, не являющимися друзьями класса? Конечно, можно, поскольку их можно реализовать с использованием открытого интерфейса `basic_string` без потери эффективности. В самом деле, спецификации стандарта выражают обе функции `resize` через другие, которые мы уже рассматривали. Например, реализовать их как обычные функции, не являющиеся друзьями класса, можно следующим образом.

```
template<class charT, class traits, class Allocator>
void resize(basic_string<charT, traits, Allocator>& s,
            typename Allocator::size_type n, charT c )
{
    if( n > s.max_size() ) throw length_error("won't fit");
    if( n <= s.size() ) {
        basic_string<charT, traits, Allocator> temp(s,0,n);
        s.swap( temp );
```

```
    } else {
        s.append( n - s.size(), c );
    }
}

template<class charT, class traits, class Allocator>
void resize(basic_string<charT, traits, Allocator>& s,
            typename Allocator::size_type n )
{
    resize( s, n, charT() );
}
```

Это — один из способов реализации функций resize, не являющихся членами, которые столь же эффективны, как и функции-члены.

assign и +=/append/push_back

2. Проанализируйте следующие функции std::string и покажите, могут ли они быть преобразованы в обычные функции-не члены. Обоснуйте ваш ответ.

а) **Присваивание, а также +=/append/push_back.**

Давайте начнем с присваивания. У нас есть шесть — посчитайте сами — шесть видов присваивания. К счастью, этот случай прост: большинство из них уже выражены друг посредством друга, и все они могут быть реализованы с использованием комбинации конструктора и оператора operator=.

Остаются operator+=, append и push_back. Что можно сказать о всех этих многочисленных операциях добавления? Только то, что их схожесть наталкивает на мысль, что членом класса, вероятно, достаточно сделать только одну из них. В конце концов, все они делают одну и ту же работу, даже если немного и разнятся в деталях ее выполнения — например, добавление символа в одном случае, строки в другом и диапазона итераторов в третьем. Все они могут быть реализованы без потери эффективности в качестве обычных функций, не являющихся друзьями класса.

- Ясно, что оператор operator+= можно реализовать посредством append, поскольку это указано в стандарте C++.
- Точно так же понятно, что пять из шести версий append могут быть обычными функциями, не являющимися друзьями класса, поскольку все они определены с использованием версии append с тремя параметрами, которая в свою очередь может быть реализована посредством insert.
- Определение статуса push_back требует немного больше усилий, поскольку его семантика определена не в пункте, посвященном классу basic_string, а в разделе о требованиях к контейнерам, и здесь мы обнаруживаем, что a.push_back(x) — это просто синоним для a.insert(a.end(),x).

"Минутку! — может сказать кое-кто из читателей. — Стандарт C++ гласит, что операторы присваивания должны быть членами, а += — оператор присваивания!" И да, и нет. Не углубляясь в детали, можно сказать, что хотя оператор += и перечислен вместе с остальными операторами типа @= как оператор присваивания в грамматике C++, членом класса должен быть только оператор присваивания operator=. Следовательно, следующий пример реализации оператора operator+= как функции-не члена представляет собой совершенно корректный исходный текст C++.

```
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
operator+=(basic_string<charT, traits, Allocator>& s,
           const basic_string<charT, traits, Allocator>& t){
    return s.append( t );
}
template<class charT, class traits, class Allocator>
```

```
basic_string<charT, traits, Allocator>&
operator+=(basic_string<charT, traits, Allocator>& s,
           const charT* p ) {
    return s.append( p );
}
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
operator+=(basic_string<charT, traits, Allocator>& s,
           charT c ) {
    return s.append( 1, c );
}
```

Что же позволяет объединить все члены этой группы, сделав их обычными функциями, которые не являются друзьями класса? Это `insert`; следовательно, именно `insert` имеет смысл сделать членом класса, выполняющим основную работу и инкапсулирующим доступ ко внутренним данным класса, необходимый для добавления к строке чего-либо. Такое решение позволит сосредоточить обращение ко внутренним данным в одной функции, не "размазывая" его по дюжине функций.

insert

б) `insert`.

Тем, кому кажется, что шести версий `assign` и шести версий `append` многовато для одного класса, рекомендую запастись валерьянкой: сейчас мы рассмотрим *восемь* версий `insert`.

Я уже предлагал использовать в качестве функции-члена трехпараметрическую версию `insert`, а сейчас я поясню, почему. Во-первых, как упоминалось ранее, `insert` — более общая операция по сравнению с `append`, так что наличие функции-члена `insert` позволяет сделать все операции `append` обычными функциями, которые не являются друзьями класса. Если мы не сделаем функцией-членом хотя бы одну из функций `insert`, то нам придется сделать членом как минимум одну из функций `append`, так что я выбрал для "членства" более фундаментальную и гибкую операцию.

Однако у нас восемь функций `insert`. Какую из них (или какие) сделать членом класса? Пять из восьми функций `insert` непосредственно определены посредством функции `insert` с тремя параметрами, да и остальные версии вполне можно эффективно реализовать при помощи этой трехпараметрической функции. Поэтому функцией-членом мы делаем именно ее, а все остальные могут быть обычными функциями, не являющимися друзьями класса.

Небольшой перерыв

Для тех из вас, кто думает, что уж восемь версий `insert` — это перебор, сообщаем — нет, это только разминка. Потерпите еще немного, и мы рассмотрим *десять* версий функции `replace`. Но, чтобы пощадить ваши нервы, мы сделаем небольшой перерыв и в начале следующей задачи рассмотрим случай попроще...

Задача 40. Ослабленная монолитность. Часть 4: новый std::string

Сложность: 6

В этой задаче мы наконец полностью разберемся с классом `std::string` и посмотрим, как выглядит строковый класс минимального размера.

Вопрос для новичка

1. Может ли `string::erase` быть функцией-не членом? Обоснуйте свой ответ.

Вопрос для профессионала

2. Проанализируйте оставшиеся функции-члены `std::string` и покажите, могут ли они быть сделаны обычными функциями-не членами. Обоснуйте ваш ответ.
   а) `replace`
   б) `copy` и `substr`
   в) `compare`
   г) Семейство `find` (`find`, `find_*` и `rfind`)

Решение

Прочие операции, которые могут не быть членами

Нам осталось рассмотреть только несколько функций, и, как вы увидите, все они могут быть реализованы как обычные функции, не являющиеся друзьями класса.

- `erase` (2 — все версии, кроме "iter, iter")
- `replace` (8 — все версии, кроме "iter, iter, num, char" и шаблонных)
- `copy`
- `substr`
- `compare` (5)
- `find` (4)
- `rfind` (4)
- `find_first_of` (4)
- `find_last_of` (4)
- `find_first_not_of` (4)
- `find_last_not_of` (4)

Небольшой перерыв на кофе

1. Может ли `string::erase` быть функцией-не членом? Обоснуйте свой ответ.

После того как мы рассмотрели и разобрались с тремя десятками версий функций `assign`, `append`, `insert` и `replace`, для вас будет большим облегчением услышать, что есть только три версии функции `erase`. После всего, с чем мы имели дело, это занятие — разминка на время перерыва для того, чтобы выпить чашечку кофе...

Тройка функций-членов `erase` малоинтересна. Как минимум одна из этих функций должна быть членом (или другом); другого пути эффективной реализации этих

функций с использованием прочих уже рассмотренных функций-членов нет. Есть два "семейства" функций erase.

```
// erase( pos, length )
basic_string& erase(size_type pos = 0, size_type n = npos);

// erase( iter, ... )
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

Сразу обратим внимание на различия возвращаемых этими семействами типов: первая версия возвращает ссылку на строку, а остальные — итератор, указывающий на позицию, следующую непосредственно за удаленным символом или диапазоном символов. Далее заметим, что различны и типы аргументов этих двух семейств. Первая версия в качестве аргументов получает смещение и длину, в то время как вторая — итератор или диапазон, определяемый итераторами; к счастью, легко выполнить как преобразование итераторов в смещения (pos = iter - begin()), так и смещений в итераторы (iter = begin() + pos).

(Кстати: стандарт этого не требует, но разработчик вполне может реализовать basic_string так, что объекты этого типа будут хранить свои данные в памяти в непрерывном массиве charT. Если это так, то очевидно, что преобразование итераторов в смещения и наоборот не приведут к каким-либо накладным расходам. Хочу, однако, заметить, что даже при использовании схемы сегментированного хранения строки можно разработать очень эффективный способ преобразования итераторов в смещения и обратно, используя только открытые интерфейсы контейнеров и итераторов. Это небольшое отступление от основной темы сделано исключительно для полноты изложения.)

Все сказанное позволяет нам выразить первые две версии функции через третью.

```
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
erase(basic_string<charT, traits, Allocator>& s,
      typename Allocator::size_type pos = 0,
      typename Allocator::size_type n =
      basic_string<charT, traits, Allocator>::npos )
{
    if( pos > s.size() )
        throw out_of_range( "yes, we have no bananas" );
    typename basic_string<charT,traits,Allocator>::iterator
        first = s.begin()+pos,
        last = n==basic_string<charT,traits,Allocator>::npos
             ? s.end() : first + min( n, s.size() - pos );
    if( first != last )
        s.erase( first, last );
    return s;
}

template<class charT, class traits, class Allocator>
typename basic_string<charT, traits, Allocator>::iterator
erase(basic_string<charT, traits, Allocator>& s,
      typename basic_string<charT, traits,
                            Allocator>::iterator position )
{
    return s.erase( position, position+1 );
}
```

Ну что же, думаю, кофе к этому времени выпит?...

replace

2. Проанализируйте оставшиеся функции-члены std::string и покажите, могут ли они быть сделаны обычными функциями-не членами. Обоснуйте ваш ответ.

а) replace

Итак, replace: по правде говоря, работать с десятью функциями-членами replace малоинтересно, скучно и утомительно.

Как минимум одна из этих функций replace должна быть функцией-членом (или другом), поскольку не имеется способа эффективно выразить replace с использованием уже рассмотренных ранее функций-членов. В частности, обратите внимание, что невозможно эффективно реализовать replace путем использования erase, за которым следует insert (или в обратном порядке), поскольку оба эти способа требуют перестановки большего количества символов, чем просто replace, а кроме того, могут вызвать перераспределение буфера в памяти.

Обратите внимание, что у нас есть два семейства функций replace.

```
// replace( pos, length, ... )
basic_string& replace(size_type pos1, size_type n1,
                      const basic_string& str);        //  1
basic_string& replace(size_type pos1, size_type n1,    //  2
                      const basic_string& str,
                      size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1,     //  3
                      const charT* s, size_type n2);
basic_string& replace(size_type pos, size_type n1,     //  4
                      const charT* s);
basic_string& replace(size_type pos, size_type n1,     //  5
                      size_type n2, charT c);

// replace( iter, iter, ... )
basic_string& replace(iterator i1, iterator i2,        //  6
                      const basic_string& str);
basic_string& replace(iterator i1, iterator i2,        //  7
                      const charT* s, size_type n);
basic_string& replace(iterator i1, iterator i2,        //  8
                      const charT* s);
basic_string& replace(iterator i1, iterator i2,        //  9
                      size_type n, charT c);
template<class InputIterator>                          // 10
  basic_string& replace(iterator i1, iterator i2,
                        InputIterator j1, InputIterator j2);
```

На этот раз типы возвращаемых значений у обоих семейств одинаковы, что не может не радовать. Однако типы аргументов, как и в случае функции erase, у разных семейств различны: одно семейство основано на смещении и длине, в то время как второе использует диапазоны, определяемые итераторами. Как и в случае функции erase, возможность преобразования итераторов в позиции и наоборот позволяет нам легко реализовать одно семейство функций при помощи другого.

Когда мы решаем, какая именно версия функции должна быть сделана членом, мы хотим выбрать наиболее гибкую и фундаментальную версию(и) и реализовать остальные функции через нее. В данном случае имеется несколько ловушек, подстерегающих нас во время анализа функций для выбора той, которая должна стать функцией-членом. Рассмотрим сначала первое семейство функций.

- *Одна функция (№ 2)?* Как известно, стандарт определяет все первое семейство через версию № 2. К сожалению, некоторые из функций при этом требуют создания временных объектов basic_string, так что данный выбор оказывается не лучшим способом решения нашей проблемы. Стандарт описывает наблюдаемое поведение функций, но это описание — не обязательно наилучший способ реализации этих функций.
- *Две функции (№ 3 и № 5)?* Можно заметить, что все функции первого семейства (кроме пятой) можно эффективно реализовать посредством третьей функции, однако для того, чтобы избежать излишнего создания временного строкового

(или эквивалентного) объекта, пятая функция должна рассматриваться как особый случай, требующий, чтобы она была функцией-членом.

Перейдем к рассмотрению второго семейства.

- *Одна функция (№ 6)?* Стандарт определяет все второе семейство через версию №6. К сожалению, в этом случае вновь некоторые из функций требуют создания временных объектов `basic_string`, так что данный выбор оказывается не лучшим способом решения нашей проблемы для второго семейства.
- *Три функции (№ 7, № 9, № 10)?* Можно заметить, что большинство функций из второго семейства можно эффективно реализовать посредством функции №7, за исключением функции №9 (по той же причине, по которой в первом семействе функция №5 оказалась вынесена в отдельный случай, а именно потому что в этом случае не имеется готового буфера с корректным содержимым) и функции №10 (в которой нельзя считать, что итераторы являются указателями, более того, что это итераторы `basic_string::iterators`!).
- *Две функции (№ 9, № 10)!* Однако, как можно видеть, все функции второго семейства, за исключением функции №9, могут быть эффективно реализованы посредством функции №10, включая функцию №7. В действительности, в предположении реализации с непрерывным размещением строки и возможности преобразования позиций и итераторов друг в друга (чем мы уже пользовались ранее), мы, вероятно, можем даже реализовать все функции первого семейства... Вот оно, искомое решение!

Похоже, лучшее, что мы можем сделать, — это две функции-члены, которые позволяют нам сделать все остальные функции обычными функциями, не являющимися друзьями класса. Эти функции-члены — версия с аргументами "iter, iter, num, char" и шаблонная версия. Все остальные функции членами не являются. (Упражнение для читателя: для всех восьми оставшихся функций самостоятельно разработайте их эффективные реализации в виде обычных функций, не являющихся друзьями класса.)

Обратите внимание, как функция №10 иллюстрирует мощь шаблонов — эта единственная функция может использоваться для реализации всех прочих функций без потери эффективности, кроме двух, у которых небольшая потеря эффективности вызвана созданием временного объекта `basic_string`, содержащего `n` копий одного и того же символа).

Сейчас самое время выпить еще одну чашечку кофе...

Второй перерыв на кофе: copy и substr

б) `copy` и `substr`

Ах, это `copy`... Обратите внимание на то, что это необычный зверь и что его интерфейс не согласуется с алгоритмом `std::copy`. Еще раз посмотрим на его сигнатуру:

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

Это константная функция, которая не изменяет строку. Вместо этого строковый объект копирует часть самого себя (до `n` символов, начиная с позиции `pos`) в целевой буфер `s` (заметьте, я специально не сказал — "в С-строку"), который должен быть достаточно большим — если это окажется не так, то программа будет писать данные куда-то в непонятное место в памяти, что тут же затянет ее в болото Неопределенного Поведения. И, что самое веселое, функция `basic_string::copy` не, повторяю — *не* добавляет нулевой объект в целевой буфер (именно поэтому он и не является С-строкой. Второй причиной является то, что тип `charT` — совсем не обязательно `char`; эта функция копирует в буфер символы любого типа, из которых состоит строка). Отсутствие нулевого завершающего символа делает эту функцию достаточно опасной.

➢ Рекомендация

Никогда не используйте функции, которые пишут в буфер, размер которого не проверяется (например strcpy, sprintf), или функции, которые не завершают нулевым символом С-строку (например strncpy, basic_string::copy). Они могут привести не только к аварийному завершению работы программы, но и представляют собой угрозу безопасности системы — атака, основанная на переполнении буфера, продолжает оставаться наиболее популярной у хакеров и разработчиков вредоносных программ. Более подробно этот вопрос поднимается в задачах 2 и 3.

Все то, что делается при помощи функции copy, можно не менее просто, но гораздо более гибко сделать при помощи старого доброго алгоритма std::copy.

```
string s = "0123456789";
char* buf1 = new char[5];
s.copy(buf1, 0, 5);  // buf содержит '0', '1', '2', '3', '4'
copy(s.begin(), s.begin()+5, buf1);
                     // buf содержит '0', '1', '2', '3', '4'

int* buf2 = new int[5];
s.copy(buf2, 0, 5);  // Ошибка: первый параметр не char*
copy(s.begin(), s.begin()+5, buf2);
                     // Все в порядке: buf2 содержит
                     // значения, соответствующие символам
                     // '0', '1', '2', '3', '4' (т.е. их
                     // ASCII-значения)
```

Кстати, этот код заодно демонстрирует, как basic_string::copy можно тривиально сделать обычной функцией, не являющейся другом класса. Проще всего сделать это при помощи алгоритма copy. Пусть это останется еще одним упражнением для читателя, надо только не забыть корректно обработать частный случай n == npos.

Переведите дыхание и сделайте очередной глоток кофе. Вот еще одна простенькая функция: substr. Вспомним ее сигнатуру[60].

```
basic_string substr(size_type pos = 0,
                    size_type n = npos) const;
```

Заметим, что substr легко реализовать как обычную функцию, не являющуюся другом класса, поскольку даже стандарт гласит, что она просто возвращает новый объект типа basic_string, сконструированный как

```
basic_string<charT,traits,Allocator>(data()+pos,min(n,size()-pos))
```

Таким образом, создание новой строки, являющейся подстрокой существующей, можно одинаково легко и просто выполнить как с применением функции substr, так и без нее, с использованием более обобщенного конструктора string.

```
string s = "0123456789";
string s2 = s.substr( 0, 5 );        // s2 содержит "01234"
```

[60] Проницательные читатели могут заметить, что эта функция получает параметры в порядке "позиция, длина", в то время как только что рассмотренная функция copy получает такие же параметры в порядке "длина, позиция". Помимо эстетического несоответствия, это может оказаться просто опасным. Попытка запомнить, в каком порядке надо передавать параметры в ту или иную функцию, может привести пользователей basic_string в полный ступор, тем более что тип этих параметров одинаков, так что компилятор пропустит неверный по сути код без замечаний, ну а дальше... На мой предвзятый взгляд, такие вещи больше всего напоминают мину-растяжку на лесной тропе, на взрывателе которой выбита надпись "Сделано комитетом по стандартизации C++". Впрочем, я отвлекся...

```
string s3( s.data(), 5 );            // s3 содержит "01234"
string s4( s.begin(), s.begin()+5 ); // s4 содержит "01234"
```

Вот и все, кофе допит, отдых закончен... Работы осталось очень немного: всего только два семейства — compare и *find*.

compare

в) compare

Предпоследнее семейство функций — compare. В нем пять членов, и можно тривиально показать, что все они эффективно реализуются как обычные функции, не являющиеся друзьями класса. Как? В стандарте эти функции определены через функции basic_string size и data (которые, как мы уже решили, являются членами класса), а также traits::compare, которая и выполняет всю работу (более полую информацию о traits::compare можно найти в [Josuttis99]).

Это было сравнительно просто? Тогда давайте вместо сравнивания сложностей займемся их поиском...

find

г) Семейство find (find, find_* и rfind)

Увы, найти их не сложно. На закуску нам осталось семейство функций, по сравнению с которым восемь insert и десять replace — так, забавы для разминки. В классе basic_string — не верите, считайте сами — 24 (да, *двадцать четыре*) варианта алгоритмов поиска.

Все их можно разбить на шесть подсемейств, каждое из которых состоит ровно из четырех функций:

- find. Прямой поиск первого вхождения строки или символа (str, s или c), начиная с позиции pos;
- rfind. Обратный поиск первого вхождения строки или символа (str, s или c), начиная с позиции pos;
- find_first_of. Прямой поиск первого вхождения любого из одного или нескольких символов (str, s или c) из списка, начиная с позиции pos;
- find_last_of. Обратный поиск первого вхождения любого из одного или нескольких символов (str, s или c) из списка, начиная с позиции pos;
- find_first_not_of. Прямой поиск первого вхождения любого символа, кроме одного или нескольких указанных символов (str, s или c), начиная с позиции pos;
- find_last_not_of. Обратный поиск первого вхождения любого символа, кроме одного или нескольких указанных символов (str, s или c), начиная с позиции pos.

Каждое подсемейство состоит из четырех членов:

- "str, pos", где str содержит искомые символы (или символы, для которых осуществляется поиск несовпадения), а pos — начальная позиция в строке;
- "ptr, pos, n", где ptr — указатель charT*, указывающий на буфер длины n, содержащий искомые символы (или символы, для которых осуществляется поиск несовпадения), а pos — начальная позиция в строке;
- "ptr, pos", где ptr — указатель charT*, указывающий на буфер, завершающийся нулевым символом, который содержит искомые символы (или символы, для которых осуществляется поиск несовпадения), а pos — начальная позиция в строке;
- "c, pos", где c — искомый символ (или символ, для которого осуществляется поиск несовпадения), а pos — начальная позиция в строке.

Все эти функции могут быть эффективно реализованы как обычные функции, которые не являются друзьями класса (оставляю это читателям в качестве очередного упражнения).

Добавлю еще одно замечание о поиске в строках. В действительности, как вы могли заметить, кроме большой группы алгоритмов `basic_string::*find*`, стандарт C++ предоставляет пусть не столь многочисленную, но полнофункциональную группу алгоритмов `std::*find*`, в частности:

- `std::find` может выполнять те же действия, что и `basic_string::find`;
- `std::find` с использованием `reverse_iterator`, а также `std::find_end` могут выполнять те же действия, что и `basic_string::rfind`;
- `std::find_first_of` или `std::find` с соответствующим предикатом могут выполнять те же действия, что и `basic_string::find_first_of`;
- `std::find_first_of` или `std::find` с соответствующим предикатом, использующие `reverse_iterator`, могут выполнять те же действия, что и `basic_string::find_last_of`;
- `std::find` с соответствующим предикатом может выполнять те же действия, что и `basic_string:: find_first_not_of`;
- `std::find` с соответствующим предикатом и использованием `reverse_iterator` может выполнять те же действия, что и `basic_string::find_last_not_of`.

Кроме того, эти алгоритмы более гибкие, так как работают не только со строками. По сути, все алгоритмы `basic_string::*find*` можно реализовать с использованием алгоритмов `std::find` и `std::find_end`, снабдив их при необходимости соответствующими предикатами и/или итераторами `reverse_iterator`.

Так что, может быть, можно просто обойтись без алгоритмов `basic_string::*find*` и посоветовать программистам использовать вместо них существующие алгоритмы `std::find*`? Можно, но осторожно: несмотря на то, что таким образом можно эмулировать все алгоритмы `basic_string::*find*`, в ряде случаев использование для этой цели реализации по умолчанию `std::find*` может привести к значительной потере производительности. Три вида каждой функции `find` и `rfind`, которые выполняют поиск подстрок (а не отдельных символов), можно реализовать гораздо более эффективно, чем методом "в лоб", когда производится проверка каждой позиции и сравнивание подстрок для каждой из них. Есть хорошо известные алгоритмы, "на лету" строящие для поиска подстрок (или доказательства их отсутствия) конечные автоматы, и вполне возможно их применение в реализации функций поиска.

Нельзя ли воспользоваться такой оптимизацией, предоставив перегрузку (не специализацию — см. задачу 7) `std::find*`, которая работает с итераторами `basic_string::iterator`? Да, но только если `basic_string::iterator` является типом класса, а не обычным указателем `charT*`. Причина в том, что если бы это был обычный указатель, то специализация `std::find` для него срабатывала бы для всех без исключения указателей этого типа — что, очевидно, не верно. Она должна срабатывать только для указателей, указывающих в буфер `std::string`. По этой причине нам определенно нужно выделить `basic_string::iterator` в отдельный тип, легко отличимый от других итераторов и указателей. Тогда специализированная версия может быть специально оптимизирована для максимально эффективного поиска подстрок.

Резюме

Декомпозиция и инкапсуляция — определенно Хорошие Вещи. В частности, лучше разделять алгоритмы и контейнеры, как это сделано в стандартной библиотеке.

Общеизвестно, что basic_string имеет слишком много функций-членов. На самом деле из 103 функций basic_string только 32 действительно должны быть членами, а 71 можно без потери эффективности сделать обычными функциями, не являющимися друзьями класса. Кроме того, многие из них дуплицируют функциональность имеющихся стандартных алгоритмов, или представляют собой алгоритмы, область применения которых может оказаться более широкой, если отделить их от basic_string, а не прятать в классе.

Не повторяйте ошибок basic_string в собственных разработках — отделяйте алгоритмы от контейнеров, используйте специализации шаблонов или перегрузку для того, чтобы обеспечить специализированное поведение алгоритмов (как в случае поиска подстрок), следуйте приведенной далее рекомендации — и пользователи скажут вам только спасибо. Так вы существенно уменьшите объем того, что им придется изучить, чтобы пользоваться вашей библиотекой.

➢ **Рекомендация**

Пользуйтесь принципом "один класс (функция) — одна задача".

Там, где это возможно, лучше использовать обычные функции, которые не являются друзьями класса.

Список литературы

Примечание: для удобства читателей весь список литературы доступен по адресу `http://www.gotw.ca/publications/xc++s/bibliography.htm`

Ссылки, выделенные полужирным шрифтом (например, **[Alexandrescu02]**), представляют собой гиперссылки в приведенной выше странице.

[Alexandrescu01] A. Alexandrescu. *Modern C++ Design.* Addison-Wesley, 2001.
Перевод: А. Александреску. *Современное проектирование на C++. Серия C++ In-Depth, т.3.* — М.: Издательский дом "Вильямс", 2002.

[Alexandrescu02] A. Alexandrescu. "Discriminated Unions (I)," "... (II)," and "... (III)". *C/C++ Users Journal*, 20(4,6,8), April/June/August 2002.

[Arnold00] M. Arnold, S. Fink, D. Grove, M. Hind, P. F. Sweeney. "Adaptive Optimization in the Jalapeco JVM". *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, 2000.

[Bentley00] J. Bentley. *Programming Pearls, Second Edition.* Addison-Wesley, 2000.
Перевод: Бентли Дж. *Жемчужины программирования. Второе издание.* — СПб.: Питер, 2002.

[Boost] C++ Boost (www.boost.org).

[BoostES] "Boost Library Requirements and Guidelines, Exception-specification rationale" (Boost website).

[Cargill94] T. Cargill. "Exception Handling: A False Sense of Security". *C++ Report.* — 9(6), November-December 1994.

[C90] ISO/IEC 9899:1990(E), *Programming Languages—C* (ISO C90 and ANSI C89 standard).

[C99] ISO/IEC 9899:1999(E), *Programming Languages—C* (revised ISO and ANSI C99 standard).

[C++98] ISO/IEC 14882:1998(E), *Programming Languages—C++* (ISO and ANSI C++ standard).

[C++03] ISO/IEC 14882:2003(E), *Programming Languages—C++* (updated ISO and ANSI C++ standard including the contents of [C++98] plus errata corrections).

[C++CLI04] *C++/CLI Language Specification, Working Draft 1.6.* Ecma International, August 2004.

[Cline99] M. Cline, G. Lomow, and M. Girou. *C++ FAQs, Second Edition.* Addison-Wesley, 1999.

[Coplien92] J. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[Dewhurst02] S. Dewhurst. "C++ Hierarchy Design Idioms". *Software Development 2002 West* conference talk, April 2002.

[Dewhurst03] S. Dewhurst. *C++ Gotchas.* Addison-Wesley, 2003.

[Ellis90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.
Перевод: Эллис М., Страуструп Б. *Справочное руководство по языку программирования Си++ с комментариями.* — М.: Мир, 1992.

[Gamma95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
Перевод: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. *Приемы*

объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2001.

[GotW] H. Sutter. *Guru of the Week* (www.gotw.ca).

[Hicks00] C. Hicks. Creating an Index Table in STL. *C/C++ Users Journal.* — 18(8), August 2000.

[Hyslop00] H. Sutter and J. Hyslop. Virtually Yours. *C/C++ Users Journal.* — 18(12), December 2000.

[Hyslop01] H. Sutter and J. Hyslop. I'd Hold Anything For You. *C/C++ Users Journal.* — 19(12), December 2001.

[JikesRVM] Jikes RVM home page.

[Jones96] R. Jones and R. Lins. *Garbage Collection.* Wiley, 1996.

[Josuttis99] N. Josuttis. *The C++ standard Library.* Addison-Wesley, 1999.
Перевод: Джосьютис Н. *C++. Стандартная библиотека.* СПб.: Питер (в печати).

[Kalev01] D. Kalev. Designing a Generic Callback Dispatcher. *DevX*, 2001.

[Koenig96] A. Koenig. When Memory Runs Low. *C++ Report.* — 8(6), June 1996.

[Langer00] A. Langer and K. Kreft. *Standard C++ IOStreams and Locales.* Addison-Wesley, 2000.

[Lippman98] S. Lippman and J. Lajoie. *C++ Primer, Third Edition.* Addison-Wesley, 1998.
Перевод: Липпман С., Лажойе Ж. *Язык программирования C++. Вводный курс (3-е издание).* СПб.: Невский Диалект, 2001.

[Liskov88] B. Liskov. Data Abstraction and Hierarchy. *SIGPLAN Notices.* — 23(5), May 1988.

[Manley02] K. Manley. Using Constructed Types in Unions. *C/C++ Users Journal.* — 20(8), August 2002.

[Martin95] R. C. Martin. *Designing Object-Oriented Applications Using the Booch Method.* Prentice-Hall, 1995.

[Marrie00] L. Marrie. Alternating Skip Lists. *Dr. Dobb's Journal.* — 25(8), August 2000.

[Meyers96] S. Meyers. *More Effective C++.* Addison-Wesley, 1996.
Перевод: Мейерс С. *Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов.* — М.: ДМК Пресс, 2000.

[Meyers97] S. Meyers. *Effective C++, Second Edition.* Addison-Wesley, 1997.
Перевод: Мейерс С. *Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов.* — М.: ДМК Пресс, 2000.

[Meyers99] S. Meyers. *Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley, 1999.

[Meyers00] S. Meyers. How Non-Member Functions Improve Encapsulation. *C/C++ Users Journal.* — 18(2), February 2000.

[Meyers01] S. Meyers. *Effective STL.* Addison-Wesley, 2001.
Перевод: Мейерс С. *Эффективное использование STL.* — СПб.: Питер, 2002.

[Newkirk97] J. Newkirk. Private Interface. Object Mentor, 1997.

[ObjectMentor] Object Mentor Inc.

[Stroustrup88] B. Stroustrup. Parameterized Types for C++. *Proc. USENIX Conference*, Denver. — October 1988.

[Stroustrup94] B. Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.
Перевод: Страуструп Б. *Дизайн и эволюция языка C++.* — М.: ДМК Пресс, 2000.

[Stroustrup99] B. Stroustrup. Learning Standard C++ as a New Language. *C/C++ Users Journal.* — 17(5), May 1999.

[Stroustrup00] B. Stroustrup. *The C++ Programming Language, Special Edition.* Addison-Wesley, 2000.
Перевод: Страуструп Б. *Язык программирования C++. Специальное издание.* — М.: Бином, 2001.

[Sutter99] H. Sutter. "ACID Programming" (Guru of the Week #61, September 1999).

[Sutter00] H. Sutter. *Exceptional C++.* Addison-Wesley, 2000.
Перевод: Г. Саттер. *Решение сложных задач на C++. Серия C++ In-Depth, т.4.* — М.: Издательский дом "Вильямс", 2002.

[Sutter02] H. Sutter. *More Exceptional C++.* Addison-Wesley, 2002.
Перевод: Г. Саттер. *Решение сложных задач на C++. Серия C++ In-Depth, т.4.* — М.: Издательский дом "Вильямс", 2002.

[Sutter02a] H. Sutter. The Group of Seven: Extensions Under Consideration for the C++ Standard Library. *C/C++ Users Journal Experts Forum,* 20(4), April 2002.

[Sutter02b] H. Sutter. Smart(er) Pointers. *C/C++ Users Journal.* — 20(8), August 2002.

[Sutter02c] H. Sutter. Standard C++ Meets Managed C++. *C/C++ Users Journal, C++ .NET Solutions Supplement,* 20(9), September 2002.

[Vandevoorde03] D. Vandevoorde and N. Josuttis. *C++ Templates: The Complete Guide.* Addison-Wesley, 2003.
Перевод: Д. Вандевурд, Н. Джосаттис. *Шаблоны C++. Справочник разработчика.* — М.: Издательский дом "Вильямс", 2003.

[Warren03] Henry S. Warren, Jr. *Hacker's Delight.* Addison-Wesley, 2003.
Перевод: Г. Уоррен. *Алгоритмические трюки для программистов.* — М.: Издательский дом "Вильямс", 2003.

Предметный указатель

У

Ф

Ч

Ш

Э

Я

Научно-популярное издание

Герб Саттер

Новые сложные задачи на C++

Литературный редактор *С.Г. Татаренко*
Верстка *О.В. Линник*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д. 43, стр. 1.

Подписано в печать 04.05.2005. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 21,9. Уч.-изд. л. 18,0.
Тираж 3000 экз. Заказ № 1689.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.