Решение сложных задач на C++

87 головоломных задач с решениями

Герб Саттер

Издательский дом “Вильямс”
Москва • Санкт-Петербург • Киев
2008

Exceptional C++

87 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

Решение сложных задач на C++

ББК 32.973.26-018.2.75
С21
УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией *А.В. Слепцов*

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

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:
info@williamspublishing.com, http://www.williamspublishing.com
115419, Москва, а/я 783; 031150, Киев, а/я 152

Саттер, Герб.

С21 Решение сложных задач на C++. Серия *C++ In-Depth*: Пер. с англ. — М. : Издательский дом “Вильямс”, 2008. — 400 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-0352-5 (рус.)

В данном издании объединены две широко известные профессионалам в области программирования на языке C++ книги Герба Саттера *Exceptional C++* и *More Exceptional C++*, входящие в серию книг *C++ In-Depth*, редактором которой является Бьерн Страуструп, создатель языка C++. Материал этой книги составляют переработанные задачи серии *Guru of the Week*, рассчитанные на читателя с достаточно глубоким знанием языка C++, однако данная книга будет полезна каждому, кто хочет углубить свои знания в этой области.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Inc, Copyright ©2002

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2008

ISBN 978-5-8459-0352-5 (рус.)
ISBN 0-201-77581-6 (англ.)

© Издательский дом “Вильямс”, 2008
© Addison-Wesley Publishing Company, Inc., 2002

Оглавление

1. Обобщенное программирование и стандартная библиотека C++ **17**

2. Вопросы и технологии безопасности исключений **104**

3. Разработка классов, наследование и полиморфизм **175**

4. Брандмауэр и идиома скрытой реализации **219**

5. Пространства и поиск имен **237**

6. Управление памятью и ресурсами **257**

7. Оптимизация и производительность **293**

8. Свободные функции и макросы **319**

9. Ловушки, ошибки и антиидиомы **337**

10. Понемногу обо всем **349**

Послесловие **389**

Список литературы **391**

Предметный указатель **393**

Содержание

1. Обобщенное программирование и стандартная библиотека C++ **17**

Задача 1.1: Итераторы. 17
Задача 1.2. Строки, нечувствительные к регистру. Часть 1 19
Задача 1.3. Строки, нечувствительные к регистру. Часть 2 22
Задача 1.4. Обобщенные контейнеры с максимальным повторным использованием. Часть 1 24
Задача 1.5. Обобщенные контейнеры с максимальным повторным использованием. Часть 2 25
Задача 1.6. Временные объекты 32
Задача 1.7. Использование стандартной библиотеки (или еще раз о временных объектах) 36
Задача 1.8. Переключение потоков 38
Задача 1.9. Предикаты. Часть 1 41
Задача 1.10. Предикаты. Часть 2 44
Задача 1.11. Расширяемые шаблоны 51
Задача 1.12. Typename 62
Задача 1.13. Контейнеры, указатели и не контейнеры 65
Задача 1.14. Использование vector и deque 73
Задача 1.15. Использование set и map 79
Задача 1.16. Эквивалентный код? 85
Задача 1.17. Специализация и перегрузка шаблонов 88
Задача 1.18. Mastermind 93

2. Вопросы и технологии безопасности исключений **104**

Задача 2.1. Разработка безопасного кода. Часть 1 104
Задача 2.2. Разработка безопасного кода. Часть 2 108
Задача 2.3. Разработка безопасного кода. Часть 3 110
Задача 2.4. Разработка безопасного кода. Часть 4 115
Задача 2.5. Разработка безопасного кода. Часть 5 117
Задача 2.6. Разработка безопасного кода. Часть 6 121
Задача 2.7. Разработка безопасного кода. Часть 7 126
Задача 2.8. Разработка безопасного кода. Часть 8 128
Задача 2.9. Разработка безопасного кода. Часть 9 130
Задача 2.10. Разработка безопасного кода. Часть 10 133
Задача 2.11. Сложность кода. Часть 1 135
Задача 2.12. Сложность кода. Часть 2 138
Задача 2.13. Исключения в конструкторах. Часть 1 142
Задача 2.14. Исключения в конструкторах. Часть 2 145
Задача 2.15. Неперехваченные исключения 151
Задача 2.16. Проблема неуправляемых указателей. Часть 1 155
Задача 2.17. Проблема неуправляемых указателей. Часть 2 158
Задача 2.18. Разработка безопасных классов. Часть 1 163
Задача 2.19. Разработка безопасных классов. Часть 2 170

3. Разработка классов, наследование и полиморфизм **175**
Задача 3.1. Механика классов 175
Задача 3.2. Замещение виртуальных функций 181
Задача 3.3. Взаимоотношения классов. Часть 1 184
Задача 3.4. Взаимоотношения классов. Часть 2 187
Задача 3.5. Наследование: потребление и злоупотребление 192
Задача 3.6. Объектно-ориентированное программирование 200
Задача 3.7. Множественное наследование 201
Задача 3.8. Эмуляция множественного наследования 205
Задача 3.9. Множественное наследование и проблема сиамских близнецов 208
Задача 3.10. (Не)чисто виртуальные функции 211
Задача 3.11. Управляемый полиморфизм 215

4. Брандмауэр и идиома скрытой реализации **219**
Задача 4.1. Минимизация зависимостей времени компиляции. Часть 1 219
Задача 4.2. Минимизация зависимостей времени компиляции. Часть 2 221
Задача 4.3. Минимизация зависимостей времени компиляции. Часть 3 225
Задача 4.4. Брандмауэры компиляции 227
Задача 4.5. Идиома “Fast Pimpl” 229

5. Пространства и поиск имен **237**
Задача 5.1. Поиск имен и принцип интерфейса. Часть 1 237
Задача 5.2. Поиск имен и принцип интерфейса. Часть 2 239
Задача 5.3. Поиск имен и принцип интерфейса. Часть 3 246
Задача 5.4. Поиск имен и принцип интерфейса. Часть 4 249

6. Управление памятью и ресурсами **257**
Задача 6.1. Управление памятью. Часть 1 257
Задача 6.2. Управление памятью. Часть 2 259
Задача 6.3. Применение auto_ptr. Часть 1 265
Задача 6.4. Применение auto_ptr. Часть 2 273
Задача 6.5. Интеллектуальные указатели-члены. Часть 1 279
Задача 6.6. Интеллектуальные указатели-члены. Часть 2 283

7. Оптимизация и производительность **293**
Задача 7.1. inline 293
Задача 7.2. Отложенная оптимизация. Часть 1 296
Задача 7.3. Отложенная оптимизация. Часть 2 299
Задача 7.4. Отложенная оптимизация. Часть 3 302
Задача 7.5. Отложенная оптимизация. Часть 4 309

8. Свободные функции и макросы **319**
Задача 8.1. Рекурсивные объявления 319
Задача 8.2. Имитация вложенных функций 324
Задача 8.3. Макросы препроцессора 330
Задача 8.4. #Definition 333
Типичные ошибки при работе с макросами 333

9. Ловушки, ошибки и антиидиомы **337**
Задача 9.1. Тождественность объектов 337
Задача 9.2. Автоматические преобразования 339
Задача 9.3. Времена жизни объектов. Часть 1 340
Задача 9.4. Времена жизни объектов. Часть 2 342

10. Понемногу обо всем **349**
Задача 10.1. Инициализация. Часть 1 349
Задача 10.2. Инициализация. Часть 2 350
Задача 10.3. Корректность const 353
Задача 10.4. Приведения 359
Задача 10.5. bool 363
Задача 10.6. Пересылающие функции 366
Задача 10.7. Поток управления 367
Задача 10.8. Предварительные объявления 373
Задача 10.9. typedef 375
Задача 10.10. Пространства имен. Часть 1 377
Задача 10.11. Пространства имен. Часть 2 380

Послесловие **389**

Список литературы **391**

Предметный указатель **393**

От редактора

Книга, которую вы держите в руках, не нуждается в представлении. Кому из серьезных программистов на C++ не известен Web-узел *Guru of the Week* и его автор Герб Саттер? На основе представленных на этом Web-узле материалов Саттер издал две книги — *Exceptional C++* и *More Exceptional C++*, и в настоящее время работает над очередной книгой этой серии.

В связи с тем, что книга *More Exceptional C++* по сути представляет собой продолжение *Exceptional C++*, было принято решение объединить эти книги при издании на русском языке в одну. Благодаря четкому разделению материала книг по темам и практически идентичному стилю книг в результате получилось не механическое объединение двух книг под одной обложкой, а единая книга, вобравшая в себя опыт множества программистов высочайшего уровня.

Изложение материала в виде задач и их решений позволяет не просто получить теоретические знания, но и тут же закрепить их путем применения на практике. Строгое разделение материала книги по темам позволяет не просто прочесть ее один раз, но и использовать ее как настольную книгу, в которой всегда можно найти подсказку о том, как решать проблемы, возникающие перед вами в процессе работы над реальными проектами.

Следует сказать несколько слов об особенностях перевода книги на русский язык. Постоянно растущая сложность языка программирования C++ вносит свои коррективы в связанную с ним терминологию. Об особенностях русскоязычной терминологии C++, использованной при работе над данной книгой, достаточно полно рассказано в предисловии к русскому изданию книги [Stroustrup97]. В частности, это касается перевода термина *statement* как *инструкция*, а не как *оператор*, который в C++ имеет свое четко определенное значение. Впрочем, для серьезного программиста, на которого и рассчитана данная книга, смысл того или иного перевода очевиден из контекста.

В заключение хотелось бы поблагодарить Герба Саттера не только за замечательную книгу, но и за ряд пояснений и комментариев, полученных от него в процессе работы над русским изданием. Особую благодарность за помощь и советы при работе над переводом книги редакция выражает Александру Кротову (NIXU Ltd).

Предисловия

Это — замечательная книга, но только заканчивая ее читать, я понял, до какой степени она замечательна. Возможно, это первая книга, написанная для тех, кто хорошо знаком с C++ — *всем* C++. От базовых возможностей языка до компонентов стандартной библиотеки и современных технологий программирования — эта книга ведет нас от задачи к задаче, заставляя все время быть начеку и акцентируя все наше внимание, — как и реальные программы на C++. Здесь перемешано все — проектирование классов, поведение виртуальных функций, зависимости компиляции, операторы присваивания, безопасность исключений... Здесь все, как в реальных программах на C++. В книге водоворот из возможностей языка, библиотечных компонент, технологий программирования — водоворот, который завораживает и притягивает. Так же, как и реальные программы на C++...

Когда я сравниваю свои решения задач из книги с ответами автора, я очень часто вижу, что в очередной раз попал в подготовленную ловушку — гораздо чаще, чем мне того хотелось бы. Некоторые могут сказать, что это доказывает лишь, что я не так хорошо знаю C++. Другие сочтут это показателем того, насколько сложен C++, и что никто не в состоянии стать настоящим мастером в программировании на C++. Я же считаю, что это всего лишь указатель на то, что, программируя на C++, надо всегда быть предельно внимательным и иметь ясное представление о том, что именно ты делаешь. C++ — мощный язык, созданный для решения различных задач, и при его использовании очень важно все время оттачивать свое знание языка, его библиотеки и идиом программирования. Широта изложенного материала в данной книге поможет вам в этом.

Читатели-ветераны групп новостей, посвященных C++, знают, насколько трудно оказаться объявленным “Гуру недели” (Guru of the Week). Ветераны-участники знают об этом еще лучше. В Internet, само собой, каждую неделю может быть только один гуру, но, снабженные информацией этой книги, вы можете рассчитывать на то, что каждая ваша программа будет иметь качество, говорящее о том, что над ней работал настоящий гуру.

Скотт Мейерс (Scott Meyers), июнь 1999 г.

Как стать экспертом? Ответ один и тот же для любой области человеческой деятельности.

1. Изучить основы.
2. Изучать этот же материал снова, но в этот раз сконцентрироваться на деталях, важность которых вы не могли осознать во время первого чтения.

Если вы обратите внимание на нужные детали и подробности и овладеете соответствующими знаниями, вы существенно приблизитесь к уровню эксперта. Но, пока вы еще не эксперт, как определить, на какие именно детали следует обратить особое внимание? Вы гораздо быстрее поднимите свой уровень и получите гораздо большее удовлетворение от изучения, если найдется кто-то, кто сможет выбрать эти детали для вас.

В качестве примера я хочу рассказать, как я однажды познакомился с владельцем небольшой фотостудии Фредом Пикером (Fred Picker). Он рассказал, что в фотографии есть две основные сложности: куда направить камеру и когда нажать спуск. Затем он потратил немало времени, рассказывая о таких технических деталях, как экспозиция, проявление и печать, — деталях, которые следует хорошо знать, чтобы иметь возможность со знанием дела сконцентрироваться на основных "сложностях".

В C++, как описано ниже, наиболее интересный способ изучения деталей — пытаться отвечать на вопросы о программах на C++, например такие.

- Одинаково ли действие "`f(a++);`" и "`f(a); ++a;`"?
- Можно ли использовать итератор для изменения содержимого контейнера `set`?
- Предположим, что вы используете `vector v`, который вырос до очень большого размера. Вы хотите очистить вектор и вернуть память системе. Поможет ли вам в этом вызов `v.clear()`?

Вы, вероятно, догадываетесь, что ответы на эти простые вопросы должны быть "нет", иначе зачем бы я вас об этом спрашивал? Но знаете ли вы, почему они отрицательны? Вы уверены?

Эта книга отвечает на все перечисленные и множество других не менее интересных вопросов о казалось бы простых программах. Имеется немного других книг, подобных этой. Большинство посвященных C++ книг, претендующих на звание книг "для профессионалов", либо посвящено узкоспециализированным темам (что неплохо, если вы хотите повысить свои знания и умения в этой конкретной области, но не совсем годится для получения более глубоких знаний для ежедневного применения), либо это "для профессионалов" указано просто для привлечения читателей.

Тщательно разобравшись с вопросами и ответами в этой книге, вам больше не придется задумываться над деталями написания ваших программ, и вы сможете полностью сконцентрироваться на решении стоящей перед вами задачи.

Эндрю Кёниг (Andrew Koenig), июнь 2001 г.

Введение

Греческий философ Сократ обучал своих учеников, задавая им вопросы, которые были разработаны таким образом, чтобы руководить учениками и помогать им сделать верные выводы из того, что они уже знают, а также показать им взаимосвязи в изучаемом материале и этого материала с другими знаниями. Этот метод обучения стал так знаменит, что сегодня мы называем его “методом Сократа”. С точки зрения учащихся подход Сократа включает их в процесс обучения, заставляет думать и помогает связать и применить уже имеющиеся знания к новой информации.

Эта книга предполагает, что вам приходится заниматься написанием промышленного программного обеспечения на языке C++, и использует вопросы и ответы для обучения эффективному использованию стандарта C++ и его стандартной библиотеки, ориентируясь на разработку надежного программного обеспечения с использованием всех современных возможностей 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, показывается, как кажущиеся на первый взгляд несвязанными между собой вещи могут совместно использоваться для получения новых решений старых хорошо известных задач. Здесь вы обнаружите материал, посвященный шаблонам и пространствам имен, исключениям и наследованию, проектированию классов и шаблонам проектирования, обобщенному программированию и магии макросов, — и не просто винегрет из этих тем, а задачи, выявляющие взаимосвязь всех этих частей современного C++.

Большинство задач первоначально было опубликовано в Internet и некоторых журналах; в частности это расширенные версии первых 62 задач, 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.

Что следует знать для чтения этой книги

Я полагаю, 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].

Как читать данную книгу

Каждая задача в книге снабжена заголовком, выглядящим следующим образом.

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

Название задачи и уровень ее сложности подсказывают вам ее предназначение. Замечу, что уровень сложности задач — мое субъективное мнение о том, 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.

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 именую "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 совершенно спокойно читать их в любом порядке. Единственная подсказка: мини-серии лучше читать вместе; во всем остальном — выбор за вами.

В этой книге немало рекомендаций, в которых следующие слова имеют особое значение.

- **Всегда**. Это абсолютно необходимо; не пренебрегайте данной рекомендацией.
- **Предпочтительно**. Обычно лучше поступать так, как сказано. Поступать иначе можно только в тех случаях, когда ситуация настоятельно того требует.
- **Избегайте**. Обычно поступать так — не лучший способ, и это может оказаться опасно. Рассмотрите альтернативные способы достижения того же результата. Поступать так можно только в тех случаях, когда ситуация настоятельно того требует.
- **Никогда**. Это очень опасно, даже не думайте так поступать!

Соглашения, используемые в этой книге

В книге я даю определенные рекомендации читателям и не хотел бы давать рекомендации, которых не придерживаюсь сам. Сюда входит код примеров программ, приведенных в книге.

Если в коде примера вы видите директиву `using` в области видимости файла, а в соседней задаче — в области видимости функции, то причина этого всего лишь в том, что именно представляется мне более корректным (и эстетичным) в каждом конкретном случае.

При объявлении параметров шаблонов можно использовать как ключевое слово `class`, так и `typename`, — никакой функциональной разницы между ними нет. Однако поскольку я иногда сталкиваюсь с людьми, которые утверждают, что использовать `class` — это слишком устарело, я постарался почти везде использовать ключевое слово `typename`.

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 подвижного, 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 найти).

Благодарности

Я выражаю особую признательность редактору серии Бьярну Страуструпу (Bjarne Stroustrup), а также Дебби Лафферти (Debbie Lafferty), Марине Ланг (Marina Lang), Тирреллу Альбах (Tyrrell Albaugh), Чарльзу Ледди (Charles Leddy) и всем остальным из

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, заслуживающая 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 Бьярну Страуструпу (Bjarne Straustrup), Скотту Мейерсу (Scott Meyers), Андрею Александреску (Andrei Alexandrescu), Стиву Клэймэджу (Steve Clamage), Стиву Дьюхарсту (Steve Dewhurst), Кэю Хорстману (Cay Horstmann), Джиму Хайслопу (Jim Hyslop), Брендону Кехоу (Brendan Kehoe), Дэннису Манклу (Dennis Mancl), Яну Кристиану ван Винклю (Jan Christiaan van Winkel), Кэвлину Хенни (Kevlin Henney), Эндрю Кёнигу (Andrew Koenig), Патрику Мак-Киллену (Patric McKillen) и ряду других анонимных рецензентов. Все оставшиеся в книге ошибки, описки и неточности — только на моей совести.

И наконец, огромное спасибо моей семье и друзьям за то, что они всегда рядом, — как в процессе работы над этим проектом, так и в любое другое время.

Герб Саттер (Herb Sutter)

1

Обобщенное программирование и стандартная библиотека C++

Одна из наиболее мощных возможностей C++ — поддержка обобщенного программирования. Эта мощь находит непосредственное отражение в гибкости стандартной библиотеки C++, особенно в той ее части, в которой содержатся контейнеры, итераторы и алгоритмы, известной как библиотека стандартных шаблонов (standard template library, STL).

Этот раздел посвящен тому, как наилучшим образом использовать стандартную библиотеку C++, в частности STL. Когда и как лучше всего применять `std::vector` и `std::deque`? Какие ловушки подстерегают вас при использовании `std::map` и `std::set` и как благополучно их избежать? Почему на самом деле `std::remove()` ничего не удаляет?

Здесь вы познакомитесь с разными полезными технологиями программирования и разнообразными ловушками при написании своего собственного обобщенного кода, включая код, который работает с STL и расширяет ее. Предикаты какого типа безопасны при использовании совместно с STL, а какие нет, и почему? Какие методы можно использовать при написании кода мощного шаблона, поведение которого может изменяться в зависимости от возможностей типов, с которыми ему предстоит работать? Как можно просто переключаться между различными типами входных и выходных потоков? Как работает специализация и перегрузка шаблонов? И что означает это странное ключевое слово `typename`?

Все это и многое другое вы найдете в задачах, посвященных общим вопросам программирования и стандартной библиотеки C++.

Задача 1.1. Итераторы — *Сложность: 7*

Каждый программист, использующий стандартную библиотеку, должен быть знаком с распространенными (и не очень распространенными) ошибками при использовании итераторов. Сколько из них вы сумеете найти?

В приведенном коде имеется, как минимум, четыре ошибки, связанные с использованием итераторов. Сколько из них вы сможете обнаружить?

```
int main()
{
    vector<Date> e;
    copy(istream_iterator<Date>(cin),
         istream_iterator<Date>(),
         back_inserter(e));
    vector<Date>::iterator first =
        find(e.begin(), e.end(), "01/01/95");
```

```
    vector<Date>::iterator last =
        find(e.begin(), e.end(), "12/31/95");
    *last = "12/30/95";
    copy(first, last,
         ostream_iterator<Date>(cout,"\n"));
    e.insert(--e.end(), TodaysDate());
    copy(first, last,
         ostream_iterator<Date>(cout,"\n"));
}
```

Решение

```
int main()
{
    vector<Date> e;
    copy(istream_iterator<Date>(cin),
         istream_iterator<Date>(),
         back_inserter(e));
```

До сих пор все в порядке. Класс `Date` снабжен функцией с сигнатурой `operator>>(istream&, Date&)`, которую `istream_iterator<Date>` использует для чтения объектов `Date` из потока `cin`. Алгоритм `copy()` заносит эти объекты в вектор.

```
vector<Date>::iterator first =
    find(e.begin(), e.end(), "01/01/95");
vector<Date>::iterator last =
    find(e.begin(), e.end(), "12/31/95");
*last = "12/30/95";
```

Ошибка: это присваивание может быть некорректным, поскольку `last` может оказаться равным `e.end()` и, таким образом, не может быть разыменован.

Если значение не найдено, алгоритм `find()` возвращает свой второй аргумент (конечный итератор диапазона). В таком случае, если `"12/31/95"` не содержится в `e`, `last` окажется равным `e.end()`, который указывает на элемент за концом контейнера и, следовательно, корректным итератором не является.

```
copy(first, last,
     ostream_iterator<Date>(cout,"\n"));
```

Ошибка заключается в том, что `[first, last)` может не быть корректным диапазоном — на самом деле `first` может находиться после `last`. Например, если `"01/01/95"` не содержится в `e`, но зато в нем имеется `"12/31/95"`, то итератор `last` будет указывать на что-то внутри контейнера (точнее — на объект `Date`, равный `"12/31/95"`), в то время как итератор `first` — за его пределы (на позицию элемента, следующего за последним). Однако `copy()` требует, чтобы `first` указывал на позицию элемента, находящегося до элемента, на который в том же множестве указывает `last`, — т.е. `[first, last)` должен быть корректным диапазоном.

Если только вы не используете версию стандартной библиотеки, которая проверяет за вас наличие такого рода проблем, то наиболее вероятный симптом происшедшей ошибки — аварийный сброс сложного для диагностики образа памяти программы в процессе выполнения `copy()` или вскоре после него.

```
e.insert(--e.end(), TodaysDate());
```

Первая ошибка: выражение `--e.end()` вполне может оказаться неверным. Причина этого проста, если немного задуматься: в популярных реализациях STL `vector<Date>::iterator` зачастую представляет собой просто `Date*`, а язык C++ не позволяет изменять временные переменные встроенного типа. Например, следующий фрагмент кода неверен.

```
Date* f();    // Функция, возвращающая Date*
p = --f();     // Ошибка. Здесь можно использовать "f()-1"
```

К счастью, мы знаем, что `vector<Date>::iterator` — итератор с произвольным доступом, так что без потери эффективности можно записать более корректный вызов `e.insert()`.

```
e.insert(e.end() - 1, TodaysDate());
```

Теперь у нас появляется вторая ошибка, заключающаяся в том, что если контейнер `e` пуст, то любые попытки получить итератор, указывающий на позицию перед `e.end()` (то, чего мы пытались добиться записью "`--e.end()`" или "`e.end()-1`"), будут некорректны.

```
copy(first, last,
     ostream_iterator<Date>(cout,"\n"));
```

Здесь ошибка в том, что `first` и `last` могут больше не быть корректными итераторами. Вектор растет "кусками", чтобы при каждой вставке элемента в вектор не приходилось увеличивать буфер последнего. Однако иногда вектор все же заполняется, и перераспределение памяти становится совершенно необходимо.

В нашей ситуации при выполнении операции `e.insert()` вектор может вырасти (а может и нет), что по сути означает, что его память может оказаться перемещена (или нет). Из-за этой неопределенности мы должны рассматривать все существующие итераторы, указывающие на элементы контейнера, как более недействительные. В приведенном коде при реальном перемещении памяти некорректное копирование вновь приведет к аварийному сбросу образа памяти.

Рекомендация

Никогда не используйте некорректные итераторы.

Резюмируем: при использовании итераторов всегда задавайте себе четыре основных вопроса.

1. Корректность значений. Возможно ли разыменование данного итератора? Например, написание "`*e.end()`" всегда является ошибкой.
2. Корректность времени жизни объекта. Остался ли корректен используемый вами итератор после выполнения предыдущих действий?
3. Корректность диапазона. Представляет ли пара итераторов корректный диапазон? Действительно ли `first` указывает на позицию, располагающуюся до позиции (или равную ей), на которую указывает `last`? Указывают ли оба итератора на элементы одного и того же контейнера?
4. Некорректные встроенные операции. Не пытается ли код модифицировать временный объект встроенного типа (как в случае "`--e.end()`")? (К счастью, такого рода ошибки компилятор часто находит сам; кроме того, для итераторов, типы которых представляют собой классы, автор библиотеки для удобства может разрешить применение таких действий.)

Задача 1.2. Строки, нечувствительные к регистру. Часть 1 *Сложность: 7*

Вам нужен строковый класс, нечувствительный к регистру символов? Ваша задача — создать его.

Эта задача состоит из трех взаимосвязанных частей.

1. Что означает “нечувствительный к регистру”?
2. Напишите класс `ci_string`, идентичный стандартному классу `std::string`, но который является нечувствительным к регистру символов в том же смысле, что и распространенное расширение библиотеки C++ `stricmp()`[1]. Класс `ci_string`[2] должен использоваться следующим образом.

```
ci_string s("AbCdE");
// Нечувствительно к регистру
assert( s == "abcde" );
assert( s == "ABCDE" );
// Остается чувствительно к регистру
assert(strcmp(s.c_str(),"AbCdE") == 0);
assert(strcmp(s.c_str(),"abcde") != 0);
```

3. Разумно ли делать нечувствительность к регистру свойством объекта?

Решение

Вот ответы на поставленные вопросы.

1. Что означает “нечувствительный к регистру”?

Что это означает, в действительности полностью зависит от вашего приложения и языка. Например, многие языки вообще не поддерживают различные регистры символов. В ходе ответа на этот вопрос вам надо также определить, считаете ли вы акцентированные символы эквивалентными неакцентированным или нет (например, одинаковы ли символы в строке “aáâãä”), и рассмотреть многие другие вопросы. Данная задача представляет собой руководство, каким образом следует реализовать нечувствительность к регистру стандартных строк в любом смысле, который вы вложите в это понятие.

2. Напишите класс `ci_string`, идентичный стандартному классу `std::string`, но который является нечувствительным к регистру символов в том же смысле, что и распространенное расширение библиотеки C++ `stricmp()`.

Вопрос *как создать нечувствительную к регистру символов строку* настолько распространен, что, вероятно, заслуживает собственного FAQ[3] (чем, пожалуй, и является эта задача). Итак, вот что мы хотим получить.

```
ci_string s("AbCdE");
// Нечувствительно к регистру
assert( s == "abcde" );
assert( s == "ABCDE" );
// Остается чувствительно к регистру
assert(strcmp(s.c_str(),"AbCdE") == 0);
assert(strcmp(s.c_str(),"abcde") != 0);
```

Ключевым моментом здесь является понимание того, чем на самом деле является `string` в стандарте C++. Если вы заглянете внутрь заголовочного файла `string`, то обнаружите там нечто типа

```
typedef basic_string<char> string;
```

[1] Функция `stricmp()`, выполняющая сравнение строк без учета регистров символов, хотя и не является частью стандарта C или C++, но достаточно широко распространена в качестве расширения стандартной библиотеки.

[2] Префикс `ci` означает “case-insensitive”, т.е. нечувствительный к регистру. — *Прим. перев.*

[3] Frequently Asked Questions — дословно: часто задаваемые вопросы; общепринятая форма представления материала (в основном для новичков). — *Прим. перев.*

Таким образом, string не является классом — это всего лишь синоним некоторого шаблона. В свою очередь, шаблон basic_string<> объявлен следующим образом (возможно, с дополнительными параметрами шаблона в конкретной реализации).

```
template<class charT,
         class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_string;
```

Итак, "string" на самом деле означает "basic_string<char, char_traits<char>, allocator<char> >", возможно, с какими-то дополнительными параметрами шаблона в конкретной используемой вами реализации. Нам не придется заботиться о части allocator, но придется как следует познакомиться с частью char_traits, поскольку именно здесь определяется, как взаимодействуют символы — в том числе и как они сравниваются!

Итак, пусть сравниваются объекты string. Шаблон basic_string снабжен функциями сравнения, которые позволяют вам определить, равны ли две строки, или одна из них больше другой. Эти функции построены на базе функций сравнения символов, которые предоставляет шаблон char_traits. В частности, этот шаблон содержит функции сравнения eq() и lt() для проверки равенства и соотношения "меньше" двух символов, а также функции compare() и find() для сравнения последовательностей символов и поиска символа в последовательности.

Если мы хотим получить поведение сравнения, отличающееся от стандартного, все, что мы должны сделать, — это обеспечить новый шаблон char_traits. Вот простейший способ сделать это.

```
struct ci_char_traits : public char_traits<char>
    // Просто наследуем все остальные функции,
    // чтобы не писать их заново
{
    static bool eq(char c1, char c2)
        { return toupper(c1) == toupper(c2); }
    static bool lt(char c1, char c2)
        { return toupper(c1) <  toupper(c2); }
    static int compare( const char * s1,
                        const char * s2,
                        size_t n )
        { return memicmp( s1, s2, n ); }
        // Используем memicmp, если эта функция
        // имеется на вашей платформе; в противном
        // случае надо писать собственную версию
    static const char *
        find( const char * s, int n, char a )
        {
            while(n-- > 0 && toupper(*s) != toupper(a))
            {
                ++s;
            }
            return n >= 0 ? s : 0;
        }
};
```

И наконец, собираем все это вместе:

```
typedef basic_string<char, ci_char_traits> ci_string;
```

Таким образом мы создаем тип ci_string, который работает в точности так же, как и стандартный тип string (в конце концов, в большинстве аспектов это и *есть* стандартный тип string), за исключением того, что правила сравнения символов определяются вместо типа char_traits<char> типом ci_char_traits. Поскольку мы сделали правила сравнения ci_char_traits нечувствительными к регистрам, класс ci_string также становится нечувствительным к регистрам без каких бы то ни было

дополнительных действий — т.е. мы получили нечувствительный к регистрам класс, совершенно не изменяя класс basic_string.

3. Разумно ли делать нечувствительность к регистру свойством объекта?

Зачастую более полезно, когда нечувствительность к регистру является свойством функции сравнения, а не объекта, как показано в приведенном решении. Например, рассмотрим следующий код.

```
string    a = "aaa";
ci_string b = "aAa";
if (a == b) /* ... */
```

Каким должен быть результат сравнения "a == b" — true или false? Можно легко обосновать точку зрения, согласно которой при наличии хотя бы одного сравниваемого объекта, нечувствительного к регистру, само сравнение также должно быть нечувствительно к регистру. Но что если мы немного изменим наш пример и введем еще один вариант basic_string, выполняющий сравнение третьим способом.

```
typedef basic_string<char, yz_char_traits> yz_string;

ci_string b = "aAa";
yz_string c = "AAa";
if (b == c) /* ... */
```

Итак, вернемся к нашему вопросу: чему равно выражение "b == c" — true или false? Я думаю, вы согласитесь, что это совершенно не очевидно, и непонятно, почему мы должны предпочесть тот или иной способ сравнения.

Рассмотрим гораздо более ясные примеры сравнения.

```
string a = "aaa";
string b = "aAa";
if (stricmp(a.c_str(), b.c_str()) == 0) /* ... */
string c = "AAa";
if (EqualUsingYZComparison(b, c)) /* ... */
```

Во многих случаях более практично, когда нечувствительность к регистру является характеристикой операции сравнения, хотя в жизни мне встречались ситуации, в которых нечувствительность к регистру как характеристика объекта (в особенности когда все или большинство сравнений должно выполняться со строками char* в стиле C) была гораздо более практична, хотя бы потому, что строковые значения при этом можно сравнивать более "естественно" (if (a == "text") ...), не вспоминая каждый раз, что нам требуется использование именно нечувствительного к регистру сравнения.

Задача 1.3. Строки, нечувствительные к регистру. Часть 2 — *Сложность: 5*

Насколько практичен созданный в задаче 1.2 класс ci_string? В этой задаче мы обратимся к вопросам его использования. Мы узнаем, с какими проблемами можем столкнуться в процессе разработки, и заполним все оставшиеся пробелы в этой теме.

Обратимся еще раз к решению задачи 1.2 (не обращая внимания на тела функций).

```
struct ci_char_traits : public char_traits<char>
{
    static bool eq(char c1, char c2)      { /* ... */ }
    static bool lt(char c1, char c2)      { /* ... */ }
    static int compare( const char * s1,
                        const char * s2,
                        size_t n )        { /* ... */ }
    static const char *
```

```
    find( const char * s, int n, char a ) { /* ... */ }
};
```

В данной задаче вы должны ответить на следующие вопросы по возможности более полно.

1. Безопасно ли наследовать `ci_char_traits` от `char_traits<char>` таким образом?
2. Почему следующий код вызывает ошибку компиляции?

```
ci_string s = "abc";
cout << s << endl;
```

3. Что можно сказать об использовании других операторов (например, +, +=, =) и о сочетании `string` и `ci_string` в качестве их аргументов? Например:

```
string    a = "aaa";
ci_string b = "bbb";
string    c = a + b;
```

Решение

Вот ответы на поставленные вопросы.

1. Безопасно ли наследовать `ci_char_traits` от `char_traits<char>` таким образом?

Открытое наследование, в соответствии с принципом подстановки Барбары Лисков (Liskov Substitution Principle, LSP), обычно должно моделировать отношение ЯВЛЯЕТСЯ/РАБОТАЕТ ПОДОБНО (см. задачу 3.3). В данном случае мы имеем дело с одним из редких исключений из LSP, поскольку `ci_char_traits` не предназначен для полиморфного использования посредством указателя или ссылки на базовый класс `char_traits<char>`. Стандартная библиотека не использует эти объекты полиморфно, и наследование использовано только из-за удобства (ну, конечно, найдутся и те, кто скажет, что это было сделано из-за лени); таким образом, в этом примере наследование не используется объектно-ориентированным путем.

Однако LSP остается применим в другом смысле: он применим во время компиляции, когда порожденный объект должен РАБОТАТЬ ПОДОБНО базовому объекту таким образом, как указывают требования шаблона `basic_string`. В статье в группе новостей Натан Майерс (Nathan Myers[4]) пояснил это так.

> *С другой стороны, LSP применим, но только во время компиляции и посредством соглашения, которое мы назовем "списком требований". Я бы выделил этот случай и назвал его обобщенным принципом подстановки Лисков (Generic Liskov Substitution Principle, GLSP): любой тип (или шаблон), передаваемый в качестве аргумента, должен соответствовать требованиям, предъявляемым к данному аргументу.*
>
> *Классы, производные от итераторов и классов-свойств, соответствуют GLSP, хотя классические признаки LSP (например, виртуальный деструктор и т.п.) могут при этом как иметься в наличии, так и отсутствовать, — в зависимости от того, определено ли полиморфное поведение времени выполнения в сигнатуре в списке требований или нет.*

Короче говоря, такое наследование безопасно, поскольку удовлетворяет GLSP (если не LSP). Однако основная причина, по которой я использовал его в данной задаче, — не удобство (хотя таким образом и удалось избежать переписывания всего на-

[4] Натан — давний член Комитета по стандарту C++ и основной автор стандартных средств `locale`.

бора функций char_traits<char>), а демонстрация *отличий*, — т.е. того факта, что для достижения наших целей мы изменили только четыре операции.

Смысл рассматриваемого вопроса — заставить вас задуматься о некоторых вещах: 1) о корректном употреблении (и некорректном злоупотреблении) наследования; 2) о смысле того факта, что у нас имеются только статические члены; 3) о том, что объекты char_traits никогда не используются полиморфно.

2. Почему следующий код вызывает ошибку компиляции?

```
ci_string s = "abc";
cout << s << endl;
```

Указание: в п. 21.3.7.9 [lib.string.io] стандарта C++ объявление оператора << для basic_string определено следующим образом:

```
template<class charT, class traits, class Allocator>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
           const basic_string<charT, traits,
                              Allocator>& str);
```

Ответ: обратите внимание на то, что cout представляет собой объект basic_ostream<char, char_traits<char> >. Теперь становится понятна и возникшая проблема: оператор << для basic_string представляет собой шаблон, но этот оператор может выводить в поток basic_ostream только строки с тем же "типом символов" и "типом свойств", что и указанные в классе string. Таким образом, оператор << может выводить ci_string в поток basic_ostream<char, ci_char_traits>, однако этот тип не совпадает с типом cout, несмотря на то, что ci_char_traits является наследником char_traits<char>.

Имеется два пути разрешения данной проблемы: мы можем определить операторы << и >> для ci_string, либо добавлять к объектам этого типа ".c_str()" для того, чтобы использовать operator<<(const char*), если, конечно, ваши строки не содержат нулевые символы:

```
cout << s.c_str() << endl;
```

3. Что можно сказать об использовании других операторов (например, +, +=, =) и о сочетании string и ci_string в качестве их аргументов? Например:

```
string    a = "aaa";
ci_string b = "bbb";
string    c = a + b;
```

Как и в предыдущем вопросе, у нас есть два варианта работы с оператором: либо мы определяем собственную функцию operator+(), либо добавляем ".c_str()" для использования operator+(const char*):

```
string c = a + b.c_str();
```

Задача 1.4. Обобщенные контейнеры с максимальным повторным использованием. Часть 1 *Сложность: 8*

Насколько гибким вы сможете сделать простой класс контейнера?

Как наилучшим образом вы можете реализовать копирующее конструирование и копирующее присваивание для приведенного класса вектора фиксированной длины? Как вы обеспечите максимальное повторное использование конструирования и присваивания? *Указание*: задумайтесь о том, что может понадобиться пользователю.

```
template<typename T, size_t size>
class fixed_vector
```

```
{
public:
    typedef T*       iterator;
    typedef const T* const_iterator;
    iterator         begin()       { return v_; }
    iterator         end()         { return v_ + size; }
    const_iterator   begin() const { return v_; }
    const_iterator   end()   const { return v_ + size; }
private:
    T v_[size];
};
```

Примечание: не делайте ничего, кроме упомянутого в задаче. Этот контейнер не планируется делать STL-совместимым; кроме того, здесь имеется как минимум одна “хитроумная” проблема. Он предназначен исключительно для иллюстрации некоторых важных вопросов в упрощенном варианте.

Решение

При решении этой задачи мы поступим несколько необычно. Я предложу вам решение, а вашей задачей будет пояснить его и раскритиковать. Рассмотрим теперь следующую задачу.

Задача 1.5. Обобщенные контейнеры с максимальным повторным использованием. Часть 2 *Сложность: 6*

Историческая справка: пример, использованный в данной задаче, создан на основе представленного Кевлином Хенни (Kevlin Henney) и позже проанализированного Джоном Джаггером (Jon Jagger) (см. британский журнал Overload, *посвященный C++, выпуски 12 и 20).*

Что и почему делает следующее решение? Поясните работу каждого конструктора и оператора. Имеются ли в приведенном коде какие-либо изъяны?

```
template<typename T, size_t size>
class fixed_vector
{
public:
    typedef T*       iterator;
    typedef const T* const_iterator;
    fixed_vector() {}

    template<typename O, size_t osize>
    fixed_vector(const fixed_vector<O,osize>& other)
    {
        copy( other.begin(),
              other.begin() + min(size, osize),
              begin());
    }

    template<typename O, size_t osize>
    fixed_vector<T,size>&
    operator=(const fixed_vector<O,osize>& other)
    {
        copy( other.begin(),
              other.begin() + min(size, osize),
              begin());
        return *this;
    }

    iterator         begin()       { return v_; }
```

```
    iterator         end()         { return v_ + size; }
    const_iterator   begin() const { return v_; }
    const_iterator   end()   const { return v_ + size; }
private:
    T v_[size];
};
```

Решение

Проанализируем приведенное решение и посмотрим, насколько хорошо оно отвечает поставленному заданию. Вспомним исходные вопросы. *Как наилучшим образом вы можете реализовать копирующее конструирование и копирующее присваивание для приведенного класса вектора фиксированной длины*? *Как вы обеспечите максимальное повторное использование конструирования и присваивания*? Указание: *задумайтесь о том, что может понадобиться пользователю.*

Копирующее конструирование и копирующее присваивание

Для начала отметим, что первый вопрос по сути был отвлекающим маневром. Вы уже поняли, в чем дело? Исходный код уже содержит конструктор копирования и оператор копирующего присваивания, которые вполне успешно работают. Приведенное же решение отвечает на второй вопрос: путем добавления шаблонных конструктора и оператора присваивания мы делаем конструирование и присваивание объектов более гибким.

```
fixed_vector() {}

template<typename O, size_t osize>
fixed_vector(const fixed_vector<O,osize>& other)
{
    copy( other.begin(),
          other.begin() + min(size, osize),
          begin());
}

template<typename O, size_t osize>
fixed_vector<T,size>&
operator=(const fixed_vector<O,osize>& other)
{
    copy( other.begin(),
          other.begin() + min(size, osize),
          begin());
    return *this;
}
```

Обратите внимание: приведенные функции *не являются* конструктором копирования и оператором копирующего присваивания. Дело в том, что конструктор копирования (оператор копирующего присваивания) создает объект (выполняет присваивание) из другого объекта в точности такого же типа, включая одинаковые аргументы шаблона, если класс представляет собой шаблон. Например:

```
struct X
{
    template<typename T>
    X( const T& );          // Не конструктор копирования -
                            // T не может быть X
    template<typename T>
    operator=( const T& ); // Не операция присваивания -
```

```
                                    // T не может быть X
};
```

“Но, — скажете вы, — эти две шаблонные функции-члены могут в точности соответствовать сигнатурам конструктора копирования и копирующего присваивания!” Вы будете неправы — в обоих случаях T не может быть X. Вот цитата из стандарта (12.8.2, примечание 4).

> *Поскольку* конструктор по шаблону никогда не является конструктором копирования*, наличие такого конструктора шаблона не запрещает неявного объявления конструктора копирования. Конструкторы шаблонов участвуют в разрешении перегрузки наряду с другими конструкторами, включая конструкторы копирования. Конструктор по шаблону может использоваться для копирования объекта, если он обеспечивает лучшее совпадение, чем все остальные конструкторы.*

Аналогичные слова сказаны и об операторе присваивания копированием (12.8.10, примечание 7). Таким образом, предлагаемое решение оставляет конструктор копирования и оператор присваивания копированием теми же, что и в исходном коде, поскольку компилятор продолжает генерировать их неявные версии. То, что было сделано нами, — это расширение гибкости конструирования и присваивания, но не замена старых версий.

В качестве еще одного примера рассмотрим следующую программу.

```
fixed_vector<char,4> v;
fixed_vector<int,4>  w;
fixed_vector<int,4>  w2(w);
        // Вызов неявного конструктора копирования
fixed_vector<int,4>  w3(v);
        // Вызов шаблона конструктора с преобразованием
w = w2; // Вызов неявного оператора присваивания копированием
w = w2; // Вызов шаблона оператора присваивания
```

Таким образом, в действительности нам надо обеспечить гибкость “присваивания и конструирования из объектов других типов fixed_vector”, а не “гибкое конструирование и присваивание копированием”, которые имеются изначально.

Вопросы применения конструирования и присваивания

Имеются два вопроса, которые нам надо рассмотреть.

1. Поддержка различных типов (включая вопросы наследования).

Вектор fixed_vector определенно представляет собой гомогенный контейнер (и должен таковым и оставаться), однако иногда имеет смысл присваивание или конструирование его из другого объекта fixed_vector, который фактически содержит объекты другого типа. При условии, что эти объекты могут быть присвоены объектам нашего типа, это должно быть осуществимо. Так, пользователь разработанного нами класса может захотеть написать нечто подобное.

```
fixed_vector<char,4> v;
fixed_vector<int,4>  w(v); // Шаблонный конструктор
w = v;                     // Шаблонное присваивание

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

fixed_vector<D*,4> x;
fixed_vector<B*,4> y(x);   // Шаблонный конструктор
y = x;                     // Шаблонное присваивание
```

Это вполне корректный и работающий код, поскольку объекты типа `D*` могут быть присвоены объектам `B*`.

2. Поддержка различных размеров.

Аналогично, пользователь класса может захотеть присвоить или сконструировать объект из объекта `fixed_vector` с отличающимся размером. Поддержка такой возможности вполне разумна. Например:

```
fixed_vector<char,6> v;
fixed_vector<int,4>  w(v);
            // Инициализация с использованием 4 значений
w = v;      // Присваивание с использованием 4 значений

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

fixed_vector<D*,16> x;
fixed_vector<B*,42> y(x);
            // Инициализация с использованием 16 значений
y = x;      // Присваивание с использованием 16 значений
```

Подход стандартной библиотеки

Как ни странно, но мне нравится синтаксис и применимость разработанных функций, хотя имеются некоторые недоработки. Рассмотрим еще один подход к задаче, следующий стилю стандартной библиотеки.

1. Копирование.

```
template<class RAIter>
fixed_vector(RAIter first, RAIter last)
{
    copy( first,
          first + min(size,(size_t)last-first),
          begin() );
}
```

Теперь при копировании вместо

```
fixed_vector<char,6> v;
fixed_vector<int,4>  w(v);
        // Инициализация с использованием 4 значений
```

мы должны писать

```
fixed_vector<char,6> v;
fixed_vector<int,4>  w(v.begin(), v.end());
        // Инициализация с использованием 4 значений
```

А теперь немного поразмыслите над этим моментом. Какой стиль конструирования лучше — предложенный нами при решении задачи или этот стиль стандартной библиотеки?

Наше решение несколько проще в применении, зато решение в стиле стандартной библиотеки несколько более гибкое (например, оно позволяет пользователю выбирать поддиапазон и выполнять копирование из контейнеров другого типа). Вы можете выбрать один из рассмотренных вариантов или оставить в классе оба.

2. Присваивание.

Обратите внимание, что мы не можем использовать шаблонное присваивание для получения диапазона итераторов, поскольку `operator=()` может иметь только один параметр. Для этого мы можем ввести в класс именованную функцию.

```
template<class Iter>
fixed_vector<T,size>&
assign(Iter first, Iter last)
{
    copy( first,
          first + min(size,(size_t)last-first),
          begin() );
    return *this;
}
```

Теперь при присваивании вместо

```
w = v; // Присваивание с использованием 4 значений
```

мы должны записать

```
w.assign(v.begin(), v.end());
       // Присваивание с использованием 4 значений
```

Технически наличие функции `assign` необязательно, поскольку мы можем получить ту же гибкость и без нее, просто несколько некрасиво и менее эффективно.

```
w = fixed_vector<int,4>(v.begin(), v.end());
  // Инициализация и присваивание с использованием 4 значений
```

И вновь остановитесь и задумайтесь над предложенными альтернативами. Какой стиль присваивания предпочтительнее — предложенный нами при решении этой задачи или стиль стандартной библиотеки?

В данной ситуации аргумент гибкости не играет столь большой роли, поскольку пользователь может самостоятельно (и даже более гибко) написать копирование самостоятельно — вместо

```
w.assign(v.begin(), v.end());
```

можно просто написать

```
copy(v.begin(), v.begin()+4, w.begin());
```

Так что в этом случае нет существенных аргументов в пользу создания отдельной функции `assign` для работы с диапазоном итераторов. Пожалуй, наиболее разумным решением будет оставить разработанный нами ранее оператор присваивания и позволить пользователям при необходимости работы с диапазоном использовать непосредственно функцию `copy()`.

Зачем нужен конструктор по умолчанию

И наконец, зачем в предложенном решении приведен пустой конструктор по умолчанию, который делает то же самое, что и конструктор по умолчанию, генерируемый компилятором?

Ответ краток и прост: как только вы определили конструктор любого типа, компилятор больше не генерирует конструктор по умолчанию, и определение его становится вашей заботой. В приведенном же выше пользовательском коде совершенно очевидна необходимость конструктора по умолчанию.

Затянувшаяся задача

В условии задачи также спрашивается, имеются ли в приведенном в условии коде какие-либо изъяны?

Возможно. Позже в этой книге мы поговорим о различных вариантах гарантий безопасной работы исключений. Так же, как и генерируемый компилятором, наш

оператор присваивания обеспечивает базовую гарантию, чего может быть вполне достаточно. Тем не менее давайте посмотрим, что случится, если мы захотим обеспечить строгую гарантию, делая оператор строго безопасным в смысле исключений. Вспомним, что шаблонный оператор присваивания определен следующим образом.

```
template<typename O, size_t osize>
fixed_vector<T,size>&
operator=(const fixed_vector<O,osize>& other)
{
    copy( other.begin(),
          other.begin() + min(size, osize),
          begin());
    return *this;
}
```

Если одно из присваиваний объекту `T` в процессе выполнения `copy` окажется неудачным, вектор окажется в несогласованном состоянии. Часть содержимого нашего вектора останется той же, что и до выполнения прерванного по исключению присваивания, а другая к этому моменту уже будет обновлена.

Увы, в разработанном к настоящему времени виде *операция присваивания класса* `fixed_vector` *не может обеспечить строгой безопасности исключений.* Почему? По следующим причинам.

- Обычно правильный (и простейший) путь решения этого вопроса — написание обеспечении атомарной и не генерирующей исключений функции `Swap()` для обмена "внутренностей" объектов `fixed_vector` с последующим использованием канонического (в нашем случае шаблонного) вида `operator=()`, который использует идиому создать-временный-объект-и-обменять (см. задачу 2.6).
- Не имеется никакой возможности написать атомарную, не генерирующую исключений функцию `Swap()` для обмена содержимого двух объектов `fixed_vector`. Это связано с тем, что содержимое `fixed_vector` организовано в виде простого массива, который не может быть скопирован за один атомарный шаг.

Только не отчаивайтесь раньше времени! На самом деле решение имеется, но оно требует изменения самой конструкции `fixed_vector`: информация должна храниться не в массиве-члене, а в динамически выделяемом массиве. Это, конечно, несколько снижает эффективность `fixed_vector`, а также означает, что мы должны создать деструктор — все это цена строгой безопасности исключений.

```
// Строго безопасная в смысле исключений версия
//
template<typename T, size_t size>
class fixed_vector
{
public:
    typedef T*       iterator;
    typedef const T* const_iterator;
    fixed_vector() : v_( new T[size] ) {}
    ~fixed_vector() { delete[] v_; }

    template<typename O, size_t osize>
    fixed_vector(const fixed_vector<O,osize>& other)
    : v_( new T[size] )
    {
        try {
            copy( other.begin(),
                  other.begin() + min(size, osize),
                  begin());
        } catch(...) { delete[] v_; throw; }
    }
```

```
    fixed_vector(const fixed_vector<T,size>& other)
    : v_( new T[size] )
    {
        try {
            copy( other.begin(), other.end(), begin() );
        } catch(...) { delete[] v_; throw; }
    }

    void Swap( fixed_vector<T,size& other ) throw()
    {
        swap(v_, other.v_);
    }

    template<typename O, size_t osize>
    fixed_vector<T,size>&
    operator=(const fixed_vector<O,osize>& other)
    {
        // Вся работа делается здесь:
        fixed_vector<T,size> temp(other);
        // Здесь исключения не генерируются:
        Swap(temp); return *this;
    }

    fixed_vector<T,size>&
    operator=(const fixed_vector<T,size>& other)
    {
        // Вся работа делается здесь:
        fixed_vector<T,size> temp(other);
        // Здесь исключения не генерируются:
        Swap(temp); return *this;
    }

    iterator         begin()       { return v_; }
    iterator         end()         { return v_ + size; }
    const_iterator   begin() const { return v_; }
    const_iterator   begin() const { return v_ + size; }
private:
    T * v_;
};
```

Распространенная ошибка

Никогда не задумывайтесь о безопасности исключений после разработки класса. Безопасность исключений влияет на устройство класса и никогда не оказывается "просто деталью реализации".

Резюмируем: надеюсь, что эта задача убедила вас в удобстве применения шаблонных функций-членов. Я также надеюсь, что она помогла вам понять, почему они так широко используются в стандартной библиотеке. Если вы еще не знакомы с ними как следует, не беспокойтесь. Не все компиляторы поддерживают шаблонные члены, но поскольку это стандарт C++, то поддержка этой возможности всеми компиляторами в будущем неизбежна.

Используйте шаблонные члены при разработке ваших собственных классов, и вы осчастливите большинство своих пользователей, которые смогут повторно использовать код, создававшийся с учетом такой возможности.

Задача 1.6. Временные объекты *Сложность: 5*

Необязательные и/или временные объекты часто "виновны" в том, что вся ваша тяжелая работа (и производительность программы) идут насмарку. Как же выявить наличие этих объектов в программе и избежать их?

("Временные объекты? — можете удивиться вы. — Но это же вопрос оптимизации? Какое отношение имеют временные объекты к обобщенному программированию и стандартной библиотеке?" Поверьте мне: причина очень скоро станет вам понятна.)

Вы просматриваете код программы. Программист написал следующую функцию, в которой необязательные временные объекты используются как минимум в трех местах. Сколько таких мест вы найдете, и каким образом можно избежать наличия этих временных объектов?

```
string FindAddr( list<Employee> emps, string name)
{
    for(list<Employee>::iterator i = emps.begin();
        i != emps.end(); i++)
    {
        if (*i == name)
        {
            return i->addr;
        }
    }
    return "";
}
```

Не изменяйте действующую семантику этой функции, даже если вы видите пути ее усовершенствования.

Решение

Поверите вы или нет, но эта короткая функция содержит три очевидных случая использования необязательных временных объектов, два немного более скрытых и два ложных.

Два наиболее очевидных временных объекта скрыты в самом объявлении функции.

```
string FindAddr( list<Employee> emps , string name )
```

Параметры должны передаваться функции не по значению, а как константные ссылки, т.е. как `const list<Employee>&` и `const string&` соответственно. Передача параметров по значению заставляет компилятор выполнять полное копирование обоих объектов, что может оказаться весьма дорогой, и к тому же в данном случае совершенно ненужной операцией.

Рекомендация

Предпочтительно передавать объекты по ссылке, как `const&`, вместо передачи их по значению.

Третий очевидный случай использования устранимого временного объекта встречается в условии завершения цикла `for`.

```
for( /* ... */; i != emps.end(); /* ... */ )
```

Для большинства контейнеров (включая `list`) вызов `end()` возвращает временный объект, который должен быть создан и уничтожен. Поскольку это значение не изме-

няется, его вычисление (а также создание и уничтожение) при каждой итерации цикла чрезвычайно неэффективно, а кроме того, просто не эстетично. Это значение должно быть однократно вычислено и сохранено в локальном объекте, после чего оно может многократно использоваться в цикле.

Рекомендация

Если некоторое значение остается неизменным, лучше вычислить его заранее и сохранить, чем постоянно вычислять его, создавая при этом излишние объекты.

Теперь рассмотрим инкремент значения `i` в цикле `for`.

```
for( /* ... */; i++ )
```

Здесь использование временного объекта более скрытое, но вполне понятное, если вспомнить разницу между операторами префиксного и постфиксного инкремента. Постфиксный инкремент обычно менее эффективен, чем префиксный, поскольку он должен запомнить и вернуть начальное значение переменной. Обычно для класса `T` постфиксный инкремент реализуется с использованием следующей канонической формы.

```
const T T::operator++(int)()
{
    T old(*this);  // Запоминаем начальное значение
    ++*this;       // Всегда реализуем постфиксный
                   // инкремент посредством префиксного
    return old;    // Возвращаем начальное значение
}
```

Теперь очень легко увидеть, почему постфиксный инкремент менее эффективен, чем префиксный. Постфиксный инкремент выполняет ту же работу, что и префиксный, но, кроме того, он должен сконструировать и вернуть объект, содержащий начальное значение.

Рекомендация

Для согласованности всегда реализуйте постфиксный инкремент посредством префиксного; в противном случае ваши пользователи получат неожиданные (и, скорее всего, неприятные) результаты.

В тексте задачи начальное значение объекта `i` при его инкременте не используется, поэтому причин для использования постфиксного инкремента нет, и следует воспользоваться префиксным инкрементом. Во врезке “Когда компилятор может оптимизировать постфиксный инкремент” рассказано, почему в общем случае компилятор не может сделать эту замену за вас автоматически.

Рекомендация

Предпочтительно использовать префиксный инкремент вместо постфиксного. Используйте постфиксный инкремент только в том случае, если вам требуется начальное значение объекта.

```
if ( *i == name )
```

Класс `Employee` в задаче не показан, однако мы можем сделать о нем некоторые умозаключения. Чтобы этот код работал, класс `Employee` должен быть снабжен оператором приведения типа к `string`, либо конструктором, в качестве параметра получающего значение типа `string`. В обоих случаях создается временный объект, вызывающий либо оператор сравнения `operator==()` класса `string`, либо оператор сравнения `operator==()` класса `Employee` (временный объект может оказаться не нужен,

если имеется преобразование типа `Employee` к ссылке `string&` или имеется оператор сравнения объектов `Employee` и `string`).

Рекомендация

Следите за скрытыми временными объектами, создаваемыми при неявных преобразованиях. Один из неплохих способов избежать такого создания — по возможности использовать объявление конструкторов как `explicit` и избегать написания операторов преобразования типов.

Когда компилятор может оптимизировать постфиксный инкремент

Допустим, что вы записали выражение с постфиксным оператором инкремента типа "`i++`", но не используете возвращаемое им значение. Может ли компилятор заметить это и просто заменить постфиксный оператор на префиксный с целью оптимизации?

Ответ однозначен — нет, по крайней мере, не в общем случае. Единственная ситуация, когда компилятор может оптимизировать постфиксный инкремент, заменив его префиксным, — при использовании встроенных и стандартных типов, таких как `int` или `complex`, семантика которых известна компилятору в силу их стандартности.

Однако семантика постфиксного и префиксного инкремента пользовательских классов компилятору не известна, более того, они могут в действительности выполнять совершенно разные действия. Поскольку компилятор не может полагаться на то, что программист в достаточной степени последователен и разработал согласованные постфиксный и префиксный операторы инкремента, в общем случае он не может заменить один другим.

Имеется один способ, который вы можете преднамеренно применить для того, чтобы позволить компилятору увидеть связь между префиксным и постфиксным операторами. Для этого вы реализуете постфиксный оператор в каноническом виде, с вызовом префиксного оператора, и объявляете его `inline`, чтобы компилятор мог "заглянуть" за границы функции (при соблюдении директивы `inline`) и обнаружить неиспользуемый временный объект. Я не рекомендую этот метод, поскольку директива `inline` ни в коей мере не может служить панацеей, более того, она может быть попросту проигнорирована компилятором. Гораздо более простое решение состоит в использовании префиксного оператора там, где вас не интересует исходное значение объекта, и описанная оптимизация вам попросту не потребуется.

```
return i->addr;
return "";
```

Вот первый ложный случай. Обе эти инструкции создают временные объекты типа `string`, но избежать этого невозможно.

Я не раз слышал людей, доказывавших, что лучше объявить локальный объект `string` для возвращаемого значения и использовать в функции единственную инструкцию `return`, возвращающую этот объект.

```
string ret;
...
ret = i->addr;
break;
...
return ret;
```

Хотя использование технологии одного входа и одного выхода (single-entry/single-exit, SE/SE) часто приводит к более удобочитаемому (а иногда и более быстрому) коду, фактическое влияние ее на производительность зависит от конкретного кода и используемого компилятора.

В этом случае проблема заключается в том, что создание локального объекта `string` с последующим присваиванием означает вызов конструктора `string` по умолчанию, а затем — оператора присваивания, в то время как в исходном варианте используется только один конструктор. “Но, — можете спросить вы, — насколько дорого обойдется вызов простого конструктора `string` по умолчанию?” Ну что ж, вот результаты исследования производительности версии с двумя `return`, выполненного с помощью одного популярного компилятора.

- При отключенной оптимизации — на 5% быстрее, чем возврат значения объекта `string`.
- При максимальной оптимизации — на 40% быстрее, чем возврат значения объекта `string`.

```
string FindAddr( /* ... */ )
```

Еще один ложный случай. Может показаться, что можно избежать появления временного объекта во всех инструкциях `return`, просто объявив возвращаемый тип как `string&` вместо `string`. В общем случае это ошибка! Если вам повезет, ваша программа аварийно остановится при первом же обращении к возвращенной ссылке, поскольку локальный объект, на который она ссылается, больше не существует. При невезении создастся впечатление работающего кода, и сбой программы будет происходить в каком-то другом месте, что заставит вас провести несколько бессонных ночей в компании с отладчиком.

Рекомендация

Следите за временем жизни объектов. Никогда, ***никогда,*** **никогда** *не возвращайте указатель или ссылку на локальный автоматический объект; это совершенно бесполезно, поскольку вызывающий код не может их использовать, но (что еще хуже) может попытаться это сделать.*

Хотя я не намерен поступать так в дальнейшем, я чувствую себя обязанным показать вариант рассматриваемой функции, которая может возвращать ссылку и тем самым избежать создания временного объекта. Вкратце:

```
const string&
FindAddr(/* Передаем epms и name по ссылке */)
{
    for(/* ... */)
    {
        if(i->name == name)
        {
            return i->addr;
        }
    }
    static const string empty;
    return empty;
}
```

Конечно, документация функции должна теперь определять время жизни ссылки. Если объект найден, мы возвращаем ссылку на объект типа `string` внутри объекта `Employee`, находящегося в объекте `list`, так что ссылка корректна только на время жизни объекта `Employee`, находящегося в списке. Кроме того, значение этой строки может быть изменено при последующем изменении объекта `Employee`.

Мне бы не хотелось использовать такой метод в дальнейшем, так как очень легко забыть обо всех этих ограничениях и наткнуться на неприятности — например, в следующем фрагменте.

```
string& a = FindAddr(emps, "John Doe");
emps.clear();
cout << a;  // Ошибка!
```

Когда ваш пользователь делает нечто подобное и использует ссылку за пределами ее времени жизни, ошибка обычно дает о себе знать только через некоторое время, и диагностировать ее оказывается чрезвычайно трудно. Кстати, одной из весьма распространенных ошибок программистов при работе со стандартной библиотекой является использование итераторов после того, как они перестают быть действительными, что аналогично описанному использованию недействительной ссылки (см. задачу 1.1, где говорится о некорректном использовании итераторов).

Имеются и другие возможности оптимизации кода, приведенного в условии задачи. Не будем обращать на них внимания и приведем исправленную версию функции, в которой устранены только все необязательные временные объекты. Заметьте, что поскольку параметр `list<Employee>` теперь константная ссылка, в теле функции мы должны использовать `const_iterator`.

```
string FindAddr( const list<Employee>& emps,
                 const string&         name )
{
    list<Employee>::const_iterator end(emps.end());
    for(list<Employee>::const_iterator i = emps.begin();
        i != end;
        ++i)
    {
        if (i->name == name)
        {
            return i->addr;
        }
    }
    return "";
}
```

Скольких ловушек в этой задаче можно было бы избежать, если бы программист использовал алгоритм стандартной библиотеки вместо создания вручную собственного цикла? Продемонстрируем это (как и в задаче 1.6, не изменяйте семантику функции, даже если функция может быть улучшена).

Итак, еще раз приведем исправленную функцию.

```
string FindAddr( const list<Employee>& emps,
                 const string&         name )
{
```

Задача 1.7. Использование стандартной библиотеки (или еще раз о временных объектах) *Сложность: 5*

Важной частью разработки программного обеспечения является эффективное повторное использование кода. Для демонстрации того, насколько лучше использовать алгоритмы стандартной библиотеки вместо написания собственных, рассмотрим еще раз предыдущую задачу. Вы увидите, какого количества проблем можно избежать, просто воспользовавшись повторным использованием кода, доступного в стандартной библиотеке.

```
    list<Employee>::const_iterator end(emps.end());
    for(list<Employee>::const_iterator i = emps.begin();
        i != end;
        ++i)
    {
```

```
        if (i->name == name)
        {
            return i->addr;
        }
    }
    return "";
}
```

Решение

Простое использование стандартного алгоритма `find()`, без каких-либо иных изменений исходного кода, позволяет избежать использования двух временных объектов и итеративного вычисления `emps.end()`.

```
string FindAddr( list<Employee> emps,
                 string         name)
{
    list<Employee>::iterator
        i(find( emps.begin(), emps.end(), name ));
    if (i != emps.end())
    {
        return i->addr;
    }
    return "";
}
```

Конечно, еще более красивый результат можно получить при использовании функторов и функции `find_if`, но даже повторное использование простейшей функции `find` сохраняет усилия программиста и высокую эффективность программы.

Рекомендация

Вместо написания собственного кода повторно используйте имеющийся, в особенности код стандартной библиотеки. Это быстрее, проще и безопаснее.

Повторное использование существующего кода обычно предпочтительнее написания собственного. Стандартная библиотека переполнена кодом, предназначенным для использования. Этот код стоил авторам многих часов труда в поте лица над повышением его практичности, эффективности работы и соответствия множеству других требований. Так что смело используйте код стандартной библиотеки и старайтесь избегать изобретения велосипеда.

Комбинируя решения, предложенные ранее, с решением данной задачи, мы получаем окончательный вариант функции.

```
string FindAddr( const list<Employee>& emps,
                 const string&         name)
{
    list<Employee>::const_iterator
        i(find( emps.begin(), emps.end(), name ));
    if (i != emps.end())
    {
        return i->addr;
    }
    return "";
}
```

Задача 1.8. Переключение потоков ***Сложность: 2***

Каким образом лучше всего использовать в программе различные потоки ввода и вывода, включая стандартные потоки ввода-вывода и файлы?

1. Какие типы имеют `std::cin` и `std::cout`?
2. Напишите программу `ECHO`, которая просто повторяет вводимую информацию и одинаково работает при двух следующих вариантах вызова.

```
ECHO  <infile  >outfile
ECHO   infile   outfile
```

В большинстве популярных сред командной строки первая команда обозначает, что программа получает ввод из `cin` и направляет вывод в `cout`. Вторая команда указывает программе, что вводимую информацию она должна брать из файла с именем `infile`, а вывод осуществлять в файл с именем `outfile`. Ваша программа должна поддерживать оба варианта указания ввода-вывода.

Решение

1. **Какие типы имеют `std::cin` и `std::cout`?**

Краткий ответ звучит так: тип `cin` —

```
std::basic_istream<char, std::char_traint<char> >
```

а `cout` —

```
std::basic_ostream<char, std::char_traint<char> >
```

Более длинный ответ показывает связь между стандартными синонимами, определяемыми посредством `typedef`, и шаблонами. `cin` и `cout` имеют типы `std::istream` и `std::ostream` соответственно. В свою очередь, они определены посредством объявления `typedef` как `std::basic_istream<char>` и `std::basic_ostream<char>`. И наконец, с учетом аргументов шаблонов по умолчанию мы получим приведенный ранее краткий ответ.

Примечание: если вы используете раннюю, до принятия стандарта реализацию подсистемы потоков ввода-вывода, вы можете встретиться с промежуточными классами типа `istream_with_assign`. В стандарте этих классов нет.

2. **Напишите программу `ECHO`, которая просто повторяет вводимую информацию и одинаково работает при двух следующих вариантах вызова.**

```
ECHO  <infile  >outfile
ECHO   infile   outfile
```

Краткое решение

Для любителей краткости приводим решение, состоящее из одной инструкции.

```
// Пример 1: решение в одну инструкцию
//
#include <fstream>
#include <iostream>

int main(int argc, char* argv[])
{
    using namespace std;
```

```
    (argc > 2
     ? ofstream(argv[2], ios::out | ios::binary)
     : cout)
    <<
    (argc > 1
     ? ifstream(argv[1], ios::in | ios::binary)
     :cin)
    .rdbuf();
}
```

Эта программа использует кооперацию двух возможностей потоков. Во-первых, basic_ios предоставляет функцию-член rdbuf(), которая возвращает объект streambuf, используемый внутри данного объекта потока (в нашем случае — cin или временный поток ifstream (связанный с файлом); они оба являются потомками basic_ios). Во-вторых, basic_ostream предоставляет operator<<(), принимающий в качестве ввода описанный объект basic_streambuf, который затем успешно считывает до конца. Вот и все.

Более гибкое решение

У подхода из первого примера имеется два главных недостатка. Во-первых, краткость должна иметь границы, и чрезмерная краткость не годится при разработке кода.

Рекомендация

Предпочитайте удобочитаемость. Избегайте чрезмерно краткого кода (краткого, но трудного для понимания и сопровождения). Избегайте крайностей!

Во-вторых, хотя первый пример и ответил на поставленный вопрос, он годится только для случая дословного копирования входной информации. Этого может оказаться достаточно сегодня, но что если завтра нам потребуется некоторая обработка входной информации? Что, если нам понадобится заменить все строчные буквы на прописные или удалить каждую третью букву? Пожалуй, стоит подумать о будущем и инкапсулировать обработку в отдельной функции, которая может использовать объекты потоков полиморфно.

```
#include <fstream>
#include <iostream>

int main(int argc, char* argv[])
{
    using namespace std;

    fstream in, out;
    if (argc > 1) in.open (argv[1], ios::in  | ios::binary);
    if (argc > 2) out.open(argv[2], ios::out | ios::binary);

    Process( in.is_open()  ? in  : cin,
             out.is_open() ? out : cout );
}
```

Но как нам реализовать функцию Process()? В C++ имеется четыре основных способа получить полиморфное поведение: виртуальные функции, шаблоны, перегрузка и преобразования типов. Первые два метода непосредственно применимы в нашей ситуации для выражения требующегося нам полиморфизма.

Шаблоны (полиморфизм времени компиляции)

Первый способ состоит в использовании полиморфизма времени компиляции посредством шаблонов, что требует просто передачи объектов с соответствующим интерфейсом (таким как функция-член `rdbuf()`).

```
//   Пример 2а: шаблонная функция Process
//
template<typename In, typename Out>
void Process(In& in, Out& out)
{
    //  Выполняется нечто более сложное,
    //  чем "out << in.rdbuf()"
}
```

Виртуальные функции (полиморфизм времени выполнения)

Второй способ — применение полиморфизма времени выполнения, который использует факт существования общего базового класса с соответствующим интерфейсом.

```
//   Пример 2б: первая попытка использования
//              виртуальности
void Process(basic_istream<char>& in,
             basic_ostream<char>& out)
{
    //  Выполняется нечто более сложное,
    //  чем "out << in.rdbuf()"
}
```

Заметим, что в этом примере параметры функции `Process()` не принадлежат типу `basic_ios<char>&`, поскольку использование этого типа не позволило бы использовать `operator<<()`.

Конечно, подход, представленный в примере *2б*, опирается на то, что входной и выходной потоки порождены из классов `basic_istream<char>` и `basic_ostream<char>`. Этого вполне достаточно для нашей задачи, но не все потоки основаны на типе `char`, и даже не на `char_traits<char>`. Например, могут использоваться потоки, базирующиеся на типе `wchar_t`, или (вспомните задачи 1.2 и 1.3) на символах с измененным пользователем поведением (как вы помните, мы использовали `ci_char_traits` для обеспечения нечувствительности к регистру).

Таким образом, еще более гибкое решение получится, если мы используем шаблоны в примере *2б*.

```
//   Пример 2в: более гибкое решение
//
template<typename C = char, typename T = char_traits<C> >
void Process(basic_istream<C,T>& in,
             basic_ostream<C,T>& out)
{
    //  Выполняется нечто более сложное,
    //  чем "out << in.rdbuf()"
}
```

Некоторые принципы проектирования

Все приведенные ответы на поставленный в условии задачи вопрос верны, но все же в этой ситуации лично я предпочитаю решение с шаблонами. Это связано с двумя полезными рекомендациями.

Рекомендация

Предпочитайте расширяемость.

Избегайте написания кода, решающего только одну непосредственно стоящую перед вами задачу. Расширенное решение почти всегда лучше — естественно, пока мы остаемся в разумных рамках. Один из показателей профессионализма программиста — его способность правильно выбрать компромисс между расширяемостью и перегруженностью решения. Такой программист понимает, как обеспечить правильный баланс между написанием специализированного кода, решающего одну непосредственно стоящую перед ним задачу, и созданием грандиозного проекта для решения всех задач, которые только могут встретиться в данной области.

По сравнению с примером 1 метод с шаблонами более сложен, но и более прост для понимания, а также легко расширяем. По сравнению с виртуальными функциями применение шаблонов проще и гибче; кроме того, это решение в большей степени адаптируется к новым ситуациям, поскольку оно избегает жесткой привязки к иерархии потоков ввода-вывода.

Итак, если у нас имеется выбор из двух вариантов, требующих одинаковых усилий при разработке и реализации и практически одинаковых с точки зрения ясности и поддержки, следует предпочесть расширяемость. Такое предпочтение не означает индульгенцию на перегрузку решения всеми возможными и невозможными расширениями и возможностями, к чему часто приходят новички. Это совет всего лишь подумать (и сделать) немного больше, чем того требует решение непосредственно стоящей перед вами задачи, и попытаться определить, частным случаем какой более общей проблемы она является. Ценность этого совета еще и в том, что проектирование с учетом расширяемости часто неявно означает проектирование с учетом инкапсуляции.

Рекомендация

Каждая часть кода — каждый модуль, класс, функция — должны отвечать за выполнение единой, четко определенной задачи.

Насколько это возможно, каждая часть кода — функция или класс — должна быть сконцентрирована и отвечать за выполнение только строго определенного круга задач.

Метод с использованием шаблонов, пожалуй, в большей степени отвечает приведенной рекомендации. Код, осведомленный о различиях в источниках и получателях ввода-вывода, отделен от кода, который осуществляет операции ввода-вывода. Такое разделение делает код более ясным, более простым для чтения и восприятия человеком. Умелое разделение задач является вторым показателем профессионализма программиста, и мы еще не раз вернемся к этому вопросу при решении наших задач.

Задача 1.9. Предикаты. Часть 1 — *Сложность: 4*

Эта задача позволяет вам проверить ваши знания о стандартных алгоритмах. Что в действительности делает алгоритм `remove()` стандартной библиотеки и как бы вы написали функцию, удаляющую из контейнера третий элемент?

1. Как функционирует алгоритм `std::remove()`? Будьте максимально точны.
2. Напишите код, удаляющий все значения, равные трем, из `std::vector<int>`.
3. Программист, работающий в вашей команде, написал следующие варианты кода для удаления *n*-го элемента контейнера.

```
// Метод 1. Разработка специализированного алгоритма
template<typename FwdIter>
FwdIter remove_nth(FwdIter first, FwdIter last, size_t n)
{
    /* ... */
}
```

```
// Метод 2. Написание объекта-функции,
// возвращающего true при n-м применении, и
// использовании его как предиката для remove_if

class FlagNth
{
public:
    FlagNth(size_t n): current_(0), n_(n) {}
    template<typename T>
    bool operator() (const T&) { return ++current_ == n_; }

private:
    size_t       current_;
    const size_t n_;
};

//
... remove_if(v.begin(), v.end(), FlagNth(3)) ...
```

а) Реализуйте недостающую часть Метода 1.

б) Какой метод лучше? Почему? Рассмотрите все возможные проблемы, которые могут возникнуть в том или ином решении.

Решение

Что удаляет remove()

1. Как функционирует алгоритм `std::remove()`? Будьте максимально точны.

Стандартный алгоритм `remove()` физически не удаляет объекты из контейнера; после того как `remove()` завершает свою работу, размер контейнера не изменяется. Вместо изменения размера `remove()` перемещает “неудаленные” объекты для заполнения промежутка, образовавшегося из-за удаления объектов, оставляя в конце контейнера для каждого удаленного объекта по одному “мертвому” объекту. И наконец, `remove()` возвращает итератор, указывающий на первый “мертвый” объект (или, если ни один объект не был удален, итератор `end()`).

Рассмотрим, например, `vector<int> v`, который содержит следующие девять элементов.

```
1 2 3 1 2 3 1 2 3
```

Допустим, мы используем следующий код для того, чтобы удалить все тройки из контейнера.

```
// Пример 1
remove(v.begin(), v.end(), 3); // Не совсем верно
```

Что же происходит? Ответ выглядит примерно так.

```
1  2  1  2  1  2  ?  ?  ?
  Не удалены     "Мертвые"
                  объекты
                  ↑ - итератор,
 возвращаемый remove(), указывает
 на третий с конца объект (так как
 удаляются три элемента)
```

Три объекта удаляются, а остальные сдвигаются таким образом, чтобы заполнить возникшие промежутки. Объекты в конце контейнера могут иметь их первоначальные

значения (1 2 3), но это не обязательно так. Еще раз обратите внимание на то, что размер контейнера остался неизменным.

Если вас удивляет, почему `remove()` работает именно так, вспомните, что этот алгоритм работает не с контейнером, а с диапазоном итераторов, а такой операции, как “удалить элемент, на который указывает итератор, из контейнера, в котором он находится”, не имеется. Чтобы выполнить такую операцию, следует обратиться к контейнеру непосредственно. За дополнительной информацией о `remove()` вы можете обратиться к [Koenig99].

2. Напишите код, удаляющий все значения, равные трем, из `std::vector<int>`.

Вот как можно сделать это (здесь `v` — объект типа `vector<int>`).

```
// Пример 2. Удаление троек из vector<int> v
v.erase(remove(v.begin(), v.end(), 3), v.end());
```

Вызов `remove(v.begin(), v.end(), 3)` выполняет основную работу и возвращает итератор, указывающий на первый “мертвый” элемент. Вызов `erase()` с диапазоном от этого элемента и до `v.end()` избавляется от всех мертвых элементов, так что по окончании его работы вектор содержит только неудаленные объекты.

3. Программист, работающий в вашей команде, написал следующие варианты кода для удаления *n*-го элемента контейнера.

```
// Пример 3а
// Метод 1. Разработка специализированного алгоритма
template<typename FwdIter>
FwdIter remove_nth(FwdIter first, FwdIter last, size_t n)
{
    /* ... */
}

// Пример 3б
// Метод 2. Написание объекта-функции,
// возвращающего true при n-м применении, и
// использовании его как предиката для remove_if

class FlagNth
{
public:
    FlagNth(size_t n): current_(0), n_(n) {}
    template<typename T>
    bool operator() (const T&) { return ++current_ == n_; }

private:
    size_t       current_;
    const size_t n_;
};

//
... remove_if(v.begin(), v.end(), FlagNth(3)) ...
```

а) Реализуйте недостающую часть Метода 1.

Люди часто предлагают реализации, содержащие ту же ошибку, что и приведенный далее код. А вы?

```
// Пример 3в. Видите ли вы ошибку?
template<typename FwdIter>
FwdIter remove_nth(FwdIter first, FwdIter last, size_t n)
{
    for(; n>0; ++first, --n)
        ;
    if (first != last)
```

```
    {
        FwdIter dest = first;
        return copy(++first, last, dest);
    }
    return last;
}
```

В примере 3*в* есть одна проблема, имеющая два аспекта.

1. Корректное предусловие. Мы не требуем, чтобы `n<=distance(first,last)`, так что начальный цикл может переместить `first` после `last`, и тогда `[first,last)` перестанет быть корректным диапазоном итераторов. Если это случится, то в остальной части функции могут произойти Ужасные Вещи.
2. Эффективность. Скажем, мы решили в качестве решения первой проблемы документировать (и проверять!) предусловие, состоящее в том, чтобы `n` было корректным для данного диапазона. Тогда мы можем обойтись без цикла и просто записать `advance(first,n)`. Стандартный алгоритм `advance()` для итераторов осведомлен о различных категориях итераторов и оптимизирован для итераторов произвольного доступа. В частности, в случае итераторов произвольного доступа выполнение алгоритма должно занимать постоянное время, в отличие от линейной зависимости времени выполнения для итераторов других типов.

   Вот более корректная реализация приведенного ранее кода.

```
// Пример 3г. Окончательное решение задачи.
// Предусловие: n не должно превышать размер диапазона
template<typename FwdIter>
FwdIter remove_nth(FwdIter first, FwdIter last, size_t n)
{
    // Проверка выполнения предусловия
    assert(distance(first,last) >= n);

    // Выполнение задания
    advance(first,n);
    if (first != last)
    {
        FwdIter dest = first;
        return copy(++first, last, dest);
    }
    return last;
}
```

б) Какой метод лучше? Почему? Рассмотрите все возможные проблемы, которые могут возникнуть при том или ином решении.

Метод 1 обладает следующими преимуществами.

1. Он корректен.
2. Он может использовать свойства конкретных итераторов, в частности их класс, и таким образом лучше работать в случае итераторов произвольного доступа.

Метод 2 имеет недостатки, соответствующие преимуществам метода 1, — об этом мы поговорим в следующей задаче.

Задача 1.10. Предикаты. Часть 2 — *Сложность: 7*

Решив задачу 1.9, рассмотрим предикаты с состояниями. Что это такое? Когда их следует применять? Насколько они совместимы со стандартными контейнерами и алгоритмами?

1. Что такое предикаты и как они используются в STL? Приведите пример.

2. Когда предикаты с состояниями могут оказаться полезными? Приведите примеры.
3. Какие требования к алгоритмам следует предъявлять, чтобы обеспечить корректную работу предикатов?

Решение

Унарные и бинарные предикаты

1. Что такое предикаты и как они используются в STL?

Предикат является указателем на функцию, либо объектом-функцией (т.е. объектом, в котором определен оператор вызова функции, `operator()()`), который на вопрос об объекте дает ответ “да” или “нет”. Многие алгоритмы используют предикаты для принятия решения о каждом объекте, с которым они работают, так что предикат `pred` должен корректно работать при использовании следующим образом.

```
// Пример 1а. Использование унарного предиката.

if (pred(*first))
{
    /* ... */
}
```

Как видно из этого примера, предикат `pred` должен возвращать значение, которое может быть проверено на истинность. Обратите внимание, что при применении разыменованного итератора предикат может использовать только `const`-функции.

Некоторые предикаты являются бинарными, т.е. в качестве аргументов они принимают два объекта (зачастую — два разыменованных итератора). Это означает, что бинарный предикат `bpred` должен корректно работать в следующей ситуации.

```
// Пример 1б. Использование бинарного предиката

if (bpred(*first1, *first2))
{
    /* ... */
}
```

Приведите пример.

Рассмотрим следующую реализацию стандартного алгоритма `find_if()`.

```
// Пример 1в. Пример find_if

template<typename Iter, typename Pred> inline
Iter find_if(Iter first, Iter last, Pred pred)
{
    while(first != last && !pred(*first))
    {
        ++first;
    }
    return first;
}
```

Эта реализация алгоритма работает поочередно со всеми элементами из диапазона `[first,last)`, применяя указатель на функцию-предикат (или объект) `pred` к каждому элементу. Если имеется элемент, для которого предикат возвращает значение `true`, `find_if()` возвращает итератор, указывающий на первый такой элемент. В противном случае `find_if()` возвращает `last` как указание на то, что элемент, удовлетворяющий предикату, не найден.

Мы можем использовать find_if() с указателем на функцию предиката следующим образом.

```
// Пример 1г. Использование find_if с указателем на функцию.

bool GreaterThanFive(int i)
{
    return i>5;
}
bool IsAnyElementGreaterThanFive(vector<int>&v)
{
    return find_if(v.begin(), v.end(), GreaterThanFive)
           != v.end();
}
```

Вот этот же пример, в котором вместо указателя на функцию используется объект-функция.

```
// Пример 1д. Использование find_if с объектом-функцией.

class GreaterThanFive
    :public std::unary_function<int,bool>
{
public:
    bool operator()(int i) const
    {
        return i>5;
    }
};
bool IsAnyElementGreaterThanFive(vector<int>&v)
{
    return find_if(v.begin(), v.end(), GreaterThanFive())
           != v.end();
}
```

В этом примере не слишком много преимуществ использования объектов-функций по сравнению с обычными функциями, не так ли? Но впереди у нас второй вопрос задачи, при рассмотрении которого вы увидите всю гибкость объектов-функций.

2. Когда предикаты с состояниями могут оказаться полезными? Приведите примеры.

Приведенный ниже код является развитием примера 1*д*. Аналогичное решение с помощью обычных функций будет не столь простым и потребует использования чего-то наподобие статических переменных.

```
// Пример 2а. Использование find_if() с более
// универсальным объектом-функцией.

class GreaterThan
    :public std::unary_function<int,bool>
{
public:
    GreaterThan(int value) : value_(value) {}
    bool operator()(int i) const
    {
        return i > value_;
    }
private:
    const int value_;
};

bool IsAnyElementGreaterThanFive(vector<int>&v)
{
    return find_if(v.begin(), v.end(), GreaterThan(5))
           != v.end();
}
```

Предикат `GreaterThan` имеет член данных, хранящий состояние, которое в нашем случае представляет собой значение, с которым должны сравниваться все элементы. Как видите, приведенная версия существенно более практична и обладает б*о*льшими возможностями повторного использования, чем узкоспециализированный код в примерах 1*г* и 1*д*, и источник этой практичности и гибкости — возможность хранения собственной информации в объекте, подобном рассмотренному.

Сделаем еще один шаг и получим еще более обобщенную версию.

```
// Пример 2б. Использование find_if() с наиболее
// универсальным объектом-функцией.

template<typename T>
class GreaterThan
    :public std::unary_function<T,bool>
{
public:
    GreaterThan(T value) : value_(value) {}
    bool operator()(const T& t) const
    {
        return t > value_;
    }
private:
    const T value_;
};

bool IsAnyElementGreaterThanFive(vector<int>&v)
{
    return find_if(v.begin(), v.end(), GreaterThan<int>(5))
           != v.end();
}
```

Итак, становятся совершенно очевидны преимущества использования предикатов с хранимыми значениями по сравнению с обычными функциями.

Следующий шаг: предикаты с состояниями

Предикаты в примерах 2*а* и 2*б* обладают одним очень важным свойством: копии объектов эквивалентны. То есть, если мы сделаем копию объекта `GreaterThan<int>`, он будет вести себя абсолютно так же, как и оригинал, и может использоваться поочередно с оригиналом. Как мы увидим при ответе на третий вопрос, это оказывается весьма важным.

Некоторые программисты идут дальше и пытаются писать предикаты с состояниями, изменяющимися при использовании этих предикатов, так что результат применения предиката к объекту зависит от истории предыдущих применений предиката. В примерах 2*а* и 2*б* объекты содержали некоторые внутренние значения, однако эти значения фиксируются в момент создания объекта и по сути состояниями, которые могут изменяться в процессе работы объекта, не являются. Когда мы говорим о предикатах с состояниями, мы, в первую очередь, имеем в виду предикаты, состояние которых *может изменяться*, так что объект предиката чувствителен к событиям, происходящим с ним во время его существования, подобно конечному автомату[1].

Примеры таких предикатов встречаются в книгах. В частности, используются предикаты, отслеживающие различную информацию об элементах, к которым они применяются. Например, предлагались предикаты, запоминающие значения объектов для проведения некоторых вычислений (типа предиката, возвращающего значение `true`,

[1] Как элегантно описал ситуацию Джон Хикин (John D. Hickin): “Вход `[first,last)` аналогичен ленте машины Тьюринга, а предикат с состояниями — программе”.

пока среднее значение объектов, к которым применялся данный предикат, больше 50, но меньше 100 и т.п.). Пример такого предиката мы только что видели в задаче 1.9.

```
// Пример 2в (пример 3б из задачи 1.9)
//
// Метод 2. Написание объекта-функции,
// возвращающего true при n-м применении, и
// использовании его как предиката для remove_if

class FlagNth
{
public:
    FlagNth(size_t n): current_(0), n_(n) {}
    template<typename T>
    bool operator() (const T&) { return ++current_ == n_; }

private:
    size_t       current_;
    const size_t n_;
};
```

Предикаты с состояниями, наподобие только что рассмотренного, чувствительны к способу их применения к элементам диапазона, с которым они работают, — в частности, их функционирование зависит как от количества применений предиката, так и от порядка его применения к элементам диапазона (если предикат используется в алгоритме типа `remove_if()`).

Основное отличие предикатов с состояниями от обычных предикатов состоит в том, что копии предикатов с состояниями *не* эквивалентны. Очевидно, что алгоритм не может сделать копию объекта `FlagNth` и применить один объект к некоторой части элементов, а второй — к оставшимся элементам. Это не приведет к желаемому результату, поскольку объекты предикатов будут обновлять свои счетчики независимо, и ни один из них не сможет корректно выделить *n*-й элемент — каждый из них в состоянии выделить только *n*-й обработанный им элемент.

Проблема заключается в том, что в примере 2*в* метод 2 может попытаться использовать объект `FlagNth` следующим образом.

```
... remove_if(v.begin(), v.end(), FlagNth(3)) ...
```

“Выглядит вполне разумно, и я бы использовал эту методику”, — скажут одни. “Я только что прочел книгу по C++, в которой используется подобная методика, так что все должно быть в порядке”, — скажут другие. Истина же в том, что этот метод вполне может работать в вашей реализации (или в реализации автора книги с ошибками), но это *не гарантирует* корректную переносимую работу этого метода во всех реализациях (и даже в очередной версии реализации, используемой вами или автором книги в настоящее время). С чем это связано, вы поймете, разобрав `remove_if()` более детально.

3. Какие требования к алгоритмам следует предъявлять, чтобы обеспечить корректную работу предикатов?

Чтобы предикаты с состояниями были действительно применимы в алгоритме, алгоритм в общем случае должен гарантировать выполнение определенных условий при работе с предикатами.

а) Алгоритм не должен делать копий предиката (т.е. он должен постоянно использовать один и тот же переданный ему объект).

б) Алгоритм должен применять предикат к элементам диапазона в некотором предопределенном порядке (например, от первого до последнего).

Увы, стандарт не требует от стандартных алгоритмов соответствия этим требованиям. Несмотря на появление предикатов с состояниями в книгах, в “битве” книг и

стандарта победителем неизменно выходит стандарт. Стандарт выдвигает другие требования к стандартным алгоритмам, такие как временн*ая* сложность алгоритма или количество применений предиката, но в частности, он никогда не накладывал ни на один алгоритм требования (а).

Рассмотрим, например, `std::remove_if()`.

а) Достаточно распространен способ реализации `remove_if()` с использованием `find_if()`; при этом предикат передается в `find_if()` по значению. Это приводит к неожиданному поведению, поскольку объект предиката, переданный алгоритму `remove_if()`, не обязательно применяется к каждому элементу диапазона. Вместо этого гарантируется, что к каждому элементу будет применен объект предиката, *либо его копия*, поскольку стандарт позволяет алгоритму `remove_if()` предполагать эквивалентность копий предикатов.

б) Стандарт требует, чтобы предикат, переданный алгоритму `remove_if()`, был применен в точности `last-first` раз, но не указывает, в каком порядке. Вполне можно разработать реализацию `remove_if()`, удовлетворяющую стандарту, которая не будет применять предикаты к элементам в определенном порядке. Главное правило: если что-то стандартом не требуется, то рассчитывать на это “что-то” нельзя.

“Хорошо, — скажете вы, — но, может, имеется какой-то способ надежного использования предикатов с состояниями (типа `FlagNth`) в стандартных алгоритмах?” К сожалению, ответ на этот вопрос отрицательный.

Да, да, я уже слышу возмущенные крики тех, кто написал предикат с использованием технологии счетчиков ссылок и тем самым решил проблему (а). Да, вы можете обеспечить совместное использование состояния предикатов и таким образом обеспечить возможность копирования при применении предикатов без изменения их семантики. Вот код, использующий эту методику (для подходящего шаблона `CountedPtr`; кстати, вот дополнительное задание: разработайте реализацию `CountedPtr`).

```
// Пример 3а. (частичное) решение с
// совместно используемым состоянием

class FlagNthImpl
{
public:
    FlagNthImpl(size_t nn) : i(0), n(nn) {}
    size_t       i;
    const size_t n;
}

class FlagNth
{
public:
    FlagNth(size_t n) :pimpl_(new FlagNthImpl(n))
    {
    }

    template<typename T>
    bool operator()(const T&)
    {
        return ++(pimpl_->i) == pimpl_->n;
    }

private:
    CountedPtr<FlagNthImpl> pimpl_;
};
```

Однако такой подход не решает, да и не может решить указанную ранее проблему (б). Это означает что вы, программист, никак не можете повлиять на порядок применения предиката алгоритмом. Нет никаких обходных путей решения этой проблемы, кроме гарантии порядка обхода элементов самим алгоритмом.

Дополнительное задание

Поставленное ранее дополнительное задание состоит в разработке подходящей реализации `CountedPtr`, интеллектуального указателя, для которого создание новой копии просто указывает на одно и то же представление, а вызов деструктора последней копии уничтожает объект. Вот один из вариантов реализации этого класса (для промышленного использования он может быть существенно улучшен).

```
template<typename T>
class CountedPtr
{
private:
    class Impl
    {
    public:
        Impl(T*pp) : p(pp), refs(1) {}
        ~Impl() { delete p; }
        T*     p;
        size_t refs;
    };
    Impl* impl_;
public:
    explicit CountedPtr(T*p)
        : impl_(new Impl(p)) {}
    ~CountedPtr() { Decrement(); }
    CountedPtr(const CountedPtr& other)
        : impl_(other.impl_)
    {
        Increment();
    }

    CountedPtr& operator = (const CountedPtr& other)
    {
        if (impl_ != other.impl_)
        {
            Decrement();
            impl_ = other.impl_;
            Increment();
        }
        return *this;
    }

    T* operator->() const
    {
        return impl_->p;
    }

    T& operator*() const
    {
        return *(impl_->p);
    }

private:
    void Decrement()
    {
        if (--(impl_->refs) == 0)
        {
            delete impl_;
        }
```

```
    }
    void Increment()
    {
        ++(impl_->refs);
    }
};
```

Задача 1.11. Расширяемые шаблоны: путем наследования или свойств? *Сложность: 7*

В этой задаче рассматриваются шаблоны со свойствами и демонстрируются некоторые интересные методы использования свойств. Даже без использования свойств — что шаблон может знать о своем типе и что он может с ним делать? Ответы на эти вопросы полезны не только тем, кто пишет библиотеки C++.

1. Что такое класс свойств?
2. Покажите, как проверить наличие определенного члена в классе-параметре шаблона в следующей ситуации: вы хотите написать шаблон класса `C`, объект которого может быть создан только для типов, имеющих константную функцию `Clone()`, которая не имеет параметров и возвращает указатель на объект такого же типа.

   ```
   // T должен иметь член T* T::Clone() const
   template<typename T>
   class C
   {
       // ...
   };
   ```

   Примечание: очевидно, что если в составе `C` имеется код, который пытается вызвать `T::Clone()` без параметров, то такой код не будет скомпилирован, если не определена функция `T::Clone()`, которая может быть вызвана без параметров. Однако этого недостаточно для ответа на наш вопрос, поскольку попытка вызова `T::Clone()` без параметров будет успешной и в том случае, когда `Clone()` имеет параметры по умолчанию и/или возвращает тип, отличный от `T*`. Задача состоит в том, чтобы гарантировать, что тип `T` имеет функцию, выглядящую в точности так: `T* T::Clone() const`.
3. Программист намерен написать шаблон, который может потребовать или хотя бы проверить, имеет ли тип, для которого производится инстанцирование, функцию-член `Clone()`. Программист решает сделать это путем требования, чтобы классы с функцией-членом `Clone()` были наследниками класса `Clonable`. Покажите, как следует написать шаблон

   ```
   template<typename T>
   class X
   {
       // ...
   };
   ```

   так, чтобы:

   а) требовалось, чтобы `T` был производным от `Cloneable`, а также

   б) обеспечить реализацию, в которой выполняются одни действия, если `T` — производный от `Cloneable` класс, и некоторые другие действия, если это не так.
4. Является ли подход, описанный в п. 3, наилучшим способом обеспечить требование (обнаружение) наличия `Clone()`? Приведите другие варианты решения.

5. Насколько полезно знание шаблона о том, что его параметр типа `T` является производным от некоторого другого типа? Дает ли такое знание какие-либо преимущества, которые не могут быть достигнуты иначе, без отношений наследования?

Решение

1. Что такое класс свойств?

Процитируем стандарт C++ [C++98], п. 17.1.18: класс свойств — это

> *класс, который инкапсулирует множество типов и функций, необходимых для шаблонных классов и функций для работы с объектами типов, для которых они инстанцируются.*[2]

Идея заключается в том, что классы свойств являются экземплярами шаблонов и используются для хранения дополнительной информации — главным образом информации, которую могут использовать другие шаблоны, — о типах, для которых инстанцируются шаблоны свойств. Самое главное их преимущество в том, что класс свойств `T<C>` позволяет нам записать такую дополнительную информацию о классе `C`, ничего не изменяя в самом классе `C`.

Например, в стандарте шаблон `std::char_traits<T>` предоставляет информацию о символоподобном типе `T`, в частности о том, как сравнивать объекты типа `T` и управлять ими. Эта информация используется в таких шаблонах, как `std::basic_string` и `std::basic_ostream`, позволяя им работать с символьными типами, отличающимися от `char` и `wchar_t`, включая работу с пользовательскими типами, для которых представлена соответствующая специализация шаблона `std::char_traits`. Аналогично, `std::iterator_traits` предоставляет информацию об итераторах, которые могут использоваться другими шаблонами, в частности алгоритмами или контейнерами. Даже `std::numeric_limits` работает как класс свойств, предоставляя информацию о возможностях и поведении различных числовых типов, реализованных в ваших конкретных платформе и компиляторе.

Дополнительную информацию вы можете найти в следующих источниках.

- В задачах 6.5 и 6.6, посвященных интеллектуальным указателям.
- В задачах 1.2 и 1.3, в которых показано, как можно изменить `std::char_traits` для изменения поведения `std::basic_string`.
- В апрельском, майском и июньском номерах *C++ Report* за 2000 год, в которых имеется превосходный материал о классах свойств.

Требование наличия функций-членов

2. Покажите, как проверить наличие определенного члена в классе-параметре шаблона в следующей ситуации: вы хотите написать шаблон класса `C`, объект которого может быть создан только для типов, имеющих константную функцию `Clone()`, которая не имеет параметров и возвращает указатель на объект такого же типа.

```
// T должен иметь член T* T::Clone() const
template<typename T>
class C
```

[2] Здесь термин *инкапсулирует* используется в смысле размещения в одном месте, а не сокрытия. Обычная практика класса свойств, — когда все его члены открыты; более того, обычно классы свойств реализуются как шаблоны структур (`struct`).

```
    {
        // ...
    };
```

Примечание: очевидно, что если в составе `C` имеется код, который пытается вызвать `T::Clone()` без параметров, то такой код не будет скомпилирован, если не имеется функции `T::Clone()`, которая может быть вызвана без параметров. Однако этого недостаточно для ответа на наш вопрос, поскольку попытка вызова `T::Clone()` без параметров будет успешной и в том случае, когда `Clone()` имеет параметры по умолчанию и/или возвращает тип, отличный от `T*`. Задача состоит в том, чтобы гарантировать, что тип `T` имеет функцию, выглядящую в точности так: `T* T::Clone() const`.

В качестве примера, иллюстрирующего последнее замечание, рассмотрим следующий код.

```
// Пример 2а. Первый вариант требования наличия Clone()
// T должен иметь /*...*/ T::Clone(/*...*/)
template<typename T>
class C
{
public:
    void SomeFunc(const T* t)
    {
        // ...
        t->Clone();
        // ...
    }
};
```

Первая проблема в примере 2*а* состоит в том, что в нем не требуется вообще ничего. В шаблоне инстанцируются только те функции, которые реально используются[3]. Если функция `SomeFunc()` никогда не используется, она не будет инстанцирована, так что `C` может оказаться инстанцирован с типом `T`, не имеющим функции-члена `Clone()`.

Решение заключается в добавлении кода, который обеспечивает требование наличия `Clone()`, в функцию, которая будет гарантированно инстанцирована. Многие помещают этот код в конструктор, поскольку невозможно использовать `C`, не вызывая его конструктор, не так ли? (Этот подход упоминается, например в [Stroustrup94].) Достаточно верно, но следует не забывать, что у класса может быть несколько конструкторов, и для безопасности этот код надо поместить в каждый из них. Более простое решение — поместить соответствующий код в деструктор. В конце концов, имеется только один деструктор, и вряд ли класс `C` будет использоваться без его вызова (создавая объект `C` динамически и никогда не удаляя его). Таким образом, вероятно, деструктор — наиболее простое место для размещения кода, требующего наличия функции-члена.

```
// Пример 2б. Второй вариант требования наличия Clone()
// T должен иметь /*...*/ T::Clone(/*...*/)
template<typename T>
class C
{
public:
    ~C()
    {
        // ...
```

[3] В конечном счете это утверждение будет справедливо для любого компилятора. В настоящий момент ваш компилятор может инстанцировать все функции, а не только реально используемые.

```
        const T t; // Расточительно, к тому же
                   // требует наличия конструктора
                   // по умолчанию
        t.Clone();
        // ...
    }
};
```

Это решение также нельзя считать полностью удовлетворительным. Пока что мы оставим его в таком виде, но вскоре мы вернемся к нему и улучшим его, а пока что давайте немного поразмышляем.

Это решение оставляет нерешенной вторую проблему: и пример 2*а*, и пример 2*б* не столько проверяют наличие функции-члена, сколько полагаются на его наличие (причем пример 2*б* еще хуже примера 2*а*, поскольку в нем только для того, чтобы убедиться в наличии функции-члена, выполняется совершенно ненужный код).

Но этого недостаточно для ответа на поставленный вопрос, поскольку попытка вызова `T::Clone()` без параметров будет такой же успешной при вызове функции `Clone()` с параметрами по умолчанию и/или возвращающей не `T*`.

Код в примерах 2*а* и 2*б* успешно выполнится, если имеется функция `T* T::Clone()`. Но не менее успешно он выполнится и при наличии функции `void T::Clone()`, или `T* T::Clone(int = 42)`, или им подобной (более того, код сработает даже при отсутствии функции `Clone()` — при наличии макроса, преобразующего имя `Clone` в нечто иное).

В ряде приложений приведенные подходы сработают, но поставленная задача формулируется строго.

Убедиться, что `T` имеет функцию-член, выглядящую в точности как `T* T::Clone() const`.

Вот один из способов решения поставленной задачи.

```
// Пример 2в. Корректное решение,
// требующее в точности T* T::Clone() const
// T должен иметь T* T::Clone() const
template<typename T>
class C
{
public:
    // Используем деструктор, чтобы не вносить код
    // в каждый из конструкторов
    ~C()
    {
        T* (T::*test)() const = &T::Clone;
        test; // Для подавления предупреждения о
              // неиспользуемой переменной. При
              // оптимизации эта переменная должна
              // быть полностью удалена

        // ...
    }
    // ...
};
```

Или немного более ясно и развернуто.

```
// Пример 2г. Еще одно корректное решение,
// требующее в точности T* T::Clone() const
// T должен иметь T* T::Clone() const
template<typename T>
class C
{
    bool ValidateRequirements() const
    {
        T* (T::*test)() const = &T::Clone;
```

```
        test; // Для подавления предупреждения о
              // неиспользуемой переменной. При
              // оптимизации эта переменная должна
              // быть полностью удалена
        // ...
        return true;
    }
public:
    // Используем деструктор, чтобы не вносить код
    // в каждый из конструкторов
    ~C()
    {
        assert(ValidateRequirements());
        // ...
    }
    // ...
};
```

Наличие функции `ValidateRequirements()` является расширением, которое резервирует место для любых проверок, которые могут потребоваться в дальнейшем. Вызов этой функции внутри `assert()` обеспечивает удаление всех проверок из окончательной версии программы.

Классы ограничений

Использование этих классов еще понятнее. Данная технология описана Бьярном Страуструпом (Bjarne Stroustrup) в его *C++ Style and Technique FAQ* [StroustrupFAQ] и основана на использовании указателей на функции Александром Степановым (Alex Stepanov) и Джереми Сиком (Jeremy Siek).[4]

Предположим, что мы написали следующий класс ограничения.

```
// Пример 2д. Использование наследования ограничений
// Класс HasClone требует наличия у класса T
// функции T* T::Colne() const
template<typename T>
class HasClone
{
public:
    static void Constraints()
    {
        T* (T::*test)() const = &T::Clone;
        test; // Для подавления предупреждения о
              // неиспользуемой переменной.
    }
    HasClone() { void (*p)() = Constraints; }
};
```

Теперь у нас есть элегантный — я даже рискну назвать его “крутым” — способ обеспечить ограничение во время компиляции.

```
template<typename T>
class C : HasClone<T>
{
    // ...
};
```

Идея проста: каждый конструктор `C` должен вызвать конструктор `HasClone<T>` по умолчанию, который не делает ничего, кроме проверки выполнения ограничения. Если проверка показывает невыполнение ограничения, большинство компиляторов вы-

[4] См. `http://www.gotw.ca/publications/mxc++/bs_constraints.htm`

даст вполне удобочитаемое сообщение об ошибке. Наследование от `HasClone<T>` обеспечивает легко диагностируемую проверку характеристик класса `T`.

Требование наследования, вариант 1

3. Программист намерен написать шаблон, который может потребовать или хотя бы проверить, имеет ли тип, для которого производится инстанцирование, функцию-член `Clone()`. Программист решает сделать это путем требования, чтобы классы с функцией-членом `Clone()` были наследниками класса `Clonable`. Покажите, как следует написать шаблон

```
template<typename T>
class X
{
    // ...
};
```

так, чтобы

а) требовалось, чтобы `T` был производным от `Cloneable`.

Мы рассмотрим два вполне работоспособных подхода. Первый подход немного более "хитрый" и сложный; второй — проще и элегантнее. Полезно рассмотреть оба подхода, поскольку каждый из них демонстрирует интересные технологии, с которыми следует быть знакомым, несмотря на то, что в данной конкретной задаче одна из них больше подходит для ее решения.

Первый подход основан на идеях Андрея Александреску (Andrei Alexandrescu) [Alexandrescu00a]. Сначала мы определим вспомогательный шаблон, который проверяет, является ли тип-кандидат `D` производным от `B`. Эта проверка выполняется путем определения того, может ли указатель на `D` быть преобразован в указатель на `B`. Вот один из способов сделать это, аналогичный примененному Александреску.

```
// Пример 3а. Вспомогательное значение IsDerivedFrom1
//
// Преимущества: может использоваться для проверки значения
//               во время компиляции
// Недостатки:  Код несколько сложноват
template<typename D, typename B>
class IsDefivedFrom1
{
    class No  { };
    class Yes { No no[2]; };
    static Yes Test( B* );    // Объявлена, но не определена
    static No  Test( ... );   // Объявлена, но не определена
public:
    enum
    { Is = sizeof(Test(static_cast<D*>(0))) == sizeof(Yes) };
};
```

Увидели? А теперь немного подумайте, перед тем как читать дальше.

* * * * *

Приведенный код использует следующие особенности.

1. `Yes` и `No` имеют различные размеры. Это гарантируется тем, что `Yes` содержит массив из более чем одного объекта `No`.
2. Разрешение перегрузки и определение значения `sizeof` выполняются во время компиляции, а не во время выполнения.
3. Значения `enum` вычисляются и могут быть использованы в процессе компиляции.

Проанализируем определение enum более детально. Его наиболее глубоко вложенная часть выглядит как Test(static_cast<D*>(0)). Все, что делает это выражение, — это упоминает функцию по имени Test и делает вид, что передает ей D* (в нашем случае это просто приведенный к данному типу нулевой указатель). Заметим, что реально ничего не выполняется, никакой код не генерируется, так что указатель никогда не разыменовывается и, собственно, даже не создается. Все, что мы делаем, — это создаем выражение типа. Компилятору известно, что собой представляет тип D, и в процессе компиляции происходит разрешение перегрузки функции Test(): если D* может быть преобразовано в B*, то будет выбрана функция Test(B*), возвращающая Yes; в противном случае будет выбрана функция Test(...), возвращающая No.

Теперь совершенно очевидно, что следующий шаг состоит в проверке, какая именно функция выбрана.

```
sizeof(Test(static_cast<D*>(0))) == sizeof(Yes)
```

Это выражение, также полностью вычисляемое при компиляции, будет равно true, если D* может быть преобразовано в B*, и false в противном случае. А это все, что мы хотели знать, поскольку D* может быть преобразовано в B* тогда и только тогда, когда D наследован от B, или D есть B[5].

Теперь в процессе компиляции мы можем вычислить все, что нам надо знать, и нам надо где-то сохранить результат вычислений. Это “где-то” должно быть установлено и допускать использование своего значения во время компиляции. К счастью, этому требованию полностью удовлетворяет enum.

```
enum {Is = sizeof(Test(static_cast<D*>(0))) == sizeof(Yes)};
```

При использовании такого подхода в нашем случае можно не беспокоиться о том, не являются ли B и D одним и тем же типом. Нам надо лишь знать, может ли D полиморфно использоваться в качестве B, а это именно то, что проверяется в классе IsDerivedFrom1. Само собой, что в тривиальном случае этим классом подтверждается возможность использования B в качестве B.

Вот и все. Теперь мы можем использовать разработанный класс для окончательного ответа на поставленный в условии задачи вопрос.

```
// Пример 3а, продолжение. Использование IsDerivedFrom1
// для гарантии порождения от Cloneable
template<typename T>
class X
{
    bool ValidateRequirements() const
    {
        // typedef необходим, поскольку иначе запятая в
        // описании типа будет воспринята как разделитель
        // параметров макроса assert.
        typedef IsDerivedFrom1<T, Cloneable> Y;
        // Проверка времени выполнения, которая легко может
        // быть преобразована в проверку времени компиляции
        assert( Y::Is );
        return true;
    }
public:
    // Используем деструктор, чтобы не вносить код
    // в каждый из конструкторов
    ~X()
    {
        assert( ValidateRequirements() );
    }
```

[5] Или B — это тип void.

```
    // ...
};
```

Требование наследования, вариант 2

К этому моменту вы, вероятно, уже заметили, что подход Страуструпа можно использовать для создания функционально эквивалентной версии, выглядящей синтаксически более привлекательно.

```
// Пример 3б. Базовый класс ограничений IsDerivedFrom2
//
// Преимущества: Вычисления в процессе компиляции
//               Проще в использовании
// Недостатки:   Невозможно непосредственное использование
//               для проверки значения во время компиляции
template<typename D, typename B>
class IsDerivedFrom2
{
    static void Constraints(D* p)
    {
        B* pb = p;
        pb = p;   // Для подавления предупреждения о
                  // неиспользуемой переменной.
    }
protected:
    IsDerivedFrom2() { void(*p)(D*) = Constraints; }
};

//
template<typename D>
class IsDerivedFrom2<D, void>
{
    IsDerivedFrom2() { char * p = (int*)0; /* Ошибка */ }
};
```

Проверка теперь становится существенно проще.

```
// Пример 3б, продолжение: использование IsDerivedFrom2
// как базового класса для гарантии порождения от
// Cloneable
template<typename T>
class X: IsDerivedFrom2<T,Cloneable>
{
    // ...
};
```

Требование наследования, вариант 3

Основное преимущество класса `IsDerivedFrom1` над классом `IsDerivedFrom2` заключается в том, что `IsDerivedFrom1` обеспечивает генерацию значения `enum` и его проверку во время компиляции. Это неважно для рассматриваемого класса `class X`, но станет важным в следующем разделе, когда нас будет интересовать возможность выбора различных реализаций свойств во время компиляции на основании этого значения. Однако `IsDerivedFrom2` обеспечивает в общем случае значительно большую простоту использования. Он в большей степени подходит для случая, когда нам достаточно поместить наше требование, гарантирующее наличие некоторой особенности, в параметр шаблона без каких-либо изысков типа выбора среди нескольких различных реализаций. Можно использовать одновременно оба подхода, но это вызовет свои проблемы, в особенности связанные с именованием. Мы не можем сделать ничего лучшего, чем просто дать этим классам разные имена, что заставит пользователя каж-

дый раз вспоминать, что же именно им надо в конкретной ситуации — IsDerivedFrom1 или IsDerivedFrom2? Это не очень-то приятно.

Но почему бы нам не попытаться убить одним выстрелом двух зайцев? Ведь это же так просто.

```
// Пример 3в: базовый класс ограничений
// IsDerivedFrom с проверяемым значением
template<typename D, typename B>
class IsDerivedFrom
{
    class No  { };
    class Yes { No no[2]; }

    static Yes Test(B*);  // Не определена
    static No Test(...);  // Не определена

    static void Constraints(D* p)
    {
        B* pb = p;
        pb = p;
    }
public:
    enum { Is = sizeof(Test(static_cast<D*>(0))) ==
                sizeof(Yes) };

    IsDerivedFrom() { void(*p)(D*) = Constraints; }
};
```

Выбор альтернативных реализаций

Решение *3а* элегантно и гарантирует, что T должно быть Clonable. Но что если T не является Cloneable? И если в этом случае мы должны предпринять некоторые альтернативные действия? Возможно, мы сможем найти более гибкое решение — отвечая на вторую часть вопроса:

б) **обеспечить реализацию, в которой выполняются одни действия, если T — производный от Cloneable класс, и некоторые другие действия, если это не так.**

Для этого мы введем навязший в зубах “дополнительный уровень косвенности”, который решает многие проблемы разработки. В нашем случае этот дополнительный уровень принимает вид вспомогательного шаблона: X будет использовать IsDerivedFrom из примера *3в* и специализации вспомогательного шаблона для переключения между реализациями “является Cloneable” и “не является Cloneable”. (Заметим, что для этого требуется проверяемое во время компиляции значение из IsDerivedFrom1, встроенное также в IsDerivedFrom, так что у нас есть возможность переключения между различными реализациями в процессе компиляции.)

```
// Пример 3г: использование IsDerivedFrom для
// использования наследования от Cloneable, если
// таковое имеет место, и выполнения некоторых
// других действий в противном случае.
template<typename T, int>
class XImpl
{
    // Общий случай: T не является наследником Cloneable
};

template<typename T>
class XImpl<T,1>
{
    // T - производный от Cloneable класс
```

```
};

template<typename T>
class X
{
    XImpl<T, IsDerivedFrom<T, Cloneable>::Is> impl_;
    // ... Делегирование выполнения impl_ ...
};
```

Вам понятно, как работает приведенный пример? Рассмотрим его работу на небольшом примере.

```
class MyCloneable: public Cloneable { /* ... */ };
X<MyCloneable> x1;
```

`impl_` имеет тип

```
XImpl<T, IsDerivedFrom<T, Cloneable>::Is>
```

В этом случае `T` является `MyCloneable`, и, соответственно, `impl_` имеет тип

```
XImpl<MyCloneable, IsDerivedFrom<MyCloneable, Cloneable>::Is>
```

что преобразуется в тип `XImpl<MyCloneable,1>`, который использует специализацию `XImpl`, где, в свою очередь, используется тот факт, что `MyCloneable` является классом, производным от `Cloneable`. Но что произойдет, если мы инстанцируем `X` с некоторым другим типом? Рассмотрим:

```
X<int> x2;
```

Здесь `T` — `int`, так что `impl_` имеет тип

```
XImpl<MyCloneable, IsDerivedFrom<int, Cloneable>::Is>
```

что преобразуется в тип `XImpl<MyCloneable, 0>`, использующий неспециализированный шаблон `XImpl`. Остроумно, не правда ли? Будучи написанным, этот код легко используется. С точки зрения пользователя, вся сложность заключена в `X`. С точки же зрения автора, `X` — это просто вопрос непосредственного повторного использования механизма, инкапсулированного в `IsDerivedFrom`, без необходимости понимания того, как работает это волхвование.

Заметим, что мы не порождаем излишних инстанцирований шаблонов. Для данного типа `T` будет инстанцирован ровно в один шаблон `XImpl<T,...>` — либо `XImpl<T,0>`, либо `XImpl<T,1>`. Хотя теоретически второй параметр шаблона `XImpl` может быть любым целым числом, в нашей ситуации реально он может принимать только два значения — 0 и 1. (В таком случае, почему мы не использовали тип `bool` вместо `int`? Ответ — для расширяемости: использование `int` ничему не мешает, зато позволяет при необходимости легко добавить дополнительные альтернативные реализации, — например, позже мы можем захотеть добавить поддержку другой иерархии, которая имеет базовый класс, аналогичный `Cloneable`, но с другим интерфейсом.)

Требования и свойства

4. Является ли подход, описанный в п. 3, наилучшим способом обеспечить требование (обнаружение) наличия `Clone()`? Приведите другие варианты решения.

Подход из ответа на вопрос 3 весьма остроумен, но лично я во многих случаях предпочитаю свойства — они и попроще (за исключением случая, когда требуется определения специализаций для каждого класса в иерархии), и более расширяемы, как вы увидите в задачах 6.5 и 6.6.

Основная идея состоит в создании шаблона свойств, единственная цель которого, в нашем случае, состоит в реализации операции `Clone()`. Шаблон свойств выглядит во многом похоже на `XImpl`; при этом имеется неспециализированная версия общего

назначения, выполняющая некие наиболее общие действия, а также может иметься множество специализированных версий, которые обеспечивают для разных классов лучшее (или просто иное) по сравнению с обобщенным клонирование объектов.

```
// Пример 4. Использование свойств вместо IsDerivedFrom
// для обеспечения возможности клонирования и выполнения
// иных действий. Требует написания специализаций для
// каждого из используемых классов.
template<typename T>
class XTraits
{
public:
    // Общий случай: использование конструктора копирования
    static T* Clone(const T* p) { return new T(*p); }
};

template<>
class XTraits<MyCloneable>
{
public:
    // MyCloneable производный от Cloneable, так что
    // просто используем Clone()
    static MyCloneable* Clone(const MyCloneable* p)
    {
        return p->Clone();
    }
};

// ... и т.д. - для каждого класса, производного
// от класса Cloneable
```

`X<T>` просто вызывает `XTraits<T>::Clone()` там, где это требуется, при этом будут выполняться корректные действия.

Основное отличие свойств от простого `XImpl` из примера 3*б* состоит в том, что при применении свойств, когда пользователь определяет некоторый новый тип, основная работа по его использованию с `X` оказывается внешней по отношению к `X` — необходимые действия выполняют шаблоны свойств. Такой подход более расширяем, чем относительно закрытый для расширения подход из ответа на вопрос 3, в котором вся работа выполняется в процессе реализации `XImpl`. Он позволяет использовать и другие методы клонирования, а не только унаследованную от специального базового класса функцию со специальным именем `Clone()` — и это тоже увеличивает расширяемость подхода, основанного на применении свойств.

Более детальный пример реализации и использования свойств можно найти в задаче 6.6, примерах 2*г* и 2*д*.

Примечание: основной недостаток описанного подхода с использованием свойств заключается в том, что для каждого класса иерархии требуется отдельная специализация шаблона свойств. Имеются способы одновременного определения свойств для всей иерархии классов вместо скучного написания множества специализаций. В [Alexandrescu00b] описана остроумная технология, позволяющая выполнить эту работу. Эта технология требует минимального вмешательства в базовый класс иерархии — в нашем случае в класс `Cloneable`.

Наследование и свойства

5. Насколько полезно знание шаблона о том, что его параметр типа `T` является производным от некоторого другого типа? Дает ли такое знание какие-либо преимущества, которые не могут быть достигнуты иначе, без отношений наследования?

Имеется небольшое дополнительное преимущество, которое шаблон может получить из знания о том, что один из его параметров является производным от некоторого заданного базового класса, состоящее в том, что при этом мы избавляемся от необходимости использовать свойства. Единственный реальный недостаток в использовании свойств заключается в том, что они могут потребовать написания большого количества специализаций для классов из большой иерархии, однако имеются технологии, которые уменьшают или вовсе устраняют этот недостаток.

Основная цель данной задачи состояла в том, чтобы продемонстрировать, что использование наследования для разделения шаблонов по категориям, вероятно, не является настолько необходимой причиной для использования наследования, как некоторые могут подумать. Свойства обеспечивают более обобщенный механизм, бо́льшая расширяемость которого имеет значение при инстанцировании существующего шаблона с новыми типами (например, типами из некоторой библиотеки стороннего производителя, когда нелегко обеспечить наследование от некоторого предопределенного базового класса).

Задача 1.12. Typename *Сложность: 7*

Эта задача призвана продемонстрировать, как и зачем используется ключевое слово `typename`.

1. Что такое `typename` и что делает это ключевое слово?
2. Какие ошибки (если таковые есть) имеются в приведенном ниже коде?

```
template<typename T>
class X_base
{
public:
    typedef T instantiated_type;
};

template<typename A, typename B>
class X: public X_base<B>
{
public:
    bool operator()(const instantiated_type& i) const
    {
        return i != instantiated_type();
    }
    // ... прочий код ...
};
```

Решение

1. Что такое `typename` и что делает это ключевое слово?

Этот вопрос возвращает нас в область поиска имен. Поясняющим примером могут служить зависимые имена в шаблонах. Так, в приведенном коде

```
// Пример 1
//
template<typename T>
void f()
{
    T::A* pa; // Что делает эта строка?
}
```

имя T::A является *зависимым именем*, поскольку оно зависит от параметра шаблона T. В данном случае программист, вероятно, ожидает, что T::A является вложенным классом (или синонимом typedef) внутри T, поскольку в ином случае (если это статическая переменная или функция) приведенный код не будет скомпилирован. Вот что говорится о данной проблеме в стандарте.

> *Имя, используемое в объявлении или определении шаблона и зависимое от параметра шаблона, считается не являющимся именем типа, если только поиск имени не находит соответствующее имя типа или имя с квалификатором typename.*

Это приводит нас к основному вопросу.

2. Какие ошибки (если таковые есть) имеются в приведенном ниже коде?

```
// Пример 2
//
template<typename T>
class X_base
{
public:
    typedef T instantiated_type;
};

template<typename A, typename B>
class X: public X_base<B>
{
public:
    bool operator()(const instantiated_type& i) const
    {
        return i != instantiated_type();
    }
    // ... прочий код ...
};
```

Использование typename для зависимых имен

Проблема с X заключается в том, что instantiated_type означает ссылку на определение типа typedef, предположительно унаследованное от базового класса X_base<B>. К сожалению, во время разбора компилятором встроенного определения X<A,B>::operator()() зависимые имена (т.е. имена, зависящие от параметров шаблона, такие как наследованное X_Base::instantiated_type) невидимы, и компилятор вынужден будет сообщить, что ему неизвестно, что должно означать instantiated_type. Зависимые имена становятся видимы только позже, в момент реального инстанцирования шаблона.

Если вас удивляет, почему компилятор не может разобраться с этими именами, встаньте на его место и спросите себя, как понять, что именно обозначает здесь instantiated_type. Вот проясняющий ситуацию пример. Вы не можете выяснить, что именно обозначает здесь instantiated_type, поскольку вы еще не знаете, что такое B, а позже вы можете столкнуться с различными специализациями X_base<B>, в которых instantiated_type оказывается чем-то неожиданным — некоторым другим именем типа или даже переменной-членом класса. В неспециализированном шаблоне X_base, приведенном ранее, X_base<T>::instantiated_type всегда представляет собой T, но ничто не мешает изменению его смысла в специализациях, например, так:

```
template<>
class X_base<int>
{
public:
```

```
    typedef Y instantiated_type;
};
```

или даже так:

```
template<>
class X_base<int>
{
public:
    double instantiated_type;
};
```

В первом случае имя явно не соответствует его содержанию, но такое его применение вполне законно. Во втором примере имя как раз в большей степени соответствует содержанию, но шаблон X не может работать с `X_base<double>` в качестве базового класса, поскольку `instantiated_type` является не именем типа, а переменной-членом.

Короче говоря, компилятор не в состоянии определить, каким образом следует разбирать определение `X<A,B>::operator()()` до тех пор, пока мы не объясним ему, что собой представляет `instantiated_type` — как минимум, является ли это имя типом или чем-то иным. В нашем случае оно должно быть типом.

Способ объяснить компилятору, что нечто представляет собой имя типа, состоит в использовании ключевого слова `typename`. В нашем конкретном примере для этого имеется два пути. Менее элегантный состоит в использовании этого ключевого слова при обращении к `instantiated_type`.

```
// Пример 2а. Немного некрасиво
//
template<typename A, typename B>
class X: public X_base<B>
{
public:
    bool operator()(
        const typename X_base<B>::instantiated_type& i
        ) const
    {
        return i != typename X_base<B>::instantiated_type();
    }
    // ... прочий код ...
};
```

Я думаю, читая это, вы не смогли не поморщиться. Конечно же, как обычно, сделать код гораздо более удобочитаемым можно с помощью `typedef` (оставив при этом по сути весь код неизменным).

```
// Пример 2б. Гораздо красивее
//
template<typename A, typename B>
class X: public X_base<B>
{
public:
    typedef typename X_base<B>::instantiated_type
            instantiated_type
    bool operator()(const instantiated_type& i) const
    {
        return i != instantiated_type();
    }
    // ... прочий код ...
};
```

Перед тем как продолжить чтение, ответьте на вопрос: не показалось ли вам необычным такое добавление конструкции `typedef`?

Еще одна тонкость

Для иллюстрации использования `typename` я мог бы использовать примеры попроще (например, из раздела 14.6.2 стандарта), но они не подходят мне по той причине, что не подчеркивают одну необычную вещь: единственная причина существования пустого базового класса `X_base` — обеспечить конструкцию `typedef`. Однако производный класс обычно заканчивает тем, что тут же переопределяет имя типа с помощью `typedef` заново.

Вам кажется, что это слишком много? Да, но только в очень небольшой степени. В конце концов, за определение конкретного типа отвечают специализации шаблона `X_base`, и этот тип может изменяться от специализации к специализации.

Стандартная библиотека содержит базовые классы, подобные рассмотренному, — так сказать, контейнеры определений типов, — и предназначенные именно для такого использования. Надеюсь, эта задача поможет предупредить некоторые вопросы о том, почему производные классы переопределяют уже определенные типы (что кажется излишним), и покажет, что этот эффект — не ошибка при разработке языка.

Постскриптум

В качестве маленькой премии — код-шутка с использованием `typename`.

```
#include <iostream>

class Rose {};

class A { public: typedef Rose rose; };

template<typename T>
class B: public T { public: typedef typename T::rose foo; };

template<typename T>
void smell(T)    { std::cout << "Жуть!"; }
void smell(Rose) { std::cout << "Прелесть!"; }

int main()
{
    smell(A::rose());
    smell(B<A>::foo()); // :-)
}
```

Задача 1.13. Контейнеры, указатели и неконтейнеры *Сложность: 5*

Масло и вода никак не смешиваются. Может, указатели и контейнеры будут смешиваться лучше?

1. Рассмотрим следующий код.

```
vector<char> v;
// ... Заполняем v ...

char* p = &v[0];

// ... Работаем с *p ...
```

   Корректен ли приведенный код? Независимо от ответа на этот вопрос, как можно улучшить приведенный код?

2. Рассмотрим следующий код.

```
template<typename T>
void f (T& t)
{
    typename T::value_type* p1 = &t[0];
    typename T::value_type* p2 = &*t.begin();

    // ... Работаем с *p1 и *p2 ...
}
```

Корректен ли данный код?

Решение

Предисловие

Маленького Никиту позвала мать, занятая уборкой в холодильнике. “Пожалуйста, — говорит она Никите, не отрываясь от работы, — дай мне какую-нибудь посудину, чтобы я могла перелить этот сок”.

Никита — послушный мальчик, кроме того, он абсолютно уверен, что знает, что такое “посудина”. Он смотрит вокруг и замечает красивый дуршлаг, который представляет собой кастрюльку с дном “в дырочку”. Но Никита точно знает, что кастрюлька — это “посудина”, и, гордый, что так быстро выполнил мамину просьбу, несет ей дуршлаг. “Замотанная” мама, не глянув хорошенько на принесенный предмет, начинает переливать сок...

Никите досталось в этот день. Но за что? Ведь он был совершенно уверен, что поступает правильно, — ведь кастрюлька с дырявым дном выглядит почти в точности так же, как и другие кастрюльки с длинными ручками. Откуда же ему знать, что эта кастрюлька не удовлетворяет поставленным мамой требованиям?

В этой задаче мы рассмотрим некоторые требования стандарта C++ к контейнерам, некоторые причины для использования указателей на содержимое контейнеров и — к возможному ужасу читателя — стандартный контейнер, который в действительности контейнером не является.

Зачем нужны указатели и ссылки на содержимое контейнеров

1. **Рассмотрим следующий код.**

```
// Пример 1a. Корректен ли данный код? Безопасен ли?
vector<char> v;
// ... Заполняем v ...

char* p = &v[0];

// ... Работаем с *p ...
```

Корректен ли приведенный код?

Стандарт гарантирует, что если соответствующая последовательность (такая, как `vector<char>`) предоставляет `operator[]()`, то этот оператор должен возвращать lvalue типа `char`, а это означает, что может быть получен его адрес[6]. Таким образом,

[6] В этой задаче используются различные выражения для этой величины — “указатель в контейнер”, “указатель на содержимое контейнера”, “указатель на объект в контейнере,” — но все они обозначают одно и то же.

ответ на поставленный вопрос следующий: да, если только вектор v не пуст, приведенный код совершенно корректен.

Безопасен ли данный код? Некоторых программистов удивляет сама идея получения указателя в контейнер, но на самом деле ответ следующий: да, этот код безопасен, только следует помнить о ситуациях, когда указатель становится недействительным (что почти всегда совпадает с ситуациями, когда становится недействительным эквивалентный итератор). Например, понятно, что если мы вставим элемент в контейнер или удалим его из контейнера, не только итераторы, но и любые указатели на содержимое контейнера становятся недействительными в силу возможных перемещений в памяти.

Но имеет ли данный код смысл? Опять же да — вполне может иметься потребность в указателе или ссылке на содержимое контейнера. Один из типичных случаев, когда вы однократно считываете структуру данных в память (например, при запуске программы), а после этого никогда не модифицируете ее. Если при этом вам требуются различные способы обращения к этой структуре, имеет смысл поддерживать дополнительную структуру данных с указателями на элементы основного контейнера для оптимизации различных методов доступа к ним. Я приведу соответствующий пример в следующем разделе.

Как улучшить приведенный код

Независимо от корректности приведенного кода, как можно улучшить его?

Пример 1*а* может быть улучшен следующим образом.

```
// Пример 1б. Возможное улучшение кода 1а
vector<char> v;
// ... Заполняем v ...

vector<char>::iterator i = v.begin();

// ... Работаем с *i ...
```

В общем случае можно придерживаться рекомендации предпочесть использование итераторов указателям при необходимости обращения к объекту в контейнере. В конце концов, итераторы становятся недействительными в основном тогда же и по той же причине, что и указатели, и единственная причина существования итераторов состоит в обеспечении способа “указать” на содержащийся в контейнере объект. Если у вас есть возможность выбора, предпочтительно использовать итераторы.

К сожалению, с помощью итераторов не всегда можно получить тот же результат, что и при использовании указателей. Вот два потенциальных недостатка применения итераторов, из-за которых мы продолжаем использовать указатели на содержимое контейнеров.

1. Не всегда удобно использовать итератор там, где можно воспользоваться указателем (см. пример ниже).
2. Когда итератор представляет собой объект, а не простой указатель, его использование может вызвать дополнительные накладные расходы как с точки зрения размера программы, так и ее производительности.

Например, пусть у нас есть контейнер `map<Name,PhoneNumber>`, загруженный при запуске программы, после чего вся работа с ним сводится только к запросам к его содержимому (т.е. идея состоит в простом поиске номера телефона для данного имени в фиксированном словаре). Но что если нам требуется осуществлять как прямой, так и обратный поиск? Очевидное решение состоит в построении второй структуры, вероятно, `map<PhoneNumber*,Name*,Deref>`, которая обеспечивает возможность обратного

поиска и при этом избегает дублирования хранимой информации. Применение указателей позволяет хранить имена и телефоны в одном экземпляре, используя во второй структуре указатели в первую структуру.

Этот метод вполне корректен; заметим только, что получить тот же эффект с использованием итераторов, а не указателей, будет существенно сложнее. Почему? Да потому, что естественным кандидатом для этой цели является итератор `map<Name,PhoneNumber>::iterator`, который указывает на `pair<Name,PhoneNumber>`, и не имеется удобного способа получения итераторов отдельно, только для имени или номера телефона. (Можно сохранить итератор целиком и всегда явно указывать `->first` и `->second`, но такой подход неудобен и может означать, что объект `map` для обратного поиска требуется переконструировать или заменить другой структурой.)

Это приводит нас к следующей (и, на мой взгляд, наиболее интересной) части задачи.

2. Рассмотрим следующий код.

```
// Пример 2. Корректен ли данный код?
//
template<typename T>
void f (T& t)
{
    typename T::value_type* p1 = &t[0];
    typename T::value_type* p2 = &*t.begin();

    // ... Работаем с *p1 и *p2 ...
}
```

Перед тем как читать дальше, подумайте немного над вопросом корректности этого кода самостоятельно. Если вы считаете, что код корректен, то при каких условиях он корректен? В частности, что можно сказать о `T`, что делает этот код корректным? (Соображения времени выполнения, такие как требование, чтобы объект `t` находился в состоянии, позволяющем вызов `t[0]`, не рассматриваются. Нас интересует только законность приведенной программы.)

Когда контейнер контейнером не является?

Итак, корректен ли пример 2? Вкратце — да, он *может* быть корректен.

Более длинный ответ включает размышления о том, каким должен быть тип `T`, чтобы код был корректен. Какими характеристиками и возможностями должен обладать тип `T`? Проведем небольшое расследование.

а) Для того чтобы выражение `&t[0]` было корректно, должен существовать `T::operator[]()`, который возвращает нечто, к чему может быть применен `operator&()`, который, в свою очередь, должен возвращать корректный указатель `T::value_type*` (или нечто, что может быть преобразовано в корректный указатель `T::value_type*`).

   В частности, это условие справедливо для контейнеров, которые отвечают требованиям стандарта к контейнерам и последовательностям и для которых реализован `operator[]()`, поскольку этот оператор должен возвращать ссылку на содержащийся в контейнере объект. По определению, после этого вы можете получить адрес содержащегося в контейнере объекта.

б) Для того чтобы выражение `&*t.begin()` было корректно, должна существовать функция `T::begin()`, которая возвращает нечто, к чему применим `operator*()`, который, в свою очередь, должен возвращать нечто, к чему применим `operator&()`, который, в свою очередь, должен возвращать корректный указатель `T::value_type*` (или нечто, что может быть преобразовано в корректный указатель `T::value_type*`).

В частности, это условие истинно для контейнеров, итераторы которых отвечают требованиям стандарта, поскольку итератор, возвращаемый функцией begin(), должен при разыменовании с использованием operator*() возвращать ссылку на содержащийся в контейнере объект. По определению, после этого вы можете получить адрес содержащегося в контейнере объекта.

Итак, код в примере 2 будет работать с любым контейнером стандартной библиотеки, который поддерживает operator[]. Если убрать из кода строку, содержащую &t[0], код будет корректен для любого контейнера стандартной библиотеки, *за исключением std::vector<bool>*.

Для того чтобы понять, почему это так, заметим, что следующий шаблон работает с любым типом T, кроме bool.

```
// Пример 3. Код работает для всех T, кроме bool
//
template<typename T>
void g(vector<T>&v)
{
    T* p = &v.front();
    // ... Работаем с *p ...
}
```

Вы поняли, почему? На первый (и даже на второй и третий) взгляд, кажется странным, что этот код работает с любым типом, кроме одного. Что же делает vector<bool> таким выдающимся? Причина столь же проста, сколь и неприятна: не все шаблоны стандартной библиотеки C++, выглядящие как контейнеры, являются контейнерами. В частности, стандартная библиотека требует наличия специализации vector<bool>, и эта специализация *не является контейнером* и, соответственно, не удовлетворяет требованиям стандартной библиотеки к контейнерам. Да, vector<bool> описывается в разделе стандарта, посвященном контейнерам и последовательностям; да, нигде нет упоминания о том, что этот тип в действительности не является ни контейнером, ни последовательностью. Тем не менее на самом деле vector<bool> контейнером не является и, соответственно, вас не должно удивлять, что он не может быть использован так же, как контейнер.[7]

Вы не одиноки — такое положение дел кажется несколько странным не только вам. Однако имеется логическое объяснение такой ситуации.

vector<bool>

Специализация vector<bool> обычно (и напрасно) используется в качестве примера из стандарта C++, показывающего, как следует писать прокси-контейнеры. “Прокси-контейнер” — это контейнер, работа с объектами которого и доступ к ним не могут выполняться непосредственно. Вместо предоставления указателя или ссылки на содержащийся в контейнере объект, прокси-контейнер предоставляет вам прокси-объект, который может использоваться для косвенного доступа или управления объектом, содержащимся в контейнере. Прокси-коллекции могут оказаться очень полезными в случаях, когда невозможно обес-

[7] Если бы vector<bool> был написан каким-то посторонним программистом, он бы был назван нестандартным, не соответствующим предъявляемым к контейнерам требованиям. Но поскольку этот тип включен в стандарт, его трудно назвать нестандартным. Пожалуй, наиболее корректным решением было бы удаление из стандарта п. 23.2.5 (требования к специализации vector<bool>), с тем, чтобы vector<bool> представлял собой не что иное, как инстанцирование шаблона vector<>, и, таким образом, был бы тем, чем и должен быть, — вектором обычных величин типа bool (кстати говоря, многие аспекты vector<bool> излишни: для такого упакованного представления логических величин имеется специально разработанный тип std::bitset).

печить надежный непосредственный доступ к объектам (как если бы эти объекты находились в памяти), например, в случае, когда объекты располагаются на диске и автоматически загружаются в память только тогда, когда в них имеется необходимость. Таким образом, идея, что vector<bool> показывает, каким образом создаются прокси-коллекции, отвечающие требованиям стандартной библиотеки к контейнерам, ошибочна.

Ложкой дегтя в бочке меда оказывается то, что vector<bool> *не является* контейнером в смысле соответствия требованиям к стандартным контейнерам и итераторам[8]. Прокси-контейнеры категорически запрещены требованиями к стандартным контейнерам и итераторам. Например, тип container<T>::reference должен быть истинной ссылкой (T&) и не может быть прокси-ссылкой. Сходные требования предъявляются и к итераторам, так что разыменование прямых, двунаправленных итераторов или итераторов произвольного доступа должно давать истинную ссылку (T&), а не прокси. В результате прокси-контейнеры не соответствуют требованиям, предъявляемым к контейнерам стандартом C++.

Причина того, что vector<bool> не соответствует стандарту, заключается в попытке оптимизации используемого пространства. Обычно объект типа bool имеет размер, как минимум, такой же, как и char; sizeof(bool) определяется реализацией языка и варьируется от компилятора к компилятору, но это значение должно быть, как минимум, единицей. Это кажется слишком расточительным? Так думают многие, и vector<bool> пытается повысить эффективность использования памяти. Вместо выделения каждому объекту типа bool отдельного char или int, логические величины упаковываются и во внутреннем представлении хранятся как отдельные биты переменной типа char или, например int. Таким образом vector<bool> выигрывает как минимум в отношении 8:1 на платформах с 8-битовым “обычным” типом bool, а на других платформах выигрыш может оказаться еще большим. (Эта упакованность логических величин уже дает возможность заметить, что имя vector<bool> не соответствует действительности, поскольку внутри этого вектора нет никаких “нормальных” bool.)

Одним очевидным следствием такого пакованного представления является то, что тип vector<bool> не в состоянии вернуть обычную ссылку bool& в результате вызова оператора operator[] или разыменования итератора[9]. Вместо этого возвращается “похожий на ссылку” прокси-объект, который очень похож на bool&, но таковым не является. К сожалению, такой способ обращения к элементам vector<bool> оказывается медленнее, чем обращение к элементам векторов других типов, поскольку вместо прямого указателя или ссылки мы имеем дело с прокси-объектами, не говоря уже о затратах на распаковку битов. Таким образом, vector<bool> по сути не является оптимизацией вектора для конкретного типа, а представляет собой компромисс между экономией памяти и потенциальным уменьшением скорости работы.

Например, в то время как функции vector<T>::operator[]() и vector<T>::front() обычно возвращают просто T& (ссылку на содержащийся в векторе объект типа T), vector<bool>::operator[]() и vector<bool>::front() возвращают прокси-объект типа vector<bool>::reference. Чтобы этот прокси-объект выглядел как можно более похожим на действительную ссылку bool&, тип reference содержит неявное преобразование типа к bool — operator bool(), так что объект типа reference может использоваться практически везде, где может использоваться объект типа bool, как минимум, в качестве значения. Кроме того, тип reference имеет функции-члены operator=(bool), operator=(reference&) и flip(), которые могут использоваться для косвенного изменения значения, находящегося в векторе, так что при программировании можно при-

[8] Очевидно, что контейнер в обобщенном смысле представляет собой коллекцию, в которую вы можете поместить объекты и из которой можете получить их обратно.

[9] Это связано с тем, что не существует стандартного пути выражения указателя или ссылки на бит.

держиваться обычного стиля работы с вектором — наподобие выражений “v[0] = v[1];”. (Заметим, что единственное, чего не может предоставить тип vector<bool>::reference, — это оператора bool* operator&(). Это связано с тем, что не существует способа получения адреса отдельного бита внутри вектора, для которого объект типа reference служит в качестве прокси-объекта.)

Прокси-контейнеры и предположения STL

Итак, std::vector<bool> может быть стандартным, но не является контейнером. (Этот тип также не является хорошей реализацией битового вектора, поскольку в нем отсутствует ряд операций с битами, которые были бы вполне уместны для битового вектора и которые имеются в std::bitset.)

Кроме того, как исходные требования STL к контейнерам, так и требования стандарта основаны на неявном предположении (среди прочих) о том, что оператор разыменования является операцией с постоянным временем выполнения и что время его выполнения пренебрежимо мало по сравнению с временем выполнения других операций. Ни одно из этих предположений в случае контейнера, хранящегося на диске, и контейнера с упакованными данными, неверно. В случае дискового контейнера доступ посредством прокси-объекта может потребовать выполнения поиска на диске, что на порядки медленнее прямого обращения к памяти. Для иллюстрации посмотрим, что будет при применении стандартного алгоритма, такого как std::find(), к такому контейнеру. Производительность по сравнению со специальными алгоритмами для работы с данными на диске будет очень низкой, в основном потому, что базовые предположения о производительности для контейнеров, содержащихся в памяти, не применимы для контейнеров, содержащихся на диске (по аналогичным причинам даже контейнеры с хранением элементов в памяти, такие как std::map, имеют собственные функции-члены find()). Для контейнеров с упаковкой данных (типа vector<bool>) доступ посредством прокси-объектов требует применения побитовых операций, которые в общем случае гораздо медленнее работы с такими встроенными типами, как int. Кроме того, если создание и уничтожение прокси-объекта не может быть оптимизировано компилятором до полного устранения, управление прокси-объектами приводит к дополнительным накладным расходам.

Хотя тип vector<bool> и используется в качестве примера, как следует писать прокси-контейнеры (так что другие программисты могут следовать ему при написании контейнеров, хранящих содержимое на диске, или контейнеров, не позволяющих непосредственный доступ к своему содержимому), он служит также и демонстрацией того, что текущие требования стандарта к контейнерам не допускают прокси-контейнеров.

Прокси-коллекции — очень полезный инструмент, зачастую наиболее подходящий для решения тех или иных задач (в особенности для очень больших коллекций). Об этом известно каждому программисту. Но они просто не соответствуют STL, как ошибочно считают многие. Несмотря на то, что vector<bool> является хорошим примером того, как следует писать прокси-коллекции, он не является “контейнером” в смысле STL и должен называться как-то иначе (предлагалось назвать его bitvector), чтобы не вводить пользователей в заблуждение.

Кто виноват

Я всегда выступаю против преждевременной оптимизации. Вот правила, которыми я призываю вас руководствоваться: “1. Не оптимизируйте преждевременно. 2. Не оптимизируйте до тех пор, пока не убедитесь в том, что это необходимо. 3. Даже в этом

случае не оптимизируйте до тех пор, пока не будете точно знать, *что и где* надо оптимизировать.” Впрочем, другие выражаются еще более кратко.

- Правило первое: никогда не оптимизируй.
- Правило второе: см. правило первое.

Вообще говоря, программисты — включая меня — как известно, славятся тем, что очень плохо находят узкие места в своих собственных программах. Если вы не использовали профайлер и у вас нет некоторых эмпирических данных, которыми вы можете руководствоваться, вы можете потратить целые дни, оптимизируя то, что оптимизировать не требуется и что заметно не повлияет ни на размер, ни на время выполнения программы. Ситуация может оказаться еще хуже, когда вы не понимаете, *что* требует оптимизации, и своими действиями непреднамеренно ухудшаете программу, “тратя на водку то, что сэкономили на спичках”. Но если у вас имеются профили производительности и результаты других тестов и вы действительно знаете, что некоторая оптимизация поможет в данной ситуации, значит, наступило время для выполнения этой оптимизации.

К этому моменту, вероятно, вы уже поняли, к чему я веду: самая преждевременная оптимизация изо всех — оптимизация, зашитая в стандарт (которая из-за этого не может быть легко устранена). В частности, `vector<bool>` преднамеренно предпочитает экономию памяти ценой производительности и заставляет *все* программы делать этот выбор, который неявно предполагает, что все пользователи вектора логических величин предпочитают использовать как можно меньше памяти, пусть даже ценой увеличения времени работы программы. Очевидно, что такое предположение ложно; в большинстве распространенных реализаций C++ размер `bool` тот же, что и размер 8-битового `char`, так что вектор из 1 000 величин типа `bool` занимает около одного килобайта памяти. Сохранение одного килобайта памяти в большинстве программ особого значения не имеет (понятно, что могут иметься программы, где этот килобайт очень важен, например, в различных встроенных системах; не менее очевидно, что при использовании векторов с миллионами логических значений речь идет о мегабайтах сохраненной памяти, чем действительно сложно пренебречь). Мораль же заключается в том, что *корректная оптимизация зависит от приложения*. Если вы пишете приложение, которое работает во внутреннем цикле с вектором, содержащим 1 000 логических значений, скорее всего, вы предпочтете более быструю работу (без накладных расходов на прокси-объекты и битовые операции) незначительному сохранению памяти (пусть даже это сохранение снизит количество нерезультативных обращений к кэшу).

Что делать

Если вы используете в своем коде `vector<bool>`, можете использовать его с алгоритмами, работающими с реальными контейнерами (которые в результате могут отказаться работать с данным типом); при этом вы пользуетесь оптимизацией, которая отдает предпочтение экономии пространства (скорее всего, весьма незначительной) перед временем работы программы. Оба замечания могут оказаться камнем преткновения в разрабатываемой вами программе. Простейший путь выяснения справедливости первого замечания — использовать свой код до тех пор, пока в один прекрасный день (который может и не наступить) новый код попросту откажется компилироваться. Что же касается второго замечания, то для выяснения его важности вам придется использовать эмпирические тесты для измерения производительности использования вашим кодом типа `vector<bool>`.

Часто разница в производительности по сравнению с использованием, например, типа vector<int> весьма незначительна.

Если вас не устраивает то, что vector<bool> не является контейнером или *если* измеренная разность в производительности достаточно существенна для вашего приложения, не используйте vector<bool>. Можно подумать над использованием вместо него vector<char> или vector<int> (т.е. о хранении значений типа bool в переменных типа char или int) с применением преобразования типов при определении значений в контейнере, но этот путь достаточно утомителен. Существенно лучшим решением является использование вместо вектора типа deque<bool>; это решение гораздо проще; информацию об использовании типа deque вы найдете в задаче 1.14.

Итак, резюмируем.:

1. vector<bool> не является контейнером.
2. vector<bool> пытается проиллюстрировать, как следует писать отвечающие стандарту прокси-контейнеры, в которых вся дополнительная работа выполняется прозрачно для пользователя. К сожалению, это не очень верная идея, поскольку хотя прокси-коллекции и могут быть важными и полезными инструментами, они по определению вынуждены нарушать требования к стандартным контейнерам. Соответственно, они не могут выступать в роли стандартных контейнеров (см. п. 1).

   Вывод: продолжайте писать и использовать прокси-коллекции, но не пытайтесь написать прокси-коллекцию, соответствующую требованиям к стандартным контейнерам и последовательностям. Во-первых, это невозможно. Во-вторых, основная причина попыток написания прокси-коллекций, удовлетворяющих стандарту, — это использование их со стандартными алгоритмами. Однако стандартные алгоритмы обычно плохо подходят для прокси-контейнеров, поскольку у последних характеристики производительности сильно отличаются от соответствующих характеристик контейнеров с непосредственным размещением объектов в памяти.
3. Имя vector<bool> несколько некорректно, поскольку объекты, хранящиеся в данном векторе, не являются объектами типа bool, несмотря на имя типа.
4. vector<bool> заставляет всех пользователей применять предопределенный стандартом тип оптимизации. Вероятно, это не самая лучшая идея, даже если реальные накладные расходы, приводящие к снижению производительности, для данного компилятора и большинства приложений незначительны — ведь требования различных пользователей отличаются друг от друга.

И последний совет: *оптимизируйте с умом.* Никогда не поддавайтесь соблазну преждевременной оптимизации, пока вы не получите четких доказательств того, что оптимизация в вашей ситуации действительно принесет пользу.

Задача 1.14. Использование vector и deque — *Сложность: 3*

В чем заключается разница между vector и deque? Когда следует использовать каждый из них? Каким образом следует корректно уменьшать эти контейнеры, когда нам больше не нужна их полная емкость?

1. В стандартной библиотеке типы vector и deque предоставляют схожие сервисы. Какой из типов вы обычно используете? Почему? При каких условиях вы используете другой тип данных?
2. Что делает следующий фрагмент кода?

```
vector<C> c(1000);
c.erase(c.begin()+10, c.end());
c.reserve(10);
```

3. И vector, и deque обычно резервируют некоторую дополнительную память для предотвращения чрезмерно частого перераспределения памяти при добавлении новых элементов. Можно ли полностью очистить vector или deque (т.е. не только удалить все содержащиеся в них элементы, но и освободить всю зарезервированную память)? Продемонстрируйте, как это сделать, или поясните, почему этого сделать нельзя.

Предупреждение: ответы на вопросы 2 и 3 могут содержать определенные тонкости. Для каждого из этих вопросов имеется лежащий на поверхности ответ, но не останавливайтесь на нем. Постарайтесь быть предельно точными в своих ответах.

Решение

Эта задача отвечает на следующие вопросы.

- Можно ли использовать вектор взаимозаменяемо с массивом языка C (например, в функциях, которые работают с массивами)? Как? Какие при этом возникают проблемы?
- Почему, как правило, следует предпочитать использовать vector вместо deque?
- Каким образом можно "ужать" вектор, емкость которого выросла больше, чем требуется? В частности, как ужать его до размера, минимально необходимого для хранения содержащихся в нем элементов, и как сделать его полностью пустым, без содержимого и памяти, выделенной для его хранения. Можно ли использовать ту же методику и для других контейнеров?

Использование вектора как массива

Программисты на C++ строго придерживаются использования стандартного шаблона vector вместо массивов C. Использование vector проще и безопаснее использования массивов, поскольку вектор является более абстрактной и инкапсулированной формой контейнера, — например, при необходимости он может увеличиваться и уменьшаться, в то время как массив имеет фиксированный размер.

Тем не менее все еще имеется огромное количество кода, предназначенного для работы с массивами в стиле C. Рассмотрим, например, следующий код.

```
// Пример 1а. Функция, работающая с массивом C
//
int FindCustomer(
    const char* szName,     // Искомое имя
    Customer  * pCustomers, // Указатель на начало массива
    size_t      nLength )   // Количество элементов массива
{
    // Выполняется (возможно, оптимизированный) поиск и
    // возвращается индекс элемента с соответствующим именем
}
```

Для использования приведенной функции мы можем следующим образом создать, заполнить и передать массив.

```
// Пример 1б. Использование массива
//
Customer c[100];
```

```
    // ... каким-то образом заполняем массив c ...
int i = FindCustomer("Fred Jones", &c[0], 100);
```

Пример 1*б* вполне корректен, но, программируя в C++, как мы можем предложить использовать `vector` вместо массива? Было бы здорово иметь возможность использовать лучшее из двух миров — использовать `vector` из-за его удобства и безопасности, но при этом иметь возможность использовать его содержимое с функциями, работающими с массивами. Это желание чаще всего возникает в организациях, переходящих на C++, и программисты, создавая код в “новом и лучшем” стиле, остаются привязаны к унаследованному от времен C коду.

А что, если попытаться использовать вектор примерно следующим образом.

```
// Пример 1в. Попытка использования вектора
//
vector<Customer> c;
    // ... каким-то образом заполняем вектор c ...
int i = FindCustomer("Fred Jones", &c[0], c.size());
```

В примере 1*в* идея заключается в том, чтобы при вызове `FindCustomer()` передать ей адрес первого элемента и их количество, что очень похоже на то, что мы делали в примере 1*б*.

Перед тем как читать дальше, остановитесь и задумайтесь над следующими вопросами. В чем заключаются преимущества кода в примере 1*в* над кодом из примера 1*б*? Видите ли вы потенциальные проблемы в примере 1*в* или вы считаете, что все будет работать так, как предполагается?

Маленькая неприятность

Пример 1*в* не высосан из пальца. Оказывается, многие программисты поступают именно так, и не без причин: код из этого примера иллюстрирует ряд важных преимуществ при использовании векторов.

- Нам не надо заранее знать размер `c`. Во многих программах на C массивы выделяются с запасом, чтобы их размера оказалось достаточно для большинства предполагаемых применений. К сожалению, это означает, что память используется понапрасну (и, что еще хуже, программа не сможет продолжать работу, если ей потребуется больший объем памяти, чем мы предполагали). В случае `vector` мы можем вместо изначальных предположений при необходимости увеличивать размер контейнера. Если же мы заранее знаем количество объектов, размещаемых в контейнере, то мы можем устранить недостатки в производительности работы вектора по сравнению с массивом путем резервирования необходимого объема (например, вызовом `c.reserve(100)` для выделения места для 100 элементов); при этом не требуется перераспределение памяти при вставке каждого нового элемента в вектор.
- Нам не надо отдельно отслеживать действительную длину массива, которую затем следует передать в качестве параметра `nLength` функции `FindCustomer()`. Конечно, можно использовать массив C и различные обходные пути для более простого запоминания или вычисления его длины,[10] но ни один из этих путей не проще, чем запись `c.size()`.

[10] Наиболее распространенные подходы заключаются в использовании для размера массива константного значения или именованной через `#define` константы; часто используется макрос с выражением `sizeof(c)/sizeof(c[0])` для вычисления размера массива (только в случае статически объявленного, а не динамически распределенного массива. — *Прим. перев.*).

Вкратце, — насколько я знаю, — пример 1*в* всегда работает с любыми существующими реализациями `std::vector`.

До 2001 года значительным недостатком было отсутствие в стандарте C++ 1998 года [C++98] гарантии переносимости кода из примера 1*в* на любую платформу. Дело в том, что пример 1*в* предполагает внутреннее хранение объектов вектора одним непрерывным блоком в том же виде, что и в массиве, а стандарт не требовал от разработчиков именно такой реализации типа `vector`. Если найдется реализация вектора, в которой элементы будут храниться в какой-то иной форме, — например, в обратном порядке или с дополнительной информацией между элементами, — то, конечно, пример 1*в* корректно работать не будет. Это упущение исправлено в 2001 году в первом исправлении стандарта (Technical Corrigendum #1), где требуется, чтобы содержимое вектора хранилось непрерывным блоком, так же, как и массив.

Даже если ваш компилятор пока не поддерживает поправки к стандарту, пример 1*в* все равно должен работать корректно. Еще раз — я не знаю ни одной коммерческой реализации `vector`, где бы элементы хранились иначе, чем непрерывным блоком.

Подытоживая, следует дать рекомендацию вместо массивов использовать тип `vector` (или `deque` — см. далее).

vector или deque?

1. **В стандартной библиотеке типы `vector` и `deque` предоставляют схожие сервисы. Какой из типов вы обычно используете? Почему? При каких условиях вы используете другой тип данных?**

В стандарте C++ [C++98] в разделе 23.1.1 указано, какой из контейнеров следует предпочесть. В нем сказано следующее.

> *`vector` представляет собой тип последовательности, который следует использовать по умолчанию. ... `deque` представляет собой структуру данных, которую следует выбирать, когда большинство вставок и удалений выполняется в начале или конце последовательности.*

Интерфейсы `vector` и `deque` очень близки друг к другу и в общем случае эти типы взаимозаменяемы. Дек[11] имеет также члены `push_front()` и `pop_front()`, которых нет у вектора (да, у `deque` нет функций-членов `capacity()` и `reserve()`, которые есть у вектора, но это нельзя считать большой потерей; на самом деле эти функции — недостаток `vector`, что я вскоре продемонстрирую).

Основное структурное различие между `vector` и `deque` в том, как организовано внутреннее хранение их содержимого. `deque` распределяет хранимые им элементы по страницам, или “кускам” (chunks), с фиксированным количеством элементов в каждой странице. Именно поэтому `deque` часто сравнивают (и соответственно произносят) с колодой карт (deck of cards),[12] хотя на самом деле это название происходит от “double-ended queue” (двусторонняя очередь), в связи с отличительной особенностью дека, заключающейся в эффективной вставке элементов с обоих концов последовательности. Вектор же выделяет для хранения данных один непрерывный блок памяти и эффективно вставлять элементы может только в конец последовательности.

Страничная организация дека дает ряд преимуществ.

- Дек позволяет выполнять операции вставки и удаления в начале контейнера, в то время как вектор такой способностью не обладает, откуда и вытекает приме-

[11] Здесь и далее используется общепринятый перевод названия этой структуры данных на русский язык (см., например, русское издание [Knuth97]). — *Прим. перев.*

[12] Насколько я знаю, первым аналогию `deque` с колодой карт провел Д. Кнут в [Knuth97].

чание в стандарте об использовании дека при необходимости выполнения вставки и удаления в концах последовательности.

- Дек работает с памятью способом, более эффективно использующим особенности операционных систем. Например, 10-мегабайтный вектор использует единый блок памяти размером в 10 Мбайт. В некоторых операционных системах такой единый блок может оказаться менее эффективным на практике, чем 10-мегабайтный дек, который может размещаться в серии меньших блоков памяти. В других операционных системах непрерывное адресное пространство само может быть построено из блоков меньшего размера, так что здесь дек не дает никакого выигрыша.
- Дек более прост в использовании и по своей природе более эффективен в смысле роста, чем вектор. Единственными операциями, которые есть у вектора и отсутствуют у дека, являются `capacity()` и `reserve()`. Их нет у дека, поскольку он в этих операциях попросту не нуждается! В случае вектора вызов `reserve()` перед большим количеством вызовов `push_back()` позволяет устранить перераспределение памяти для постоянно растущего буфера всякий раз, когда его размера становится недостаточно для добавления нового элемента. У дека такой проблемы нет, и возможность вызова `deque::reserve()` перед выполнением большого количества `push_back()` не устранила бы ни излишнее перераспределение памяти, ни какую-либо другую работу. Дек в этом случае получил бы то же дополнительное количество страниц, что и при поочередном добавлении элементов.

Интересно заметить, что в стандарте C++ для создания стека предпочитается использование дека. Хотя стек растет только в одном направлении и никогда не требует вставки в средину иди другой конец последовательности элементов, реализация по умолчанию должна использовать дек.

```
namespace std
{
    template<typename T, typename Container = deque<T> >
    class stack
    {
        // ...
    };
}
```

Несмотря на это, для простоты и удобочитаемости предпочитайте по умолчанию использовать в своих программах векторы. Отступайте от этого правила только в том случае, когда вам требуется эффективная вставка в начало и конец контейнера (и при этом не обязательно использовать для хранения элементов непрерывный блок памяти). Вектор в C++ является тем же, чем и массив в C — типом контейнера по умолчанию для использования в ваших программах, который вполне успешно справляется со всеми своими обязанностями до тех пор, пока вам не потребуется что-то иное.

Несжимаемый вектор

Как уже упоминалось, вектор сам управляет хранением своих данных, и мы можем помочь ему подсказкой о том, какую емкость ему следует иметь для размещения добавляемых данных. Если у нас есть `vector<T> c`, в котором мы намерены разместить 100 элементов, мы можем сначала вызвать `c.reserve(100)`, чтобы перераспределить память для этих элементов только один раз, обеспечивая максимальную эффективность роста вектора.

Но что делать в противоположном случае? Что, если у нас имеется очень большой вектор, из которого мы удалили много элементов и хотим, чтобы вектор уменьшил

свои размеры, т.е. чтобы он избавился от излишней емкости. Можно подумать, что этого можно добиться следующим образом.

2. Что делает следующий фрагмент кода?

```
// Пример 2а. Наивная попытка сжать вектор
//
vector<C> c(1000);
```

Сейчас `c.size() == 1000` и `c.capacity() >= 1000`. Удалим все элементы, кроме первых десяти.

```
c.erase(c.begin()+10, c.end());
```

Теперь `c.size() == 10`; пусть нам известно, что больше в программе вектор расти не будет. Но наша пытка

```
c.reserve(10);
```

не приводит к уменьшению размера внутреннего буфера вектора. Последняя строка из примера 2*а* не делает того, на что мы надеялись, — она не уменьшает емкость вектора. Вызов `reserve()` может только увеличивать емкость вектора (и ничего не делает, если емкость достаточна).

К счастью, имеется корректный путь получения желаемого эффекта.

```
// Пример 2б. Корректное решение уменьшения емкости
//            вектора до объема его элементов
vector<Customer> c(10000);
// ... теперь c.capacity() >= 10000 ...

// Удаляем все элементы, кроме первых 10
c.erase(c.begin()+10, c.end());

// Следующая строка сжимает внутренний буфер вектора
// до размера его элементов
vector<Customer>(c).swap(c);

// ... теперь c.capacity() == c.size(), или,
// возможно, немного превосходит эту величину ...
```

Вам понятно, как работает приведенный код? Здесь есть свои хитрости.

1. Сначала мы создаем временный (безымянный) `vector<Customer>` и инициализируем его содержимым `c`. Основное отличие между временным вектором и `c` в том, что в то время как емкость `c` гораздо больше размера хранящихся в нем элементов, емкость временного вектора в точности соответствует количеству хранящихся в нем элементов (в некоторых реализациях она может немного превышать это значение).
2. Затем мы обмениваем содержимое безымянного вектора и вектора `c`. Теперь буфер слишком большого размера принадлежит временному вектору, в то время как размер буфера вектора `c` теперь соответствует количеству хранящихся в нем элементов.
3. И наконец, временный вектор выходит из области видимости, унося с собой в небытие и буфер чрезмерного размера, который будет удален при уничтожении временного объекта. Мы остаемся с вектором `c`, внутренний буфер которого теперь имеет “правильную” емкость.

Заметим, что описанная процедура весьма эффективна и не выполняет никаких излишних действий. Даже если бы вектор имел специальную функцию-член `shrink_to_fit()`, она бы выполняла ту же работу, что и описанная только что.

3. **И `vector`, и `deque` обычно резервируют некоторую дополнительную память для предотвращения чрезмерно частого перераспределения памяти при добавлении новых элементов. Можно ли полностью очистить `vector` или `deque` (т.е. не только удалить все содержащиеся в них элементы, но и освободить всю зарезервированную память)? Продемонстрируйте, как это сделать, или поясните, почему этого сделать нельзя.**

Код для полной очистки вектора (в результате которой в нем не будет никакого содержимого и никакой излишней емкости) практически идентичен приведенному ранее. Просто в этот раз мы используем пустой безымянный вектор.

```
// Пример 3. Корректная очистка вектора
//
vector<Customer> c(10000);
// ... теперь c.capacity() >= 10000 ...

// Следующая строка делает вектор с пустым
vector<Customer>().swap(c);

// ... теперь c.capacity() == 0 (кроме тех реализаций,
// где даже для пустых векторов предусмотрена некоторая
// минимальная емкость ...
```

Обратите внимание, что, как в предыдущем случае конкретные реализации вектора могли оставлять определенное количество зарезервированной памяти, так и в данном случае некоторые реализации вектора могут резервировать некоторое количество памяти даже в случае пустого вектора. Приведенный метод тем не менее гарантирует, что это будет минимальный резервируемый объем, такой же, как и для пустого вектора.

Этот способ применим и к деку, но не применим к таким классам, как `list`, `set`, `map`, `multiset` или `multimap`, выделение памяти в которых всегда происходит только при необходимости, так что в них никогда не должна возникать проблема излишней емкости.

Резюме

Векторы можно безопасно использовать вместо массивов C, и следует всегда отдавать предпочтение векторам или декам перед массивами. Используйте вектор в качестве типа контейнера по умолчанию, если только вы не уверены, что вам требуется какой-то иной тип и не требуется хранения элементов контейнера одним непрерывным блоком. Для сжатия существующих векторов и деков воспользуйтесь идиомой обмена с временным контейнером. И наконец, как упоминалось в задаче 1.13, если только вам не требуется оптимизация использования памяти, используйте `deque<bool>` вместо `vector<bool>`.

Задача 1.15. Использование `set` и `map` — *Сложность: 5*

В задаче 1.14 мы рассматривали vector и deque. Сейчас мы остановимся на ассоциативных контейнерах `set`, `multiset`, `map` и `multimap`[13].

1. а) Что неверно в приведенном коде? Каким образом его можно исправить?

```
map<int,string>::iterator i = m.find(13);
if (i != m.end())
```

[13] Заметим, что `set` и `multiset` не являются действительно “ассоциативными” в обычном смысле связывания ключа поиска с другим значением, как в словаре. Это делается в `map` и `multimap`. Однако все эти контейнеры в стандарте C++ описаны под заголовком “Ассоциативные контейнеры”, так что я решил следовать стандарту.

```
{
    const_cast<int&>(i->first) = 9999999;
}
```

б) В какой степени проблемы решаются, если изменить код следующим образом?

```
map<int,string>::iterator i = m.find(13);
if (i != m.end())
{
    string s = i->second;
    m.erase(i);
    m.insert(make_pair(9999999,s));
}
```

2. Можно ли модифицировать содержимое контейнера `set` посредством `set::iterator`? Почему да или почему нет?

Решение

Ассоциативные контейнеры: обзор

Сначала познакомимся с кратким обзором ассоциативных контейнеров и узнаем, что собой представляют и как работают их итераторы.

На рис. 1.1 приведены четыре стандартных ассоциативных контейнера. Каждый из них поддерживает внутреннее представление элементов, обеспечивающее быстрый обход в неубывающем порядке ключей и быструю выборку по ключу; ключи всегда сравниваются в соответствии с предоставляемым контейнеру типом `Compare` (по умолчанию это `less<Key>`, предоставляющий `operator<()` для объектов `Key`). При вставке нового ключа контейнер находит соответствующее место для вставки, при котором сохраняется корректное упорядочение внутренней структуры данных.

```
set<Key, Compare>
multiset<Key, Compare>
map<Key, Value, Compare>
multimap<Key, Value, Compare>
```

Рис. 1.1. Стандартные ассоциативные контейнеры (распределители памяти опущены)

Итераторы просты для реализации, поскольку они предполагают, что обход выполняется в неубывающем порядке значений ключа, и контейнер хранит элементы так, чтобы естественным образом поддерживать это упорядочение.

Требования к ключу

Главное требование, соблюдение которого необходимо для правильной работы контейнеров, заключается в том, что, будучи вставленным в контейнер, ключ не должен изменять свое значение способом, который может привести к изменению его относительного положения в контейнере. Если это случится, контейнер не будет знать о происшедшем, и его предположения об упорядоченности элементов окажутся нарушенными. Соответственно, поиск требуемых записей окажется неверным, итераторы перестанут обходить контейнер упорядоченно, и, вообще говоря, в результате могут произойти разные Ужасные Вещи.

Я продемонстрирую вам один конкретный пример, а потом мы посмотрим, что можно сделать в этом случае.

Поясняющий пример

Рассмотрим объект m типа map<int, string>, содержимое которого показано на рис. 1.2. Каждый узел m показан в виде пары pair<const int, string>. Я представляю внутреннюю структуру объекта в виде дерева, поскольку именно такой способ используется во всех реально имеющихся реализациях стандартной библиотеки шаблонов.

При вставке ключей поддерживается структура дерева, обеспечивающая выполнение обычного неупорядоченного обхода дерева в порядке less<int>. До этого момента все идет хорошо.

Однако предположим теперь, что посредством итератора мы изменяем ключ второго элемента, используя примерно следующий код.

1. а) Что неверно в приведенном коде? Каким образом его можно исправить?

```
// Пример 1. Некорректный способ изменения
// ключа в map<int, string> m.
//
map<int,string>::iterator i = m.find(13);
if (i != m.end())
{
    const_cast<int&>(i->first) = 9999999;
}
```

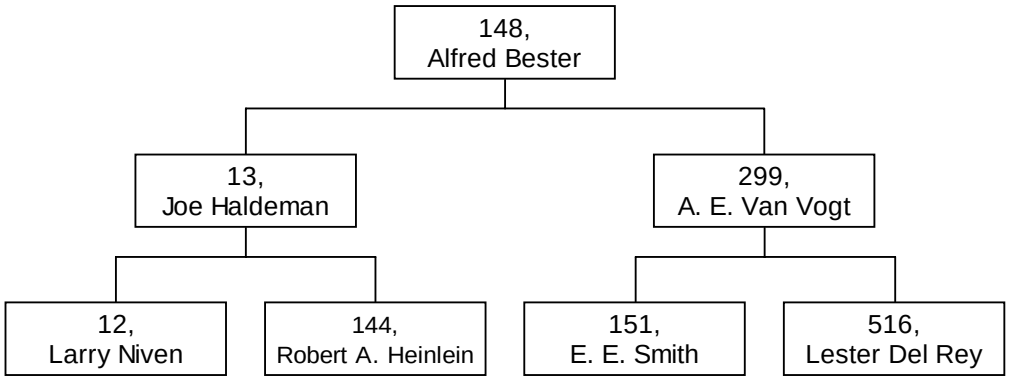


Рис. 1.2. Пример внутреннего содержимого map<int, string>

Заметим, что для того, чтобы этот код компилировался, требуется явное приведение константного типа. Остановимся на этом моменте. Проблема заключается в том, что код изменяет внутреннее представление объекта map не предусмотренным образом; соответственно, результаты такого воздействия не могут быть корректно обработаны объектом.

Код из примера 1 портит внутреннюю структуру объекта map (рис. 1.3). Теперь, например, обход с использованием итератора вернет содержимое объекта map не в упорядоченном по значениям ключа порядке, как этого можно было бы ожидать. Поиск ключа 144, вероятно, будет неуспешен, несмотря на наличие такого ключа в контейнере. Вообще говоря, контейнер оказывается в рассогласованном и непригодном для использования состоянии. Заметим, что требовать от типа map автоматической защиты от такого недозволенного вмешательства бессмысленно — такое вмешательст-

во попросту не может быть обнаружено. В примере 1 мы выполнили изменения посредством ссылки на элемент контейнера, без вызова какой-либо функции-члена.

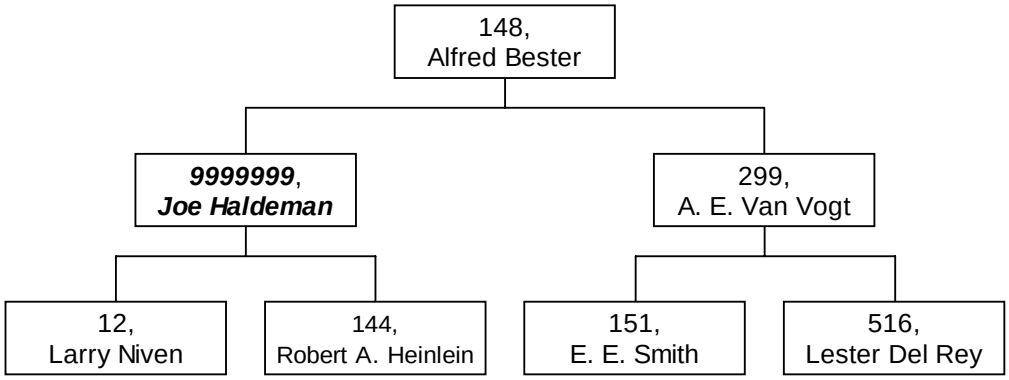


Рис. 1.3. Внутреннее содержимое `map<int, string>` с рис. 1.2 после выполнения фрагмента кода из примера 1

Если бы код из примера изменил значение ключа таким образом, что относительный порядок элементов остался неизменным, то никаких проблем в данной реализации `map` не возникло бы. Проблема возникает только тогда, когда код пытается изменить относительный порядок ключей, когда они уже находятся в контейнере.

Каков же лучший способ решения этой проблемы? В идеале мы бы хотели предотвратить такое вмешательство, используя соответствующее стандартное правило программирования. Что же должно гласить это правило?

Вариант 1. const означает const! (недостаточное решение)

В примере 1 для того, чтобы изменить ключ, нам требуется явное приведение типа для устранения его константности. Это связано с тем, что стандарт C++ активно пытается предотвратить код от изменений относительного порядка ключей. На самом деле стандарт C++ еще более строг в этом отношении и требует, чтобы ключи в типах `map` и `multimap` не изменялись вообще. Итератор `map<Key,Value>::iterator` указывает на `pair<const Key,Value>`, что, соответственно, позволяет вам модифицировать часть, представляющую значение, но не ключ. Это, в свою очередь, предохраняет ключ от любых изменений, и уж тем более от изменений, при которых изменяется относительное положение ключа во внутреннем представлении объекта типа `map`.

Следовательно, одна возможность решения проблемы — напечатать большой плакат с лозунгом "**`const` — значит `const`!!!**", собрать митинг и пригласить на него знаменитых ораторов, купить время на телевидении и разрекламировать в прессе и довести эту простую истину до программистов. Правда, это надежно гарантированно убережет от кода наподобие приведенного в примере 1?

Неплохая идея, но этого явно недостаточно. Например, если тип `Key` имеет член, объявленный как `mutable`, влияющий на способ сравнения посредством `Compare` объектов типа `Key`, то вызов константной функции-члена может изменить относительный порядок ключей.

Вариант 2. Изменение путем удаления и вставки (уже лучше!)

Лучшее, но все еще не окончательное решение состоит в следовании следующей стратегии: для изменения ключа следует удалить его и вставить повторно.

б) В какой степени проблемы решаются, если изменить код следующим образом?

```
// Пример 2. Более корректный способ изменения
// ключа в map<int, string> m.
//
map<int,string>::iterator i = m.find(13);
if (i != m.end())
{
    string s = i->second;
    m.erase(i);
    m.insert(make_pair(9999999,s));
}
```

Этот способ лучше предыдущего, поскольку позволяет избежать любых изменений ключей, даже ключей с `mutable`-членами, играющими роль при упорядочении. Этот способ работает даже с нашим примером. Так что, это и есть окончательное решение?

К сожалению, в общем случае этого недостаточно, поскольку ключи все еще могут быть изменены в тот момент, когда они находятся в контейнере. “Что?! — можете спросить вы. — Как же ключи могут быть изменены, находясь в контейнере, если мы приняли решение никогда не изменять объекты непосредственно?” Вот два контрпримера.

1. Пусть, например, тип `Key` имеет некоторую структуру, доступ к которой может получить некоторый внешний код, например, указатель на совместно используемый буфер, который может быть модифицирован другими частями системы без участия объекта `Key`. Кроме того, пусть эта доступная извне структура участвует в сравнениях, выполняемых `Compare`. В таком случае возможно внесение таких изменений в описанную структуру без каких-либо знаний об объекте `Key` или о коде, использующем ассоциативный контейнер, которые приведут к изменению относительного упорядочения ключей. Таким образом, в этом случае, даже при строгом следовании стратегии удаления и вставки, все равно оказывается возможным изменение относительного порядка ключей.
2. Рассмотрим тип `Key`, представляющий собой `string`, и тип `Compare`, который рассматривает ключ как имя файла и выполняет сравнение *содержимого* файлов. Таким образом, даже при неизменных ключах относительный их порядок может быть изменен при изменении файлов другим *процессом* или даже в случае совместного использования файла в сети другим пользователем, работающим на другой машине, расположенной на другом конце света.

Как работает класс set

Рассмотренный материал привел нас к типу `set`. Поскольку мы так и не нашли полностью удовлетворяющего нас ответа для `map`, посмотрим, как работает тип `set`.

2. Можно ли модифицировать содержимое контейнера `set` посредством `set::iterator`? Почему да или почему нет?

Тип `set` можно представить как тип `map<Key,Value>` в предельном случае, когда тип `Value` представляет собой `void`. Можно ожидать, что тип `set` будет работать с ключами так же, как и `map`, и что оба итератора — `set<Key>::iterator` и

set<Key>::const_iterator — должны указывать на const Key. Это должно предостеречь программистов от модификации указываемых ключей.

Увы, в этом вопросе в случае set стандарт не так ясен, как в случае map. В конце концов, содержащийся в set объект может содержать не только информацию, влияющую на результат сравнения, так что вполне реальна ситуация, когда возможность внесения изменений может оказаться желательной. Имеется два различных мнения о том, должен ли итератор set::iterator указывать на неконстантный объект, так что вас не должно удивлять, что в одних реализациях стандартной библиотеки этот итератор указывает на константный объект, а в других — нет. Другими словами, на практике вы не можете переносимо использовать возможность модификации элемента множества set посредством set::iterator без применения const_cast. В настоящий момент в комитет по стандартам внесено предложение формально потребовать, чтобы как set::iterator, так и set::const_iterator были константными итераторами, так что для внесения изменений следует использовать такое мощное орудие, как const_cast.

Однако для использования этого орудия имеются веские причины. Вспомним, что set и map предназначены для разных целей. Пусть, например, мы хотим реализовать поиск по заданному имени клиента его адреса или номера телефона. В таком случае мы можем использовать класс map<string, CustData>, где ключом является имя клиента, а значением — структура, содержащая его адрес и номер телефона.

Но что если у нас есть детально разработанный класс Customer, в котором имеется как информация об адресе и телефоне клиента, так и его имя? В таком случае создание нового класса CustData, содержащего всю ту же информацию, но без имени клиента, явно излишне. Конечно, можно использовать класс map<string, Customer>, но при этом явно избыточным окажется использование имени клиента, которое содержится в классе Customer. В действительности нет никакой необходимости дважды сохранять имя клиента. Вместо этого можно воспользоваться классом set<Customer>, который не требует новой структуры данных и дублирования имени. map<string, Customer> позволяет изменять объект типа Customer, находящийся в контейнере, и так же должен поступать и set<Customer>, но в действительности вы должны делать это с использованием const_cast. Ряд современных реализаций set позволяет сделать это без приведения типа, другие — нет. Но даже если используемая вами реализация set позволяет изменять объекты в контейнере, помните, что вы не должны изменять объект таким образом, чтобы изменился относительный порядок в контейнере.

Наши попытки написать правила для использования map не охватывали все возможные ситуации, а с учетом того, что set и map работают немного по-разному, пожалуй, лучше всего сформулировать общее правило.

Ключевое требование

Какому же правилу надо следовать для того, чтобы гарантировать корректное использование ассоциативного контейнера? Хотя хотелось бы указать более точную стратегию, описать ее как простое правило кодирования не удается. Так что мы не можем поступить более точно, чем, описав правило, добавить: “Вы должны знать, что вы делаете”.

Ключевое правило ассоциативного контейнера

После того как ключ вставлен в ассоциативный контейнер, он никогда не должен изменять свое относительное положение в этом контейнере.

Не забывайте как о прямых, так и о косвенных путях, которыми ключ может изменить относительный порядок, и избегайте их. Тем самым вы избежите массы неприятностей при работе со стандартными ассоциативными контейнерами.

Задача 1.16. Эквивалентный код? *Сложность: 5*

Могут ли незначительные различия в коде иметь большое значение, в частности, в таких простых случаях, как постинкремент параметра функции? Эта задача посвящена изучению интересных взаимодействий, которые становятся важными в коде в стиле STL.

1. Опишите, что делает следующий код.

```
// Пример 1.
//
f( a++ );
```

   Дайте как можно более полный ответ.

2. В чем состоит различие, если таковое имеется, между двумя приведенными фрагментами кода?

```
// Пример 2а.
//
f( a++ );
```

```
// Пример 2б.
//
f( a );
a++;
```

3. В вопросе 2 сделаем предположение, что `f()` — функция, принимающая аргумент по значению, и что класс объекта имеет `operator++(int)` с естественной семантикой. В чем теперь состоит различие (если таковое имеется) между примерами 2*а* и 2*б*?

Решение

1. **Опишите, что делает следующий код.**

```
// Пример 1.
//
f( a++ );
```

   Дайте как можно более полный ответ.

Полный список может выглядеть просто устрашающе, так что перечислим только основные возможности.

Итак, `f` может представлять собой следующее.

1. Макрос

В этом случае инструкция может означать почти что угодно, и `a++` может вычисляться как множество раз, так и ни одного раза.

```
#define   f(x)   x                      // 1 раз
#define   f(x)   (x,x,x,x,x,x,x,x,x)    // 9 раз
#define   f(x)                          // Ни разу
```

Рекомендация

Избегайте использования макросов. Это обычно затрудняет понимание и, соответственно, сопровождение кода.

2. Функция

В этом случае сначала вычисляется a++, после чего результат передается функции в качестве параметра. Обычно постинкремент возвращает первоначальное значение a в виде временного объекта, так что f() может получать параметры либо по значению, либо по ссылке на const объект, но не по ссылке на неконстантный объект, поскольку такая ссылка не может быть связана с временным объектом.

3. Объект

В этом случае f может быть объектом-функцией, т.е. объектом, для которого определен operator()(). Опять-таки, если постинкремент возвращает, как и должен, значение a, то operator()() объекта f может принимать параметры либо по значению, либо по ссылке на const объект.

4. Имя типа

В этом случае инструкция сначала вычисляет a++ и использует результат этого выражения для инициализации временного объекта типа f.

* * * * *

В свою очередь, a может представлять собой следующее.

1. Макрос

В этом случае a может означать почти что угодно.

2. Объект (возможно, встроенного типа)

В этом случае для типа должен быть определен подходящий оператор постинкремента operator++(int).

Обычно постинкремент реализуется посредством преинкремента и возвращает первоначальное значение a.

```
// Каноническая форма постинкремента
T T::operator++(int)
{
    T old( *this );  // Запоминаем начальное значение

    ++*this;         // Реализуем постинкремент путем
                     // использования преинкремента

    return old;      // Возвращаем первоначальное значение
}
```

При перегрузке оператора вы имеете возможность изменить его обычную семантику на нечто необычное. Например, следующая перегрузка сделает для большинства видов f выполнение кода из примера 1 невозможным (здесь a имеет тип A).

```
void A::operator++(int) // Постинкремент ничего не возвращает
```

Не поступайте так. Следуйте разумному правилу.

Рекомендация

Всегда сохраняйте естественную семантику при перегрузке операторов. “Поступайте так, как поступает *int”, — т.е. следуйте семантике встроенных типов [Meyers96].*

3. Значение, такое как адрес

Например, a может быть указателем.

Побочные действия

В остальной части данной задачи для простоты я полагаю, что f() не является макросом и что a — объект с естественной постинкрементной семантикой.

2. В чем состоит различие, если таковое имеется, между двумя приведенными фрагментами кода?

```
// Пример 2a.
//
f( a++ );
```

В примере 2*а* выполняется следующее.

1. a++: увеличивается значение a и возвращается его старое значение.
2. f(): выполняется f(), которой в качестве аргумента передается старое значение a.

Пример 2*а* гарантирует выполнение постинкремента, и, следовательно, a получает новое значение до выполнения f(). Как уже упоминалось, f() может быть функцией, объектом-функцией или именем типа, приводящим к вызову конструктора.

Некоторые стандарты кодирования требуют, чтобы операции типа ++ всегда располагались в отдельных строках в связи с определенной опасностью выполнения нескольких операций типа ++ в одной инструкции (см. задачи 2.16 и 2.17). Такие стандарты предусматривают стиль, использованный в примере 2*б*.

```
// Пример 2б.
//
f( a );
a++;
```

В этом примере выполняются следующие действия.

1. f(): выполняется f(), которой в качестве аргумента передается старое значение a.
2. a++: увеличивается значение a и возвращается его старое значение, которое игнорируется.

В обоих случаях f() получает старое значение a. “Так в чем же такое большое различие?” — спросите вы. Дело в том, что пример 2*б* иногда дает не тот же результат, что 2*а*, поскольку выполнение постинкремента и получение объектом a нового значения происходит уже после того, как будет выполнен вызов f().

Это приводит к двум основным последствиям. Во-первых, если f() генерирует исключение, в примере 2*а* гарантируется полное выполнение a++ и всех его побочных действий; в примере же 2*б* гарантируется, что a++ выполнено не будет и что какие-либо его побочные действия не будут иметь место.

Во-вторых, даже в случае отсутствия исключений, при наличии побочных действий порядок выполнения f() и a.operator++(int) может иметь значение. Рассмотрим, например, что произойдет, если f() имеет побочное действие, влияющее на состояние a. Такое предположение вовсе не притянуто за уши и не так уж редко встречается на практике и может произойти даже в том случае, когда f() не изменяет (да и не может) a непосредственно. Проиллюстрируем это на примере.

Спички, дороги и итераторы

3. В вопросе 2 сделаем предположение, что `f()` — функция, принимающая аргумент по значению, и что класс объекта имеет `operator++(int)` с естественной семантикой. В чем теперь состоит различие (если таковое имеется) между примерами 2*а* и 2*б*?

Отличие состоит в том, что код из примера 2*а* может быть корректен, а код из примера 2*б* — нет. Это связано с тем, что в примере 2*а* имеется период времени, когда одновременно существуют объекты со старым и новым значениями `a`. В примере 2*б* такого перекрытия нет.

Рассмотрим, что произойдет, если мы заменим `f()` функцией `list::erase()`, а `a` — `list::iterator`. Первый пример остается корректен.

```
// Пример 3а
//
// l имеет тип list<int>
// i - корректный не конечный итератор l
l.erase(i++); // OK, увеличивается корректный итератор
```

Второй же пример в этом случае некорректен.

```
// Пример 3б
//
// l имеет тип list<int>
// i - корректный не конечный итератор l
l.erase(i);
i++; // Ошибка, i - некорректный итератор
```

Причина некорректности примера 3*б* заключается в том, что вызов `l.erase(i)` делает итератор `i` недействительным, так что после этого не может быть вызван его `operator++()`.

Предупреждение. Некоторые программисты используют код в стиле примера 3*б*, в основном руководствуясь рекомендацией удалять операторы типа ++ из вызовов функций. Возможно даже, что они постоянно (и безнаказанно) поступают именно так, и ни разу не сталкивались с неприятностями — в основном потому, что они работают с конкретной версией компилятора и библиотеки. Но знайте: код, наподобие приведенного в примере 3*б*, непереносим, и вы можете получить массу неприятностей при переходе на другой компилятор (или даже при обновлении версии). Когда это произойдет, вам придется потратить массу усилий на поиск неполадок, так как ошибку использования некорректного итератора очень трудно обнаружить. Заботливые мамаши-программистки всегда учат своих детей хорошо себя вести (и нам стоит прислушаться к их советам).

1. Не играть со спичками.
2. Не играть на проезжей части.
3. Не играть с недействительными итераторами.

Но, вообще говоря, за исключением примеров типа рассмотренного, лучше все же следовать приемам, описанным в задачах 2.16 и 2.17 и избегать использования операторов, наподобие ++, в вызовах функций.

Задача 1.17. Специализация и перегрузка шаблонов *Сложность: 6*

Каким образом создаются специализированные и перегруженные шаблоны? Как определить, какой из шаблонов будет вызван? Проверьте себя на приведенных ниже примерах.

1. Что такое специализация шаблона? Приведите пример.
2. Что такое частичная специализация? Приведите пример.

3. Рассмотрим следующие объявления:

```
template<typename T1, typename T2>
void g(T1, T2);                          //  1
template<typename T> void g( T );        //  2
template<typename T> void g( T, T );     //  3
template<typename T> void g( T* );       //  4
template<typename T> void g( T*, T );    //  5
template<typename T> void g( T, T* );    //  6
template<typename T> void g( int, T* );  //  7
template<> void g<int>( int );           //  8
void g( int, double );                   //  9
void g( int );                           // 10
```

Какие из этих функций будут вызваны в каждой из следующих инструкций? Где это возможно, укажите типы параметров шаблонов.

```
int             i;
double          d;
float           f;
complex<double> c;

g( i );           // a
g<int>( i );      // b
g( i, i );        // c
g( c );           // d
g( i, f );        // e
g( i, d );        // f
g( c, &c );       // g
g( i, &d );       // h
g( &d, d );       // i
g( &d );          // j
g( d, &i );       // k
g( &i, &i );      // l
```

Решение

Шаблоны обеспечивают наиболее мощный вид обобщенности в C++. Они позволяют вам написать обобщенный код, который способен работать со многими типами несвязанных объектов, — например, разработать строки, содержащие разные виды символов, контейнеры, способные хранить объекты произвольных типов, алгоритмы, которые могут работать с последовательностями разных типов.

1. Что такое специализация шаблона? Приведите пример.

Специализации шаблонов позволяют последним отдельно работать с частными случаями. Иногда обобщенный алгоритм способен работать гораздо более эффективно для последовательностей определенного типа (например, при наличии итераторов с произвольным доступом), так что для такого случая имеет смысл разработать отдельную специализацию, а для всех остальных ситуаций использовать обобщенный, хотя и более медленный алгоритм. Производительность — наиболее распространенная причина использования специализаций, но далеко не единственная. Например, вы можете специализировать шаблон для работы с некоторыми объектами, которые не соответствуют интерфейсу, необходимому для работы обобщенного шаблона.

Для работы с такими частными случаями могут использоваться два метода — явная специализация и частичная специализация.

Явная специализация

Явная специализация позволяет вам написать конкретную реализацию для некоторой комбинации параметров шаблонов. Например, рассмотрим следующий шаблон функции.

```
template<typename T> void sort( Array<T>&v ) { /* ... */ };
```

Если имеется более быстрый (или отличающийся каким-либо другим образом) способ работы с массивом элементов типа `char*`, мы можем явно специализировать `sort()` для этого случая.

```
template<> void sort<char*>( Array<char*>& );
```

После этого в процессе компиляции компилятор будет выбирать наиболее точно соответствующий шаблон, например:

```
Array<int>   ai;
Array<char*> apc;

sort( ai );        // Вызов sort<int>
sort( apc );       // Вызов специализации sort<char*>
```

Частичная специализация

2. Что такое частичная специализация? Приведите пример.

При работе с шаблонами классов можно определить частичные специализации, которые не обязаны определять все исходные (неспециализированные) параметры шаблона класса.

Вот пример из стандарта C++ ([C++98], раздел 14.5.4 [temp.class.spec]). Первый шаблон представляет собой исходный шаблон класса.

```
template<class T1, class T2, int I>
         class A             { }; // #1
```

Мы можем специализировать его для случая, когда `T2` представляет собой `T1*`.

```
template<class T, int I>
         class A<T, T*, I>   { }; // #2
```

Для случая, когда `T1` является указателем.

```
template<class T1, class T2, int I>
         class A<T1*, T2, I> { }; // #3
```

Для случая, когда `T1` представляет собой `int`, `T2` — произвольный указатель, а `I` равно 5.

```
template<class T>
         class A<int, T*, 5> { }; // #4
```

Для случая, когда `T2` представляет собой произвольный указатель.

```
template<class T1, class T2, int I>
         class A<T1, T2*, I> { }; // #5
```

Объявления со второго по пятое объявляют частичные специализации исходного шаблона. При наличии этих объявлений компилятор будет выбирать наиболее подходящий из указанных шаблонов. В стандарте C++ ([C++98], раздел 14.5.4.1) приведены следующие примеры.

```
A<int, int, 1>   a1;  // Используется #1
A<int, int*, 1>  a2;  // Используется #2, T - int, I - 1
A<int, char*, 5> a3;  // Используется #4, T - char
A<int, char*, 1> a4;  // Используется #5,
```

```
                        // T1 - int, T2 - char, I - 1
A<int*, int*, 2> a5;    // Неоднозначность: подходят
                        // как #3, так и #5
```

Перегрузка шаблонов функций

Рассмотрим теперь перегрузку шаблонов функций. Перегрузка — не то же, что и специализация, но достаточно тесно связана с ней.

C++ позволяет вам перегружать функции, гарантируя, что будет вызвана необходимая в данной ситуации функция.

```
int  f( int );
long f( double );

int    i;
double d;

f( i );  // Вызывается f( int )
f( d );  // Вызывается f( double )
```

Точно так же можно перегружать и шаблоны функций, что и приводит нас к последнему вопросу.

3. Рассмотрим следующие объявления.

```
template<typename T1, typename T2>
void g(T1, T2);                          //  1
template<typename T> void g( T );        //  2
template<typename T> void g( T, T );     //  3
template<typename T> void g( T* );       //  4
template<typename T> void g( T*, T );    //  5
template<typename T> void g( T, T* );    //  6
template<typename T> void g( int, T* );  //  7
template<> void g<int>( int );           //  8
void g( int, double );                   //  9
void g( int );                           // 10
```

Для начала немного упростим свою задачу, заметив, что здесь имеются две группы перегруженных функций `g()` — с одним и с двумя параметрами.

```
template<typename T1, typename T2>
void g(T1, T2);                          //  1
template<typename T> void g( T, T );     //  3
template<typename T> void g( T*, T );    //  5
template<typename T> void g( T, T* );    //  6
template<typename T> void g( int, T* );  //  7
void g( int, double );                   //  9

template<typename T> void g( T );        //  2
template<typename T> void g( T* );       //  4
template<> void g<int>( int );           //  8
void g( int );                           // 10
```

Заметьте, что я сознательно не усложнял задачу, добавляя перегрузку с двумя параметрами, где второй параметр имеет значение по умолчанию. Если бы в задании были такие функции, то при решении задачи они должны были бы рассматриваться в обоих списках — и как функции с одним параметром (и вторым со значением по умолчанию), и как функции с двумя параметрами.

Теперь поочередно рассмотрим каждый из вызовов.

Какие из этих функций будут вызваны в каждой из следующих инструкций? Где это возможно, укажите типы параметров шаблонов.

```
int             i;
double          d;
float           f;
complex<double> c;

g( i );           // a
```

Здесь вызывается № 10, поскольку вызов в точности соответствует этому объявлению, а нешаблонные функции всегда имеют преимущество перед шаблонами (см. раздел 13.3.3 стандарта).

```
g<int>( i );      // b
```

Здесь происходит вызов № 8, поскольку этот шаблон абсолютно точно соответствует `g<int>()`.

```
g( i, i );        // c
```

Здесь вызывается функция № 3 (где `T` представляет собой `int`), поскольку именно она наиболее соответствует вызову `g(i, i)`.

```
g( c );           // d
```

Здесь вызывается функция № 2 (где `T` представляет собой `complex<double>`), поскольку данному вызову не соответствует никакой другой шаблон.

```
g( i, f );        // e
```

Здесь вызывается функция № 1 (где `T1` представляет собой `int`, а `T2` — `float`). Вы можете решить, что в большей степени данному случаю соответствует шаблон № 9, но нешаблонная функция имеет преимущество перед шаблоном только при абсолютно точном соответствии. В случае же № 9 соответствие близкое, но не точное.

```
g( i, d );        // f
```

Здесь используется вызов функции № 9, так как она точно соответствует вызову и имеет преимущество, так как не является шаблоном.

```
g( c, &c );       // g
```

Здесь вызывается функция № 6 (где `T` представляет собой `complex<double>`), поскольку этот шаблон является наиболее соответствующей перегруженной функцией. Шаблон № 6 представляет перегрузку `g()`, где второй параметр представляет собой указатель на тот же тип, что и у первого параметра.

```
g( i, &d );       // h
```

Здесь вызывается функция № 7 (где `T` представляет собой `double`), поскольку этот шаблон является наиболее соответствующей перегруженной функцией.

```
g( &d, d );       // i
```

Здесь вызывается функция № 5 (где `T` представляет собой `double`). Шаблон № 5 представляет перегрузку `g()`, где первый параметр представляет собой указатель на тот же тип, что и у второго параметра.

```
g( &d );          // j
```

Понятно, что здесь вызывается функция № 4 (где `T` представляет собой `double`).

```
g( d, &i );       // k
```

Этому вызову близки несколько перегрузок, но наиболее соответствует ему шаблон № 1, где `T1` представляет собой `double`, а `T2` — `int*`.

```
g( &i, &i );      // l
```

Здесь будет применен шаблон № 3 (где `T` представляет собой `int*`), наиболее соответствующий вызову, несмотря на то, что в некоторых других шаблонах явно указан параметр-указатель.

Хорошей новостью является то, что современные компиляторы обычно обеспечивают неплохую поддержку шаблонов, так что вы можете использовать описанные возможности более надежно и переносимо, чем раньше.

(Потенциально) плохой новостью может оказаться то, что если вы правильно ответили на все вопросы, то, возможно, вы знаете правила C++ лучше, чем ваш компилятор.

Задача 1.18. Mastermind ***Сложность: 8***

Завершающая этот раздел задача позволит вам закрепить рассмотренный материал, самостоятельно разработав объекты-функции и использовав встроенные алгоритмы стандартной библиотеки.

Напишите программу для игры в упрощенную версию Mastermind[14] с использованием контейнеров, алгоритмов и потоков лишь из стандартной библиотеки. Как правило, цель задач данной книги состоит в демонстрации строгой и надежной разработки программного обеспечения. Сейчас же наша цель несколько иная — с использованием минимального количества `if`, `while`, `for` и прочих ключевых слов, а также минимального количества инструкций получить работающую программу. В этой конкретной задаче компактность кода только приветствуется.

Упрощенные правила игры

В начале игры программа случайным образом составляет строку из четырех символов, каждый из которых может принимать одно из трех значений — R, G или B (что соответствует ряду из четырех фишек разных цветов — красного (Red), зеленого (Green) и синего (Blue)). Например, программа может выбрать “RRBB”, “GGGG” или “BGRG”.

Игрок последовательно выдвигает свои предположения о строке, до тех пор, пока не укажет точный порядок символов в строке. На каждое предположение программа отвечает двумя числами. Первое из них — количество угаданных символов (угаданных цветов фишек, независимо от их расположения). Второе число — угаданные символы, стоящие на корректных местах (количество фишек, цвет и расположение которых угадано).

Вот пример игры, в которой программа “загадала” строку “RRBB”.

```
guess--> RBRR
3  1
guess--> RBGG
2  1
guess--> BBRR
4  0
guess--> RRBB
4  4 - solved!
```

Решение

Решения, представленные здесь, не являются единственно верными решениями. Оба решения легко расширяемы как в плане дополнительных цветов, так и в плане добавления фишек, поскольку ни цвета фишек, ни их количество не закодированы в алгоритме. Цвета могут быть добавлены путем изменения строки `colors`, а длина

[14] У нас эта игра известна под названием “быки и коровы”. — *Прим. перев.*

строки увеличена в результате непосредственного изменения длины строки comb. В то же время оба решения обеспечивают недостаточную проверку ошибок.

Одним из требований задачи является минимизация количества инструкций, поэтому там, где это возможно, вместо точки с запятой в качестве разделителя инструкций будет использоваться запятая. Большинство из нас никогда не используют в своем коде большого количества запятых, но на самом деле запятые могут использоваться в гораздо большем количестве мест, чем это могло бы показаться на первый взгляд. Другое требование состоит в уменьшении количества используемых ключевых слов, так что в нашей программе тернарный оператор ?: будет часто заменять if/else.

Решение № 1

Идея первого решения состоит в выполнении всех необходимых вычислений во время единственного прохода по строке запроса, вводимой игроком. Первое решение использует одну инструкцию while. Во втором решении (при определенной потере ясности) мы заменим ее вызовом find_if.

Чтобы увидеть основную структуру кода, начнем с функции main().

```
typedef map<int, int> M;

int main() {
    const string colors("BGR"); // Возможные цвета
    string comb(4, '.'),        // Загаданная комбинация
           guess;               // Текущий запрос
```

Расширение этой программы для работы с большим количеством цветов или большей длиной строки выполняется очень легко: чтобы изменить набор доступных цветов, измените строку colors; чтобы изменить длину отгадываемой строки, сделайте строку comb длиннее или короче.

```
    int cok, pok = 0;           // Корректные цвет и позиция
    M cm, gm;                   // Вспомогательные структуры
```

Нам требуется строка для хранения загаданной комбинации фишек и еще одна — для хранения запроса игрока. Остальные переменные используются при обработке каждого запроса игрока.

Прежде всего нам надо сгенерировать исходную комбинацию фишек.

```
    srand(time(0)),
    generate(comb.begin(), comb.end(),
             ChoosePeg( colors ));
```

После инициализации встроенного генератора случайных чисел мы используем функцию стандартной библиотеки generate() для генерации отгадываемой комбинации. В функцию передается объект типа ChoosePeg, который используется при генерации. Для каждой позиции фишки ее цвет определяется путем вызова оператора функции operator()() данного объекта. О том, как работает ChoosePeg, вы узнаете немного позже.

Теперь перейдем к основному циклу, который выводит приглашение и обрабатывает запросы игрока.

```
while( pok < comb.size() )
    cout << "\n\nguess--> ",
    cin >> guess,
    guess.resize( comb.size(), ' ' ),
```

Здесь реализована простейшая проверка ошибок ввода. Если игрок вводит слишком длинный запрос, он будет сокращен; слишком короткий запрос будет дополнен пробелами. Такое добавление позволяет игроку сжульничать, сперва используя одно-

символьные запросы для поиска нужной буквы, затем двухсимвольные для поиска второй буквы и т.д. Наш код разрешает такое мелкое мошенничество.

Далее мы очищаем рабочую область и приступаем к обработке запроса.

```
cm = gm = M(),
```

Одновременно мы выполняем две обработки запроса игрока. Мы определяем количество фишек, имеющих правильный цвет и стоящих в верных позициях (pok — “place okay”, “размещено верно”). Отдельно мы определяем количество фишек, которые имеют верный цвет, но при этом не обязательно находятся на верных позициях (cok — “color okay”, “цвет верный”). Для того чтобы избежать двух проходов по запросу, мы сначала используем алгоритм transform(), работающий с двумя диапазонами, в котором объект-функция CountPlace накапливает информацию о том, сколько фишек каждого цвета встречается в загаданной комбинации и запросе.

```
transform( comb.begin(), comb.end(),
           guess.begin(), guess.begin(),
           CountPlace( cm, gm, pok )),
```

Алгоритм стандартной библиотеки transform() работает с двумя входными диапазонами — в нашем случае со строками искомой комбинации и запроса, рассматриваемыми как последовательность символов. На самом деле алгоритм transform() выполняет более сложную задачу, генерируя выходную информацию, но она нас не интересует. Простейший способ игнорировать выходную информацию — это разработать объект-функцию так, чтобы она не изменяла последовательность, т.е. чтобы эта функция возвращала то же значение, что и символ строки запроса. Таким образом строка запроса останется неизменной.

Заметим, что объект-функция CountPlace, кроме вычисления значения pok, заполняет также отображения cm и gm. Эти вспомогательные отображения затем используются во второй фазе, когда цикл for_each проходит по цветам (не по строке запроса игрока!) и вычисляет количество совпадений фишек каждого цвета.

```
for_each( colors.begin(), colors.end(),
          CountColor( cm, gm, cok )),
```

Алгоритм for_each(), вероятно, наиболее известный алгоритм стандартной библиотеки. Все, что он делает, — это проходит по заданному диапазону и применяет переданный объект-функцию к каждому элементу диапазона. В нашем случае объект CountColor отвечает за подсчет количества совпадающих фишек одного цвета в двух строках.

После того как эта работа выполнена, мы сообщаем ее результаты.

```
cout << cok << ' ' << pok;
```

Заметим, что вокруг тела цикла while не нужны фигурные скобки, поскольку все тело, благодаря использованию запятых, состоит из одной инструкции.

И наконец, по завершении цикла мы сообщаем о завершении работы.

```
    cout << " - solved!\n";
}
```

Теперь, когда мы познакомились с работой программы в целом, рассмотрим три вспомогательных объекта-функции.

ChoosePeg

Простейшим из вспомогательных объектов является используемый для генерации объект типа ChoosePeg.

```
class ChoosePeg
{
public:
```

```
    ChoosePeg( const string& colors)
        : colors_(colors) { }

    char operator()() const
        { return colors_[rand() % colors_.size()]; }

private:
    const string& colors_;
};
```

Каждый вызов `operator()()` генерирует цвет очередной фишки.

Если бы не требовалось запоминать строку возможных цветов, вместо объекта можно было бы использовать обычную функцию. Конечно, можно сделать `colors` глобальной строкой и засорить глобальное пространство имен. Наше решение элегантнее — оно избегает использования глобальных данных и делает `CoosePeg` не зависящим от них.

CountPlace

Этот объект отвечает за вычисление `pok`, количества фишек, имеющих верный цвет и находящихся в верных позициях. Кроме того, этот объект отвечает также за накопление статистической информации об обрабатываемом запросе игрока и загаданной комбинации для дальнейшей обработки объектом `CountColor`.

```
class CountPlace
{
public:
    CountPlace( M& cm, M& gm, int& pok )
        : cm_(cm), gm_(gm), pok_(pok = 0) { }
    char operator()( char c, char g )
    {
        return ++cm_[c],
               ++gm_[g],
               pok_ += (c == g),
               g;
    }

private:
    M &cm_, &gm_;
    int& pok_;
};
```

Каждый вызов `operator()()` сравнивает одну из составляющих комбинацию фишек `c` с соответствующей фишкой `g` из запроса. Если они одинаковы, мы увеличиваем значение `pok_`. Теперь цель используемых объектов типа `map` становится понятнее. Каждое отображение хранит количество фишек данного цвета (значение цвета типа `char` преобразуется в `int`), встречающихся в данной строке.

В соответствии с семантикой `transform()`, оператор `operator()()` должен вернуть некоторое значение типа `char`. На самом деле не имеет значения, какую величину мы будем возвращать. Однако возвращаемые значения направляются в строку `guess`, которую, возможно, мы захотим использовать в дальнейшем. Для простоты и общей чистоты кода будем возвращать значение цвета из строки `guess`, оставляя ее тем самым неизменной в результате работы алгоритма.

CountColor

Последний из объектов, объект типа `CountColor`, отвечает за вычисление `cok` с использованием накопленной объектом `CountPlace` статистической информации. Это делается очень просто: для каждого из возможных цветов объект суммирует число соответствующих фишек независимо от их положения в строке. Это значение представ-

ляет собой минимальное значение среди количества фишек данного цвета в искомой комбинации и запросе игрока.

```
class CountColor
{
public:
    CountColor( M& cm, M& gm, int& cok )
        : cm_(cm), gm_(gm), cok_(cok = 0) { }

    void operator()( char c ) const
    { cok_ += min( cm_[c], gm_[c] ); }

private:
    M &cm_, &gm_;
    int& cok_;
};
```

Полный код

Собирая все рассмотренные части кода вместе, мы получим полный текст решения поставленной задачи.

```
#include <iostream>
#include <algorithm>
#include <map>
#include <string>
#include <ctime>
#include <cstdlib>
using namespace std;

typedef map<int, int> M;

class ChoosePeg
{
public:
    ChoosePeg( const string& colors)
        : colors_(colors) { }

    char operator()() const
        { return colors_[rand() % colors_.size()]; }

private:
    const string& colors_;
};

class CountPlace
{
public:
    CountPlace( M& cm, M& gm, int& pok )
        : cm_(cm), gm_(gm), pok_(pok = 0) { }
    char operator()( char c, char g )
    {
        return ++cm_[c],
               ++gm_[g],
               pok_ += (c == g),
               g;
    }

private:
    M &cm_, &gm_;
    int& pok_;
};

class CountColor
{
```

```
public:
    CountColor( M& cm, M& gm, int& cok )
        : cm_(cm), gm_(gm), cok_(cok = 0) { }

    void operator()( char c ) const
    { cok_ += min( cm_[c], gm_[c] ); }

private:
    M &cm_, &gm_;
    int& cok_;
};

int main() {
    const string colors("BGR"); // Возможные цвета
    string comb(4, '.'),        // Загаданная комбинация
           guess;               // Текущий запрос
    int cok, pok = 0;           // Корректные цвет и позиция
    M cm, gm;                   // Вспомогательные структуры

    srand(time(0)),
    generate(comb.begin(), comb.end(),
             ChoosePeg( colors ));

    while( pok < comb.size() )
        cout << "\n\nguess--> ",
        cin >> guess,
        guess.resize( comb.size(), ' ' ),
        cm = gm = M(),
        transform( comb.begin(), comb.end(),
                   guess.begin(), guess.begin(),
                   CountPlace( cm, gm, pok )),
        for_each( colors.begin(), colors.end(),
                  CountColor( cm, gm, cok )),
        cout << cok << ' ' << pok;

    cout << " - solved!\n";
}
```

Решение № 2

Идея второго решения состоит в том, чтобы предельно упростить функцию `main()`, просто поставив задачу: “найти комбинацию во входном потоке”. Вот как выглядит функция `main()`.

```
int main()
{
    srand( time(0) ),
    find_if( istream_iterator<string>(cin),
             istream_iterator<string>(),
             Combination() );
}
```

Это все. Вся работа выполняется предикатом `Combination`.

Combination

Понятно, что конструктор по умолчанию предиката `Combination` должен сгенерировать искомую комбинацию.

```
class Combination
{
public:
    Combination()
        : comb_(4,'.')
```

```
{
    generate(comb_.begin(), comb_.end(), ChoosePeg),
    Prompt();
}
```

Заметим, что здесь `ChoosePeg()` отличается от своего тезки в первом решении; подробнее об этом — чуть позже. После генерации комбинации конструктор выводит приглашение для игрока ввести запрос, и на этом его обязанности заканчиваются.

Далее `Combination::operator()()` выполняет сравнение каждой введенной строки запроса с исходной комбинацией и возвращает `true`, если она совпадает с комбинацией, и `false` — если нет.

```
bool operator()(string guess) const // Один ход
{
    int cok, pok; // Корректные цвет и позиция

    return
        guess.resize(comb_.size(),' '),
```

Здесь, как и в первом решении, выполняется небольшая обработка ошибок ввода, изменяющая при необходимости размер введенной строки `guess`.

Основная работа, как и ранее, выполняется в две стадии. Отличие заключается в том, что в этот раз стадии выполняются в обратном порядке, и мы больше не беспокоимся о том, чтобы ограничиться одним проходом по входной строке. Сперва мы определяем `cok`, количество фишек, имеющих верный цвет, но при этом не обязательно располагающихся в верных позициях.

```
cok = accumulate(colors.begin(), colors.end(),
                 ColorMatch(0, &comb_, &guess),
                 ColorMatch::Count),
```

Вспомогательный класс `ColorMatch` будет рассмотрен чуть позже.

Вторая стадия состоит в вычислении `pok`, количества фишек верного цвета, находящихся в верных позициях строки. Для этого мы используем алгоритм, который, на первый взгляд, может показаться неподходящим.

```
pok = inner_product(comb_.begin(), comb_.end(),
                    guess.begin(), 0,
                    plus<int>(),
                    equal_to<char>()),
```

Мы же не производим математические вычисления с матрицами, так почему же мы используем `inner_product()`? Краткий ответ — потому что!

А длинный ответ состоит в том, что это алгоритм, поведение которого наиболее близко к тому, которое нам требуется. Алгоритм `inner_product()` выполняет следующие действия:

- принимает в качестве входа два диапазона;
- выполняет определенную операцию (назовем ее *op2*) над парой первых элементов диапазонов, затем над парой вторых и т.д.;
- выполняет еще одну операцию (назовем ее *op1*) для получения результата.

По умолчанию *op1* и *op2* представляют собой соответственно сложение и умножение. То, что требуется нам, ненамного отличается от работы по умолчанию. Сложение в качестве *op1* нас вполне устраивает, но *op2* должен возвращать 1, если оба символа одинаковы, и 0, если нет, что и выполняется при проверке равенства посредством `equal_to<char>` и преобразовании полученного значения типа `bool` в тип `int`.

Если имя `inner_product()` продолжает вас раздражать, попытайтесь вместо него использовать алгоритмы типа `accumulate()`, который производит вычисления с одним диапазоном, и `transform()`, работающий с двумя входными диапазонами и гиб-

кой настройкой действий, выполняемых с элементами этих диапазонов. Математическое скалярное произведение представляет собой частный случай описанной ситуации, но алгоритм inner_product() может быть адаптирован для решения разнообразных задач комбинации и вычислений над входными последовательностями, — в чем мы только что убедились на собственном опыте.

```
        cout << cok << ' ' << pok,
        (pok == comb_.size())
            ? (cout << " - solved!\n", true)
            : (Prompt(), false);
}
```

И наконец, рассматриваемый оператор обеспечивает пользовательский интерфейс, включая вывод результатов расчетов и приглашение ввести новый запрос.

Что касается нестатических членов-данных, то нам нужен только один такой член.

```
private:
    string comb_;    // Загаданная комбинация
```

Остальные члены класса являются статическими.

```
static char ChoosePeg()
    { return colors[rand() % colors.size()]; }
```

Заметим, что в связи с изменением инкапсуляции (строка colors теперь находится в той же области видимости, что и ChoosePeg()) мы можем упростить ChoosePeg() и сделать этот объект просто функцией, без запоминания состояния.

```
    static void Prompt() { cout << "\n\nguess--> "; }
    static const string colors;  // Возможные цвета
};

const string Combination::colors = "BGR";
```

Поскольку Combination теперь единственный класс, нуждающийся в информации о возможных цветах и длине запроса игрока, мы можем скрыть эту информацию, спрятав ее внутри области видимости Combination.

ColorMatch

Кроме класса Combination, нам требуется еще один вспомогательный объект-функция. Вспомним, что он используется в инструкции

```
cok = accumulate(colors.begin(), colors.end(),
                 ColorMatch(0, &comb_, &guess),
                 ColorMatch::Count),
```

Здесь имеется целый ряд новшеств. Заметим, что в действительности типом аккумулируемого значения является ColorMatch. Это означает, что accumulate() возвращает окончательное значение ColorMatch, которое мы рассматриваем как значение типа int. Все очень просто — для этого и служит оператор преобразования к типу int.

```
class ColorMatch
{
public:
    ColorMatch( int i, const string* s1, const string* s2 )
        : cok_(i), s1_(s1), s2_(s2) { }

    operator int() const { return cok_; }
```

Теперь при использовании алгоритма accumulate() так, как показано выше, происходит не простое прибавление значений типа int, а подсчет посредством Count() с использованием объектов ColorMatch. То есть для каждого char из строки colors алгоритм accumulate() применяет функцию ColorMatch::Count(), которая получает

ссылку на объект ColorMatch, хранящий вычисляемое значение, и очередной обрабатываемый символ из строки colors.

```
static ColorMatch Count( ColorMatch& cm, char c )
    {
        return
            ColorMatch(
              cm.cok_ +
              min(count(cm.s1_->begin(), cm.s1_->end(), c),
                  count(cm.s2_->begin(), cm.s2_->end(), c)),
              cm.s1_, cm.s2_ );
    }
```

Работа Count() состоит в определении того, сколько фишек цвета c имеется в обеих строках (необязательно в одних и тех же позициях), и добавлении этого количества к результирующему значению ColorMatch. Это делается посредством использования еще одного стандартного алгоритма — count(), — который определяет количество фишек заданного цвета в каждой из строк. Затем к общему результату добавляется минимальное из полученных значений. После этого Count() возвращает новый объект ColorMatch, содержащий результат вычислений. Поскольку Count() вызывается по одному разу для каждого возможного цвета, значение члена cok_ результирующего объекта ColorMatch будет именно той величиной, которая нас и интересует, — количеством фишек верного цвета, находящихся не обязательно в верных позициях.

И наконец, объект-функция содержит закрытое состояние.

```
private:
    int cok_;
    const string *s1_, *s2_;
};
```

Полный код

Собирая все рассмотренные части кода вместе, мы получим полный текст второго решения поставленной задачи.

```
#include <iostream>
#include <algorithm>
#include <numeric>
#include <functional>
#include <string>
#include <ctime>
#include <cstdlib>
using namespace std;

class ColorMatch
{
public:
    ColorMatch( int i, const string* s1, const string* s2 )
        : cok_(i), s1_(s1), s2_(s2) { }

    operator int() const { return cok_; }

    static ColorMatch Count( ColorMatch& cm, char c )
    {
        return
            ColorMatch(
              cm.cok_ +
              min(count(cm.s1_->begin(), cm.s1_->end(), c),
                  count(cm.s2_->begin(), cm.s2_->end(), c)),
              cm.s1_, cm.s2_ );
    }

private:
```

```
    int cok_;
    const string *s1_, *s2_;
};

class Combination
{
public:
    Combination()
        : comb_(4,'.')
    {
        generate(comb_.begin(), comb_.end(), ChoosePeg),
        Prompt();
    }

    bool operator()(string guess) const // Один ход
    {
        int cok, pok; // Корректные цвет и позиция

        return
            guess.resize(comb_.size(),' '),
            cok = accumulate(colors.begin(), colors.end(),
                             ColorMatch(0, &comb_, &guess),
                             ColorMatch::Count),
            pok = inner_product(comb_.begin(), comb_.end(),
                                guess.begin(), 0,
                                plus<int>(),
                                equal_to<char>()),
            cout << cok << ' ' << pok,
            (pok == comb_.size())
                ? (cout << " - solved!\n", true)
                : (Prompt(), false);
    }

private:
    string comb_;    // Загаданная комбинация

    static char ChoosePeg()
    { return colors[rand() % colors.size()]; }
    static void Prompt() { cout << "\n\nguess--> "; }
    static const string colors;  // Возможные цвета
};

const string Combination::colors = "BGR";

int main()
{
    srand( time(0) ),
    find_if( istream_iterator<string>(cin),
             istream_iterator<string>(),
             Combination() );
}
```

Сравнение решений

Эта задача преследует две цели. Одна из них — предоставить возможность “поразвлекаться” с некоторыми возможностями C++ (например, получить удовольствие от необычного использования запятой, которое редко встретишь в реальных задачах).

Вторая цель, однако, гораздо более серьезна и состоит в углубленном изучении обобщенного программирования и возможностей стандартной библиотеки C++. Каждое из решений достигает поставленной цели различными путями, решая несколько отличные друг от друга подзадачи. В первом решении главным было избежать нескольких проходов по входной строке. В данном случае это не связано с производи-

тельностью, так как программа большую часть времени находится в состоянии ожидания ввода пользователя. Тем не менее устранение излишних проходов — весьма полезная технология, используемая в реальных задачах. Когда устранение излишних проходов становится существенным? Наиболее простой пример: при очень больших входных данных, в особенности когда данные столь велики, что требуется их чтение с диска при каждом проходе (очевидно, что в этой ситуации крайне желательно избежать излишних обращений к медленному устройству вторичной памяти). Еще один пример: ситуация, когда повторное обращение к данным попросту невозможно (например, при использовании итератора ввода).

Главной задачей второго решения была максимальная инкапсуляция внутреннего кода и максимальное упрощение функции `main()`. Хотя это решение несколько длиннее первого, оно более четкое и ясное, так как избегает недостатков первого решения, связанных с косвенной координацией двух фаз основного цикла посредством вспомогательных структур. При разрешении двух проходов по входным данным эти вспомогательные структуры оказываются не нужными.

Оба решения служат отличным примером использования стандартной библиотеки и демонстрируют технологии, которые могут быть с успехом применены в промышленном коде (конечно, за исключением этого безобразия с запятыми, которые использовались только для развлечения).

2

Вопросы и технологии безопасности исключений

Сначала — краткая предыстория вопроса. В 1994 году Том Каргилл (Tom Cargill) опубликовал основополагающую статью "Exception Handling: A False Sense of Security" ("Обработка исключений: ложное чувство безопасности") [Cargill94].[1] Он убедительно показал, что в то время программисты на C++ не совсем хорошо понимали, как следует писать безопасный в смысле исключений код. Более того, они даже не понимали всю важность вопросов безопасности исключений или как корректно ее обосновать. Каргилл предложил всем читателям найти полное решение поставленной им задачи. Прошло три года; за это время читатели предложили частичные решения различных аспектов приведенной Каргиллом задачи, но исчерпывающее решение так и не было найдено.

Затем, в 1997 году, в группе новостей *comp.lang.c++.moderated* восьмой выпуск *Guru of the Week* вызвал длительную дискуссию, завершившуюся первым полным решением поставленной Каргиллом задачи. Позже в осенних выпусках *C++ Report* была опубликована статья "Exception-Safe Generic Containers" (копия которой имеется в [Martin00]), учитывающая предлагаемые изменения в стандарте C++ и демонстрирующая не менее трех полных решений поставленной задачи. В 1999 году Скотт Мейерс (Scott Meyers) включил материал о задаче Каргилла и ее решениях в [Meyers99].

Сейчас практически невозможно написать надежный код без знания вопросов безопасности исключений. Если вы используете стандартную библиотеку C++, пусть даже только один оператор `new`, — вы должны быть готовы к работе с исключениями.

Приведенная серия задач прошла долгий путь от первой публикации в *Guru of the Week*. Я надеюсь, что она понравится вам и принесет реальную пользу. Я бы хотел особо поблагодарить Дэйва Абрахамса (Dave Abrahams) и Грега Колвина (Greg Colvin) за их помощь в работе над данным материалом.

Задача 2.1. Разработка безопасного кода. Часть 1 ***Сложность: 7***

Обработка исключений и шаблоны представляют собой две наиболее мощные возможности C++. Однако написание кода, безопасного в смысле исключений, может оказаться весьма сложной задачей — в особенности при разработке шаблонов, когда вы можете не знать, в какой момент и какое исключение может сгенерировать та или иная функция.

Приведенная здесь серия задач объединяет обе возможности языка, рассматривая написание безопасных в смысле исключений (т.е. корректно работающих при нали-

[1] Данная статья доступна по адресу `http://www.gotw.ca/publications/xc++/sm_effective.htm`.

чии исключений) и нейтральных по отношению к исключениям (т.е. передающих все исключения вызывающей функции) обобщенных контейнеров. Однако скоро сказка сказывается, да не скоро дело делается...

Итак, приступим к реализации простого контейнера (стека, в который пользователи могут помещать элементы на вершину и снимать их оттуда) и рассмотрим вопросы, возникающие при попытках сделать его безопасным и нейтральным в смысле исключений.[2]

Мы начнем с того, на чем остановился Каргилл, — а именно с постепенного создания безопасного шаблона `Stack`. Позже мы существенно усовершенствуем наш контейнер `Stack`, снижая требования к типу содержимого стека `T`, и познакомимся с технологиями безопасного управления ресурсами. Попутно мы найдем ответы на следующие вопросы.

- Какие “уровни” безопасности исключений имеются?
- Обязаны ли обобщенные контейнеры быть полностью нейтральными по отношению к исключениям?
- Являются ли контейнеры стандартной библиотеки безопасными и нейтральными?
- Влияет ли требование безопасности на дизайн открытого интерфейса ваших контейнеров?
- Должны ли обобщенные контейнеры использовать спецификации исключений?

Ниже приведено объявление шаблона `Stack`, по существу такое же, как и в статье Каргилла. Ваша задача: сделать шаблон `Stack` безопасным и нейтральным. Это означает, что объекты типа `Stack` должны всегда находиться в корректном и согласованном состоянии, независимо от наличия любых исключений, которые могут быть сгенерированы в процессе выполнения функций-членов класса `Stack`. При генерации любого исключения оно должно быть в неизменном виде передано вызывающей функции, которая может поступать с ним так, как сочтет нужным, ибо она осведомлена о конкретном типе `T`, в то время как вы при разработке ничего о нем не знаете.

```
template<class T> class Stack
{
public:
    Stack();
    ~Stack();

    /* ... */

private:
    T*     v_;     //Указатель на область памяти, достаточную
    size_t vsize_; //для размещения vsize_ объектов типа T
    size_t vused_; //Количество реально используемых объектов
};
```

Напишите конструктор по умолчанию и деструктор класса `Stack` так, чтобы была очевидна их безопасность (чтобы они корректно работали при наличии исключений) и нейтральность (чтобы они передавали все исключения в неизменном виде вызывающим функциям, и при этом не возникало никаких проблем с целостностью объекта `Stack`).

---

[2] Далее в этой главе под безопасным кодом, если не оговорено иное, будет пониматься код, безопасный в смысле исключений, т.е. код, корректно работающий при наличии исключений. То же относится и к нейтральному коду, под которым подразумевается код, передающий все сгенерированные исключения вызывающей функции. — *Прим. перев.*

Решение

Сразу же становится очевидно, что `Stack` должен управлять ресурсами динамической памяти. Ясно, что одной из ключевых задач становится устранение утечек даже при наличии генерации исключений операциями `T` и при стандартном распределением памяти. Пока мы будем управлять этими ресурсами памяти в каждой функции-члене `Stack`, но позже усовершенствуем управление ресурсами путем использования закрытого базового класса для инкапсуляции владения ресурсами.

Конструктор по умолчанию

Сначала рассмотрим один из возможных конструкторов по умолчанию.

```
// Безопасен ли данный код?

template<class T>
Stack<T>::Stack()
    : v_(0),
      vsize_(10),
      vused_(0)         // Пока что ничего не использовано
{
    v_ = new T[vsize_]; // Начальное распределение памяти
}
```

Является ли этот конструктор нейтральным и безопасным? Для того чтобы выяснить это, рассмотрим, что может генерировать исключения. Коротко говоря — любая функция. Так что первый шаг состоит в анализе кода и определении того, какие функции будут реально вызываться — включая обычные функции, конструкторы, деструкторы, операторы и прочие функции-члены.

Конструктор `Stack` сначала устанавливает `vsize_` равным 10, а затем делает попытку выделения начальной памяти путем использования оператора `new T[vsize_]`. Этот оператор сначала вызывает `operator new[]()` (либо оператор `new` по умолчанию, либо переопределенный в классе `T`) для выделения памяти, а затем `vsize_` раз вызывает конструктор `T::T()`. Во-первых, сбой может произойти при работе оператора `operator new[]()`, который при этом сгенерирует исключение `bad_alloc`. Во-вторых, конструктор по умолчанию `T` может сгенерировать любое исключение; в этом случае все созданные к этому моменту объекты уничтожаются, а выделенная память гарантированно освобождается вызовом оператора `operator delete[]()`.

Следовательно, приведенный выше код полностью безопасен и нейтрален, и мы можем перейти к следующему... Что? Вы спрашиваете, почему эта функция вполне благонадежна? Ну что ж, рассмотрим ее немного детальнее.

1. *Приведенный код нейтрален.* Мы не перехватываем ни одно исключение, так что все исключения, которые могут быть сгенерированы, корректно передаются вызывающей функции.

Рекомендация

Если функция не обрабатывает (не преобразует или преднамеренно не перехватывает) исключение, она должна разрешить его передачу вызывающей функции, где исключение и будет обработано.

3. *Приведенный код не вызывает утечки памяти.* Если вызов `operator new[]()` приводит к генерации исключения `bad_alloc`, то память не выделяется вовсе, так что никаких утечек нет. Если же генерирует исключение один из конструк-

торов T, то все полностью созданные объекты типа T корректно уничтожаются, а после этого автоматически вызывается operator delete[](), который освобождает выделенную память. Это защищает нас от утечек.

Я не рассматриваю возможность генерации исключения деструктором T, что приведет к вызову terminate() и завершению программы. Более подробно такая ситуация рассматривается в задаче 2.9.

4. *Независимо от наличия исключений состояние объекта остается согласованным.* Вы можете подумать, что при генерации исключения оператором new значение vsize_ равно 10, в то время как выделения памяти в действительности не произошло. Не является ли это несогласованным состоянием? В действительности нет, поскольку это не имеет никакого значения. Ведь когда new генерирует исключение, оно передается за пределы нашего конструктора. Но, по определению, выход из конструктора посредством исключения означает, что прото-объект Stack никогда не становится полностью сконструированным объектом. Его время жизни никогда не начиналось, так что состояние объекта не имеет никакого значения, поскольку этот объект никогда не существовал. Так что значение vsize_ после генерации исключения имеет не больше смысла, чем после выхода из деструктора объекта. Все, что остается от попытки создания объекта, — это память, дым и пепел...

Рекомендация

Всегда структурируйте ваш код таким образом, чтобы при наличии исключений ресурсы корректно освобождались, а данные находились в согласованном состоянии.

Ну что ж, я готов даже признать ваши претензии и изменить конструктор следующим образом.

```
template<class T>
Stack<T>::Stack()
    : v_(new T[10]), // Начальное распределение памяти
      vsize_(10),
      vused_(0)      // Пока что ничего не использовано
{
}
```

Обе версии конструктора практически эквивалентны. Лично я предпочитаю последний вариант, поскольку он следует общепринятой практике по возможности инициализировать члены в списке инициализации.

Деструктор

Деструктор выглядит намного проще, поскольку мы принимаем существенно упрощающее предположение.

```
template<class T>
Stack<T>::~Stack()
{
    delete[] v_;  // Тут исключения генерироваться не могут
}
```

Почему delete[] не может генерировать исключения? Вспомним, что при этом для каждого объекта в массиве вызывается деструктор T::~T(), а затем operator delete[]() освобождает выделенную память. Мы знаем, что освобождение

памяти этим оператором не может генерировать исключений, поскольку стандарт требует, чтобы его сигнатура была одной из:[3]

```
void operator delete[] ( void* ) throw();
void operator delete[] ( void*, size_t ) throw();
```

Следовательно, единственное, что может сгенерировать исключение, — это деструктор `T::~T()`. Соответственно, наш класс `Stack` требует, чтобы деструктор `T::~T()` не мог генерировать исключения. Почему? Да просто потому, что иначе мы не в состоянии сделать деструктор `Stack` полностью безопасным. Однако это требование не является таким уж обременительным, поскольку имеется множество других причин, по которым деструкторы никогда не должны генерировать исключений.[4] Любой класс, деструктор которого может сгенерировать исключение, рано или поздно вызовет те или иные проблемы. Вы даже не можете надежно создавать и уничтожать массивы таких объектов посредством операторов `new[]` и `delete[]`.

Рекомендация

Никогда не позволяйте исключениям выйти за пределы деструктора или переопределенных операторов `delete()` или `delete[]()`. Разрабатывайте каждую из этих функций так, как если бы они имели спецификацию исключений `throw()`.

Задача 2.2. Разработка безопасного кода. Часть 2 — *Сложность: 8*

Теперь, когда у нас есть конструктор по умолчанию и деструктор, мы можем решить, что написать другие функции так же просто. Разработайте безопасные и нейтральные конструктор копирования и оператор присваивания и решите, так ли это просто, как кажется на первый взгляд.

Обратимся вновь к шаблону `Stack` Каргилла.

```
template<class T> class Stack
{
public:
    Stack();
    ~Stack();
    Stack(const Stack&);
    Stack& operator=(const Stack&);

    /* ... */

private:
    T*     v_;     //Указатель на область памяти, достаточную
    size_t vsize_; //для размещения vsize_ объектов типа T
    size_t vused_; //Количество реально используемых объектов
};
```

Приступим к разработке конструктора копирования и оператора присваивания таким образом, чтобы они были безопасны (корректно работали при наличии исключе-

[3] Как указал Скотт Мейерс (Scott Meyers) в частном сообщении, строго говоря, это не предотвращает возможного использования переопределенного оператора `operator delete[]`, который может генерировать исключения, но такое переопределение нарушает смысл стандарта и должно рассматриваться как некорректное.

[4] Честно говоря, вы не сильно ошибетесь, если будете писать `throw()` после объявления каждого разрабатываемого вами деструктора. Если же спецификации исключений приводят при использовании вашего компилятора к большим затратам на выполнение проверок, то по крайней мере разрабатывайте все деструкторы так, как если бы они были специфицированы как `throw()`, т.е. никогда не позволяйте исключениям покидать деструкторы.

ний) и нейтральны (передавали все исключения вызывающей функции без каких-либо нарушений целостности объекта `Stack`).

Решение

При реализации конструктора копирования и оператора копирующего присваивания для управления распределением памяти воспользуемся общей вспомогательной функцией `NewCopy`. Эта функция получает указатель (`src`) на существующий буфер объектов `T` и его размер (`srcsize`) и возвращает указатель на новую, возможно, большего размера, копию буфера, передавая владение этим буфером вызывающей функции. Если при этом генерируются исключения, `NewCopy` корректно освобождает все временные ресурсы и передает исключение вызывающей функции таким образом, что при этом не образуется никаких утечек памяти.

```
template<class T>
T* NewCopy( const T* src,
            size_t   srcsize,
            size_t   destsize )
{
    assert( destsize >= srcsize );
    T* dest = new T[destsize];
    try
    {
        copy( src, src + srcsize, dest );
    }
    catch(...)
    {
        delete[] dest;  // Здесь исключений не может быть
        throw;          // Передаем исходное исключение
    }
    return dest;
}
```

Проанализируем пошагово приведенный код.

1. В инструкции `new` при выделении памяти может быть сгенерировано исключение `bad_alloc`; кроме того, конструктором `T::T()` могут быть сгенерированы исключения любого типа. В любом случае выделения памяти при этом не происходит, и мы просто позволяем исключению быть перехваченным вызывающей функцией.
2. Далее мы присваиваем все существующие значения с помощью оператора `T::operator=()`. При сбое любого из присваиваний мы перехватываем исключение, освобождаем выделенную память и передаем исходное исключение дальше. Таким образом, наш код нейтрален по отношению к генерируемым исключениям, и, кроме того, мы избегаем утечки памяти. Однако здесь есть одна тонкость: требуется, чтобы в случае исключения в `T::operator=()` объект, которому присваивается новое значение, мог быть уничтожен путем вызова деструктора.[5]
3. Если и выделение памяти, и копирование прошли успешно, мы возвращаем указатель на новый буфер и передаем владение им вызывающей функции (которая теперь отвечает за его дальнейшую судьбу). Возврат просто копирует значение указателя, так что сгенерировать исключения он не может.

[5] Дальше мы усовершенствуем `Stack` настолько, что сможем избежать использования `T::operator=()`.

Конструктор копирования

При наличии такой вспомогательной функции разработка конструктора копирования оказывается очень простым делом.

```
template<class T>
Stack<T>::Stack( const Stack<T>& other )
    : v_(NewCopy( other.v_,
                  other.vsize_,
                  other.v_size_)),
      vsize_( other.vsize_ ),
      vused_( other.vused_ )
{
}
```

Единственный возможный источник исключений здесь — функция `NewCopy()`, которая сама управляет своими ресурсами.

Копирующее присваивание

Возьмемся теперь за копирующее присваивание.

```
template<class T>
Stack<T>& Stack<T>::operator=( const Stack<T>& other )
{
    if ( this != &other )
    {
        T* v_new = NewCopy( other.v_,
                            other.vsize_,
                            other.vsize_);
        delete[] v_;  // Здесь исключений не может быть
        v_ = v_new;   // Вступаем во владение
        vsize_ = other.vsize_;
        vused_ = other.vused_;
    }
    return *this; // Безопасно, копирование не выполняется
}
```

Здесь, после того как выполнена простейшая защита от присваивания объекта самому себе, генерировать исключения может только вызов `NewCopy()`. Если это происходит, мы просто передаем это исключение вызывающей функции, никак не воздействуя на состояние объекта. С точки зрения вызывающей функции при генерации исключения состояние остается неизменным, а если генерации исключения не произошло, значит, присвоение и все его побочные действия успешно завершены.

Здесь мы познакомились с очень важной идиомой безопасности исключений.

Рекомендация

В каждой функции следует собрать весь код, который может генерировать исключения, и выполнить его отдельно, безопасным с точки зрения исключений способом. Только после этого, когда вы будете знать, что вся реальная работа успешно выполнена, вы можете изменять состояние программы (а также выполнять другие необходимые действия, например, освобождение ресурсов) посредством операций, которые не генерируют исключений.

Задача 2.3. Разработка безопасного кода. Часть 3 ***Сложность: 9.5***

Вы уже уловили суть безопасности исключений? Ну что ж, тогда самое время испытать ваши силы.

Рассмотрим последнюю часть объявления шаблона `Stack`, приведенного Каргиллом.

```
template<class T> class Stack
{
public:
    Stack();
    ~Stack();
    Stack(const Stack&);
    Stack& operator=(const Stack&);
    size_t Count() const;
    void Push(const T&);
    T    Pop(); // Если стек пуст, генерируется исключение
private:
    T*     v_;     //Указатель на область памяти, достаточную
    size_t vsize_; //для размещения vsize_ объектов типа T
    size_t vused_; //Количество реально используемых объектов
};
```

Напишите последние три функции: `Count()`, `Push()` и `Pop()`. Помните о безопасности и нейтральности!

Решение

Count()

Простейшей для реализации из всех функций-членов `Stack()` является `Count()`, в которой выполняется копирование встроенного типа, не генерирующее исключений.

```
template<class T>
size_t Stack<T>::Count() const
{
    return vused_; // Безопасно; исключения не генерируются
}
```

Никаких проблем.

Push()

В этой функции мы должны применить ставшие уже привычными средства обеспечения безопасности.

```
template<class T>
void Stack<T>::Push( const T& t )
{
    if ( vused_ == vsize_ )
    {
        // При необходимости увеличиваем
        // размер выделенной памяти
        size_t vsize_new = vsize_*2 + 1;
        T* v_new = NewCopy( v_, vsize_, vsize_new );
        delete[] v_;
        v_ = v_new;
        vsize_ = vsize_new;
    }
    v_[vused_] = t;
    ++vused_;
}
```

Если у нас не хватает пространства, мы сначала посредством `NewCopy()` выделяем новое пространство для буфера и копируем в новый буфер элементы старого. Если в этой функции генерируется исключение, состояние нашего объекта остается неиз-

менным, а исключение передается вызывающей функции. Удаление исходного буфера и вступление во владение новым включают только те операции, которые не могут генерировать исключения. Таким образом, весь блок `if` содержит безопасный код.

После этого мы пытаемся скопировать новое значение, после чего увеличиваем значение `vused_`. Таким образом, если в процессе присваивания генерируется исключение, увеличения `vused_` не происходит, и состояние объекта `Stack` остается неизменным. Если же присваивание прошло успешно, состояние объекта `Stack` изменяется и отражает наличие нового значения.

А теперь еще раз повторим каноническое правило безопасности исключений.

Рекомендация

В каждой функции следует собрать весь код, который может генерировать исключения, и выполнить его отдельно, безопасным с точки зрения исключений способом. Только после этого, когда вы будете знать, что вся реальная работа успешно выполнена, вы можете изменять состояние программы (а также выполнять другие необходимые действия, например, освобождение ресурсов) посредством операций, которые не генерируют исключений.

Pop()

Осталось написать только одну функцию. Это не так уж и сложно, не правда ли? Впрочем, радоваться пока рано, поскольку именно эта функция наиболее проблематична, и написание полностью безопасного кода этой функции — не такая простая задача, как кажется. Наша первая попытка могла бы выглядеть примерно так.

```
// Насколько на самом деле безопасен этот код?
template<class T>
T Stack<T>::Pop()
{
    if ( vused_ == 0 )
    {
        throw "pop from empty stack";
    }
    else
    {
        T result = v_[vused_-1];
        --vused_;
        return result;
    }
}
```

Если стек пуст, мы генерируем соответствующее исключение. В противном случае мы создаем копию возвращаемого объекта `T`, обновляем состояние нашего объекта `Stack` и возвращаем объект `T`. Если первое копирование из `v_[vused_-1]` сопровождается генерацией исключения, оно будет передано вызывающей функции, и состояние объекта `Stack` останется неизменным, — что и требуется. Если же данное копирование прошло успешно, состояние объекта обновляется, и объект вновь находится в согласованном состоянии.

Итак, все работает? Ну, как будто да. Но тут есть одна небольшая неприятность, находящаяся уже вне границ `Stack::Pop()`. Рассмотрим следующий код.

```
string s1(s.Pop());
string s2;
s2 = s.Pop();
```

Заметим, что мы говорили о “первом копировании” (из `v_[vused_-1]`). Это связано с тем, что имеется еще одно копирование,[6] проявляющееся в любом из приведенных случаев, а именно копирование возвращаемого временного объекта в место назначения. Если конструктор копирования или присваивание завершатся неуспешно, то возникнет ситуация, когда `Stack` полностью выполнит побочные действия, сняв элемент с вершины стека, но этот элемент будет навсегда утерян, поскольку он не достигнет места назначения. Что и говорить, ситуация не из приятных. По сути это означает, что любая версия `Pop()`, возвращающая, как в данном случае, временный объект (и таким образом ответственная за два побочных действия), не может быть полностью безопасной. Если даже реализация функции сама по себе технически безопасна, она заставляет клиентов класса `Stack` писать небезопасный код. Вообще говоря, функции-мутаторы не должны возвращать объекты по значению (о вопросах безопасности функций с несколькими побочными действиями см. задачу 2.12).

Вывод — и он очень важен! — следующий: *безопасность исключений влияет на разработку класса*. Другими словами, вы должны изначально думать о безопасности, которая никогда не является “просто деталью реализации”.

Распространенная ошибка

Никогда не откладывайте продумывание вопросов безопасности исключений. Безопасность влияет на конструкцию класса и не может быть “просто деталью реализации”.

Суть проблемы

Один из вариантов решения — с минимально возможным изменением[7] — состоит в переопределении `Pop()` следующим образом.

```
template<class T>
void Stack<T>::Pop( T& result )
{
    if ( vused_ == 0 )
    {
        throw "pop from empty stack";
    }
    else
    {
        result = v_[vused_-1];
        --vused_;
    }
}
```

Такое изменение гарантирует, что состояние `Stack` останется неизменным до тех пор, пока копия снимаемого со стека объекта не окажется “в руках” вызывающей функции.

---

[6] Для такого опытного читателя, как вы, уточним — “ни одного или одно копирование”, поскольку компилятор при оптимизации может избежать одного копирования. Но дело в том, что такое копирование может существовать, а значит, вы должны быть готовы к этому.

[7] С минимально возможным *приемлемым* изменением. Вы можете изменить исходную версию таким образом, чтобы функция `Pop()` вместо `T` возвращала `T&` (ссылку на снятый с вершины стека объект, поскольку определенное время этот объект продолжает физически существовать в вашем внутреннем представлении), и пользователь вашего класса сможет написать безопасный код. Но такой подход, — возврат ссылки на то, чего быть не должно, — изначально порочен. Если в будущем вы измените реализацию вашего класса, такое решение может стать невозможным. Не прибегайте к такому методу!

Проблема заключается в том, что `Pop()` выполняет две задачи, а именно снимает элемент с вершины стека и возвращает снятое значение.

Рекомендация

Прилагайте максимум усилий к тому, чтобы каждая часть кода — каждый модуль, класс, функция, — отвечали за выполнение одной четко определенной задачи.

Соответственно, второй вариант (на мой взгляд, предпочтительный) состоит в разделении функции `Pop()` на две — одна из которых возвращает значение элемента на вершине стека, а вторая — для его удаления с вершины. Вот как выглядят эти функции.

```
template<class T>
T& Stack<T>::Top()
{
    if ( vused_ == 0 )
    {
        throw "empty stack";
    }
    else
    {
        return v_[vused_-1];
    }
}

template<class T>
void Stack<T>::Pop()
{
    if ( vused_ == 0 )
    {
        throw "pop from empty stack";
    }
    else
    {
        --vused_;
    }
}
```

Кстати, вы никогда не жаловались на то, что функции типа `pop()` у стандартных контейнеров (например, `list::pop_back`, `stack::pop` и другие) не возвращают удаляемые значения? Одну причину этого вы уже знаете: это позволяет избежать снижения уровня безопасности.

Вы, вероятно, уже обратили внимание, что приведенные выше функции `Top()` и `Pop()` соответствуют сигнатурам функций-членов `top` и `pop` класса `stack<>` из стандартной библиотеки. Это не случайное совпадение. У нас действительно отсутствуют только две открытые функции-члена из класса `stack<>` стандартной библиотеки, а именно:

```
template<class T>
const T& Stack<T>::Top() const
{
    if ( vused_ == 0 )
    {
        throw "empty stack";
    }
    else
    {
        return v_[vused_-1];
    }
}
```

версия `Top()` для объектов `const Stack` и

```
template<class T>
bool Stack<T>::Empty() const
{
    return (vused_ == 0);
}
```

Конечно, стандартный класс `stack<>` в действительности представляет собой контейнер, который реализован с использованием другого контейнера, но открытый интерфейс у него тот же, что и у рассмотренного нами класса; все остальное представляет собой детали реализации.

Напоследок я рекомендую вам немного поразмышлять над следующим.

Распространенная ошибка

"Небезопасный по отношению к исключениям" и "плохой дизайн" идут рука об руку. Если часть кода не безопасна, обычно это не означает ничего страшного и легко исправляется. Но если часть кода не может быть сделана безопасной из-за лежащего в ее основе дизайна, то практически всегда это говорит о плохом качестве дизайна. Пример 1: трудно сделать безопасной функцию, отвечающую за выполнение двух различных задач. Пример 2: оператор копирующего присваивания, написанный так, что он должен проверять присваивание самому себе, вероятнее всего, не является строго безопасным.

Пример 2 будет вскоре вам продемонстрирован. Заметим, что проверка присваивания самому себе вполне может присутствовать в операторе копирующего присваивания, — например, в целях повышения эффективности. Однако оператор, в котором такая проверка *обязана* выполняться (иначе оператор будет работать некорректно), вероятно, не является строго безопасным.

Задача 2.4. Разработка безопасного кода. Часть 4 *Сложность: 8*

Небольшая интерлюдия: чего мы так долго добиваемся?

Теперь, когда мы реализовали безопасный и нейтральный класс `Stack<T>`, ответьте как можно точнее на следующие вопросы.

1. Какие важные гарантии безопасности исключений существуют?
2. Какие требования следует предъявить к типу `T` реализованного нами класса `Stack<T>`?

Решение

Существует множество способов написать безопасный в смысле исключений код. В действительности же наш выбор при обеспечении гарантии безопасности кода ограничивается двумя основными альтернативами. Впервые гарантии безопасности в приведенном далее виде были сформулированы Дэйвом Абрахамсом (Dave Abrahams).

1. *Базовая гарантия: даже при наличии генерируемых `T` или иных исключений утечки ресурсов в объекте `Stack` отсутствуют.* Заметим, что отсюда вытекает, что контейнер можно будет использовать и уничтожать, даже если генерация исключений происходит при выполнении некоторых операций с этим контейнером. При генерации исключений контейнер будет находиться в согласованном, однако не обязательно предсказуемом состоянии. Контейнеры, поддерживающие базовую гарантию, могут безопасно использоваться в ряде случаев.

2. *Строгая гарантия: если операция прекращается из-за генерации исключения, состояние программы остается неизменным.* Это требование всегда приводит к семантике принятия-или-отката (commit-or-rollback). В случае неуспешного завершения операции все ссылки и итераторы, указывающие на элементы контейнера, будут действительны. Например, если пользователь класса `Stack` вызывает функцию `Top()`, а затем функцию `Push()`, при выполнении которой генерируется исключение, то состояние объекта `Stack` должно оставаться неизменным, и ссылка, возвращенная предыдущим вызовом `Top()`, должна оставаться корректной. Дополнительную информацию по вопросам строгой гарантии вы можете найти по адресу `http://www.gotw.ca/publications/xc++/da_stlsafety.htm`.

Вероятно, наиболее интересным при этом является то, что когда вы реализуете базовую гарантию, строгая гарантия зачастую реализуется сама по себе.[8] Например, в нашей реализации класса `Stack` почти все, что мы делали, требовалось только для обеспечения базовой гарантии. Однако полученный результат очень близок к требованиям строгой гарантии — достаточно небольших доработок.[9] Неплохой результат, не правда ли?

В дополнение к двум перечисленным гарантиям имеется третья, требованиям которой для обеспечения полной безопасности исключений должны удовлетворять некоторые функции.

3. *Гарантия отсутствия исключений: функция не генерирует исключений ни при каких обстоятельствах.* До тех пор, пока определенные функции могут генерировать исключения, полная безопасность невозможна. В частности, мы видели, что это справедливо для деструкторов. Позже мы увидим, что эта гарантия требуется и для ряда других вспомогательных функций, таких как `Swap()`.

Рекомендация

Не забывайте о базовой, строгой гарантиях и гарантии отсутствия исключений при разработке безопасного кода.

Теперь у нас есть некоторый материал для размышлений. Заметим, что мы смогли реализовать `Stack` так, что он оказался не только безопасным, но и полностью нейтральным, использовав при этом один блок `try/catch`. Как мы увидим позже, использование более мощных технологий инкапсуляции может избавить нас даже от этого блока. Это означает, что мы можем написать строго безопасный и нейтральный обобщенный контейнер, не используя при этом `try` или `catch`, — очень изящно и элегантно.

Разработанный нами шаблон `Stack` требует от инстанцирующего типа соблюдения следующих условий.

---

[8] Обратите внимание: я говорю “зачастую”, но не “всегда”. Например, в стандартной библиотеке таким контрпримером является вектор, который удовлетворяет требованиям базовой гарантии, но при этом удовлетворение требованиям строгой гарантии само по себе не осуществляется.

[9] Имеется одно место, где данная версия `Stack` все еще нарушает условия строгой гарантии. Если при вызове `Push()` происходит увеличение внутреннего буфера, а затем при присваивании `v_[vused_] = t` генерируется прерывание, то `Stack` остается в согласованном состоянии, однако буфер внутренней памяти при этом оказывается перемещен на новое место. Соответственно, все ссылки, возвращенные предыдущими вызовами функции `Top()`, оказываются недействительными. Исправить ситуацию можно, переместив часть кода и добавив блок `try`. Однако имеется и лучшее решение (с которым вы встретитесь немного позже), которое полностью удовлетворяет требованиям строгой гарантии.

- Наличия конструктора по умолчанию (для создания буфера v_).
- Наличия копирующего конструктора (если Pop() возвращает значение).
- Деструктор, не генерирующий исключений (чтобы иметь возможность гарантировать безопасность кода).
- *Безопасное* присваивание (для установки значений в буфере v_; если копирующее присваивание генерирует исключение, оно должно гарантировать, что целевой объект остается корректным объектом типа T. Заметим, что это единственная функция-член T, которая должна быть безопасна для того, чтобы класс Stack был безопасен).

Дальше мы увидим, каким образом можно уменьшить предъявляемые к типу T требования, не влияя на степень безопасности класса Stack. Попутно мы рассмотрим действия стандартной инструкции delete[] x более подробно.

Задача 2.5. Разработка безопасного кода. Часть 5 *Сложность: 7*

Вы уже передохнули? Закатывайте рукава и готовьтесь к продолжению работы.

Теперь мы готовы "углубиться" в наш пример и написать целых две новых улучшенных версии класса Stack. Кроме того, попутно мы ответим на несколько интересных вопросов.

- Каким образом можно использовать более продвинутые технологии для упрощения управления ресурсами (и кроме того, для устранения использованного блока try/catch)?
- Каким образом можно усовершенствовать класс Stack, снизив требования к классу T элементов стека?
- Должны ли обобщенные контейнеры использовать спецификации исключений?
- Что в действительности делают операторы new[] и delete[]?

Ответ на последний вопрос может существенно отличаться от того, который вы можете ожидать. Написание безопасных контейнеров в C++ — не квантовая механика и требует только тщательности и хорошего понимания работы языка программирования. В частности, помогает выработанная привычка смотреть с небольшим подозрением на все, что может оказаться вызовом функции, включая среди прочих определенные пользователем операторы, преобразования типов и временные объекты, поскольку любой вызов функции может генерировать исключения.[10]

Один из способов существенного упрощения реализации безопасного контейнера Stack состоит в использовании инкапсуляции. В частности, мы можем инкапсулировать всю работу с памятью. Наибольшее внимание при написании исходного безопасного класса Stack мы уделяли корректному распределению памяти, так что введем вспомогательный класс, который и будет заниматься этими вопросами.

```
template<class T>
class StackImpl
{
/*????*/:
    StackImpl(size_t size = 0);
    ~StackImpl();
    void Swap(StackImpl& other) throw();

    T*     v_;     //Указатель на область памяти, достаточную
```

[10] Кроме функций со спецификацией исключений throw() и ряда функций стандартной библиотеки, о которых в стандарте сказано, что они не могут генерировать исключения.

```
    size_t vsize_; //для размещения vsize_ объектов типа T
    size_t vused_; //Количество реально используемых объектов

private:
    // Закрыты и не определены: копирование запрещено
    StackImpl( const StackImpl& );
    StackImpl& operator =( const StackImpl& );
};
```

Заметим, что `StackImpl` содержит все данные-члены исходного класса `Stack`, так что по сути мы переместили исходное представление класса в `StackImpl`. Класс `StackImpl` содержит вспомогательную функцию `Swap`, которая обменивается "внутренностями" объекта `StackImpl` с другим объектом того же типа.

Ваша задача состоит в следующем.

1. Реализовать все три функции-члена `StackImpl`, но не так, как мы делали это раньше. Считаем, что в любой момент времени буфер `v_` должен содержать ровно столько сконструированных объектов `T`, сколько их имеется в контейнере, — не больше и не меньше. В частности, неиспользуемое пространство в буфере `v_` не должно содержать сконструированных объектов типа `T`.
2. Опишите обязанности класса `StackImpl`. Зачем он нужен?
3. Чем должно быть `/*????*/` в приведенном выше листинге? Как выбор этого модификатора может повлиять на использование класса `StackImpl`? Дайте как можно более точный ответ.

Решение

Мы не будем тратить много времени на анализ приведенного далее кода и доказательство того, что он безопасен и нейтрален, поскольку этот вопрос уже неоднократно рассматривался нами ранее.

Конструктор

Конструктор прост и понятен. Мы используем оператор `new()` для выделения буфера в виде обычного блока памяти (если бы мы использовали выражение типа `new T[size]`, то буфер был бы инициализирован объектами `T`, сконструированными по умолчанию, что явным образом запрещено условием задачи).

```
template<class T>
StackImpl<T>::StackImpl( size_t size )
    : v_( static_cast<T*>
          ( size == 0
            ? 0
            : operator new( sizeof(T)*size ) ) ),
    vsize_(size),
    vused_(0)
{
}
```

Деструктор

Деструктор реализовать легче всего — достаточно вспомнить, что мы говорили об операторе `delete` немного ранее. (См. врезку "Некоторые стандартные вспомогательные функции", где приведено описание функций `destroy()` и `swap()`, участвующих в следующем фрагменте.)

```
template<class T>
StackImpl<T>::~StackImpl()
{
    // Здесь исключений не может быть
    destroy( v_, v_ + vused_ );

    operator delete( v_ );
}
```

Некоторые стандартные вспомогательные функции

Классы `Stack` и `StackImpl`, представленные в этом решении, используют три вспомогательные функции, одна из которых (`swap()`) имеется в стандартной библиотеке: `construct()`, `destroy()` и `swap()`. В упрощенном виде эти функции выглядят следующим образом.

```
// construct() создает новый объект в указанном месте
// с использованием определенного начального значения
//
template<class T1, class T2>
void construct( T1* p, const T2& value )
{
    new(p) T1( value );
}
```

Приведенный оператор `new` называется “размещающим `new`” и вместо выделения памяти для объекта просто размещает его в памяти по адресу, определяемому указателем `p`. Любой объект, созданный таким образом, должен быть уничтожен явным вызовом деструктора (как в следующих двух функциях), а не посредством оператора `delete`.

```
// destroy() уничтожает объект или
// диапазон объектов
//
template<class T>
void destroy( T* p )
{
    p->~T();
}
template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
    while( first != last )
    {
        destroy( &*first );
        ++first;
    }
}
// swap() просто обменивает два значения
//
template<class T>
void swap( T& a, T& b )
{
    T temp(a);
    a = b;
    b = temp;
}
```

Наиболее интересной функцией из приведенных является `destroy(first,last)`. Мы еще вернемся к ней — в ней заложено гораздо больше, чем можно увидеть на первый взгляд.

Swap

И наконец, последняя и наиболее важная функция. Поверите вы или нет, но именно эта функция делает класс `Stack` таким элегантным, в особенности его оператор `operator=()`, как вы вскоре увидите.

```
template<class T>
void StackImpl<T>::Swap(StackImpl&other) throw()
{
    swap( v_,     other.v_);
    swap( vsize_, other.vsize_ );
    swap( vused_, other.vused_ );
}
```

Для того чтобы проиллюстрировать работу функции `Swap()`, рассмотрим два объекта `StackImpl<T>` — `a` и `b`, — показанные на рис. 2.1. Выполнение `a.Swap(b)` изменяет состояние объектов, которое становится таким, как показано на рис. 2.2.

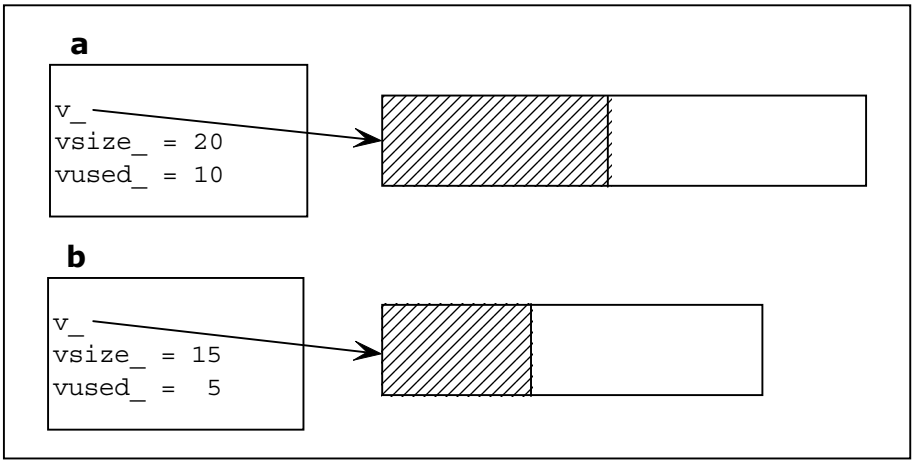


Рис. 2.1. Объекты `a` и `b` типа `StackImpl<T>`

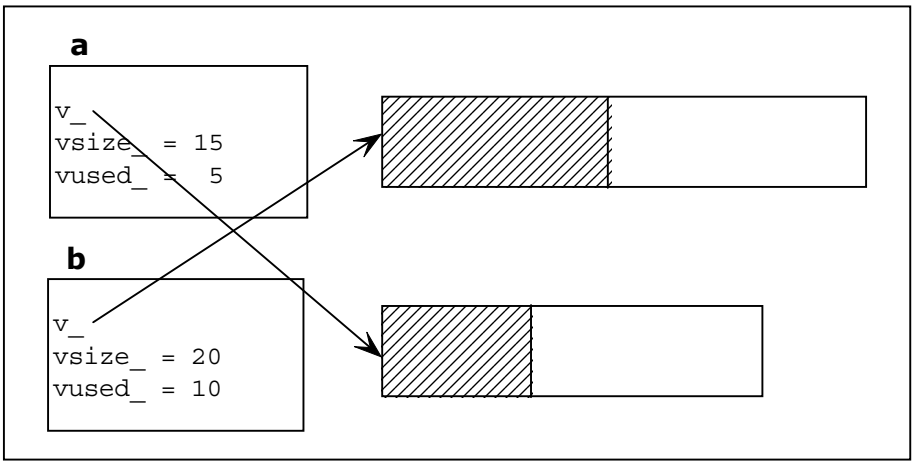


Рис. 2.2. Те же объекты, что и на рис. 2.1, после выполнения `a.Swap(b)`

Заметим, что `Swap()` удовлетворяет требованиям строжайшей гарантии, а именно гарантии отсутствия исключений. `Swap()` гарантированно не генерирует исключений ни при каких обстоятельствах. Это свойство весьма важно и является ключевым звеном в цепи доказательства безопасности класса `Stack` в целом.

Зачем нужен класс `StackImpl`? В этом нет ничего таинственного: `StackImpl` отвечает за управление памятью и ее освобождение по завершении работы, так что любой класс, использующий данный, не должен беспокоиться об этих деталях работы с памятью.

Рекомендация

Прилагайте максимум усилий к тому, чтобы каждая часть кода — каждый модуль, класс, функция, — отвечали за выполнение одной четко определенной задачи.

Чем же должен быть заменен комментарий `/*????*/` в приведенном в условии задачи листинге? Подсказка: само имя `StackImpl` говорит о том, что при использовании его классом `Stack` между ними существует некое отношение "реализован посредством", которое в C++ реализуется в основном двумя методами.

Метод 1: закрытый базовый класс. При этом `/*????*/` следует заменить модификатором `protected` или `public` (но не `private` — в этом случае воспользоваться классом никому не удастся). Сначала рассмотрим, что случится, если мы используем модификатор `protected`.

Использование `protected` означает, что `StackImpl` предназначен для использования в качестве закрытого базового класса. Соответственно, `Stack` "реализован посредством `StackImpl`" с четким разделением обязанностей между классами. Базовый класс `StackImpl` заботится о работе с буфером памяти и уничтожении всех сконструированных объектов, находящихся в нем при деструкции класса `Stack`, в то время как последний заботится о конструировании всех объектов, размещаемых в буфере. Вся работа с памятью вынесена за пределы `Stack`, так что, например, начальное выделение памяти должно пройти успешно еще до того, как будет вызван конструктор `Stack`. В следующей задаче мы займемся разработкой именно этой версии `Stack`.

Метод 2: закрытый член. Теперь рассмотрим, что произойдет, если вместо `/*????*/` использовать модификатор доступа `public`.

Этот модификатор указывает, что `StackImpl` предназначен для использования в качестве структуры некоторым внешним клиентом, поскольку открыты его члены-данные. И вновь `Stack` "реализуется посредством `StackImpl`", но в этот раз вместо закрытого наследования используется связь СОДЕРЖИТ (ВКЛЮЧАЕТ). Наблюдается все то же четкое разделение обязанностей между классами. Базовый класс `StackImpl` заботится о работе с буфером памяти и уничтожении всех сконструированных объектов, находящихся в нем во время деструкции класса `Stack`, в то время как последний заботится о конструировании всех объектов, размещаемых в буфере. Поскольку данные-члены инициализируются до входа в тело конструктора, работа с памятью вынесена за пределы `Stack`; например, начальное выделение памяти должно успешно пройти еще до того, как будет осуществлен вход в тело конструктора `Stack`.

Как мы увидим, второй метод лишь немного отличается от первого.

Задача 2.6. Разработка безопасного кода. Часть 6 — *Сложность: 9*

Разработаем теперь улучшенную версию `Stack` со сниженными требованиями к типу `T` и с весьма элегантным оператором присваивания.

Представим, что комментарий /*????*/ в StackImpl заменен модификатором protected. Реализуйте все функции-члены следующей версии Stack посредством StackImpl, используя последний в качестве закрытого базового класса.

```
template<class T>
class Stack : private StackImpl<T>
{
public:
    Stack(size_t size = 0);
    ~Stack();
    Stack(const Stack&);
    Stack& operator = (const Stack&);
    size_t Count() const;
    void   Push(const T&);
    T&     Top(); // Если стек пуст, генерируется исключение
    void   Pop(); // Если стек пуст, генерируется исключение
};
```

Как обычно, все функции должны быть безопасны и нейтральны.

Указание: имеется очень элегантное решение для оператора присваивания. Сможете ли вы найти его самостоятельно?

Решение

Конструктор по умолчанию

При использовании метода закрытого базового класса наш класс Stack выглядит следующим образом (для краткости код показан как встраиваемый).

```
template<class T>
class Stack : private StackImpl<T>
{
public:
    Stack(size_t size = 0)
        : StackImpl<T>(size)
    {
    }
```

Конструктор по умолчанию просто вызывает конструктор StackImpl по умолчанию, который устанавливает состояние стека как пустого, и выполняет необязательное начальное выделение памяти. Единственная операция, которая может сгенерировать исключение, — это оператор new в конструкторе StackImpl, что не влияет на рассмотрение безопасности самого класса Stack. Если исключение будет сгенерировано, мы не попадем в конструктор Stack, и соответствующий объект создан не будет, и сбой при начальном выделении памяти в базовом классе на класс Stack никак не повлияет (см. также примечания о выходе из конструктора из-за исключения в задаче 2.1).

Заметим, что мы слегка изменили интерфейс исходного конструктора Stack с тем, чтобы позволить передачу указания о количестве выделяемой памяти. Мы используем эту возможность при написании функции Push().

Рекомендация

Всегда используйте идиому "захвата ресурса при инициализации" для отделения владения и управления ресурсами.

Деструктор

Вот первый красивый момент: нам не нужен деструктор класса `Stack`. Нас вполне устроит деструктор, сгенерированный компилятором по умолчанию, поскольку он вызывает деструктор `StackImpl`, который уничтожает все объекты в стеке и освобождает память. Изящно.

Конструктор копирования

Заметим, что конструктор копирования `Stack` не вызывает конструктор копирования `StackImpl`. О том что собой представляет функция `construct()`, см. в решении предыдущей задачи.

```
Stack(const Stack& other)
    : StackImpl<T>(other.vused_)
{
    while( vused_ < other.vused_ )
    {
        construct( v_ + vused_, other.v_[vused_] );
        ++vused_;
    }
}
```

Конструктор копирования эффективен и понятен. Наихудшее, что может здесь случиться, — это сбой в конструкторе `T`; в этом случае деструктор `StackImpl` корректно уничтожит все корректно созданные объекты и освободит занятую память. Одно из важных преимуществ наследования от класса `StackImpl` заключается в том, что мы можем добавить любое количество конструкторов без размещения в каждом из них кода освобождения захваченных ресурсов.

Элегантное копирующее присваивание

Далее приведен невероятно изящный и остроумный способ написания безопасного оператора копирующего присваивания. Это решение по своей элегантности превосходит все рассмотренное нами ранее.

```
Stack& operator = (const Stack& other)
{
    Stack temp(other); // Вся работа выполняется здесь
    Swap( temp );      // Здесь исключений не может быть
    return *this;
}
```

Ну и как вам этот фрагмент? Он стоит того, чтобы вы остановились и подумали над ним перед тем, как читать дальше.

Эта функция представляет собой воплощение важного правила, с которым мы уже познакомились ранее.

Рекомендация

В каждой функции следует собрать весь код, который может генерировать исключения, и выполнить его отдельно, безопасным с точки зрения исключений способом. Только после этого, когда вы будете знать, что вся реальная работа успешно выполнена, вы можете изменять состояние программы (а также выполнять другие необходимые действия, например, освобождение ресурсов) посредством операций, которые не генерируют исключений.

Это удивительно изящное решение. Мы конструируем временный объект на основе объекта `other`, затем вызываем `Swap` для обмена внутреннего содержимого нашего

и временного объектов. Затем временный объект выходит из области видимости, и его деструктор автоматически выполняет все необходимые действия по очистке и освобождению захваченных ресурсов.

Заметим, что когда мы таким образом делаем оператор присваивания безопасным, мы получаем побочный эффект, заключающийся в том, что при этом автоматически корректно отрабатывается присваивание объекта самому себе (например, `Stack s; s = s;`). (Поскольку присваивание самому себе встречается исключительно редко, я опустил традиционную проверку `if(this!=&other)`, имеющую, кстати говоря, свои тонкости, о которых вы узнаете подробнее в задаче 9.1.)

Заметим, что поскольку вся реальная работа выполняется при конструировании `temp`, все исключения, которые могут быть сгенерированы (при распределении памяти или в конструкторе копирования `T`), никак не влияют на состояние нашего объекта. Кроме того, при таком методе не может быть никаких утечек или других проблем, связанных с объектом `temp`, поскольку, как мы уже выяснили, конструктор копирования строго безопасен с точки зрения исключений. После того как вся необходимая работа выполнена, мы просто обмениваем внутренние представления нашего объекта и объекта `temp`, а при этом генерация исключений невозможна, поскольку `Swap` имеет спецификацию `throw()`, и в этой функции не выполняется никаких действий, кроме копирования объектов встроенных типов. На этом работа оператора присваивания заканчивается.

Еще раз обратите внимание, насколько этот метод элегантнее метода, представленного в решении задачи 2.2. В приведенном здесь решении требуются гораздо меньшие усилия для гарантии безопасности.

Если вы из тех людей, кто обожает сестру таланта, вы можете записать канонический вид оператора `operator=()` более компактно, создавая временную переменную посредством передачи аргумента функции по значению.

```
Stack& operator=( Stack temp )
{
    Swap( temp );
    return *this;
}
```

Stack<T>::Count()

Никаких неожиданностей — эта функция остается простейшей в написании.

```
size_t Count() const
{
    return vused_;
}
```

Stack<T>::Push()

Эта функция требует несколько большего внимания. Рассмотрим ее код, перед тем как читать дальше.

```
void   Push(const T&)
{
    if ( vused_ == vsize_ )
    {
        Stack temp( vsize_*2 + 1 );
        while( temp.Count() < vused_ )
        {
            temp.Push( v_[temp.Count()] );
        }
        temp.Push( t );
        Swap( temp );
```

```
    }
    else
    {
        construct( v_ + vused_, t );
        ++vused_;
    }
}
```

Рассмотрим сначала более простой случай `else`: если у нас имеется место для нового объекта, мы пытаемся сконструировать его. Если это нам удается, мы обновляем величину `vused_`. Здесь все просто, понятно и безопасно.

В противном случае, когда у нас не хватает памяти для нового элемента, мы инициируем перераспределение памяти. В этом случае мы просто создаем временный объект `Stack`, помещаем в него новый элемент и обмениваем внутренние данные нашего и временного объектов.

Является ли приведенный код безопасным? Да. Судите сами.

- Если происходит сбой при конструировании временного объекта, состояние нашего объекта остается неизменным, и не происходит утечки никаких ресурсов, так что тут все в порядке.
- Если произойдет сбой с генерацией исключения при загрузке содержимого в объект `temp` (включая конструктор копирования нового объекта), он будет корректно уничтожен вызовом деструктора при выходе объекта из области видимости.
- Ни в каком из случаев мы не изменяем состояние нашего объекта до тех пор, пока вся работа не будет успешно завершена.

Заметим, что тем самым обеспечивается гарантия принятия-или-отката, поскольку `Swap()` выполняется, только когда успешна полная операция перераспределения памяти и внесения элементов в стек. Никакие ссылки, возвращаемые `Top()`, или итераторы (если позже мы будем их использовать), не окажутся недействительными из-за перераспределения памяти до тех пор, пока вставка не завершится полностью успешно.

Stack<T>::Top()

Функция `Top()` не претерпела никаких изменений.

```
T&     Top()
{
    if ( vused_ == 0 )
    {
        throw "empty stack";
    }
    return v_[vused_ - 1];
}
```

Stack<T>::Pop()

Эта функция использует вызов описанной ранее функции `destroy()`.

```
void   Pop()
{
    if ( vused_ == 0 )
    {
        throw "pop from empty stack";
    }
    else
    {
        --vused_;
```

```
            destroy(v_ + vused_ );
        }
    }
};
```

Резюмируя, можно сказать, что функция `Push()` упростилась, но наибольшая выгода от инкапсуляции владения ресурсами в отдельном классе видна в конструкторе и деструкторе `Stack`. Благодаря `StackImpl` мы можем написать любое количество конструкторов, не беспокоясь об освобождении ресурсов, в то время как в предыдущем решении соответствующий код должен был содержаться в каждом конструкторе.

Вы также можете заметить, что устранен даже имевшийся в предыдущей версии блок `try/catch`, — так что нам удалось написать полностью безопасный и нейтральный код, не использовав ни одного `try`! (Кто там говорил о том, что написать безопасный код трудно?[11])

Задача 2.7. Разработка безопасного кода. Часть 7 *Сложность: 5*

Небольшая разминка.

Представим, что комментарий `/*????*/` в `StackImpl` представляет собой `public`. Реализуйте все функции-члены приведенной далее версии `Stack`, реализованной посредством `StackImpl` с использованием его в качестве объекта-члена.

```
template<class T>
class Stack
{
public:
    Stack(size_t size = 0);
    ~Stack();
    Stack(const Stack&);
    Stack& operator=(const Stack&);
    size_t Count() const;
    void   Push(const T&);
    T&     Top(); // Если стек пуст, генерируется исключение
    void   Pop(); // Если стек пуст, генерируется исключение
private:
    StackImpl<T> impl_; // Сокрытие реализации
};
```

Не забывайте о безопасности кода.

Решение

Эта реализация класса `Stack` лишь немного отличается от предыдущей. Например, `Count()` возвращает `impl_.vused_` вместо унаследованного `vused_`.

Вот полный код класса.

```
template<class T>
class Stack
{
public:
    Stack(size_t size = 0)
        : impl_(size)
    {
    }
    Stack(const Stack& other)
```

[11] Непереводимая игра слов, основанная на том, что слово *trying* означает “утомительный, трудный”. — *Прим. перев.*

```
        : impl_(other.impl_.vused_)
    {
        while(impl_.vused_ < other.impl_.vused_)
        {
            construct(impl_.v_ + impl_.vused_,
                      other.impl_.v_[impl_.vused_]);
            ++impl_.vused_;
        }
    }
    Stack& operator=(const Stack& other)
    {
        Stack temp(other);
        impl_.Swap(temp.impl_); // Здесь исключений нет
        return *this;
    }
    size_t Count() const
    {
        return impl_.vused_;
    }
    void   Push(const T& t)
    {
        if (impl_.vused_ == impl_.vsize_)
        {
            Stack temp(impl_.vsize_*2 + 1);
            while(temp.Count() < impl_.vused_)
            {
                temp.Push(impl_.v_[temp.Count()]);
            }
            temp.Push(t);
            impl_.Swap(temp.impl_);
        }
        else
        {
            construct(impl_.v_ + impl_.vused_, t);
            ++impl_.vused_;
        }
    }
    T&     Top()
    {
        if (impl_.vused_ == 0)
        {
            throw "empty stack";
        }
        return impl_.v_[impl_.vused_ - 1];
    }
    void   Pop()
    {
        if (impl_.vused_ == 0)
        {
            throw "pop from empty stack";
        }
        else
        {
            --impl_.vused_;
            destroy(impl_.v_ + impl_.vused_);
        }
    }
private:
    StackImpl<T> impl_; // Сокрытие реализации
};
```

Вот и все.

Задача 2.8. Разработка безопасного кода. Часть 8 *Сложность: 9*

После проделанной работы пришло время перевести дух и ответить на несколько вопросов.

1. Какая технология лучше — использование `StackImpl` в качестве закрытого базового класса или в качестве объекта-члена?
2. Насколько повторно используемы последние версии `Stack`? Какие требования предъявляются к типу элементов стека `T`? (Другими словами, с объектами каких типов может работать `Stack`? Чем меньше требований к типу `T`, тем больше степень повторного использования `Stack`.)
3. Следует ли использовать спецификации исключений для функций-членов `Stack`?

Решение

Будем отвечать на поставленные вопросы по порядку.

1. Какая технология лучше — использовать `StackImpl` в качестве закрытого базового класса или в качестве объекта-члена?

Оба метода дают по сути одинаковый результат, при этом четко разделяя вопросы управления памятью и создания и уничтожения объектов.

При принятии решения об использовании закрытого наследования или объекта-члена класса я всегда предпочитаю использовать последний, применяя наследование только там, где это совершенно необходимо. Обе технологии означают “реализован посредством” и предполагают разделение различных аспектов работы, поскольку использование класса клиентом ограничено только доступом к открытому интерфейсу класса. Используйте наследование только тогда, когда это абсолютно необходимо, что означает следующее:

- вам необходим доступ к защищенным членам класса, или
- вам требуется переопределение виртуальной функции, или
- объект должен быть сконструирован раньше других базовых подобъектов.[12]

2. Насколько повторно используемы последние версии `Stack`? Какие требования предъявляются к типу элементов стека `T`? (Другими словами, с объектами каких типов может работать `Stack`? Чем меньше требований к типу `T`, тем больше степень повторного использования `Stack`.)

При написании шаблона класса, в частности, такого, который потенциально может использоваться как обобщенный контейнер, всегда задавайте себе один ключевой вопрос: насколько повторно используем ваш класс? Или — какие ограничения я накладываю на пользователей этого класса и не будут ли эти ограничения чрезмерно ограничивать возможности пользователей при работе с классом?

Последняя реализация `Stack` имеет два основных отличия от первой версии. Одно из них мы уже рассматривали. Последняя версия `Stack` отделяет управление памятью от создания и уничтожения хранимых объектов, что очень приятно, но никак не влияет на пользователя. Однако есть еще одно важное отличие — новый `Stack` создает и уничтожает отдельные объекты по мере необходимости, вместо создания объектов `T` по умолчанию, заполняющих весь буфер, с последующим присваиванием новых значений по мере необходимости.

---

[12] Конечно, в случае наследования соблазнительным фактором является то, что нам не придется так много раз писать “`impl_.`”.

Это отличие дает нам существенные преимущества: бо́льшую эффективность и сниженные требования к типу хранимых объектов `T`. Вспомним, что первая версия `Stack` требовала от `T` наличия следующего.

- Конструктора по умолчанию (для заполнения буфера `v_`).
- Конструктора копирования (если функция `Pop()` возвращает объект по значению).
- Деструктора, не генерирующего исключений (для обеспечения безопасности).
- Безопасного копирующего присваивания (для установки значений `v_` и для обеспечения неизменности целевого объекта при генерации исключений в копирующем присваивании; заметим, что это единственная функция-член `T`, которая должна быть безопасной для того, чтобы обеспечить безопасность нашего класса `Stack`).

В последней версии нам не требуется наличие конструктора по умолчанию, поскольку нами используется только конструктор копирования. Кроме того, не требуется копирующее присваивание, поскольку ни в `Stack`, ни в `StackImpl` не происходит присваивания хранящихся в стеке объектов. Это означает, что требования новой версии `Stack` снижены до наличия

- конструктора копирования и
- деструктора, не генерирующего исключений (для обеспечения безопасности).

Насколько это влияет на используемость класса `Stack`? В то время как множество классов имеет конструктор по умолчанию и оператор копирующего присваивания, у немалого количества вполне полезных и широко используемых классов их нет (кстати, некоторые объекты просто не могут быть присвоены, например, объекты, содержащие члены-ссылки, поскольку последние не могут быть изменены). Теперь `Stack` может работать и с такими объектами, в то время как исходная версия такой возможности не давала. Это определенно большое преимущество новой версии, и использовать новую версию `Stack` сможет большее количество пользователей.

Рекомендация

При разработке всегда помните о повторном использовании.

3. Следует ли использовать спецификации исключений для функций-членов `Stack`?

Вкратце: нет, поскольку мы, авторы `Stack`, не обладаем достаточными знаниями (да и при полных знаниях вряд ли станем это делать). То же справедливо в отношении, по сути, любого из обобщенных контейнеров.

Во-первых, рассмотрим, что мы, как авторы `Stack`, знаем о типе `T`? В частности, мы не знаем заранее, какие именно операции `T` могут привести к исключениям и к каким именно. Конечно, мы всегда можем предъявить дополнительные требования к типу `T`, которые увеличат наши знания о типе `T` и позволят добавить спецификации исключений к функциям-членам `Stack`. Однако наложение дополнительных ограничений противоречит цели сделать класс `Stack` максимально повторно используемым, так что данный метод нам не подходит.

Во-вторых, вы можете заметить, что некоторые операции контейнера (например, `Count()`) просто возвращают скалярное значение и гарантированно не могут генерировать исключений. Следовательно, их можно описать с указанием спецификации исключений `throw()`? Да, можно, но не нужно. Тому есть две основные причины.

- Указание ограничения `throw()` ограничит в будущем ваши возможности по изменению реализации этих функций — вы не будете иметь права генерировать в них исключения. Ослабление же спецификации исключений всегда несет опре-

деленный риск нанесения вреда существующему клиенту (новая версия класса нарушает старое обещание), так что ваш класс будет плохо поддаваться внесению изменений. (Указание спецификации исключений у виртуальных функций, кроме того, снижает повторное использование класса, поскольку существенно ограничивает возможности программистов, которые могут захотеть использовать ваш класс в качестве базового для создания своих собственных классов, производных от вашего. Использование спецификации может иметь смысл, но такое решение требует длительных размышлений.)

- Спецификации исключений могут привести к повышенным накладным расходам независимо от того, генерируется ли на самом деле исключение или нет (хотя многие современные компиляторы и минимизируют эти расходы). Для часто используемых операций и контейнеров общего назначения лучше все же избегать дополнительных расходов и не использовать спецификации исключений.

Задача 2.9. Разработка безопасного кода. Часть 9 ***Сложность: 8***

Насколько хорошо вы понимаете, что делает безобидное выражение `delete[]`? Что оно означает с точки зрения безопасности исключений?

Вот и настало время рассмотреть давно ожидаемый вопрос — “Деструкторы, генерирующие исключения, и почему они неприемлемы”.

Рассмотрим выражение `delete[] p;`, где `p` указывает на корректный массив в памяти, который был корректно выделен и инициализирован с использованием оператора `new[]`.

1. Что в действительности делает `delete[]p;`?
2. Насколько безопасно данное выражение? Будьте максимально точны в своем ответе.

Решение

Мы подошли к ключевой теме, а именно к невинно выглядящему оператору `delete[]`. В чем заключается его работа? Насколько он безопасен?

Деструкторы, генерирующие исключения, и почему они неприемлемы

Для начала вспомним нашу стандартную вспомогательную функцию `destroy` из врезки “Некоторые стандартные вспомогательные функции” в задаче 2.5.

```
template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
    while( first != last )
    {
        destroy( &*first ); // Вызывает деструктор *first
        ++first;
    }
}
```

В рассматривавшейся задаче эта функция была безопасна, поскольку мы выдвигали требование, чтобы деструктор `T` не мог генерировать исключений. Но что если деструкторы объектов, содержащихся в нашем контейнере, будут способны генерировать исключения? Рассмотрим, что произойдет, если передать функции `destroy` диапазон из, скажем, 5 объектов. Если исключение сгенерирует первый же деструктор, то про-

изойдет выход из функции destroy, и оставшиеся четыре объекта никогда не будут уничтожены. Совершенно очевидно, что это весьма некорректное поведение.

“Но, — можете прервать вы меня, — разве нельзя написать код, который бы корректно работал даже при условии, что деструктор T может генерировать исключения?” Ну, это не так просто, как можно подумать. Наверное, вы бы начали с кода наподобие этого.

```
template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
    while( first != last )
    {
        try
        {
            destroy( &*first );
        }
        catch(...)
        {
            /* И что дальше? */
        }
        ++first;
    }
}
```

Вся хитрость заключается в части, где находится комментарий “И что дальше?”. На самом деле у нас только три варианта действий: catch генерирует то же самое исключение, преобразует его в некоторое другое исключение, и не генерирует ничего, и выполнение цикла продолжается.

1. Если тело catch генерирует перехваченное исключение, то функция destroy, очевидно, соответствует требованиям нейтральности, поскольку свободно пропускает любые исключения T и передает их вызывающей функции. Но при этом функция нарушает требование безопасности об отсутствии утечек при возникновении исключений. Поскольку функция destroy не может сообщить о том, сколько объектов не было успешно уничтожено, эти объекты никогда так и не будут корректно уничтожены, а значит, связанные с ними ресурсы будут безвозвратно утрачены. Определенно, это не самое хорошее решение.
2. Если тело catch конвертирует исключение в некоторое иное, наша функция, очевидно, перестает соответствовать требованиям как нейтральности, так и безопасности. Этим сказано вполне достаточно.
3. Если тело catch не генерирует никаких исключений, то функция destroy, очевидно, соответствует требованию отсутствия утечек при генерации исключений.[13] Однако совершенно очевидно, что функция при этом нарушает требования нейтральности, которые говорят о том, что любое сгенерированное T исключение должно в неизменном виде дойти до вызывающей функции, в то время как в этой версии функции исключения поглощаются и игнорируются (с точки зрения вызывающей функции исключения игнорируются, даже если тело catch выполняет некоторую их обработку).

Мне встречались предложения перехватывать исключения и “сохранять” их, тем временем продолжая работу по уничтожению остальных объектов, а по окончании работы — повторно генерировать это исключение. Такой метод также не

[13] Вообще-то, если деструктор T может генерировать исключения таким образом, что при этом связанные с объектом ресурсы не освобождаются полностью, то утечки могут иметь место. Однако это уже не проблема функции destroy... это всего лишь означает, что тип T не является безопасным. Однако функция destroy не имеет утечек в том плане, что она не допускает сбоев при освобождении ресурсов, за которые отвечает (а именно — при освобождении переданных ей объектов T).

является решением, например, он не может корректно обработать ситуацию генерации нескольких исключений (даже если все они будут сохранены, то сгенерировать вы сможете только одно из них, а остальные будут молча поглощены). Вы можете искать другие пути решения проблемы, но, поверьте мне, все они сведутся к написанию некоторого кода наподобие рассмотренного, поскольку у вас есть множество объектов и все они должны быть уничтожены. Если деструктор `T` может генерировать исключения, в лучшем случае все заканчивается написанием небезопасного кода.

Это приводит нас к невинно выглядящим операторам `new[]` и `delete[]`. Вопросы, связанные с ними, по сути те же, что и описанные при рассмотрении функции `destroy`. Рассмотрим, например, следующий код.

```
T* p = new T[10];
delete[] p;
```

Выглядит очень просто и безвредно, не так ли? Но вы никогда не интересовались, что будут делать операторы `new[]` и `delete[]`, если деструктор `T` окажется способен генерировать исключения? Даже если интересовались, то вы все равно не знаете ответа, по той простой причине, что его не существует. Стандарт гласит, что если где-то в этом коде произойдет генерация исключения деструктором `T`, поведение кода будет непредсказуемым. Это означает, что любой код, который выделяет или удаляет массив объектов, деструкторы которых могут генерировать исключения, может привести к непредсказуемому поведению. Не смотрите так удивленно — сейчас вы увидите, почему это так.

Во-первых, рассмотрим, что произойдет, если все вызовы конструкторов успешно завершены, а затем, при выполнении операции `delete[]`, пятый деструктор `T` генерирует исключение. В этом случае возникает та же проблема, что и описанная выше при рассмотрении функции `destroy`. С одной стороны, исключению нельзя выйти за пределы оператора, поскольку при этом оставшиеся объекты `T` навсегда окажутся не удаленными, но, с другой стороны, нельзя и преобразовать или игнорировать исключение, так как это нарушит нейтральность кода.

Во-вторых, рассмотрим, что произойдет при генерации исключения пятым конструктором. В этом случае вызывается деструктор четвертого объекта, затем третьего и т.д., пока все успешно созданные объекты не будут уничтожены, а память не будет освобождена. Но что случится, если не все пройдет так гладко? Если, например, после генерации исключения пятым конструктором произойдет генерация исключения деструктором четвертого объекта? И (если мы его проигнорируем) третьего тоже? Вы представляете, куда это может нас завести?...

Если деструктор может генерировать исключения, то ни `new[]`, ни `delete[]` не могут быть сделаны безопасными и нейтральными. Вывод прост: *никогда не пишите деструкторы, которые позволяют исключениям покинуть их.*[14] Если вы пишете класс с таким деструктором, вы не будете способны обеспечить безопасность и нейтральность даже создания и уничтожения массива таких объектов посредством `new[]` и `delete[]`. Все деструкторы всегда должны разрабатываться так, как если бы они имели спецификацию исключений `throw()`, — т.е. ни одному исключению не должно быть позволено выйти за пределы деструктора.

[14] На заседании в Лондоне в июле 1997 года в черновой вариант стандарта было внесено следующее: “ни одна операция уничтожения, определенная в стандартной библиотеке C++, не генерирует исключений”. Это относится не только к самим стандартным классам, но и, в частности, запрещает инстанцировать стандартные контейнеры с типами, деструкторы которых могут генерировать исключения.

Рекомендация

Никогда не позволяйте исключениям покинуть деструктор или переопределенные операторы `delete()` и `delete[]()`. Разрабатывайте каждый деструктор и функции удаления объектов так, как если бы они имели спецификацию исключений `throw()`.

Допускаю, что некоторым такое состояние дел может показаться попросту плачевным, так как одна из основных причин введения исключений заключалась именно в том, чтобы иметь возможность сообщать о сбоях в конструкторах и деструкторах (поскольку у них нет возвращаемых значений). Однако это не совсем так, поскольку основным предназначением исключений было их использование в конструкторах (в конце концов, деструкторы предназначены для уничтожения объектов, так что область видимости их сбоев определенно меньше, чем у конструкторов). А в этой части все в порядке — исключения отлично служат для сообщения об ошибках конструкторов, включая сбои при создании массивов посредством оператора `new[]`. В этом случае поведение программы вполне предсказуемо, даже при генерации исключений в конструкторе.

Безопасность исключений

Правило "тише едешь — дальше будешь"[15] вполне применимо и при написании безопасного кода контейнеров и других объектов. Для успешного решения этой задачи вам нередко придется быть исключительно внимательным и осторожным. Однако пугаться исключений не стоит — достаточно следовать приведенным выше правилам, и у вас все отлично получится. Просто не забывайте об изоляции управления ресурсами, используйте идиому "создай и обменяй временный объект", и никогда не позволяйте исключениям выскользнуть из деструкторов — и у вас будет получаться отличный, безопасный и нейтральный код.

Для удобства соберем все правила безопасности в одном месте. Не забывайте время от времени освежать их в памяти!

Рекомендация

Не забывайте о канонических правилах безопасности исключений. 1. Никогда не позволяйте исключениям покинуть деструктор или переопределенные операторы `delete()` и `delete[]()`. Разрабатывайте каждый деструктор и функции удаления объектов так, как если бы они имели спецификацию исключений `throw()`. 2. Всегда используйте идиому "захвата ресурса при инициализации" для отделения владения и управления ресурсами. 3. В каждой функции следует собрать весь код, который может генерировать исключения, и выполнить его отдельно, безопасным с точки зрения исключений способом. Только после этого, когда вы будете знать, что вся реальная работа успешно выполнена, вы можете изменять состояние программы (а также выполнять другие необходимые действия, например, освобождение ресурсов) посредством операций, которые не генерируют исключений.

Задача 2.10. Разработка безопасного кода. Часть 10 ***Сложность: 9.5***

Вам еще не надоели задачи с одним и тем же названием "Разработка безопасного кода"? Это последняя. Спасибо за то, что вы дочитали до конца эту серию. Надеюсь, она вам понравилась.

[15] В оригинале "be aware, drive with care". — *Прим. перев.*

Пожалуй, вы уже утомлены и опустошены чтением предыдущих задач. Это понятно. Завершая путешествие по первой серии задач этой главы, позвольте преподнести прощальный подарок. Здесь хочется вспомнить всех, кто способствовал тому, чтобы рассмотренные нами принципы и гарантии были включены в стандартную библиотеку. Я хотел бы выразить искреннюю признательность Дэйву Абрахамсу (Dave Abrahams), Грегу Колвину (Greg Colvin) Мэтту Остерну (Matt Austern) и другим, кому удалось добиться внесения гарантий безопасности в стандартную библиотеку буквально за несколько дней до финального заседания ISO WG21/ANSI J16 в ноябре 1997 года в Морристауне, Нью-Джерси.

Является ли стандартная библиотека C++ безопасной?

Поясните.

Решение

Безопасность исключений и стандартная библиотека

Является ли стандартная библиотека C++ безопасной? Краткий ответ: да.[16]

- Все итераторы, возвращаемые стандартными контейнерами, безопасны и могут копироваться без генерации исключений.
- Все стандартные контейнеры должны реализовывать базовую гарантию для всех операций: они всегда разрушаемы и всегда находятся в согласованном (если не предсказуемом) состоянии даже при наличии исключений.
- Для того чтобы это стало возможным, ряд важных функций должен реализовывать гарантию отсутствия исключений, включая `swap` (важность которой была продемонстрирована в предыдущей задаче), `allocator<T>::deallocate` (важность этой функции продемонстрирована в процессе рассмотрения оператора `delete()`) и некоторых операций типов-параметров шаблонов (в особенности деструктора, о чем говорилось в задаче 2.9, в разделе “Деструкторы, генерирующие исключения, и почему они неприемлемы”).
- Все стандартные контейнеры должны также реализовывать строгую гарантию для всех операций (с двумя исключениями). Они должны всегда следовать семантике принятия-или-отката, так что операции типа вставки должны быть либо полностью успешны, либо не изменять состояние программы вовсе. “Не изменять состояние” означает, что неуспешно завершившаяся операция не влияет на корректность любого итератора, который в начале операции указывает внутрь контейнера.

  Имеется два исключения из этого правила. Во-первых, для всех контейнеров множественная вставка (вставка диапазона, указываемого итераторами) не является строго безопасной. Во-вторых, у `vector<T>` и `deque<T>` (и только у них) вставка и удаление (неважно, одно- или многоэлементные) являются строго безопасными при условии, что конструктор копирования и оператор присваивания `T` не генерируют исключений. Обратите внимание на результаты этих ограничений. К сожалению, среди прочего это означает, что вставка и удаление для контейнеров `vector<string>` или, например, `vector<vector<int> >` не являются строго безопасными.

[16] Я рассматриваю контейнеры и итераторы стандартной библиотеки. Другие элементы библиотеки, — например потоки, — обеспечивают как минимум базовые гарантии безопасности.

Откуда взялись эти ограничения? Они возникли постольку, поскольку откат любой операции такого типа невозможен без дополнительных расходов времени и памяти, и стандарт не требует идти на них во имя безопасности исключений. Все остальные операции могут быть реализованы как безопасные без дополнительных накладных расходов. Таким образом, при внесении диапазона элементов в контейнер (или если вы вносите диапазон элементов в `vector<T>` или `deque<T>` и при этом конструктор копирования или оператор присваивания `T` может генерировать исключения) он не обязательно будет иметь предсказуемое содержимое, а итераторы, указывающие внутрь него, могут стать недействительными.

Что это означает для вас? Если вы пишете класс, в составе которого имеется член-контейнер, и выполняете вставку диапазона (или, если этот член `vector<T>` или `deque<T>` и при этом конструктор копирования или оператор присваивания `T` может генерировать исключения), то вы должны выполнить дополнительную работу для того, чтобы гарантировать предсказуемость состояния вашего собственного класса при генерации исключений. К счастью, эта "дополнительная работа" достаточно проста. При вставке в контейнер или удалении из него сначала скопируйте его, затем измените копию. И, если все прошло успешно, воспользуйтесь функцией `swap`, чтобы обменять содержимое исходного контейнера и измененного.

Задача 2.11. Сложность кода. Часть 1 *Сложность: 9*

В этой задаче представлен вопрос "на засыпку": сколько путей выполнения может быть в функции из трех строк? Правильный ответ наверняка удивит вас.

Сколько путей выполнения может быть в приведенном коде?

```
String EvaluateSalaryAndReturnName( Employee e )
{
    if ( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        cout<<e.First()<<" "<<e.Last()<<" is overpaid"<<endl;
    }
    return e.First() + " " + e.Last();
}
```

Первоначально примем следующие предположения (без которых позже попытаемся обойтись).

1. Игнорируем различный порядок вычисления параметров функции, как и возможные сбои в работе деструкторов.
2. Рассматриваем вызываемые функции как атомарные.
3. При подсчете различных путей выполнения учитываем, что каждый из них должен состоять из своей уникальной последовательности вызовов и выходов из функций.

Решение

Сначала рассмотрим следствия предложенных в условии предположений.

1. Игнорируем различный порядок вычисления параметров функции, как и возможные сбои в работе деструкторов.[17] Сразу же вопрос для героев: насколько изменится количество путей выполнения, если деструкторы могут генерировать исключения?
2. Рассматриваем вызываемые функции как атомарные. Например, вызов `e.Title()` может генерировать исключения по самым разным причинам (генерация исключения может быть вызвана кодом самой функции; функция может не перехватывать исключение, сгенерированное другой функцией; исключение может генерироваться конструктором временного объекта и т.д.). Все, что нас интересует при рассмотрении данной функции, — сгенерировано ли исключение при ее работе или нет.
3. При подсчете различных путей выполнения учитываем, что каждый из них должен состоять из своей уникальной последовательности вызовов и выходов из функций.

Итак, сколько возможных путей выполнения имеется в представленном коде? Ответ: 23 (в каких-то трех строках кода!).

Если ваш ответ:	*Ваш уровень:*
3	Средний
4–14	Специалист по исключениям
15–23	Гуру

Все 23 пути состоят из

- 3 путей без исключений;
- 20 невидимых путей, связанных с исключениями.

Под *путями без исключений* я подразумеваю пути выполнения, которые возможны даже в случае, если не генерируется ни одно исключение. Эти пути — результат нормального выполнения программы C++. Под путями, связанными с исключениями, я подразумеваю пути выполнения, получившиеся в результате генерации или распространения исключения, и рассматриваю такие пути отдельно.

Пути без исключений

Для того чтобы подсчитать пути без исключений, достаточно знать о сокращенных вычислениях логических выражений в C/C++.

```
if ( e.Title() == "CEO" || e.Salary() > 100000 )
```

1. Если выражение `e.Title() == "CEO"` истинно, то вторая часть выражения не вычисляется (и `e.Salary()` не вызывается), но вывод в `cout` осуществляется.

   При наличии перегрузки операторов `==`, `||` и/или `>` в условии `if`, оператор `||` может на самом деле оказаться вызовом функции. Если это так, то сокращенного вычисления не будет, и обе части выражения будут вычисляться в любом случае.
2. Если `e.Title() != "CEO"`, а `e.Salary() > 100000`, будут вычислены обе части выражения и выполнен вывод в `cout`.
3. Если `e.Title() != "CEO"`, а `e.Salary() <= 100000`, будут вычислены обе части выражения, но вывод в `cout` выполняться не будет.

[17] Никогда не позволяйте исключениям покидать деструкторы. Такой код никогда не будет корректно работать. Почему это так, см. в разделе “Деструкторы, генерирующие исключения, и почему они неприемлемы” задачи 2.9.

Пути с исключениями

Здесь мы рассматриваем пути, связанные с исключениями.

```
String EvaluateSalaryAndReturnName( Employee e )
  ^*^                                     ^4^
```

4. Аргумент передается по значению, что приводит к вызову конструктора копирования `Employee`. Эта операция копирования может вызвать генерацию исключения.

(*) Конструктор копирования `String` может генерировать исключение при копировании временного возвращаемого значения. Мы игнорируем эту возможность, поскольку такая генерация исключения происходит за пределами рассматриваемой функции (у нас и без того достаточно работы).

```
 if ( e.Title() == "CEO" || e.Salary() > 100000 )
         ^5^   ^7^  ^6^ ^11^   ^8^    ^10^  ^9^
```

5. Функция-член `Title()` может генерировать исключение, как сама по себе, так и в процессе копирования в случае возвращения объекта по значению.

6. Для использования в операторе `operator==()` строка должна быть преобразована во временный объект (вероятно, того же типа, что и возвращаемое `e.Title()` значение), при конструировании которого может быть сгенерировано исключение.

7. Если `operator==()` представляет собой пользовательскую функцию, то она также может генерировать исключения.

8. Аналогично п. 5, функция-член `Salary()` может генерировать исключение, как сама по себе, так и в процессе копирования в случае возвращения объекта по значению.

9. Аналогично п. 6, здесь может потребоваться создание временного объекта, в процессе чего может быть сгенерировано исключение.

10. Аналогично п. 7, если `operator>()` представляет собой пользовательскую функцию, то она также может генерировать исключения.

11. Аналогично пп. 7 и 10, если `operator||()` представляет собой пользовательскую функцию, то она также может генерировать исключения.

```
  cout << e.First() << " " << e.Last() << " is overpaid" << endl;
      ^12^  ^17^   ^13^   ^14^  ^18^  ^15^               ^16^
```

12–16. Как документировано стандартом C++, любой из пяти вызовов оператора << может генерировать исключения.

17–18. Аналогично п. 5, `First()` и/или `Last()` могут генерировать исключения, как сами по себе, так и в процессе копирования в случае возвращения объекта по значению.

```
 return e.First()  +   " "   +  e.Last();
          ^19^   ^22^ ^21^ ^23^   ^20^
```

19–20. Аналогично п. 5, `First()` и/или `Last()` могут генерировать исключения, как сами по себе, так и в процессе копирования в случае возвращения объекта по значению.

21. Аналогично п. 6, может потребоваться создание временного объекта, конструктор которого может генерировать исключения.

22–23. Аналогично п. 7, если оператор представляет собой пользовательскую функцию, то она также может генерировать исключения.

Рекомендация

Всегда помните об исключениях. Вы должны точно знать, где именно они могут генерироваться.

Одна из целей данной задачи — продемонстрировать, как много невидимых путей выполнения может иметься в простом коде в языке, позволяющем работу с исключениями. Влияет ли эта невидимая сложность на надежность и тестируемость функций? Об этом вы узнаете из следующей задачи.

Задача 2.12. Сложность кода. Часть 2 *Сложность: 7*

Сделайте три строки кода из предыдущей задачи безопасными с точки зрения исключений. Эта задача преподнесет важный урок, касающийся безопасности.

Является ли функция из задачи 2.11 безопасной (корректно работающей при наличии исключений) и нейтральной (передающей все исключения вызывающей функции)?

```
String EvaluateSalaryAndReturnName( Employee e )
{
    if ( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        cout<<e.First()<<" "<<e.Last()<<" is overpaid"<<endl;
    }
    return e.First() + " " + e.Last();
}
```

Поясните ваш ответ. Если эта функция безопасна, то поддерживает ли она базовую гарантию, строгую гарантию или гарантию отсутствия исключений? Если нет, каким образом ее следует изменить для поддержки одной из этих гарантий?

Предположим, что все вызываемые функции безопасны (могут генерировать исключения, но не имеют побочных эффектов при их генерации) и что использование любых объектов, включая временные, также безопасно (при уничтожении объектов освобождаются все захваченные ресурсы).

Для того чтобы вспомнить о разных типах гарантий, обратитесь еще раз к задаче 2.4. Вкратце: базовая гарантия обеспечивает уничтожение объектов и отсутствие утечек; строгая, кроме того, обеспечивает семантику принятия-или-отката. Гарантия отсутствия исключений гарантирует, что функция не генерирует исключений.

Решение

Вначале, до того как мы приступим к решению задачи, несколько слов о предположениях из ее условия.

Как указано, мы предполагаем, что все вызываемые функции — включая функции потока — безопасны, как и все используемые объекты, включая временные.

Потоки, однако, портят картину, поскольку могут иметь “неоткатываемые” побочные эффекты. Например, оператор << может сгенерировать исключение после вывода части строки, которую уже не вернуть назад из потока; кроме того, может устанавливаться состояние ошибки потока. Этот вопрос нами игнорируется, так как цель нашей задачи — изучить, каким образом можно сделать функцию безопасной, если она имеет два побочных действия.

Итак, является ли функция из задачи 2.11 безопасной (корректно работающей при наличии исключений) и нейтральной (передающей все исключения вызывающей функции)?

```
String EvaluateSalaryAndReturnName( Employee e )
{
    if ( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        cout<<e.First()<<" "<<e.Last()<<" is overpaid"<<endl;
    }
    return e.First() + " " + e.Last();
}
```

В таком виде функция удовлетворяет базовой гарантии: при наличии исключений отсутствует утечка ресурсов.

Эта функция *не* удовлетворяет строгой гарантии. Строгая гарантия гласит, что если функция завершается неуспешно из-за генерации исключения, то состояние программы не должно изменяться. Однако функция `EvaluateSalaryAndReturnName()` имеет два побочных действия.

- В поток `cout` выводится сообщение "...overpaid...".
- Возвращается строка с именем.

В принципе второе действие не мешает условию строгой гарантии, поскольку при генерации исключения в функции строка с именем не будет возвращена. Однако, рассмотрев первое действие, можно сделать вывод о том, что функция небезопасна по двум причинам.

- Если исключение генерируется после того, как в `cout` выведена первая часть сообщения, но до завершения всего вывода (например, если исключение генерируется четвертым оператором <<), то в `cout` будет выведена только часть сообщения.[18]
- Если же сообщение выведено полностью, но после этого в функции генерируется исключение (например, во время построения возвращаемого значения), то сообщение оказывается выведенным, несмотря на то, что выполнение функции завершается неуспешно.

И наконец, очевидно, что функция не удовлетворяет гарантии отсутствия исключений: множество операций внутри нее могут генерировать исключения, в то время как в самой функции не имеется ни блоков `try/catch`, ни спецификации `throw()`.

Рекомендация

Четко различайте требования различных гарантий безопасности.

Для соответствия строгой гарантии требуется, чтобы либо полностью были завершены оба действия, либо, при генерации исключения, ни одно из них.

Можем ли мы добиться этого? Мы можем попытаться сделать это, например, таким образом.

```
// Попытка №1. Улучшение.
//
String EvaluateSalaryAndReturnName( Employee e )
{
    String result = e.First() + " " + e.Last();
    if ( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        String message = result + " is overpaid\n";
```

[18] Если вы считаете, что беспокойство о том, выведено ли сообщение полностью или только частично, — проявление излишнего педантизма, вы в чем-то правы. Однако рассматриваемые нами принципы применимы к любой функции, выполняющей два побочных действия, так что такой педантичный подход в нашем случае оправдан и весьма полезен.

```
        cout << message;
    }
    return result;
}
```

Получилось неплохо. Заметьте, что мы заменили endl символом \n (который, вообще говоря, не является точным эквивалентом), для того чтобы вывод строки осуществлялся одним оператором <<. (Конечно, это не гарантирует, что вывод в поток не может оказаться неполным, но это все, что мы можем сделать на таком высоком уровне.)

В приведенном решении все еще имеется одна неприятность, иллюстрируемая следующим кодом клиента.

```
// Маленькая неприятность...
//
String theName;
theName = EvaluateSalaryAndReturnName( bob );
```

Конструктор копирования String вызывается вследствие того, что результат возвращается по значению, а для копирования результата в переменную theName вызывается копирующее присваивание. Если любое из этих копирований выполняется неуспешно, функция все равно полностью выполняет все свои побочные действия, но результат функции оказывается безвозвратно утерян...

Как же нам поступить? Возможно, мы сможем избежать проблемы, избежав копирования? Например, можно использовать дополнительный параметр функции, передаваемый по ссылке и предназначенный для сохранения возвращаемого значения.

```
// Попытка №2. Стала ли функция безопаснее?
//
String EvaluateSalaryAndReturnName( Employee e,
                                    String& r )
{
    String result = e.First() + " " + e.Last();
    if ( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        String message = result + " is overpaid\n";
        cout << message;
    }
    r = result;
}
```

Выглядит это решение несколько лучше, но на самом деле это не так, поскольку присваивание результата r может оказаться неуспешным, что приведет к выполнению одного действия функции, оставив незавершенным другое. Следовательно, наша вторая попытка неудачна.

Один из способов решения задачи состоит в возврате указателя на динамически выделенный объект типа String. Однако лучшее решение состоит в добавлении еще одного шага и возврате указателя “в обертке” auto_ptr.

```
// Попытка №3. Наконец-то успешная!
//
auto_ptr<String>
    EvaluateSalaryAndReturnName( Employee e )
{
    auto_ptr<String> result
        = new String( e.First() + " " + e.Last() );
    if ( e.Title() == "CEO" || e.Salary() > 100000 )
    {
        String message = result + " is overpaid\n";
        cout << message;
    }
    return result; // Передача владения; здесь
                   // исключения генерироваться не могут
}
```

Здесь используется определенная хитрость, заключающаяся в том, что мы успешно скрываем всю работу по построению второго действия функции (т.е. возвращаемого результата), и при этом гарантируем, что возврат результата будет выполнен без генерации исключений после полного завершения первого действия (вывода сообщения). Мы знаем, что если функция будет успешно завершена, то возвращаемое значение будет успешно передано вызывающей функции, и во всех случаях будет корректно выполнен код освобождения захваченных ресурсов. Если вызывающая функция принимает возвращаемое значение, то при этом произойдет передача владения объектом; если же вызывающая функция не принимает возвращаемое значение, — например, просто проигнорировав его, — динамически выделенный объект `String` будет автоматически уничтожен (и освобождена занимаемая им память) при уничтожении хранящего его временного объекта `auto_ptr`. Какова цена этой дополнительной безопасности? Как это часто случается при реализации строгой безопасности, она обеспечивается (как правило, небольшими) потерями эффективности — в нашем случае за счет дополнительного динамического выделения памяти. Однако когда встает вопрос поиска компромисса между эффективностью, с одной стороны, и предсказуемостью и корректностью, с другой, — предпочтение отдается предсказуемости и корректности.

Рассмотрим еще раз вопрос о безопасности исключений и множественных побочных действиях. В этом случае можно использовать подход из третьей попытки, где оба действия выполняются по сути с использованием семантики принятия-или-отката (за исключением вывода в поток). Это стало возможным благодаря использованию методики, в соответствии с которой оба действия могут быть выполнены атомарно, т.е. вся "реальная" подготовительная работа для обоих действий может быть полностью выполнена таким образом, что сами видимые действия выполняются только с использованием операций, не генерирующих исключения.

Хотя в этот раз нам повезло, вообще говоря, это далеко не простая задача. Невозможно написать строго безопасную функцию, которая имеет два или большее количество несвязанных побочных действий, которые не могут быть выполнены атомарно (например, что если эти два действия состоят в выводе двух сообщений в потоки `cout` и `cerr`?), поскольку строгая гарантия гласит, что при наличии исключений "состояние программы останется неизменным", — другими словами, при наличии исключений не должны выполняться никакие побочные действия. Когда вы работаете с ситуацией, в которой два побочных действия не могут быть выполнены атомарно, обычно единственным способом обеспечить строгую безопасность является разбиение одной функции на две другие, которые могут быть выполнены атомарно. Таким образом достигается, как минимум, то, что вызывающей функции явно указывается невозможность атомарного выполнения этих действий.

Рекомендация

Прилагайте максимум усилий к тому, чтобы каждая часть кода — каждый модуль, класс, функция, — отвечали за выполнение одной четко определенной задачи.

Эта задача проиллюстрировала три важных момента.

1. Обеспечение строгой гарантии часто (но не всегда) требует от вас идти на компромисс со снижением производительности.
2. Если функция имеет множественные несвязанные побочные действия, она не всегда может быть сделана строго безопасной. Если сделать ее строго безопасной невозможно, ее следует разбить на ряд функций, в каждой из которых побочные действия могут быть выполнены атомарно.

3. Не все функции обязаны быть строго безопасными. Как исходный код, так и код из первой попытки удовлетворяют условиям базовой гарантии. Для множества клиентов попытки №1 (минимизирующей возможности побочных действий при наличии исключений без снижения производительности) будет вполне достаточно.

Небольшой постскриптум к задаче, касающийся потоков и побочных действий.

В условии задачи было сказано: предположим, что все вызываемые функции безопасны (могут генерировать исключения, но не имеют побочных эффектов при их генерации) и что использование любых объектов, включая временные, также безопасно (при уничтожении объектов освобождаются все захваченные ресурсы).

Опять же, это предположение о том, что ни одна из вызываемых функций не имеет побочных действий, не может быть полностью справедливым. В частности, нет возможности гарантировать, что операции с потоками не дадут сбой после частичного вывода. Это означает, что мы не в состоянии достичь точной семантики принятия-или-отката при выполнении вывода в поток (как минимум, не при работе со стандартными потоками).

Еще одна проблема состоит в том, что при сбое вывода в поток его состояние изменяется. Здесь мы не проверяли состояние потока и не восстанавливали его, но с целью дальнейшего усовершенствования функции можно перехватывать генерируемые потоком исключения и сбрасывать флаг ошибки потока `cout` перед тем, как передать перехваченное исключение вызывающей функции.

Задача 2.13. Исключения в конструкторах. Часть 1 *Сложность: 4*

Что именно происходит при генерации исключения конструктором? Что происходит, когда исключение генерируется при попытке создания подобъекта или объекта-члена?

1. Рассмотрим следующий класс.

```
// Пример 1
//
class C: private A
{
    B b_;
};
```

Как в конструкторе `C` можно перехватить исключение, сгенерированное в конструкторе базового подобъекта (такого, как `A`) или объекта-члена (такого, как `b_`)?

2. Рассмотрим следующий код.

```
// Пример 2
//
{
    Parrot p;
}
```

Когда начинается время жизни объекта? Когда оно заканчивается? Каково состояние объекта за рамками его времени жизни? И наконец, что будет означать генерация исключения в конструкторе объекта?

Решение

Try-блоки функций были введены в C++ для небольшого увеличения области видимости исключений, которые могут генерироваться функцией. В этой задаче мы рассмотрим:

- что означает конструирование объектов и сбой конструктора в C++;
- как использовать try-блоки функций для преобразования (не подавления) исключений, сгенерированных конструкторами базового подобъекта или объекта-члена.

Для удобства в этой задаче под понятием “член” подразумевается “нестатический член данных класса”, если явно не оговорено иное.

Try-блоки функций

1. **Рассмотрим следующий класс.**

```
// Пример 1
//
class C: private A
{
    B b_;
};
```

Как в конструкторе `C` можно перехватить исключение, сгенерированное в конструкторе базового подобъекта (такого, как `A`) или объекта-члена (такого, как `b_`)?

Вот для чего нужны try-блоки функций.

```
// Пример 1a: try-блок конструктора
//
C::C()
try
    : A ( /* ... */ ) // Необязательный список инициализации
    , b_( /* ... */ )
{
}
catch( ... )
{
    /* Здесь мы перехватываем исключения, сгенерированные
       A::A() или B::B().

       Если A::A() успешно завершается, после чего B:B()
       генерирует исключение, язык гарантирует, что для
       уничтожения уже созданного базового объекта типа A
       до входа в блок catch будет вызван деструктор A::~A()
}
```

Но вот гораздо более интересный вопрос: почему мы хотим перехватить исключение? Этот вопрос приводит нас к первому из двух ключевых моментов данной задачи.

Время жизни объекта и исключения в конструкторе

Давайте рассмотрим вопрос о том, может ли (или должен) конструктор `C` поглощать исключения конструкторов `A` и `B` и не выпускать никаких исключений вообще. Однако, перед тем как мы ответим на этот вопрос, нам следует убедиться, что мы хорошо понимаем вопросы времени жизни объектов и что означает генерация исключений в конструкторах.

2. **Рассмотрим следующий код.**

```
// Пример 2
//
{
    Parrot p;
}
```

Когда начинается время жизни объекта? Когда оно заканчивается? Каково состояние объекта за рамками его времени жизни? И наконец, что будет означать генерация исключения в конструкторе объекта?

Будем отвечать на вопросы последовательно.

Вопрос: Когда начинается время жизни объекта?

Ответ: Когда его конструктор успешно завершает свою работу и осуществляется обычный выход из него (т.е. когда управление достигает конца тела конструктора или завершается до этого посредством инструкции `return`).

Вопрос: Когда завершается время жизни объекта?

Ответ: Когда начинается выполнение деструктора (т.е. когда управление достигает начала тела деструктора).

Вопрос: Каково состояние объекта по окончании его времени жизни?

Ответ: Заметим в скобках: все гуру в области программирования злоупотребляют антропоморфизмами, говоря о коде, объектах и других сущностях как об одушевленных существах.

```
// Пример 3
//
{
    Parrot& perch = Parrot();
    // ...
}
// <-- Приведенный далее монолог
//     относится к этому месту.
```

Его нет! Он ушел! Этого `Parrot` больше нет! Он прекратил свое существование! Он испустил последний вздох и ушел на встречу со своим творцом! Он преставился! Сыграл в ящик! Приказал долго жить! Перекинулся! Ноги протянул! Дуба дал! Упокоился в мире! (Причем даже раньше — перед концом блока.) Это — бывший `Parrot`!

Д-р. М. Питон (M. Python)[19]

Кроме шуток, это очень важный момент, что состояние объекта до начала его времени жизни совершенно то же, что и по его окончании: объект не существует. Точка. Это приводит нас к следующему ключевому вопросу.

Вопрос: Что означает генерация исключения конструктором?

Ответ: Это означает, что при работе конструктора происходит сбой, что объект никогда не существовал и что его время жизни никогда не начиналось. Единственный способ сообщить о сбое при работе конструктора — т.е. о невозможности корректно создать объект данного типа — состоит в генерации исключения. (Ранее использовавшееся соглашение, гласящее “при возникновении неприятностей установите соответствующий флаг статуса и позвольте вызывающей функции проверить его посредством функции `IsOK()`”, более недействительно.)

Кстати говоря, именно поэтому при сбое конструктора никогда не вызывается деструктор: ему просто нечего уничтожать. Объект, который никогда не жил, не может и умереть. Заметим, что этот факт делает фразу “объект, конструктор которого генерирует исключение” некорректной. Это не объект, и даже меньше, чем бывший объект — это просто не объект.

Мы можем подвести итог обсуждения конструкторов C++ следующим образом.

[19] Мои извинения Монти Питону (Monty Python).

1. Либо выход из конструктора происходит естественным путем (по достижении его конца, либо посредством инструкции `return`), и объект начинает свое существование.
2. Либо выход из конструктора осуществляется посредством генерации исключения, и соответствующий объект не только не существует, но и никогда не существовал в качестве объекта.

Третьего не дано. Вооруженные этой информацией, мы теперь можем приступить к ответу на вопрос о поглощении исключений.

Задача 2.14. Исключения в конструкторах. Часть 2 ***Сложность: 7***

Эта задача детально анализирует сбои конструктора, показывает, почему правила C++ должны быть именно такими, и демонстрирует следствия спецификаций исключений конструктора.

1. Если в примере 1 из задачи 2.13 конструктор `A` или `B` генерирует исключение, может ли конструктор `C` поглотить его и в результате не выпустить никаких исключений вообще? Обоснуйте ваш ответ, подтвердив его примерами.
2. Каким минимальным требованиям должны удовлетворять `A` и `B` для того, чтобы мы могли безопасно указать пустую спецификацию исключений конструктора `C`?

Решение

Нельзя задерживать не перехваченные исключения

1. **Если в примере 1 из задачи 2.13 конструктор `A` или `B` генерирует исключение, может ли конструктор `C` поглотить его и в результате не выпустить никаких исключений вообще?**

Если не рассматривать правила времени жизни объектов, мы можем попытаться сделать что-то наподобие следующего.

```
// Пример 1a: поглощение исключения?
//
C::C()
try
    : A ( /* ... */ ) // Необязательный список инициализации
    , b_( /* ... */ )
{
}
catch( ... )
{
    // ?
}
```

Каким образом может осуществляться выход из обработчика try-блока конструктора[20]?

- Обработчик исключения не может использовать инструкцию `return;` — это запрещено правилами языка.
- Если обработчик воспользуется инструкцией `throw;`, то это будет означать, что он просто передает вызывающей функции исключения, изначально сгенерированные в конструкторах `A::A()` и `B::B()`.

[20] Здесь и далее используется сокращенный перевод термина “constructor (destructor) function try block” как “try-блок конструктора (деструктора)”. — *Прим. перев.*

- Если обработчик генерирует некоторое другое исключение, то это исключение заменит сгенерированное конструктором базового подобъекта или подобъекта-члена.
- Если выход из обработчика не осуществляется путем генерации исключения (либо нового, либо передачи исходного исключения вызывающей функции посредством инструкции `throw;`) и управление достигает конца catch-блока в конструкторе или деструкторе, то исходное исключение автоматически передается вызывающей функции, как если бы была использована инструкция `throw;`. Это не очевидно, но явно указано в стандарте C++.

Подумайте о том, что это означает: try-блок конструктора или деструктора *обязан* завершиться генерацией исключения. Другого пути нет. До тех пор пока вы не пытаетесь нарушить спецификации исключений, языку нет дела до того, какое исключение выйдет за пределы обработчика — исходное или преобразованное, — но это исключение должно быть! Одним словом,

> *в C++ при генерации исключения конструктором любого базового подобъекта или подобъекта-члена конструктор всего объекта также* должен *сгенерировать исключение.*

Невозможно никакое восстановление после генерации исключения конструктором базового подобъекта или подобъекта-члена. Нельзя даже перевести свой собственный объект в состояние "конструирование неуспешно", распознаваемое компилятором. Объект *не* создан и никогда не будет создан, какие бы усилия вы не прилагали. Все деструкторы базовых подобъектов и подобъектов-членов будут вызваны в соответствии с правилами языка автоматически.

Но что, если ваш класс имеет состояние "конструирование частично неуспешное" — т.е. класс имеет некоторые "необязательные" члены данных, которые не являются совершенно необходимыми, и объект может кое-как обойтись и без них, возможно, со сниженной функциональностью? В этом случае стоит использовать идиому скрытой реализации (Pimpl), описанную в задачах 4.2—4.5, для хранения "необязательной" части (обратитесь также к задачам 5.1—5.4 о злоупотреблениях наследованием). Кстати говоря, эта идея "необязательной части объекта" является еще одной причиной использовать делегирование вместо наследования, когда это только возможно. Базовые подобъекты никогда не могут быть сделаны необязательными, поскольку они не могут быть помещены в скрытую реализацию.

Как бы я не любил или ненавидел исключения, я всегда был согласен с тем, что исключения предоставляют самый корректный способ сообщить о сбоях конструктора, поскольку конструктор не в состоянии сообщить об ошибках с помощью возвращаемого значения (так же, как и большинство операторов). Я считаю, что метод сообщения об ошибках путем установки соответствующего флага с последующей его проверкой устарел, опасен, утомителен и ни в чем не превосходит использование исключений. То же самое относится и к двойнику этого метода — двухфазному конструированию. И я не единственный рассматривающий эти методы как одиозные. Не кто иной, как Страуструп [Stroustrup00] назвал этот стиль ни чем иным, как реликтом C++ до появления в нем исключений.

Мораль сей басни такова...

Кстати, рассмотренный материал означает, что единственное (повторяю — единственное) возможное применение try-блока конструктора состоит в том, чтобы транслировать исключение, сгенерированное конструктором базового подобъекта или подобъекта-члена. Это мораль №1. Мораль №2 гласит, что try-блоки деструктора совершенно бе...

“Минутку! — слышу я чей-то голос из зала. — Я не согласен с моралью №1. Я думаю, что у try-блока конструктора есть и другое применение, — он может использоваться для освобождения ресурсов, выделенных в списке инициализации или в теле конструктора!”

Извините, нет. Вспомните, что когда вы входите в обработчик try-блока конструктора, все локальные переменные в теле конструктора находятся вне зоны видимости, а кроме того, больше не существуют ни базовые подобъекты, ни подобъекты-члены. Точка. Вы не можете даже обратиться к их именам. Часть объектов могла никогда не быть созданной, остальные объекты, которые были созданы, к этому моменту уже уничтожены. Так что никакие захваченные базовыми подобъектами и подобъектами-членами ресурсы вы освободить не сможете (и, кстати, для чего тогда существуют их деструкторы?).

Отступление: почему C++ поступает таким образом?

Для того чтобы понять, почему C++ поступает именно таким образом и почему это правильно, давайте на минутку уберем все ограничения и представим, что C++ позволяет использовать имена членов в обработчике try-блока конструктора. Затем представим (и попробуем разрешить) следующую ситуацию: должен ли обработчик удалять `t_` или `z_` в приведенном ниже коде? (Не обращайте внимания на то, что в действительности вы даже не сможете обратиться к `t_` и `z_`.)

```
// Пример 1б. Очень некорректный класс
//
class X : Y
{
    T* t_;
    Z* z_;
public:
    X()
    try
        : Y(1)
        , t_( new T( static_cast<Y*>(this) ))
        , z_( new Z( static_cast<Y*>(this), t_ ))
    {
        /* ... */
    }
    catch(...) // Исключение сгенерировано одним из
               // конструкторов: Y::Y(), T::T(),
               // Z::Z() или X::X()
    {
        // Следует ли удалять здесь t_ или z_?
        // (Примечание: этот код в C++ некорректен!)
    }
};
```

Во-первых, мы не имеем возможности даже узнать, была ли выделена память для `t_` или `z_`. Таким образом, их удаление может оказаться небезопасным.

Во-вторых, даже если бы мы знали, что одно из выделений выполнено, мы, вероятно, не могли бы уничтожить `*t_` или `*z_`, поскольку они обращаются к `Y` (а возможно, и к `T`), которые больше не существуют, а они могут при уничтожении обратиться к `Y` (а возможно, и к `T`). Между прочим, это говорит о том, что не только мы не можем уничтожить `*t_` и `*z_`, но и никто не сможет этого сделать.

Мне приходилось встречать людей, которые разрабатывали код в духе приведенного, не представляя, что они создают объекты, которые в случае неприятностей не могут быть уничтожены! Хорошо, что имеется очень простой путь избежать этих неприятностей. Достаточно вместо членов типа `T*` использовать `auto_ptr` или аналогичные объекты-менеджеры (об опасностях использования членов `auto_ptr` и о том, как их избежать, говорится в задачах 6.5 и 6.6).

И наконец, если `Y::~Y()` может генерировать исключения, то вообще невозможно надежно создать объект `X`! Если все изложенное выше вас не отрезвило, то уж это соображение должно привести вас в чувства. Если `Y::~Y()` может генерировать исключения, то даже написание `X x;` чревато опасностями. Это подтверждает мысль о том, что деструкторы ни при каких условиях не должны генерировать исключения, и написание такого деструктора следует рассматривать как ошибку.

Но, пожалуй, достаточно об этом. Надеюсь, что это отступление помогло вам понять, почему C++ поступает именно таким образом и почему это правильно. В C++ вы попросту не сможете обратиться к `t_` или `z_` в обработчике исключения конструктора. Обычно я воздерживаюсь от чрезмерного цитирования стандарта, но иногда я все же это делаю. Итак, вот цитата из стандарта [C++98, 15.3.10]: "*Обращение к любому нестатическому члену или базовому классу объекта в обработчике try-блока конструктора данного объекта приводит к неопределенному поведению*".

Мораль о try-блоках функций

Таким образом, резюмировать ситуацию можно следующим образом.

Мораль №1. Обработчик try-блока конструктора пригоден только для трансляции исключения, сгенерированного в конструкторе базового подобъекта или подобъекта-члена (а также, возможно, для соответствующего занесения информации в журнальный файл или каких-то других побочных действий в ответ на генерацию исключения). Ни в каких других целях обработчик служить не может.

Мораль №2. Try-блоки деструкторов практического применения не имеют, поскольку деструкторы не должны генерировать исключения.[21] Следовательно, не должно быть ничего, что могло бы быть перехвачено try-блоком деструктора и не перехвачено обычным try-блоком в его теле. Даже если из-за некачественного кодирования нечто подобное и наблюдается (а именно подобъект-член, деструктор которого может генерировать исключения), то все равно использование try-блока деструктора по сути бесполезно, поскольку оно не в состоянии подавить это исключение. Лучшее, что можно сделать в таком блоке, — это занести сообщение в журнальный файл или пожаловаться на горе-программистов каким-то подобным способом.

Мораль №3. Все остальные try-блоки функций практического применения не имеют. Try-блок обычной функции не может перехватить что-то, что не может перехватить обычный try-блок в теле функции.

Мораль о безопасном кодировании

Мораль №4. Неуправляемое[22] выделение ресурсов всегда выполняйте в теле конструктора и никогда — в списках инициализации. Другими словами, либо используйте прин-

---

[21] Try-блок деструктора неприменим даже для журнальной записи или других аналогичных действий, поскольку исключения не должны генерироваться ни деструкторами базовых классов, ни деструкторами членов класса. Следовательно, все исключения, которые перехватывает try-блок деструктора, могут быть с тем же успехом перехвачены обычным try-блоком внутри тела деструктора.

[22] Под *неуправляемым* (unmanaged) выделением ресурсов подразумевается выделение, требующее явного освобождения, в отличие от управляемого (managed) выделения, при котором выделение и освобождение выполняются автоматически конструктором и деструктором соответствующего класса-менеджера, в соответствии с принципом "выделение ресурса есть инициализация". — *Прим. перев.*

цип “выделение ресурса есть инициализация” (тем самым полностью избегая неуправляемых ресурсов), либо выполняйте выделение ресурса в теле конструктора.[23]

Пусть в примере 1*б* тип `T` представляет собой тип `char`, а `t_` — обычный старый массив `char*`, который выделяется посредством оператора `new[]` в списке инициализации. В таком случае может не оказаться способа освободить выделенную память оператором `delete[]` — ни в обработчике исключений, ни где-либо еще. Исправить ситуацию можно, либо использовав для динамически выделяемой памяти “обертку” (например, изменив `char*` на `string`), либо выделяя память в теле конструктора, где она может быть корректно освобождена, например, при использовании локального try-блока.

Мораль №5. Ресурсы, выделенные неуправляемым способом, следует освобождать в обработчике локального try-блока в теле конструктора или деструктора, но никогда — в обработчике try-блока конструктора или деструктора.

Мораль №6. Если конструктор имеет спецификацию исключений, то она должна объединять все возможные исключения, которые могут быть сгенерированы базовыми подобъектами и подобъектами-членами.

Мораль №7. Используйте для хранения “необязательных частей” внутреннего представления класса идиому указателя на реализацию. Если конструктор объекта-члена может генерировать исключения, но вы можете продолжать работу и без указанного члена, то используйте для его хранения указатель (и нулевое значение указателя как указание, что конструирование этого объекта прошло неуспешно). Используйте для группирования таких “необязательных” членов (чтобы их можно было выделять однократно) идиому скрытой реализации.

И наконец, последняя мораль (которая частично перекрывается прочими, но стоит того, чтобы выделить ее отдельно).

Мораль №8. Для управления ресурсами используйте принцип “выделение ресурса есть инициализация”. Это спасет вас от такой головной боли, которую вы не можете и представить. Ряд причин для такой боли нами уже был разобран в этой книге.

Другой дороги нет

Вернемся к последней части первого вопроса.

Обоснуйте ваш ответ, подтвердив его примерами.

Способ, которым работает язык, совершенно корректен и легко обосновывается — стоит только вспомнить о том, что такое время жизни объекта в C++.

Исключение конструктора должно быть передано вызывающей функции. Другого пути для сообщения о сбое в работе конструктора нет. На ум приходят два случая. Первый случай.

```
// Пример 1в. Автоматический объект
//
{
    X x;
    g( x ); // Выполняем некоторые действия
}
```

Если конструктор `X` завершается неуспешно, — неважно, из-за генерации исключения в теле конструктора `X`, или в конструкторе базового подобъекта, или подобъекта-члена, — выполнение не должно продолжаться в области видимости данного блока кода. В конце концов, объекта `x` не существует! Единственный способ предотвратить продолжение выполнения — генерация исключения. Таким образом, сбой конструктора автоматического объекта должен приводить к генерации исключения некоторого типа, вызванного сбоем конструирования базового подобъекта или подобъекта-члена, или транслированного исключения, выдаваемого try-блоком конструктора `X`.

[23] См. раздел 16.5 в [Stroustrup94] и раздел 14.4 в [Stroustrup00].

Второй случай.

```
// Пример 1г. Массив объектов
//
{
    X ax[10];
    // ...
}
```

Предположим, что произошел сбой конструктора пятого объекта `X` — опять-таки, неважно, из-за генерации исключения в теле конструктора `X`, или в конструкторе базового подобъекта, или подобъекта-члена. В любом случае выполнение не должно продолжаться в области видимости данного блока кода. Если вы все же попытаетесь продолжить работу, то столкнетесь с “дырявым массивом” — массивом, часть которого в действительности не существует.

Конструкторы с защитой от сбоев

2. Каким минимальным требованиям должны удовлетворять `A` и `B` для того, чтобы мы могли безопасно указать пустую спецификацию исключений конструктора `C`?

Рассмотрим вопрос: можно ли использовать пустую спецификацию исключений для конструктора класса (подобного классу `C` из примера 1 задачи 2.13), если какой-то из базовых конструкторов или конструкторов членов может генерировать исключения? Ответ — нет. Вы, конечно, можете указать пустую спецификацию исключений, по сути сказав, что ваш конструктор не генерирует исключений, но это будет ложь, поскольку обеспечить отсутствие исключений вы не в состоянии. Для того чтобы обеспечить гарантию отсутствия исключений, вы должны быть в состоянии поглотить все возможные исключения кода нижнего уровня и избежать их случайной передачи вызывающей функции. Если вы действительно хотите написать конструктор, который не генерирует исключений, то вы в состоянии перехватить исключения, генерируемые конструкторами подобъектов-членов (например, работая с ними через указатели), но вы никак не можете избежать исключений, генерируемых конструкторами базовых подобъектов (вот и еще одна причина избегать излишнего наследования).

Для того чтобы конструктор мог иметь пустую спецификацию исключений, все базовые подобъекты и подобъекты-члены не должны генерировать исключений, независимо от того, имеют ли они пустую спецификацию исключений или нет. Пустая спецификация исключений говорит всем, что создание объекта не может завершиться неуспешно. Если же в силу каких-то причин это может произойти, то использование пустой спецификации исключений недопустимо.

Что же произойдет, если вы укажете пустую спецификацию исключений у конструктора, а конструктор базового подобъекта или подобъекта-члена сгенерирует исключение? Краткий ответ: будет вызвана функция `terminate()`, и никакие try-блоки при этом не помогут. Ответ несколько более полный: будет вызвана функция `unexpected()`, которая может либо сгенерировать исключение, допустимое спецификацией исключений (выход не для нашего случая — спецификация исключений пуста), либо вызвать функцию `terminate()`, которая, в свою очередь, немедленно прекратит выполнение программы.[24] Используя автомобильную аналогию, — это визг тормозов, за которым следует хруст удара...

[24] Вы можете использовать функции `set_unexpected()` и `set_terminate()` для того, чтобы указать собственные обработчики этой ситуации. Эти обработчики дают вам возможность вывести запись в журнальный файл или освободить какой-то ресурс, но завершаются они так же, как и только что рассмотренные исходные обработчики.

Резюме

Время жизни объекта в C++ начинается только после удачного завершения его конструктора. Следовательно, генерация исключений конструктором всегда означает (и является единственным средством сообщить об этом), что произошел сбой в работе конструктора. Не существует способа восстановления после сбоя конструктора базового подобъекта или подобъекта-члена, так что если неуспешно создание любого базового подобъекта или подобъекта-члена, то и создание объекта в целом также должно оказаться неуспешным.

Избегайте использования try-блоков функций, не в силу какой-то их "злокозненности", а просто потому, что они имеют слишком мало преимуществ перед обычными try-блоками (если имеют их вообще). Такой выбор в пользу обычных try-блоков следует из принципа выбора наиболее простого решения при одинаковой эффективности и создания ясного и понятного кода. Try-блок конструктора может быть полезен только для преобразования исключений, сгенерированных базовыми конструкторами или конструкторами членов. Все прочие функции try-блоков редко (если вообще) применимы.

И наконец, как постоянно указывалось в задачах этой главы, используйте для управления ресурсами собственные объекты и принцип "выделение ресурса есть инициализация", и это, как правило, позволит вам избежать необходимости использовать в вашем коде `try` и `catch` и забыть о try-блоках функций.

Задача 2.15. Неперехваченные исключения ***Сложность: 6***

Что собой представляет стандартная функция `uncaught_exception()` и когда она должна использоваться? Ответ на этот вопрос для многих обычно оказывается неожиданным.

1. Что делает функция `std::uncaught_exception()`?
2. Рассмотрим следующий код.

```
T::~T()
{
    if ( !std::uncaught_exception() )
    {
        // код, который может генерировать исключения
    }
    else
    {
        // код, не генерирующий исключения
    }
}
```

   Насколько корректна данная методика? Приведите аргументы "за" и "против".
3. Имеется ли еще какое-нибудь корректное применение функции `uncaught_exception()`? Рассмотрите этот вопрос и изложите свои выводы.

Решение

Резюмируем uncaught_exception()

1. **Что делает функция `std::uncaught_exception()` ?**

Стандартная функция `uncaught_exception()` обеспечивает способ узнать, не является ли в настоящее время активным какое-то исключение. Важно отметить, что это *не* одно и то же, что и знание, является ли генерация исключения безопасной.

Цитируем стандарт (15.5.3/1).

> *Функция `bool uncaught_exception()` возвращает значение `true` после завершения вычисления генерируемого объекта и до полной инициализации объявления исключения в соответствующем обработчике (сюда включается и свертка стека). При повторной генерации исключения `uncaught_exception()` возвращает `true` с момента повторной генерации исключения и до его перехвата.*

Как обычно, это описание слишком компактно, чтобы быть полезным.

Проблема деструкторов, генерирующих исключения

Если деструктор генерирует исключения, могут произойти Ужасные Вещи. В частности, рассмотрим следующий код.

```
// Пример 1. Постановка задачи
//
class X {
public:
    ~X() { throw 1; }
};

void f() {
    X x;
    throw 2;
} // Вызывается X::~X (генерирует исключение),
  // после чего вызывается terminate()
```

Если деструктор генерирует исключение в то время, когда активно другое исключение (т.е. в процессе свертки стека), программа прекращает работу. Обычно это не самый лучший выход.

Перечитайте в свете сказанного еще раз раздел “Деструкторы, генерирующие исключения, и почему они неприемлемы” из задачи 2.9.

Неверное решение

“Ага, — могут сказать некоторые, — вот и решение: используем `uncaught_exception()` для того, чтобы определить, когда можно генерировать исключения!” И приведут код, аналогичный коду из условия задачи.

2. Рассмотрим следующий код.

```
T::~T()
{
    if ( !std::uncaught_exception() )
    {
        // код, который может генерировать исключения
    }
    else
    {
        // код, не генерирующий исключения
    }
}
```

Насколько корректна данная методика? Приведите аргументы “за” и “против”.

Вкратце: нет, эта методика некорректна, несмотря на все ее старания разрешить описанную проблему. Против этой методики имеются существенные технические

возражения, но меня сейчас в большей степени интересуют, скажем так, философские соображения.

Идея, стоящая за использованной в примере 2 идиомой, состоит в том, что мы используем генерацию исключений тогда, когда это безопасно. Однако здесь имеются две ошибки. Во-первых, этот код не справляется с поставленной задачей. Во-вторых (что, на мой взгляд, существенно важнее), перефразируя Джеффа Питерса, можно сказать, что эта идея и есть свое самое слабое место. Рассмотрим обе эти ошибки по отдельности.

Почему это решение ошибочно

Первая проблема заключается в том, что в некоторых ситуациях предложенное в примере 2 решение не будет работать так, как предполагается. Дело в том, что возможен выбор пути выполнения без генерации исключений в ситуации, когда такая генерация вполне безопасна.

```
// Пример 2а. Почему неверное решение неверно
//
U::~U()
{
    try
    {
        T t;
        // Выполнение некоторых действий
    }
    catch( ... )
    {
        // Освобождение ресурсов
    }
}
```

Если объект `U` уничтожается из-за свертки стека в процессе распространения исключения, `T::~T()` будет ошибочно использовать путь, который не генерирует исключений, несмотря на то, что в данной ситуации генерация исключений совершенно безопасна: `T::~T()` не знает, что он защищен отдельным блоком `catch(...)`.

Заметим, что по сути рассмотренный пример ничем не отличается от следующего.

```
// Пример 3. Еще одно неверное решение
//
Transaction::~Transaction()
{
    if (uncaught_exception())
    {
        RollBack();
    }
    else
    {
        // ...
    }
}
```

Здесь также осуществляется неверный выбор при осуществлении транзакции в деструкторе в процессе свертки стека.

```
// Пример 3а. Использование неверного
//            решения из примера 3
U::~U() {
    try {
        Transaction t(/* ... */);
        // Выполнение некоторых действий
    }
    catch( ... )
```

```
    {
        // Освобождение ресурсов
    }
}
```

Итак, мы убедились, что предложенное в примере 2 решение не работает так, как ожидалось. Этого достаточно, чтобы отказаться от данного решения, но это — не главная причина.

Почему это решение неэтично

Вторая и существенно более фундаментальная проблема лежит не в технической, а в этической плоскости. Наличие двух режимов сообщения об ошибках в деструкторе `T::~T()` говорит о непродуманности дизайна. Сообщать об одной ошибке двумя разными путями — признак дурного вкуса. Это излишнее усложнение интерфейса и семантики, усложняющее к тому же задачу использующего класс программиста: в вызывающей функции он должен предусмотреть два варианта реакции на ошибку, и это при том, что слишком многие программисты вообще не уделяют должного внимания обработке ошибок.

Иногда, когда я веду машину, мне попадаются водители, которые ухитряются ехать передо мной сразу по двум полосам дороги. В таких ситуациях мне так и хочется приоткрыть окно и крикнуть такому водителю: “Эй, дружище! Выбери себе полосу — все равно какую, но только одну, и не мешай ехать порядочным водителям!” Пока этот горе-водитель едет по разделительной линии, мне надо постоянно быть готовым к двум вариантам его движения. Это не только раздражает, но и затормаживает движение на дороге.

Иногда, когда я пишу программу, мне приходится использовать классы или функции других программистов, имеющие шизоидные интерфейсы, в частности, пытающиеся сообщать об одной и той же ошибке разными способами, вместо того чтобы выбрать и последовательно использовать одну семантику. В таких ситуациях мне так и хочется крикнуть такому программисту: “Эй, дружище! Выбери один способ — все равно какой, но только один, и не мешай работать порядочным людям!”

Верное решение

Правильное решение проблемы из примера 1 выглядит гораздо проще.

```
// Пример 4. Верное решение
//
T::~T() /* throw() */
{
    // код, не генерирующий исключений
}
```

Пример 4 демонстрирует, как надо поступать без всяких долгих пространных рассуждений.

Заметим, что пустая спецификация исключений — не более чем комментарий. Я всегда следую этому стилю, в частности, потому что использование спецификаций исключений, как правило, приносит больше вреда, чем пользы. Следует ли явно указывать спецификацию исключений или нет — дело вашего вкуса. Главное, чтобы эта функция не выпускала никаких исключений вовне. (Возможность генерации исключений деструкторами членов `T` рассматривалась в задаче 2.14.)

При необходимости класс `T` может содержать “деструктивную” функцию, например `T::Close()`, которая может генерировать исключения и которая выполняет все действия по уничтожению объекта и освобождению захваченных ресурсов. При этом

такая функция может генерировать исключения для указания наличия серьезных ошибок, а деструктор `T::~T()` реализуется как вызов `T::Close()` в блоке `try/catch`.

```
// Пример 5. Еще одно корректное решение
//
T::Close()
{
    // Код, который может генерировать исключения
}

T::~T() /* throw() */
{
    try
    {
        Close();
    }
    catch( ... ) { }
}
```

Такой подход четко следует принципу “одна функция, одно задание”. Проблема в исходном коде заключалась в том, что одна и та же функция отвечала за выполнение двух задач: за уничтожение объекта и за окончательное освобождение ресурсов и сообщения об ошибках.

Обратитесь также к задачам 2.13 и 2.14, чтобы понять, почему мы используем try-блок внутри тела деструктора, а не try-блок функции деструктора.

Рекомендация

Никогда не позволяйте исключениям покинуть деструктор или переопределенные операторы `delete()` и `delete[]()`. Разрабатывайте каждый деструктор и функции удаления объектов так, как если бы они имели спецификацию исключений `throw()` (использовать ли эту спецификацию в коде или нет — дело вашего вкуса).

Рекомендация

Если деструктор вызывает функции, которые могут генерировать исключения, всегда осуществляйте эти вызовы в блоках `try/catch` для предотвращения выхода исключения за пределы деструктора.

3. **Имеется ли еще какое-нибудь корректное применение функции `uncaught_exception()`? Рассмотрите этот вопрос и изложите свои выводы.**

К сожалению, я не знаю ни одного полезного и безопасного применения функции `uncaught_exception()`. Мой совет: не используйте ее.

Задача 2.16. Проблема неуправляемых указателей. Часть 1 *Сложность: 6*

Читатели уже знают, что проблему безопасности исключений можно назвать какой угодно, но только не тривиальной. Эта задача освещает сравнительно недавно открытую проблему безопасности исключений и показывает, каким образом избежать ее при разработке своего кода.

1. Что вы можете сказать о порядке вычислений функций `f`, `g` и `h` и выражений `expr1` и `expr2` в каждом из приведенных фрагментов? Полагаем, что выражения `expr1` и `expr2` вызовов функций не содержат.

```
// Пример 1а
//
f( expr1, expr2 );

// Пример 1б
//
f( g( expr1 ), h( expr2 ) );
```

2. Роясь в архивах, вы нашли такой фрагмент кода.

```
// Пример 2
//

// В заголовочном файле:
void f( T1*, T2* );

// В файле реализации
f( new T1, new T2 );
```

Имеются ли в этом коде потенциальные проблемы, связанные с безопасностью исключений? Поясните свой ответ.

Решение

Порядок и беспорядок вычислений

1. **Что вы можете сказать о порядке вычислений функций `f`, `g` и `h` и выражений `expr1` и `expr2` в каждом из приведенных фрагментов? Полагаем, что выражения `expr1` и `expr2` вызовов функций не содержат.**

```
// Пример 1а
//
f( expr1, expr2 );

// Пример 1б
//
f( g( expr1 ), h( expr2 ) );
```

Игнорируя отсутствующие в стандарте C++ потоки выполнения (threads), при ответе на первый вопрос следует опираться на следующие базовые правила.

1. До того как будет вызвана функция, все ее аргументы должны быть полностью вычислены. Сюда включается завершение выполнения всех побочных действий выражений, используемых в качестве аргументов функций.
2. После того как началось выполнение функции, никакие выражения из вызывающей функции не начинают и не продолжают вычисляться до тех пор, пока не будет полностью завершена вызванная функция. Выполнения функций никогда не чередуются друг с другом.
3. Выражения, используемые в качестве аргументов функций, вообще говоря, могут вычисляться в любом порядке, включая чередование, если только это не запрещено какими-либо другими правилами.

Исходя из данных правил, рассмотрим первый из наших фрагментов.

```
// Пример 1а
//
f( expr1, expr2 );
```

В этом примере все, что мы можем сказать, — это то, что до вызова функции `f()` должны быть вычислены как `expr1`, так и `expr2`. Вот именно. Компилятор может вычислить выражение `expr1` до, после, или даже *чередуя* это вычисление с вычислением `expr2`. Имеется немалое количество людей, для которых это оказывается большим сюрпризом, так что в соответствующих группах новостей вопросы на эту тему отнюдь не редкость. Но такой ответ — всего лишь прямое следствие правил C и C++.

```
// Пример 1б
//
f( g( expr1 ), h( expr2 ) );
```

В примере 1*б* функции и выражения могут вычисляться в любом порядке, который удовлетворяет следующим правилам.

- `expr1` должно быть вычислено до вызова `g()`.
- `expr2` должно быть вычислено до вызова `h()`.
- И `g()`, и `h()` должны быть вычислены до вызова `f()`.
- Вычисление выражений `expr1` и `expr2` может чередоваться друг с другом, но не с вызовом любой функции. Например, между началом выполнения `g()` и ее завершением не могут осуществиться ни никакая часть вычисления выражения `expr2`, ни вызов `h()`.

Все. Это означает, например, что возможна последовательность действий, когда начинается вычисление `expr1`, которое прерывается вызовом `h()`; после завершения этой функции вычисление `expr1` благополучно продолжается (то же может быть и с `expr2`, и `g()`).

Вызов функций и безопасность

2. Роясь в архивах, вы нашли такой фрагмент кода.

```
// Пример 2
//

// В заголовочном файле:
void f( T1*, T2* );

// В файле реализации
f( new T1, new T2 );
```

Имеются ли в этом коде потенциальные проблемы, связанные с безопасностью исключений? Поясните свой ответ.

Да, здесь имеется несколько потенциальных проблем, связанных с безопасностью исключений.

Вспомним, что делает выражение типа `new T1`.

- Выделяет память.
- Конструирует новый объект в выделенной памяти.
- В случае генерации исключения освобождает выделенную память.

Так что по сути каждое выражение `new` является серией из двух вызовов функций: вызова `operator new()` (либо глобального оператора, либо предоставляемого типом создаваемого объекта) с последующим вызовом конструктора.

Рассмотрим работу фрагмента из примера 2, если компилятор генерирует код следующей последовательности действий.

1. Выделение памяти для `T1`.

2. Конструирование `T1`.
3. Выделение памяти для `T2`.
4. Конструирование `T2`.
5. Вызов `f()`.

Проблема заключается в следующем: если третий или четвертый шаг выполняется неуспешно и генерирует исключение, то происходит утечка памяти, поскольку стандарт C++ не требует уничтожения объекта `T1` и освобождения выделенной ему памяти. Совершенно очевидно, что это одна из Ужасных Вещей.

Другой вариант последовательности действий может выглядеть таким образом.

1. Выделение памяти для `T1`.
2. Выделение памяти для `T2`.
3. Конструирование `T1`.
4. Конструирование `T2`.
5. Вызов `f()`.

При использовании такой последовательности количество проблем увеличивается — теперь их две.

а. Если шаг 3 оказывается неуспешен и генерирует исключение, то память, выделенная для объекта `T1`, автоматически освобождается. Однако стандарт не требует освобождения памяти, выделенной для объекта `T2`. Происходит утечка памяти.

б. Если неуспешным и генерирующим исключение оказывается шаг 4, объект `T1` к этому времени оказывается полностью построенным. Однако стандарт не требует его уничтожения и освобождения выделенной памяти. Происходит утечка объекта `T1`.

“А почему такая дыра вообще существует? — можете поинтересоваться вы. — Почему бы стандарту не потребовать от компилятора правильно поступать в этой ситуации и выполнять освобождение ресурсов?”

Следуя духу C в его стремлении к эффективности, стандарт C++ предоставляет компилятору определенную свободу в порядке выполнения вычислений, поскольку это позволяет компилятору выполнять оптимизацию, которая иначе была бы невозможна. Поэтому правила вычисления выражений не являются безопасными с точки зрения исключений, так что если вы хотите разработать безопасный код, вы должны знать о существовании описанных проблем и избегать их. К счастью, сделать это достаточно просто. Возможно, нам поможет в этом управляемый указатель типа `auto_ptr`? Вы найдете ответ на этот вопрос в задаче 2.17.

Задача 2.17. Проблема неуправляемых указателей. Часть 2 *Сложность: 8*

Решает ли проблему из задачи 2.16 применение `auto_ptr`?

1. Продолжая рыться в архивах, вы находите фрагмент кода, очень похожий на найденный в задаче 2.16, но с определенными изменениями.

```
// Пример 1
//

// В заголовочном файле:
void f( auto_ptr<T1>, auto_ptr<T2> );
```

```
// В файле реализации
f( auto_ptr<T1>( new T1 ), auto_ptr<T2>( new T2 ) );
```

В чем состоит данное улучшение кода из задачи 2.16, если таковое имеется? Остаются ли в данном варианте кода проблемы с безопасностью исключений? Поясните свой ответ.

2. Покажите, каким образом следует написать вспомогательное средство `auto_ptr_new`, которое решит проблемы безопасности из первой части задачи и которое можно использовать следующим образом.

```
// Пример 2
//

// В заголовочном файле:
void f( auto_ptr<T1>, auto_ptr<T2> );

// В файле реализации
f( auto_ptr_new<T1>(), auto_ptr_new<T2>() );
```

Решение

1. **Продолжая рыться в архивах, вы находите фрагмент кода, очень похожий на найденный в задаче 2.16, но с определенными изменениями.**

```
// Пример 1
//

// В заголовочном файле:
void f( auto_ptr<T1>, auto_ptr<T2> );

// В файле реализации
f( auto_ptr<T1>( new T1 ), auto_ptr<T2>( new T2 ) );
```

В чем состоит данное улучшение кода из задачи 2.16, если таковое имеется? Остаются ли в данном варианте кода проблемы с безопасностью исключений? Поясните свой ответ.

Приведенный фрагмент кода пытается решить проблему, указанную в задаче 2.16. Многие программисты считают `auto_ptr` некоторой панацеей в смысле безопасности исключений, своего рода амулетом, который должен отпугнуть все проблемы, связанные с исключениями.

Но это не так. Ничего не изменилось. Пример 1 остается небезопасным, по той же самой причине, что и раньше.

На этот раз проблема состоит в том, что ресурсы безопасны только тогда, когда они действительно создаются под управлением `auto_ptr`, но в данном случае исключение может произойти до того, как будет достигнут конструктор `auto_ptr`. Это связано с порядком, вернее, с беспорядком вычисления аргументов функции. Вот один пример последовательности вычислений, повторяющий ранее рассмотренную последовательность.

1. Выделение памяти для `T1`.
2. Конструирование `T1`.
3. Выделение памяти для `T2`.
4. Конструирование `T2`.
5. Конструирование `auto_ptr<T1>`.
6. Конструирование `auto_ptr<T2>`.
7. Вызов `f()`.

В приведенной последовательности генерация исключений на шаге 3 или 4 приводит к тем же проблемам, что и ранее. Аналогично, в следующей последовательности действий генерация исключений на шаге 3 или 4 будет приводить к тем же проблемам, что описаны в предыдущей задаче.

1. Выделение памяти для `T1`.
2. Выделение памяти для `T2`.
3. Конструирование `T1`.
4. Конструирование `T2`.
5. Конструирование `auto_ptr<T1>`.
6. Конструирование `auto_ptr<T2>`.
7. Вызов `f()`.

К счастью, проблема не в `auto_ptr`, а в его неверном использовании. Все можно легко исправить, причем даже не единственным способом.

Отступление от темы: некорректное решение

Заметим, что попытка

```
// В заголовочном файле
void f( auto_ptr<T1> = auto_ptr<T1>( new T1 ),
        auto_ptr<T2> = auto_ptr<T1>( new T2 ) );

// В файле реализации
f();
```

решением не является. Это связано с тем, что с точки зрения вычисления выражений этот код ничем не отличается от кода из примера 1. Аргументы по умолчанию рассматриваются как создаваемые при вызове функции, несмотря на то, что записаны они где-то в другом месте в объявлении функции.

Ограниченное решение

2. **Покажите, каким образом следует написать вспомогательное средство `auto_ptr_new`, которое решит проблемы безопасности из первой части задачи и которое можно использовать следующим образом.**

```
// Пример 2
//

// В заголовочном файле:
void f( auto_ptr<T1>, auto_ptr<T2> );

// В файле реализации
f( auto_ptr_new<T1>(), auto_ptr_new<T2>() );
```

Простейший способ заключается в создании шаблона функции следующего вида.

```
// Пример 2a. Частичное решение
//
template<typename T>
auto_ptr<T> auto_ptr_new()
{
    return auto_ptr<T>( new T );
}
```

Таким способом решаются проблемы безопасности исключений. Никакая последовательность генерируемого кода не может привести к утечке ресурсов, поскольку теперь все, что мы имеем, — две функции, о которых мы знаем, что одна из них должна быть полностью выполнена до другой. Рассмотрим следующую последовательность вычислений.

1. Вызов `auto_ptr_new<T1>()`.
2. Вызов `auto_ptr_new<T2>()`.

Если генерация исключения происходит на первом шаге, утечек не возникает, поскольку шаблон `auto_ptr_new()` является строго безопасным.

Если же генерация исключения происходит на втором шаге, будет ли временный объект `auto_ptr<T1>`, созданный на первом шаге, гарантированно уничтожен с освобождением всех захваченных ресурсов? Да, будет. Может показаться, что этот способ очень похож на создание объекта путем вызова `new T1` в примере 2 задачи 2.16, в котором не происходит уничтожения временного объекта. На самом деле эти ситуации существенно различны, так как `auto_ptr<T1>` — это истинный временный объект, корректное уничтожение которого требуется стандартом (12.2.3).

> *Временные объекты уничтожаются на последнем шаге вычисления полного выражения, которое (лексически) содержит точку их создания. Это справедливо и в случае, когда вычисление прекращается из-за генерации исключения.*

Однако приведенное в примере 2*а* решение является ограниченным: оно работает только с конструктором по умолчанию, что не позволяет применить его в случае отсутствия у типа такого конструктора или если использовать его нежелательно. Таким образом, нам следует подумать о более общем решении.

Обобщение auto_ptr_new()

Как указал Дэйв Абрахамс (Dave Abrahams), мы можем расширить это решение для поддержки других конструкторов путем обеспечения семейства перегруженных шаблонов функций.

```
// Пример 2б. Улучшенное решение
//
template<typename T>
auto_ptr<T> auto_ptr_new()
{
    return auto_ptr<T>( new T );
}

template<typename T, typename Arg1>
auto_ptr<T> auto_ptr_new( const Arg1& arg1 )
{
    return auto_ptr<T>( new T( arg1 ) );
}

template<typename T, typename Arg1, typename Arg2>
auto_ptr<T> auto_ptr_new( const Arg1& arg1,
                          const Arg2& arg2 )
{
    return auto_ptr<T>( new T( arg1, arg2 ) );
}

// etc.
```

Теперь `auto_ptr_new()` полностью и естественным образом поддерживают конструкторы с различными аргументами.

Лучшее решение

Хотя предложенное решение неплохо работает, нет ли способа обойтись вовсе без всяких вспомогательных функций? Может, можно избежать проблем, если придерживаться некоторого лучшего стандарта кодирования? Да, если, например, следовать следующему правилу. *Никогда не выделяйте ресурсы (например, посредством `new`) в том же выражении, где другой код может генерировать исключение. Это правило должно использоваться даже в том случае, если выделенный посредством `new` ресурс станет управляемым (например, будет передан конструктору `auto_ptr`) в том же выражении.*

В примере 1 удовлетворить правилу можно, если перенести один из временных объектов `auto_ptr` в отдельную именованную переменную.

```
// Пример 1а. Корректное решение
//

// В заголовочном файле
void f( auto_ptr<T1>, auto_ptr<T2> );

// В файле реализации
{
    auto_ptr<T1> t1( new T1 );
    f( t1, auto_ptr<T2>( new T2 ) );
}
```

Приведенный фрагмент удовлетворяет приведенному правилу, поскольку, хотя выделение ресурсов и происходит, их утечки вследствие генерации исключения быть не может: выделение ресурсов и возможная генерация исключений другим кодом отделены друг от друга.

Вот еще одно возможное правило (которое даже проще и легче реализуемо при кодировании и обнаружении).

Выполняйте каждое явное выделение ресурса (например, посредством `new`) в отдельной инструкции, в которой тут же выполняется его передача управляющему объекту (например, `auto_ptr`).

В случае примера 1 применение этого правила приводит к вынесению обоих временных объектов `auto_ptr` в отдельные именованные переменные.

```
// Пример 1б. Корректное решение
//

// В заголовочном файле
void f( auto_ptr<T1>, auto_ptr<T2> );

// В файле реализации
{
    auto_ptr<T1> t1( new T1 );
    auto_ptr<T2> t2( new T2 );
    f( t1, t2 );
}
```

Этот фрагмент удовлетворяет второму правилу и требует гораздо меньше размышлений. Каждый новый ресурс создается в своем собственном выражении и немедленно передается управляющему объекту.

Резюме

Мои рекомендации следующие.

Рекомендация

Выполняйте каждое явное выделение ресурсов (например, посредством `new`) в своей собственной инструкции кода, которая немедленно передает выделенный ресурс управляющему объекту (например, `auto_ptr`).

Эту рекомендацию легко понять и запомнить; она позволяет избежать всех проблем, связанных с безопасностью исключений как в исходной задаче, так и во многих других ситуациях, и является отличным кандидатом на включение в стандарты кодирования вашей организации.

Благодарности

Эта задача была предложена в группе новостей *comp.lang.c++.moderated*. Данное решение основано на материалах, представленных Джеймсом Канце (James Kanze), Стивом Клеймеджем (Steve Clamage) и Дэйвом Абрахамсом (Dave Abrahams).

Задача 2.18. Разработка безопасных классов. Часть 1 *Сложность: 7*

Можно ли сделать любой класс C++ строго безопасным, например, сделать строго безопасным оператор копирующего присваивания? Если да, то как? Какие при этом возникают вопросы и какие из этого следуют выводы? В этой задаче в качестве иллюстративного материала используется пример Каргилла `Widget`.

1. Перечислите три общие уровня безопасности исключений. Кратко опишите каждый из них и поясните его важность.
2. Как выглядит каноническая форма строго безопасного копирующего присваивания?
3. Рассмотрим следующий класс.

```
// Пример 1. Пример Widget Каргилла
//
class Widget
{
public:
    Widget& operator=( const Widget& );
    // ...
private:
    T1 t1_;
    T2 t2_;
};
```

   Предположим, что любые операции `T1` и `T2` могут генерировать исключения. Можно ли, не изменяя структуру класса, написать строго безопасный оператор `Widget::operator(const Widget&)`? Если да, то как? Если нет, то почему? Какие из этого можно сделать выводы?
4. Опишите и продемонстрируйте простое преобразование, работающее с любым классом и делающее копирующее присваивание этого класса (почти) строго безопасным. Где мы встречались с этой технологией (в другом контексте) ранее?

Решение

Данная задача отвечает на следующие вопросы.

- Можно ли сделать любой произвольный класс безопасным, не изменяя его структуру?
- Если нет (т.е. если безопасность влияет на конструкцию класса), то существует ли простое изменение, которое всегда позволяет нам сделать произвольный класс безопасным?

- Влияет ли способ выражения взаимоотношений между классами на безопасность? Конкретно — имеет ли значение, выражаем ли мы это взаимоотношение с использованием наследования или с использованием делегирования?

Каноническая форма безопасности

1. Перечислите три общие уровня безопасности исключений. Кратко опишите каждый из них и поясните его важность.

Канонические гарантии безопасности, определенные Дэйвом Абрахамсом (Dave Abrahams), следующие.

1. *Базовая гарантия*: при наличии исключений утечки ресурсов отсутствуют, а объект остается используемым и уничтожаемым и находится в согласованном, однако не обязательно предсказуемом состоянии. Это наиболее слабый уровень безопасности исключений, он применим тогда, когда вызывающий код в состоянии справиться со сбойными операциями, которые вносят изменения в состояние объекта.
2. *Строгая гарантия*: при наличии исключений состояние программы остается неизменным. Этот уровень приводит к семантике принятия-или-отката (commit-or-rollback), что включает, в частности, действительность всех ссылок и итераторов, указывающих на элементы контейнера.

В дополнение некоторые функции должны удовлетворять еще более строгой гарантии для обеспечения описанных выше уровней безопасности исключений.

3. *Гарантия отсутствия исключений*: функция не генерирует исключений ни при каких обстоятельствах. Оказывается, иногда невозможно реализовать строгую или даже базовую гарантию до тех пор, пока не будет достигнута гарантия того, что определенные функции не будут генерировать исключений (например, деструкторы или функции освобождения памяти). Как мы увидим ниже, очень важным свойством стандартного класса `auto_ptr` является то, что его операции не генерируют исключений.

Освежив наши знания, перейдем ко второму вопросу задачи.

2. Как выглядит каноническая форма строго безопасного копирующего присваивания?

Каноническая форма копирующего присваивания включает два шага. Во-первых, разрабатывается не генерирующая исключений функция `Swap()`, которая обменивает внутренние состояния двух объектов.

```
void T::Swap( T& other ) /* throw() */
{
    // Обменивает внутренние состояния *this и other
}
```

Во-вторых, с использованием идиомы “создать временный объект и обменять” реализуется `operator=()`.

```
T& T::operator=( const T& other )
{
    T temp( other ); // Выполняется вся необходимая работа
    Swap( temp );    // и принятие ее результата посредством
    return *this;    // операций, не генерирующих исключений
}
```

Проанализируем пример Widget

Рассмотренный материал приводит нас к задаче, предложенной Томом Каргиллом (Tom Cargill).

3. Рассмотрим следующий класс.

```
// Пример 1. Пример Widget Каргилла
//
class Widget
{
public:
    Widget& operator=( const Widget& );
    // ...
private:
    T1 t1_;
    T2 t2_;
};
```

Предположим, что любые операции `T1` и `T2` могут генерировать исключения. Можно ли, не изменяя структуру класса, написать строго безопасный оператор `Widget::operator(const Widget&)`? Если да, то как? Если нет, то почему? Какие из этого можно сделать выводы?

Вкратце: вообще говоря, безопасность исключений не может быть достигнута без изменения структуры `Widget`. В примере 1 вообще невозможно разработать безопасный `Widget::operator=()`. Мы не в состоянии даже гарантировать, что объект `Widget` будет находиться в согласованном состоянии в случае генерации исключения, так как не имеется ни способа атомарного изменения обоих членов класса `t1_` и `t2_`, ни способа надежного отката в согласованное состояние (пусть даже отличное от исходного). Пусть наш `Widget::operator=()` пытается изменить `t1_`, затем `t2_` (т.е. сначала изменяется состояние одного члена, затем другого; порядок изменения не имеет значения). При этом возникает две проблемы.

Если попытка изменения `t1_` генерирует исключение, `t1_` должно остаться неизменным, т.е. для того, чтобы `Widget::operator=()` был безопасным, требуется гарантия безопасности класса `T1`, а именно оператора `T1::operator=()` или другого, используемого нами для изменения `t1_`: оператор должен либо успешно завершить работу, либо не изменять состояние объекта вообще. Таким образом, мы пришли к требованию строгой гарантии безопасности оператора `T1::operator=()`. Очевидно, что то же требование относится и к оператору `T2::operator=()`.

Если попытка изменения `t1_` завершилась успешно, а попытка изменения `t2_` привела к генерации исключения, мы попадаем в промежуточное состояние и, вообще говоря, не можем осуществить откат изменений, уже проведенных над `t1_`. Например, что произойдет, если попытка присвоить `t1_` его первоначальное значение также окажется неуспешной и приведет к генерации исключения? В таком случае объект `Widget` не сможет быть восстановлен в согласованном состоянии вообще.

Итак, если структура объекта `Widget` такая, как показанная в примере 1, его `operator=()` невозможно сделать строго безопасным (см. также врезку “Более простой пример”, где представлено подмножество рассматриваемой проблемы).

Наша цель состоит в написании оператора `Widget::operator=()`, который был бы строго безопасен, не делая при этом никаких предположений о безопасности каких бы то ни было операций `T1` или `T2`. Можно ли это сделать или все попытки — пустая трата времени?

Более простой пример

Заметим, что даже в упрощенном случае

```
class Widget2
{
   // ...
private:
    T1 t1_;
};
```

первая проблема остается. Если `T1::operator=()` может генерировать исключения, когда в состояние `t1_` уже внесены изменения, то не существует способа сделать оператор `Widget2::operator=()` строго безопасным. (Если только `T1` не обеспечивает подходящего для этого средства с использованием некоторой другой функции. Но если такая функция существует, то почему это же не сделано и для оператора `T1::operator=()`?)

Использование скрытой реализации

4. Опишите и продемонстрируйте простое преобразование, работающее с любым классом и делающее копирующее присваивание этого класса (почти) строго безопасным. Где мы встречались с этой технологией (в другом контексте) ранее?

Хорошая новость заключается в том, что несмотря на то, что сделать `Widget::operator=()` строго безопасным без изменения структуры класса нельзя, описанная далее простая трансформация всегда позволяет получить *почти* строго безопасное копирующее присваивание. Следует хранить объекты-члены посредством указателя, а не по значению, причем предпочтительно хранение с использованием одного указателя (более подробно о скрытой реализации, ее цене и том, как эту цену минимизировать, речь идет в задачах 4.1—4.5).

Пример 2 иллюстрирует обобщенное преобразование, способствующее обеспечению безопасности (в качестве альтернативного варианта `pimpl_` может быть простым указателем; кроме того, вы можете использовать некоторый другой объект для управления указателем).

```
// Пример 2. Обобщенное решение задачи Каргилла
//
class Widget
{
public:
    Widget();  // Инициализация pimpl_ значением
               // new WidgetImpl
    ~Widget(); // Деструктор должен быть указан явно;
               // неявная версия приводит к проблемам
               // при использовании (см. задачи 4.4 и 4.5)
    Widget& operator=( const Widget& );
    // ...
private:
    class WidgetImpl;
    auto_ptr<WidgetImpl> pimpl_;
};

// Обычно в отдельном файле реализации:
//
class Widget::WidgetImpl
{
public:
    // ...
    T1 t1_;
```

```
    T2 t2_;
};
```

Небольшая ремарка: заметим, что если мы используем член `auto_ptr`, то: а) мы должны либо обеспечить определение `WidgetImpl` вместе с определением `Widget`, либо, если вы хотите скрыть `WidgetImpl`, вы должны написать свой собственный деструктор `Widget`, даже если это тривиальный деструктор[25]; кроме того б) вы должны обеспечить собственный конструктор копирования и копирующее присваивание для `Widget`, поскольку обычно семантика передачи владения для членов классов не используется. Если у вас имеется некоторый другой тип интеллектуальных указателей, подумайте о его применении вместо `auto_ptr` (но принципы, описанные в данной задаче, остаются неизменно важными в любом случае).

Теперь мы в состоянии легко реализовать не генерирующую исключений функцию `Swap()`, что означает, что мы можем легко реализовать *почти* строгую гарантию безопасности. Сначала реализуем не генерирующую исключений функцию `Swap()`, которая обменивает внутреннее состояние двух объектов. Заметим, что эта функция в состоянии обеспечить гарантию отсутствия исключений, поскольку ни одна из операций `auto_ptr` исключений не генерирует.[26]

```
void Widget::Swap( Widget& other ) /* throw() */
{
    auto_ptr<WidgetImpl> temp( pimpl_ );
    pimpl_ = other.pimpl_;
    other.pimpl_ = temp;
}
```

Теперь реализуем безопасный оператор присваивания с использованием идиомы создания временного объекта и обмена.

```
Widget& Widget::operator=( const Widget& other )
{
    Widget temp(other); // Вся работа выполняется здесь,
    Swap( temp );       // после чего работа "принимается"
    return *this;       // с использованием операций, не
                        // генерирующих исключений
}
```

Мы получили почти строгую безопасность. “Почти”, поскольку гарантии полной неизменности состояния программы при генерации исключения у нас нет. Видите ли вы, почему? Дело в том, что когда мы создаем временный объект `Widget` и, соответственно, члены `t1_` и `t2_` объекта `pimpl_`, создание этих членов (и/или их уничтожение при генерации исключений) может иметь побочные действия, например, такие как изменение глобальной переменной, и не имеется никаких способов узнать об этом или управлять этим процессом.

Потенциальные возражения и их необоснованность

Некоторые горячие головы могут вскочить с криками: “Ага! Вот и доказательство того, что безопасность исключений в общем случае недостижима, поскольку вы не в состоянии сделать произвольный класс строго безопасным без внесения изменений в него!” Я затронул этот вопрос потому, что подобные возражения мне в действительности встречались.

---

[25] Если вы используете деструктор, автоматически генерируемый компилятором, то этот деструктор будет определен в каждой единице трансляции; следовательно, определение `WidgetImpl` должно быть видимо в каждом модуле трансляции.

[26] Заметим, что замена трехстрочного тела `Swap()` одной строкой “`swap(pimpl_, other.pimpl_);`” не гарантирует корректную работу, поскольку `std::swap()` не обязательно корректно работает с `auto_ptr`.

Такой вывод представляется необоснованным. Описанное преобразование вносит минимальные изменения в структуру и является общим решением поставленной задачи. Подобно многим другим целям реализации, безопасность исключений влияет на разработку класса. Так же, как никто не ожидает полиморфной работы класса при отсутствии изменений в производном классе, не следует ожидать и безопасной работы класса без внесения небольших изменений для хранения его членов на некотором расстоянии. В качестве иллюстрации рассмотрим три некорректных утверждения.

- Некорректное утверждение №1. Полиморфизм в C++ не работает, поскольку мы не в состоянии сделать произвольный класс работающим вместо `Base&` без внесения в него изменений (для наследования его от `Base`).
- Некорректное утверждение №2. Контейнеры STL в C++ не работают, поскольку произвольный класс нельзя сделать содержимым контейнера без внесения изменений (обеспечивающего наличие оператора присваивания).
- Некорректное утверждение №3. Безопасность исключений в C++ не работает, поскольку мы не в состоянии обеспечить безопасность произвольного класса без внесения в него изменений (сокрытия внутреннего представления в классе реализации).

Приведенные выше аргументы одинаково некорректны, и использование преобразования реализации класса является общим решением, дающим гарантии безопасности (в действительности — почти строгую гарантию) без каких-либо знаний о безопасности членов данных класса.

Выводы

Итак, чему же нас научила данная задача?

Вывод 1. Безопасность влияет на разработку класса

Безопасность исключений не является и не может являться “деталью реализации”. Описанное в задаче преобразование скрытой реализации — простое структурное изменение, но от этого оно не перестает быть изменением.

Вывод 2. Код всегда можно сделать (почти) строго безопасным

Вот один важный принцип.

> *То, что класс не является безопасным, не является основанием для того, чтобы использующий его код не был безопасным (за исключением побочных действий).*

Вполне можно строго безопасно использовать класс, у которого отсутствует безопасное копирующее присваивание (за исключением, само собой, побочных действий, — например, изменения глобальных переменных, о которых нам неизвестно и управлять которыми мы не в состоянии). Другими словами, можно достигнуть того, что может быть названо локальной строгой гарантией.

- *Локальная строгая гарантия.* При генерации исключения состояние программы остается неизменным по отношению к контролируемым объектам. Этот уровень безопасности всегда предполагает локальную семантику принятия-или-отката, включая корректность всех ссылок и итераторов в случае неуспешности операции.

Технология сокрытия деталей за указателем может быть с равным успехом использована как разработчиком `Widget`, так и его пользователем. Если этим занимается

разработчик, то пользователь может безопасно использовать класс непосредственно; в противном случае пользователю можно поступить примерно следующим образом.

```
// Пример 3. Пользователь делает то,
//           что не сделал автор Widget
//
class MyClass
{
    auto_ptr<Widget> w_; // Хранит небезопасный с точки
                         // зрения копирования объект
public:
    void Swap( MyClass& other ) /* throw */
    {
        auto_ptr<Widget> temp( w_ );
        w_ = other.w_;
        other.w_ = temp;
    }

    MyClass& operator=( const MyClass& other )
    {
        MyClass temp(other); // Вся работа выполняется здесь,
        Swap( temp );     // после чего работа "принимается"
        return *this;     // с использованием операций, не
                          // генерирующих исключений
    }
}
```

Вывод 3. Благоразумно используйте указатели

Скотт Мейерс (Scott Meyers) пишет следующее.[27]

Когда я говорю об обработке исключений, я учу слушателей двум вещам.

1. *УКАЗАТЕЛИ — ВАШИ ВРАГИ, поскольку они приводят к проблемам, для решения которых и был разработан* `auto_ptr`.

То есть простые указатели обычно должны принадлежать управляющим объектам, которые являются владельцами ресурсов, на которые указывает простой указатель, и выполняют их автоматическое освобождение.

2. *УКАЗАТЕЛИ — ВАШИ ДРУЗЬЯ, поскольку операции с указателями не генерируют исключений.*

   На этом я прощаюсь со слушателями.

Скотт хорошо подметил двойственность в использовании указателей. К счастью, на практике все не так уж плохо — вы можете (и должны) взять лучшее из обоих миров.

- *Используйте указатели, поскольку они ваши друзья*, — ведь операции с указателями не могут генерировать исключения.
- *Работайте с ними с использованием управляющих объектов*, таких как `auto_ptr`, поскольку тем самым обеспечивается освобождение ресурсов. Использование управляющих объектов не затрагивает отсутствия генерации исключений при работе с указателями, так как операции `auto_ptr` также не генерируют исключений (кроме того, вы всегда можете воспользоваться реальным указателем, получив его, например, вызовом `auto_ptr::get()`).

Зачастую наилучший путь использования идиомы скрытой реализации — посредством использования указателя (который не может генерировать исключений), как было показано в примере 2; при этом динамический ресурс “заворачивается” в управ-

[27] Скотт Мейерс, частное письмо.

ляющий объект (в приведенном примере — в `auto_ptr`). Только запомните, что если вы используете `auto_ptr`, то вы должны обеспечить ваш класс собственным деструктором, конструктором копирования и копирующим присваиванием с корректной семантикой (либо запретить копирующее присваивание и конструирование, если таковые не имеют смысла для вашего класса).

В следующей задаче мы применим изученный материал для анализа наилучшего пути выражения взаимоотношения классов.

Задача 2.19. Разработка безопасных классов. Часть 2 ***Сложность: 6***

Что означает "реализован посредством"? Это может вас удивить, но выбор между наследованием и делегированием может привести к определенным последствиям с точки зрения безопасности исключений. Можете ли вы определить, к каким?

1. Что означает "реализован посредством"?
2. В C++ отношение "реализован посредством" может быть выражено либо закрытым или защищенным наследованием, либо содержанием/делегированием, т.е. при написании класса `T`, реализованного посредством класса `U`, двумя основными вариантами являются закрытое наследование от `U` и включение `U` в качестве члена объекта.

   Влияет ли выбор между этими методами на безопасность исключений? Поясните. (Игнорируйте все прочие вопросы, не связанные с безопасностью.)

Решение

Реализован посредством

Мы используем выражения типа СОДЕРЖИТ, ЯВЛЯЕТСЯ, ИСПОЛЬЗУЕТ[28] как удобные сокращения для описания различных типов взаимоотношений в коде.

Отношение "ЯВЛЯЕТСЯ" (или более точно — "ЯВЛЯЕТСЯ ПОДСТАНОВКОЙ") должно следовать принципу подстановки Барбары Лисков (Barbara Liskov) (Liskov's Substitutability Principle, LSP), в котором она определила, что означает, что тип `S` может быть подставлен вместо типа `T`.

> *Если для каждого объекта `o1` типа `S` существует объект `o2` типа `T` такой, что для всех программ `P`, определенных в терминах `T`, поведение `P` останется неизменным при подстановке `o1` вместо `o2`, то `S` является подтипом `T`. [Liskov88]*

ЯВЛЯЕТСЯ обычно используется для описания открытого наследования, которое соответствует LSP и сохраняет весь открытый интерфейс. Например, "`D` ЯВЛЯЕТСЯ `B`" означает, что код, принимающий объект базового класса `B` по указателю или по ссылке, может естественным образом использовать вместо него объекты открытого производного класса `D`.

Однако очень важно помнить, что в C++ имеются и другие пути для выражения отношения "ЯВЛЯЕТСЯ" помимо открытого наследования. ЯВЛЯЕТСЯ может также описывать не связанные отношением наследования классы, которые поддерживают одинаковые интерфейсы и, таким образом, могут взаимозаменяемо использоваться в шаблонах, использующих эти интерфейсы. LSP применим к

[28] Has-A, Is-A, Uses-A. — *Прим. перев.*

этому виду подстановок в той же мере, что и к другим видам. В данном контексте, например, “X ЯВЛЯЕТСЯ Y” сообщает, что код шаблона, который принимает объекты типа Y, будет принимать и объекты типа X, поскольку и те и другие имеют одинаковый интерфейс. Более подробное рассмотрение этого вопроса вы можете найти в работе [Henney00].

Понятно, что оба типа подстановок зависят от контекста реального использования объектов; главное, что отношение ЯВЛЯЕТСЯ может быть выражено различными путями. Но вернемся к отношению “реализован посредством”.

1. Что означает “реализован посредством”?

Не менее часто, чем ЯВЛЯЕТСЯ, используется отношение “реализован посредством[29]”. Тип T реализован посредством другого типа U, если T в своей реализации некоторым образом использует U. Выражение “использует некоторым образом” можно толковать довольно широко — от использования T в качестве оболочки или прокси для U и до случайного использования U при реализации каких-то деталей собственных сервисов T.

Обычно “T реализован посредством U” означает, что либо T СОДЕРЖИТ U, как показано в примере 1*а*:

```
// Пример 1а. T реализован посредством U
//            с использованием присваивания
//
class T
{
    // ...
private:
    U* u_;  // Возможно, по значению или по ссылке
};
```

либо T выводится из U закрытым образом, как показано в примере 1*б*:[30]

```
// Пример 1б. T реализован посредством U
//            с использованием наследования
class T: private U
{
    // ...
};
```

Это приводит нас к естественному вопросу: какой выбор мы должны сделать при реализации отношения “реализован посредством”? На какие компромиссы следует идти? На что следует обратить внимание в каждом из случаев?

Реализация посредством: наследование или делегирование?

2. В C++ отношение “реализован посредством” может быть выражено либо закрытым или защищенным наследованием, либо содержанием/делегированием, т.е. при написании класса T, реализованного посредством класса U, двумя основными вариантами являются закрытое наследование от U и включение U в качестве члена объекта.

Как я уже говорил раньше, наследованием часто злоупотребляют даже опытные разработчики. Вполне разумное правило, которого стоит придерживаться при

[29] Is-Implemented-In-Terms-Of (дословно — “реализован в терминах”). — *Прим. перев.*

[30] Открытое наследование также моделирует отношение “реализован посредством”, но основное предназначение открытого наследования — моделирование отношения ЯВЛЯЕТСЯ.

разработке программного обеспечения, — минимизировать взаимосвязь элементов. Если одно и то же взаимоотношение может быть выражено несколькими способами, желательно выбирать тот, который использует наиболее слабую взаимосвязь. Наследование же представляет собой практически наиболее тесную взаимосвязь в C++, уступая только дружбе,[31] и его следует применять только там, где нет иной, более слабой в плане связи альтернативы. Если взаимоотношения классов можно выразить, ограничившись делегированием, — предпочтительно использовать этот способ.

Очевидно, что принцип минимизации взаимосвязи непосредственно влияет на надежность вашего кода, время компиляции и другие его характеристики. Интересно, что выбор между делегированием и наследованием для осуществления отношения "реализации посредством" влияет и на вопросы безопасности исключений (хотя после прочтения данной главы это уже не должно вас удивлять).

Принцип минимизации взаимосвязей гласит следующее.

> *Меньшая степень взаимосвязи способствует корректности программы (включая безопасность исключений), в то время как тесная взаимосвязь снижает максимально возможную корректность программы.*

Это естественно. В конце концов, чем менее тесно связаны между собой объекты реального мира, тем меньшее воздействие оказывают они друг на друга. Вот почему люди размещают брандмауэры в домах и водонепроницаемые переборки на кораблях. Если в одном отсеке возникнут неполадки, то чем сильнее этот отсек будет изолирован от других, тем менее вероятно распространение этих неприятностей на другие отсеки до того момента, как мы возьмем ситуацию под контроль.

А теперь вновь вернемся к примерам 1*а* и 1*б* и вновь рассмотрим класс `T`, реализованный посредством `U`. Рассмотрим оператор копирующего присваивания: каким образом выбор выражения отношения "реализован посредством" влияет на разработку `T::operator=()`?

Следствия безопасности исключений

> **Влияет ли выбор между этими методами на безопасность исключений? Поясните. (Игнорируйте все прочие вопросы, не связанные с безопасностью.)**

Сначала рассмотрим, каким образом мы можем написать `T::operator=()`, если отношение "реализован посредством" выражается с использованием отношения СОДЕРЖИТ. Наверняка у вас уже выработалась привычка использовать технологию, заключающуюся в том, чтобы выполнить всю работу отдельно, а потом принять ее с использованием операций, не генерирующих исключения. Использование такой технологии, обеспечивающей максимальную безопасность исключений, приводит к коду следующего вида.

```
// Пример 2а. T реализован посредством U
// с использованием отношения СОДЕРЖИТ.
//
class T
{
    // ...
private:
```

[31] Друг класса `X` имеет более тесную взаимосвязь с ним, чем наследник, поскольку имеет доступ ко всем его членам, в то время как наследник ограничен доступом к открытым и защищенным членам предка.

```
    U* u_;
};

T& T::operator=( const T& other )
{
    U* temp = new U( *other.u_ ); // Вся работа выполняется
                                  // отдельно,
    delete u_;    // а затем принимается с использованием
    u_ = temp;    // операций, не генерирующих исключения
    return *this;
}
```

Это довольно хорошее решение. Не делая никаких предположений об U, мы в состоянии написать T::operator=(), который почти строго безопасен (за исключением возможных побочных действий U).

Даже если объект типа U содержится в объекте типа T по значению, а не по указателю, преобразовать класс таким образом, чтобы объект U хранился с использованием указателя, достаточно легко. Объект U может также быть размещен в скрытой реализации, как описывалось в первой части данной задачи. Именно факт использования делегирования (отношения СОДЕРЖИТ) дает нам возможность легко и просто написать безопасный оператор копирующего присваивания T::operator=() без каких бы то ни было предположений об U.

Теперь рассмотрим, как изменится проблема, если взаимоотношения между T и U выражаются каким-либо видом наследования.

```
// Пример 2б. T реализован посредством U
// с использованием наследования
//
class T : private U
{
    // ...
};

T& T::operator=( const T& other )
{
    U::operator=( other );   // ???
    return *this;
}
```

Проблема заключается в вызове оператора U::operator=(). Как упоминалось во врезке “Более простой пример” в задаче 2.18, если U::operator=() может генерировать исключения таким образом, что при этом происходит изменение целевого объекта, то невозможно написать строго безопасный оператор T::operator=(), конечно, если U не предоставляет соответствующей возможности посредством некоторой другой функции (но если такая функция имеется, то почему это не U::operator=()?).

Другими словами, возможность достижения гарантии безопасности исключений оператора T::operator=() неявным образом зависит от гарантий безопасности U. Есть ли в этом что-то удивительное? Нет, поскольку пример *2б* использует наиболее тесную, насколько это возможно, взаимосвязь для выражения соотношения между T и U.

Резюме

Меньшая степень взаимосвязи способствует корректности программы (включая безопасность исключений), в то время как тесная взаимосвязь снижает максимально возможную корректность программы.

Наследованием часто злоупотребляют даже опытные разработчики. В задаче 3.5 приведена дополнительная информация о других, помимо вопросов безопасности, причинах для использования делегирования вместо наследования, где это возможно. Всегда следует минимизировать взаимосвязи объектов. Если взаимоотношения классов могут быть выражены несколькими способами, используйте тот, при котором взаимосвязь наименее сильная (но при этом способ выражения остается применим с точки зрения практичности). В частности, используйте наследование только там, где недостаточно использования делегирования.

3

Разработка классов, наследование и полиморфизм

Задача 3.1. Механика классов *Сложность: 7*

Насколько хорошо вы разбираетесь в деталях написания классов? Эта задача посвящена не только явным ошибкам при создании классов; скорее, она в большей степени относится к профессиональному стилю разработки. Понимание основных принципов поможет вам при разработке более надежных и легче поддерживаемых классов.

Вы просматриваете код. Программист создал приведенный далее класс, при написании которого показал плохой стиль и допустил ряд ошибок. Сколько из них вы сможете найти и как бы вы их исправили?

```
class Complex
{
public:
    Complex( double real, double imaginary = 0 )
        : _real( real ), _imaginary( imaginary )
    {
    }
    void operator + ( Complex other )
    {
        _real = _real + other._real;
        _imaginary = _imaginary + other._imaginary;
    }
    void operator << ( ostream os )
    {
        os << "(" << _real << "," << _imaginary << ")";
    }
    Complex operator ++ ()
    {
        ++_real;
        return *this;
    }
    Complex operator ++ ( int )
    {
        Complex temp = *this;
        ++_real;
        return temp;
    }
private:
    double _real, _imaginary;
};
```

Решение

В этом классе множество проблем — больше, чем я покажу вам в решении этой задачи. Главная цель этой задачи — показать механику классов (рассмотрев вопросы типа “какова каноническая форма оператора <<?” и “должен ли оператор + быть членом?”), а не только указать некорректности при разработке интерфейса. Однако я начну с одного, пожалуй, самого полезного замечания: зачем писать класс `Complex`, если таковой уже имеется в стандартной библиотеке? Помимо всего прочего, стандартный класс разработан с учетом многолетнего опыта лучших программистов и не болен “детскими болезнями крутизны в программизме”. Будьте скромнее и воспользуйтесь трудами ваших предшественников.

Рекомендация

Вместо написания собственного кода повторно используйте имеющийся, в особенности код стандартной библиотеки. Это быстрее, проще и безопаснее.

Вероятно, наилучший способ решить все проблемы — это не использовать класс `Complex` вообще, заменив его стандартным классом `std::complex`.

Если же мы не намерены идти легким путем, пойдем сложным и приступим к исправлению ошибок. Начнем с конструктора.

1. Конструктор позволяет неявное преобразование.

```
Complex( double real, double imaginary = 0 )
    : _real( real ), _imaginary( imaginary )
{
}
```

Поскольку второй параметр имеет значение по умолчанию, эта функция может использоваться в качестве конструктора с одним параметром и, следовательно, как неявное преобразование `double` в `Complex`. В рассматриваемом случае такое преобразование, вероятно, вполне корректно, но, как мы видели в задаче 1.6, неявное преобразование не всегда входит в планы разработчика. Вообще говоря, желательно объявлять все ваши конструкторы как `explicit`, кроме тех случаев, когда вы намерены разрешить неявное преобразование (подробнее об этом можно узнать из задачи 9.2).

Рекомендация

Не забывайте о скрытых временных объектах, создаваемых при неявном преобразовании. Избежать их появления можно, объявляя по возможности разрабатываемые вами конструкторы как `explicit` и избегая написания операторов преобразования.

2. Оператор + недостаточно эффективен.

```
void operator + ( Complex other )
{
    _real = _real + other._real;
    _imaginary = _imaginary + other._imaginary;
}
```

Для повышения эффективности желательно передавать параметры по ссылке на константный объект.

Рекомендация

По возможности передавайте объекты в качестве параметров функции по ссылке, как `const&`*, а не по значению.*

Кроме того, в обоих случаях “a=a+b” следует переписать как “a+=b”. В данном конкретном случае это не приведет к повышению эффективности, поскольку у нас выполняется суммирование переменных типа `double`, но может привести к ее существенному повышению при работе с различными типами классов.

Рекомендация

Предпочтительно использовать “a `op=` *b;” вместо “a = a* `op` *b;” (где* `op` *означает некоторый оператор). Такой код проще для понимания и зачастую более эффективен.*

Причина более высокой эффективности оператора += заключается в том, что он работает с объектом слева от оператора непосредственно, а кроме того, возвращает ссылку, а не временный объект (`operator+()` должен возвращать временный объект). Чтобы понять, почему это так, рассмотрим канонические виды операторов, обычно реализуемые для некоторого типа `T`.

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

const T operator+( const T& a, const T& b )
{
    T temp( a );
    temp += b;
    return temp;
}
```

Обратите внимание на взаимоотношения операторов + и +=. Первый из них реализуется посредством второго — как для простоты реализации, так и для последовательности семантики.

Рекомендация

Если вы предоставляете автономную версию некоторого оператора (например, `operator+`*), всегда предоставляйте и присваивающую версию этого оператора (*`operator+=`*), причем предпочтительна реализация первого оператора посредством второго. Кроме того, сохраняйте естественные взаимоотношения между* `op` *и* `op=` *(здесь* `op` *означает произвольный оператор).*

3. `operator+` не должен быть функцией-членом.

```
void operator + ( Complex other )
{
    _real = _real + other._real;
    _imaginary = _imaginary + other._imaginary;
}
```

Если `operator+` является функцией-членом, как в данном случае, то данный оператор не будет работать настолько естественно, насколько этого могут ожидать пользователи разрабатываемого класса, исходя из разрешения неявного преобразования типов. В частности, при сложении объекта

Complex с числовыми значениями вы можете написать “a = b + 1.0”, но не можете — “a = 1.0 + b”, поскольку функция-член operator+ требует, чтобы левым аргументом сложения был объект типа Complex, но не const double.

Кроме того, если вы хотите обеспечить пользователям удобство сложения объектов Complex с величинами типа double, имеет смысл предоставить также перегруженные функции operator+(const Complex&, double) и operator+(double, const Complex&).

Рекомендация

Стандарт требует, чтобы операторы =, [], () и -> были членами, а операторы new, new[], delete и delete[] — статическими членами. Для всех остальных функций действует следующее правило.

Если функция представляет собой operator>> или operator<< для потокового ввода-вывода, или если она требует преобразования типа левого аргумента, или если она может быть реализована только с использованием открытого интерфейса класса,

ее следует сделать нечленом класса (в первых двух случаях — другом класса). Если требуется виртуальное поведение данной функции,

добавьте виртуальную функцию-член и реализуйте ее посредством соответствующей функции-нечлена,

в противном случае

сделайте ее членом класса.

4. operator+ не должен изменять значение объекта и должен возвращать временный объект, содержащий сумму.

```
void operator + ( Complex other )
{
    _real = _real + other._real;
    _imaginary = _imaginary + other._imaginary;
}
```

Обратите внимание, что тип возвращаемого временного объекта должен быть const Complex (а не просто Complex), чтобы предотвратить возможность присваивания типа “a+b=c”.[1]

5. Если вы определяете оператор op, вам следует определить и оператор op=. В данном случае, поскольку у вас определен operator+, вы должны определить и operator+= и реализовать оператор + посредством оператора +=.

6. operator<< не должен быть функцией-членом (см. обсуждение этого вопроса при рассмотрении оператора +).

```
void operator << ( ostream os )
{
    os << "(" << _real << "," << _imaginary << ")";
}
```

Параметрами оператора << должны быть (ostream&, const Complex&). Обратите внимание, что оператор-нечлен << обычно должен быть реализован посредством

[1] Историческая справка: в предварительных версиях стандарта было запрещено вызывать неконстантную функцию-член для rvalue. Таким образом достигался тот же эффект — невозможность выполнения присваивания a+b=c. Но затем К□ниг предложил снять этот запрет, и вместо этого возвращать const-объекты. — *Прим. ред.*

(зачастую виртуальных) функций-членов, которые и выполняют всю работу по выводу информации в поток и обычно именуемых наподобие `Print()`.

Кроме того, для реального оператора << следует позаботиться о флагах формата вывода, с тем чтобы обеспечить максимальные возможности при его использовании. За деталями реализации флагов обратитесь к стандартной библиотеке либо к соответствующей литературе.

7. `operator<<` должен иметь возвращаемый тип `ostream&` и должен возвращать ссылку на поток, чтобы обеспечить возможность использования цепочек выводов типа "`cout << a << b`".

Рекомендация

Всегда возвращайте ссылку на поток из операторов `operator<<` и `operator>>`.

8. Возвращаемый тип оператора преинкремента некорректен.

```
Complex operator ++ ()
{
    ++_real;
    return *this;
}
```

Преинкремент должен возвращать ссылку на неконстантный объект — в нашем случае `Complex&`. Это обеспечивает более интуитивный и эффективный код клиента.

9. Возвращаемый тип оператора постинкремента некорректен.

```
Complex operator ++ ( int )
{
    Complex temp = *this;
    ++_real;
    return temp;
}
```

Постинкремент должен возвращать константное значение — в данном случае `const Complex`. Запрещая изменения возвращаемого объекта, мы предупреждаем возможное использование кода типа "`a++++`", что могут попытаться сделать некоторые наивные пользователи.

10. Постинкремент должен быть реализован посредством преинкремента (канонический вид постинкремента приведен в задаче 1.6.

Рекомендация

Для согласованности всегда реализуйте постфиксный инкремент посредством префиксного; в противном случае ваши пользователи получат неожиданные (и, скорее всего, неприятные) результаты.

11. Избегайте использования зарезервированных имен.

```
private:
    double _real, _imaginary;
```

В ряде популярных книг типа [Gamma95] используются ведущие подчеркивания в именах переменных, но вы не берите с них пример. Стандарт резервирует для целей реализации некоторые идентификаторы с именами, начинающимися с подчеркиваний, но соответствующие правила достаточно сложно запомнить,

поэтому проще не использовать ведущие подчеркивания вовсе.[2] Что касается меня, то я предпочитаю следовать соглашению, по которому переменные-члены класса именуются с замыкающими подчеркиваниями.

Вот и все. Итак, корректная версия класса выглядит следующим образом (игнорируя вопросы дизайна и стиля программирования, не затронутые при нашем рассмотрении).

```
class Complex
{
public:
    explicit Complex( double real, double imaginary = 0 )
        : real_( real ), imaginary_( imaginary )
    {
    }

    Complex& operator += ( const Complex& other )
    {
        real_      += other.real_;
        imaginary_ += other.imaginary_;
        return *this;
    }

    Complex& operator ++ ()
    {
        ++real_;
        return *this;
    }

    Complex operator ++ ( int )
    {
        Complex temp( *this );
        ++*this;
        return temp;
    }

    ostream& Print( ostream& os ) const
    {
        return os << "(" << real_ << ","
                  << imaginary_ << ")";
    }

private:
    double real_, imaginary_;
};

const Complex operator+( const Complex& lhs,
                         const Complex& rhs )
{
    Complex ret( lhs );
    ret += rhs;
    return ret;
}
```

[2] Например, имена с ведущими подчеркиваниями технически зарезервированы только для имен, не являющихся членами, поэтому некоторые читатели могут возразить, что члены класса могут иметь имена, начинающиеся с подчеркиваний, и это не будет нарушением стандарта. На практике это не совсем так, поскольку ряд реализаций определяют макросы, имена которых содержат ведущие подчеркивания, а макросы не подчиняются правилам поиска имен и их действие не ограничено областями видимости. Макросы подавляют имена членов так же легко, как и имена нечленов класса. Гораздо проще оставить ведущие подчеркивания реализациям языка и совсем не использовать их в своем коде.

```
ostream& operator<<( ostream& os, const Complex& c )
{
    return c.Print( os );
}
```

Задача 3.2. Замещение виртуальных функций *Сложность: 6*

Виртуальные функции — очень привлекательное фундаментальное свойство C++, которое, впрочем, может легко загнать в ловушку недостаточно хорошо разбирающегося в этом вопросе программиста. Если вы справитесь с данной задачей, значит, вы хорошо знакомы с виртуальными функциями, и вам не придется тратить много времени на отладку запутанного кода, подобного приведенному далее.

В архивах исходных текстов вашей компании вы нашли следующий фрагмент кода, написанный неизвестным программистом. Похоже, программист просто экспериментировал, выясняя, как работают различные возможности C++. Какой вывод программы мог ожидать этот неизвестный программист и что он в результате получил?

```
#include <iostream>
#include <complex>

using namespace std;

class Base
{
public:
    virtual void f( int );
    virtual void f( double );
    virtual void g( int i = 10 );
};

void Base::f( int )
{
    cout << "Base::f(int)" << endl;
}

void Base::f( double )
{
    cout << "Base::f(double)" << endl;
}

void Base::g( int i )
{
    cout << i << endl;
}

class Derived: public Base
{
public:
    void f( complex<double> );
    void g( int i = 20 );
};

void Derived::f( complex<double> )
{
    cout << "Derived::f(complex)" << endl;
}

void Derived::g( int i )
{
    cout << "Derived::g() " << i << endl;
}
```

```
void main()
{
    Base    b;
    Derived d;
    Base*   pb = new Derived;
    b.f(1.0);
    d.f(1.0);
    pb->f(1.0);
    b.g();
    d.g();
    pb->g();
    delete pb;
}
```

Решение

Начнем с того, что рассмотрим некоторые вопросы стиля программирования, а также одну реальную ошибку.

1. `void main()` является нестандартной инструкцией и, соответственно, непереносимой.

   Да, я знаю, что, к сожалению, именно такая запись часто встречается в книгах. Некоторые авторы даже утверждают, что такая запись соответствует стандарту, но это не так. Это никогда не было верным, даже в 1970-е годы, в достандартном C.

   Несмотря на нестандартность `void main()`, многие компиляторы допускают такое объявление функции `main`; однако его использование рано или поздно приведет к непереносимости программы на новый компилятор (а, возможно, и на новую версию того же компилятора). Лучше всего выработать привычку использовать одно из двух стандартных объявлений функции `main`.

   ```
   int main()
   int main( int argc, char* argv[] )
   ```

   Вы можете заметить еще одну проблему — отсутствие инструкции `return` в функции `main`. Однако несмотря на то, что тип возвращаемого функцией `main` значения — `int`, ошибкой отсутствие инструкции `return` не является (хотя предупреждение об этом факте со стороны компилятора определенно является хорошим тоном). Дело в том, что в соответствии со стандартом отсутствие инструкции `return` аналогично наличию инструкции `return 0;`. Но учтите, что многие компиляторы до сих пор не реализовали это правило и реагируют на отсутствие инструкции `return` в функции `main` как на ошибку.

2. Использование `delete pb;` небезопасно.

   Эта инструкция выглядит совершенно безвредно. И она бы и была такой, если бы разработчик не забыл снабдить класс `Base` виртуальным деструктором. Удаление же указателя на базовый класс без виртуального инструктора некорректно, и лучшее, на что можно надеяться, — нарушение целостности и согласованности памяти, поскольку при этом вызывается неверный деструктор, а оператор `delete` получает неверный размер объекта.

Рекомендация

Делайте деструктор базового класса виртуальным (за исключением тех случаев, когда вы совершенно точно знаете, что не будет никаких попыток удаления производного объекта посредством указателя на базовый).

Для понимания дальнейшего материала следует четко различать три термина.

- *Перегрузка* (overload[3]) функции `f()` означает наличие другой функции с тем же именем (`f`) в той же области видимости, но с другими типами параметров. При вызове `f()` компилятор пытается выбрать наиболее подходящую для типов фактических параметров функцию.
- *Замещение*, или *переопределение* (override) виртуальной функции `f()` означает наличие в производном классе другой функции с тем же именем (`f`) и теми же типами параметров.
- *Сокрытие* (hide) функции `f()` в окружающей области видимости (базовый класс, внешний класс или пространство имен) означает наличие другой функции с тем же именем (`f`) во вложенной области видимости (производный класс, вложенный класс или пространство имен), так что эта функция скрывает функцию с тем же именем из объемлющей области видимости.

3. `Derived::f` не является перегруженной. Класс `Derived` не перегружает, а скрывает `Base::f`. Это отличие очень важно, поскольку означает, что `Base::f(int)` и `Base::f(double)` не видимы в области видимости `Derived`. (Удивительно, но некоторые популярные компиляторы даже не выводят предупреждения об этом, так что знать о сокрытии важнее, чем можно было бы предполагать.)

Если автор класса `Derived` намерен скрыть функции класса `Base` с именем `f`, то у него это получилось.[4] Однако обычно сокрытие происходит неожиданно и непреднамеренно; корректный путь внесения имен в область видимости `Derived` состоит в использовании объявления `using Base::f;`.

Рекомендация

При использовании функции с тем же именем, что и у унаследованной функции, убедитесь, что унаследованные функции внесены в область видимости с помощью объявления `using`, если только вы не намерены скрывать их.

4. `Derived::g` замещает `Base::g`, но при этом изменяет параметр по умолчанию. Это, несомненно, не лучшее решение. Если только вы не хотите специально запутать пользователя, не изменяйте параметры по умолчанию замещаемых вами наследуемых функций. Да, это разрешено в C++; да, результат строго определен; но — нет, не пользуйтесь этой возможностью.

Далее мы увидим, насколько такое переопределение параметра по умолчанию запутывает пользователей.

Рекомендация

Никогда не изменяйте параметры по умолчанию замещаемых унаследованных функций.

Теперь мы перейдем к функции `main` и рассмотрим, чего же хотел добиться программист и что получилось на самом деле.

```
void main()
{
    Base    b;
```

[3] В русскоязычной литературе встречается также перевод данного термина как “совместное использование”. — *Прим. ред.*

[4] При преднамеренном сокрытии базового имени проще сделать это путем указания закрытого (`private`) объявления `using` для данного имени.

```
    Derived d;
    Base*   pb = new Derived;

    b.f(1.0);
```

Никаких проблем. Здесь, как и ожидается, вызывается функция `Base::f(double)`.

```
d.f(1.0);
```

Здесь происходит вызов `Derived::f(complex<double>)`. Почему? Вспомним, что в классе `Derived` нет объявления "`using Base::f;`", так что функции с именем `f` из класса `Base` не попадают в область видимости, а следовательно, ни `Base::f(int)`, ни `Base::f(double)` вызваны быть не могут. Их нет в той же области видимости, где находится `Derived::f(complex<double>)`, так что в перегрузке эти функции не участвуют.

Программист мог ожидать, что здесь будет вызвана функция `Base::f(double)`, но в данном случае не приходится ожидать даже сообщения об ошибке, поскольку, к счастью(?), `complex<double>` может неявно преобразоваться из типа `double`, так что компилятор интерпретирует данный вызов как `Derived::f(complex<double>(1.0))`.

```
pb->f(1.0);
```

Несмотря на то что `Base*pb` указывает на объект `Derived`, здесь будет вызвана функция `Base::f(double)`, поскольку разрешение перегрузки выполняется на основе статического типа (`Base`), а не динамического (`Derived`).

По той же причине вызов `pb->f(complex<double>(1.0));` скомпилирован не будет, поскольку интерфейс `Base` не содержит соответствующей функции.

```
b.g();
```

Здесь выводится "`10`", поскольку происходит простой вызов `Base::g(int)` с параметром по умолчанию, равным 10. Никаких проблем.

```
d.g();
```

Здесь выводится "`Derived::g() 20`", поскольку происходит простой вызов `Derived::g(int)` с параметром по умолчанию, равным 20. Также никаких проблем.

```
pb->g();
```

Здесь выводится "`Derived::g() 10`".

"Минутку! — протестуете вы. — Что же происходит?!" Ничего страшного, компилятор совершенно прав. Вспомните, что значения по умолчанию, как и разрешение перегрузки, определяются на основании статического типа (в нашем случае `Base`) объекта, так что значение по умолчанию при вызове равно 10. Однако вызываемая функция виртуальна, поэтому вызов функции осуществляется с учетом динамического типа (в данном случае `Derived`) объекта.

Если вы смогли понять последние абзацы, в которых речь шла о сокрытии имен и использовании статических и динамических типов, значит, вы разобрались в данной теме. Поздравляю!

```
    delete pb;
}
```

Здесь, как я уже говорил выше, удаление в состоянии вызвать нарушение целостности памяти и оставить объект частично не удаленным.

Задача 3.3. Взаимоотношения классов. Часть 1 ***Сложность: 5***

Каковы ваши познания в области объектно-ориентированного проектирования? Эта задача иллюстрирует ряд распространенных ошибок при проектировании классов.

Сетевое приложение имеет два варианта сессии связи, каждый со своим собственным протоколом сообщений. У обоих протоколов есть одинаковые части (некоторые вычисления и даже некоторые сообщения одинаковы), так что программист решает задачу путем инкапсулирования общих действий и сообщений в классе `BasicProtocol`.

```
class BasicProtocol /* : возможные базовые классы */
{
public:
    BasicProtocol();
    virtual ~BasicProtocol();
    bool BasicMsgA( /* ... */ );
    bool BasicMsgB( /* ... */ );
    bool BasicMsgC( /* ... */ );
};

class Protocol1: public BasicProtocol
{
public:
    Protocol1();
    ~Protocol1();
    bool DoMsg1( /* ... */ );
    bool DoMsg2( /* ... */ );
    bool DoMsg3( /* ... */ );
    bool DoMsg4( /* ... */ );
};

class Protocol2: public BasicProtocol
{
public:
    Protocol2();
    ~Protocol2();
    bool DoMsg1( /* ... */ );
    bool DoMsg2( /* ... */ );
    bool DoMsg3( /* ... */ );
    bool DoMsg4( /* ... */ );
    bool DoMsg5( /* ... */ );
};
```

Каждая функция-член `DoMsg…()` вызывает для выполнения общей работы функции `BasicProtocol::Basic…()`, но реальная передача сообщений осуществляется функциями `DoMsg…()`. В каждом классе содержатся и другие члены, но можно считать, что все существенные члены классов приведены в данном коде.

Прокомментируйте этот проект. Есть ли в нем что-то, что вы бы хотели изменить? Если да, то что?

Решение

В этой задаче показаны наиболее распространенные ловушки при проектировании объектно-ориентированного взаимоотношения классов. Кратко резюмируем состояние дел: классы `Protocol1` и `Protocol2` открыто выводятся из общего базового класса `BasicProtocol`, выполняющего некоторую общую работу.

Ключевой при постановке задачи является следующая фраза.

> *Каждая функция-член `DoMsg…()` вызывает для выполнения общей работы функции `BasicProtocol::Basic…()`, но реальная передача осуществляется функциями `DoMsg…()`.*

Итак, ситуация описывается типичным отношением, которое в C++ пишется как “реализуется посредством”, а читается — как “закрытое наследование” или “принадлежность”. К сожалению, многие программисты ошибочно произносят

“открытое наследование”, тем самым путая наследование реализации и наследование интерфейса. Это разные вещи, и такая путаница лежит в основе данной задачи.

Распространенная ошибка

Никогда не используйте открытое наследование, за исключением модели ЯВЛЯЕТСЯ и РАБОТАЕТ КАК. Все замещающие функции-члены при этом не должны предъявлять повышенные требования к условиям своей работы и не должны иметь пониженную функциональность по сравнению с оригиналом.

Кстати, программисты, имеющие обыкновение допускать эту ошибку, обычно в результате строят большие иерархии наследования. Это резко усложняет поддержку программ из-за внесения излишней сложности, необходимости изучения пользователем интерфейсов большого количества классов даже в том случае, когда используется только один определенный производный класс. Такое проектирование может также повлиять на использование памяти и производительность программ за счет добавления излишних виртуальных таблиц и косвенных обращений к классам, которые реально не являются необходимыми. Если вы обнаружили, что у вас подозрительно часто образуются глубокие иерархические структуры, вам следует пересмотреть свой стиль проектирования. Глубокие иерархические структуры редко являются необходимыми и почти никогда не являются хорошим решением поставленной задачи. Если вы не в состоянии поверить в это и считаете, что объектно-ориентированное программирование без интенсивного использования наследования таковым не является, в качестве контрпримера взгляните на стандартную библиотеку.

Рекомендация

Никогда не используйте открытое наследование для повторного использования кода (базового класса). Используйте открытое наследование только для того, чтобы быть повторно использованным кодом, полиморфно использующим базовые объекты.[5]

Рассмотрим вопрос более детально.

1. `BasicProtocol` не имеет виртуальных функций (кроме деструктора, о котором чуть позже).[6] Это означает, что он не предназначен для полиморфного использования, что само по себе является аргументом против открытого наследования.
2. `BasicProtocol` не имеет защищенных функций или членов. Это означает, что отсутствует “интерфейс наследования”, что является аргументом против любого наследования, как открытого, так и закрытого.
3. `BasicProtocol` инкапсулирует общую деятельность, но, как указывалось, он не выполняет реальную передачу — это делают производные классы. Это означает, что объект `BasicProtocol` не РАБОТАЕТ КАК производный объект протокола и не ИСПОЛЬЗУЕТСЯ КАК производный объект протокола. Открытый интерфейс должен использоваться только в одном случае — для взаимоотношения ЯВЛЯЕТСЯ, подчиняющегося принципу подстановки Лисков (LSP). Для боль-

[5] Я получил эту рекомендацию (и с благодарностью ее использую) от Маршалла Клайна (Marshall Cline), соавтора классической книги *C++ FAQs* [Cline95].

[6] Даже если `BaseProtocol` унаследован от другого класса, мы придем к тому же заключению, поскольку класс `BaseProtocol` не предоставляет ни одной новой виртуальной функции. Если некоторый базовый класс имеет виртуальные функции, то это означает, что для полиморфного использования предназначен базовый класс, а не `BaseProtocol`. Так что, скорее всего, нам надо использовать наследование от этого базового класса вместо наследования от `BaseProtocol`.

шей ясности я обычно называю это взаимоотношение РАБОТАЕТ КАК или ИСПОЛЬЗУЕТСЯ КАК.

4. Производный класс использует только открытый интерфейс `BasicProtocol`. Это означает, что в данном случае наследование не дает никаких преимуществ, а вся необходимая работа может быть выполнена с помощью отдельного вспомогательного объекта типа `BasicProtocol`.

Из детального рассмотрения вытекают еще несколько вопросов. Во-первых, поскольку, как выяснилось, класс `BasicProtocol` не предназначен для наследования, его виртуальный деструктор не является необходимым (более того, он сбивает с толку пользователей класса) и должен быть устранен. Во-вторых, классу `BasicProtocol` стоит дать другое имя, не вводящее пользователей в заблуждение, например, `MessageCreator` или `MessageHelper`.

После того как вы внесете все эти изменения, что вы используете для данной модели "реализован посредством" — закрытое наследование или делегирование? Ответ достаточно очевиден.

Рекомендация

При моделировании отношения "реализован посредством" предпочтительно использовать делегирование, а не наследование. Используйте закрытое наследование только тогда, когда оно совершенно необходимо, т.е. когда вам требуется доступ к защищенным членам или требуется заместить виртуальную функцию. Никогда не используйте открытое наследование для повторного использования кода.

Использование делегирования приводит к более полному разделению ответственности, поскольку использующий класс является обычным клиентом, обладающим доступом только к открытому интерфейсу используемого класса. Используйте делегирование, и вы обнаружите, что ваш код становится яснее, его проще читать и легче поддерживать.

Задача 3.4. Взаимоотношения классов. Часть 2 *Сложность: 6*

Шаблоны проектирования представляют собой важный инструмент при написании повторно используемого кода. Распознаете ли вы шаблоны проектирования в данной задаче? Если да, сможете ли вы их улучшить?

Программе управления базой данных часто требуется выполнить некоторые действия над каждой записью (или выделенными записями) данной таблицы. Сначала выполняется считывающий проход по таблице для кэширования информации о том, какие записи таблицы следует обработать, а затем — второй проход, который вносит изменения в таблицу.

Вместо того чтобы каждый раз переписывать во многом схожую логику, программист пытается предоставить обобщенную повторно используемую схему в следующем абстрактном классе. Смысл заключается в том, что абстрактный класс должен инкапсулировать все повторяющиеся действия, во-первых, по сборке списка строк таблицы, над которыми должны будут выполняться некоторые действия, и во-вторых, выполнение этих действий над каждой строкой из списка. Производные классы детализируют выполняемые операции.

```
// -----------------------------------------
// Файл gta.h
// -----------------------------------------
class GenericTableAlgorithm
{
```

```
public:
    GenericTableAlgorithm( const string& table );
    virtual ~GenericTableAlgorithm();

    // Process() возвращает true тогда и только тогда,
    // когда выполнение функции завершается успешно.
    // Функция выполняет следующую работу:
    // а) физически читает записи таблицы, вызывая для
    //    каждой из них Filter() для определения того,
    //    должна ли данная запись быть обработана
    // б) после составления списка обрабатываемых строк
    //    вызывает ProcessRow() для каждой строки из
    //    списка.
    //
    bool Process();

private:
    // Filter() возвращает true тогда и только тогда,
    // когда строка должна быть включена в список
    // обрабатываемых строк. Действие по умолчанию -
    // возврат true - все строки должны быть обработаны.
    //
    virtual bool Filter( const Record& );

    // ProcessRow() вызывается для каждой записи из списка
    // обрабатываемых строк. Здесь конкретный класс
    // указывает, какие действия должны быть выполнены над
    // строкой. (Примечание: это означает, что каждая
    // обрабатываемая строка будет считана дважды, однако
    // будем считать, что это необходимо, и не будем
    // рассматривать вопросы производительности.)
    //
    virtual bool ProcessRow( const PrimaryKey& ) = 0;

    struct GenericTableAlgorithmImpl* pimpl_;
};
```

1. Этот проект реализует хорошо известный шаблон проектирования. Какой? Почему он применим в данном случае?
2. Не изменяя основ проекта, проанализируйте способ его осуществления. Что бы вы предпочли сделать иначе? В чем состоит предназначение члена `pimpl_`?
3. Приведенный проект на самом деле может быть значительно усовершенствован. За что отвечает класс `GenericTableAlgorithm`? Если он одновременно решает несколько задач, то каким образом можно улучшить их инкапсуляцию? Поясните, как ваш ответ влияет на повторное использование класса и, в частности, на его расширяемость.

Решение

Будем отвечать на вопросы в порядке их поступления.

1. **Этот проект реализует хорошо известный шаблон проектирования. Какой? Почему он применим в данном случае?**

В данном случае реализован шаблон проектирования под названием “Метод шаблона” (Template Method [Gamma95]) — не спутайте его случайно с шаблонами C++. Применимость данного шаблона проектирования обусловлена наличием общего способа выполнения, состоящего из одних и тех же шагов (различие имеется лишь в деталях, которые указываются в производных классах). Кроме того, производный класс

также может использовать рассматриваемый шаблон проектирования, так что различные шаги будут детализироваться на разных уровнях иерархии классов.

(Идиома указателя на реализацию Pimpl в данном случае предназначена для сокрытия реализации класса. Более детально с этой идиомой мы познакомимся в задачах 4.1—4.5.)

Рекомендация

Избегайте открытых виртуальных функций; вместо них предпочтительнее использовать шаблон проектирования "Метод шаблона".

Рекомендация

Не забывайте о существовании шаблонов проектирования.

2. Не изменяя основ проекта, проанализируйте способ его осуществления. Что бы вы предпочли сделать иначе? В чем состоит предназначение члена `pimpl_`?

В данном проекте в качестве кодов возврата используются величины типа `bool`; при этом не имеется другого способа (коды состояния или исключения) для сообщения об ошибках. В зависимости от требований задачи этого может оказаться достаточно, но далеко не всегда.

Член `pimpl_` предназначен для сокрытия реализации за непрозрачным указателем. Структура, на которую указывает `pimpl_`, содержит все закрытые функции- и переменные-члены, так что любые изменения в ней не будут вызывать необходимости перекомпилировать весь клиентский код. Эта технология, описанная в [Lakos96], помогает скомпенсировать недостаточность модульности в C++.

Рекомендация

Для широко используемых классов предпочтительно использовать идиому указателя на реализацию (Pimpl) для сокрытия деталей реализации. Для хранения закрытых членов (как функций, так и переменных) используйте непрозрачный указатель (указатель на объявленный, но не определенный класс), объявленный как "`struct XxxxImpl* pimpl_;`", например "`class Map { private: struct MapImpl* pimpl_ };`".

3. Приведенный проект на самом деле может быть значительно усовершенствован. За что отвечает класс `GenericTableAlgorithm`? Если он решает несколько задач, то каким образом можно улучшить их инкапсуляцию? Поясните, как ваш ответ влияет на повторное использование класса и, в частности, на его расширяемость.

Класс `GenericTableAlgorithm` может быть значительно усовершенствован, поскольку в настоящий момент он выполняет две задачи. Человек испытывает определенный стресс, если ему необходимо решать одновременно две задачи, в частности, из-за возрастающей нагрузки и ответственности. Так и рассматриваемый класс только выиграет от разделения стоящих перед ним задач между двумя классами.

В исходном варианте `GenericTableAlgorithm` загружен двумя различными несвязанными задачами, которые могут быть легко разделены.

- Код клиента ИСПОЛЬЗУЕТ (специализированный соответствующим образом) обобщенный алгоритм.
- `GenericTableAlgorithm` ИСПОЛЬЗУЕТ специализированный конкретный класс "деталей", определяющий его действия в конкретной ситуации.

Рекомендация

Каждая часть кода — каждый модуль, класс, функция — должны отвечать за выполнение единой, четко определенной задачи.

Вот как выглядит несколько усовершенствованный код.

```
// ------------------------------------------
// Файл gta.h
// ------------------------------------------
// Задача 1. Обеспечивает открытый интерфейс, который
// инкапсулирует общую функциональность методом шаблона.
// Данный класс не требует наследования и может быть
// легко изолирован. Целевая аудитория - внешние
// пользователи класса GenericTableAlgorithm.
//
class GTAClient;

class GenericTableAlgorithm
{
public:
    // Конструктор в качестве параметра получает
    // объект с конкретной реализацией.
    //
    GenericTableAlgorithm( const string& table,
                           GTAClient&    worker );

    // Поскольку мы устранили необходимость в
    // наследовании, деструктор может не быть
    // виртуальным.
    //
    ~GenericTableAlgorithm();

    bool Process();    // Без изменений

private:
    struct GenericTableAlgorithmImpl* pimpl_;
};

// ------------------------------------------
// Файл gtaclient.h
// ------------------------------------------
// Задача 2. Обеспечивает расширяемый абстрактный
// интерфейс. Здесь содержатся детали реализации
// класса GenericTableAlgorithm, которые легко
// выделяются в отдельный абстрактный класс протокола.
// Целевая аудитория этого класса – разработчики
// конкретного класса, работающего с классом
// GenericTableAlgorithm и определяющего его действия.
//
class GTAClient
{
public:
    virtual GTAClient() = 0;
    virtual bool Filter( const Record& );
    virtual bool ProcessRow( const PrimaryKey& ) = 0;
};

// ------------------------------------------
// Файл gtaclient.cpp
// ------------------------------------------
bool GTAClient::Filter( const Record& )
{
    return true;
}
```

Эти два класса, как видите, должны находиться в разных заголовочных файлах. Как же после внесения этих изменений будет выглядеть код клиента? Да практически так же, как и раньше.

```
class MyWorker : public GTAClient
{
    // Замещаем Filter() и ProcessRow() для
    // реализации конкретных операций
};

int main()
{
    GenericTableAlgorithm a( "Customer", MyWorker() );
    a.Process();
}
```

Хотя код практически не изменился, следует рассмотреть три важных следствия усовершенствования проекта.

1. Что произойдет при изменении открытого интерфейса `GenericTableAlgorithm`? В исходной версии потребуется перекомпиляция всех конкретных рабочих классов, поскольку они являются производными от `GenericTableAlgorithm`. В усовершенствованной версии все изменения открытого интерфейса `GenericTableAlgorithm` надежно изолированы и никак не влияют на конкретные рабочие классы.
2. Что произойдет при изменении расширяемого протокола `GenericTableAlgorithm` (например, при добавлении аргументов по умолчанию к функциям `Filter()` или `ProcessRow()`)? В исходной версии требуется перекомпиляция всех внешних клиентов `GenericTableAlgorithm` (несмотря на то, что открытый интерфейс остался неизменным), поскольку производный интерфейс видим в определении класса. В усовершенствованной версии все изменения интерфейса протокола `GenericTableAlgorithm` надежно изолированы и никак не влияют на внешних пользователей.
3. Любой конкретный рабочий класс теперь может быть использован не только в алгоритме `GenericTableAlgorithm`, но и в любом другом, который работает с использованием интерфейса `Filter()/ProcessRow()`. То, что мы получили в результате усовершенствования, очень напоминает стратегический шаблон проектирования (Strategy pattern).

Запомните один из лозунгов информатики: множество проблем могут быть решены, если ввести дополнительный уровень косвенности. Конечно, этот принцип необходимо сочетать с бритвой Оккама: “не преумножайте сущности сверх необходимости”. В нашем случае правильный баланс между этими двумя принципами повышает используемость и облегчает поддержку полученного решения за очень небольшую цену.

Попробуем еще в большей степени обобщить наше решение. Вы могли заметить, что `GenericTableAlgorithm` может в действительности быть не классом, а функцией (возможно, некоторым легче будет понять это, если функцию `Process()` переименовать в `operator()()`). Причина, по которой класс можно заменить функцией, состоит в том, что в описании задачи ничего не говорится о необходимости сохранения состояния между вызовами `Process()`. Поэтому мы можем изменить наш код следующим образом.

```
bool GenericTableAlgorithm(
    const string& table,
    GTAClient& method)
{
    // Здесь размещается код бывшей
    // функции Process()
}
```

```
int main()
{
    GenericTableAlgorithm("Customer", MyWorker());
}
```

Итак, мы получили обобщенную функцию, поведение которой может быть “специализировано”. Если известно, что объекту method никогда не потребуется хранение состояния (т.е. все экземпляры функционально эквивалентны и содержат только данные виртуальные функции), то method можно сделать параметром шаблона.

```
template<typename GTACworker>
bool GenericTableAlgorithm( const string& table )
{
    // Здесь размещается код бывшей
    // функции Process()
}

int main()
{
    GenericTableAlgorithm<MyWorker>("Customer");
}
```

Впрочем, я не думаю, что добавление шаблона подкупит вас и вы предпочтете использовать его, только чтобы избавиться от одной запятой в коде, так что предыдущее решение явно лучше. Словом, еще один принцип состоит в том, чтобы не поддаваться искушению написать чересчур вычурный код.

В любом случае, использовать ли в данной ситуации функцию или класс зависит от того, чего именно вы пытаетесь добиться; в данном конкретном случае лучшим решением представляется написание обобщенной функции.

Задача 3.5. Наследование: потребление и злоупотребление ***Сложность: 6***

Почему наследование?
Наследованием часто чрезмерно увлекаются даже опытные разработчики. Однако следует всегда минимизировать взаимосвязи объектов. Если взаимоотношение между классами может быть выражено несколькими способами, следует использовать тот, который при сохраняющейся практичности приводит к наиболее слабым связям. Наследование, будучи одной из самых сильных связей (второй после отношения дружбы), которые можно выразить средствами C++, может применяться только тогда, когда нет эквивалентной альтернативы с более слабым взаимодействием.
В данной задаче нас интересуют вопросы наследования. Попутно мы создадим практически полный список всех обычных и необычных причин для использования наследования.

Приведенный шаблон предоставляет функции управления списком, включая возможность работы с элементами списка в определенных позициях.

```
// Пример 1
//
template<class T>
class MyList
{
public:
    bool   Insert( const T&, size_t index );
    T      Access( size_t index ) const;
    size_t Size() const;
private:
    T*     buf_;
    size_t bufsize_;
};
```

Рассмотрим следующий код, демонстрирующий два способа написания класса `MySet` с использованием `MyList`. Предполагается, что показаны все важнейшие элементы класса.

```
// Пример 1а
//
template<class T>
class MySet1 : private MyList<T>
{
public:
    bool Add( const T& );           // Вызывает Insert()
    T    Get( size_t index ) const; // Вызывает Access()
    using MyList<T>::Size;
    // ...
};

// Пример 1б
//
template<class T>
class MySet2
{
public:
    bool   Add( const T& );  // Вызывает impl_.Insert()
    T      Get( size_t index const );
                             // Вызывает impl_.Access()
    size_t Size() const;     // Вызывает impl_.Size()
    // ...
private:
    MyList<T> impl_;
};
```

Подумайте над этими альтернативами и ответьте на следующие вопросы.

1. Имеется ли различие между `MySet1` и `MySet2`?
2. Говоря более обобщенно, в чем заключается различие между закрытым наследованием и включением? Приведите по возможности более полный список причин, по которым могло бы потребоваться использовать наследование, а не включение.
3. Какую бы из версий вы предпочли — `MySet1` или `MySet2`?
4. И наконец, приведите по возможности более полный список причин, по которым могло бы потребоваться открытое наследование.

Решение

Приведенный пример помогает проиллюстрировать некоторые вопросы, сопровождающие использование наследования, в особенности вопросы выбора между закрытым наследованием и включением.

1. Имеется ли различие между `MySet1` и `MySet2`?

Ответ на этот вопрос прост — между `MySet1` и `MySet2` нет никакого существенного различия. Функционально они идентичны.

2. Говоря более обобщенно, в чем заключается различие между закрытым наследованием и включением? Приведите по возможности более полный список причин, по которым могло бы потребоваться использовать наследование, а не включение.

Этот вопрос приводит нас к серьезному рассмотрению сути наследования и включения.

- *Закрытое наследование* всегда должно выражать отношение РЕАЛИЗОВАН ПОСРЕДСТВОМ (за одним редким исключением, о котором я скажу чуть позже). Оно делает использующий класс зависящим от открытых *и защищенных* частей используемого класса.
- *Включение* всегда выражает отношение СОДЕРЖИТ и, следовательно, РЕАЛИЗОВАН ПОСРЕДСТВОМ. Включение делает использующий класс зависящим только от открытых частей используемого класса.

Можно легко показать, что наследование является надмножеством над одинарным включением, т.е. нет ничего такого, что мы могли бы сделать с одиночным членом `MyList<T>` и не могли бы при наследовании от `MyList<T>`. Конечно, использование наследования ограничивает нас, позволяя иметь только один объект `MyList<T>` (в качестве базового подобъекта); если нам требуется несколько экземпляров `MyList<T>`, мы должны использовать не наследование, а включение.

Рекомендация

Предпочтительно использовать включение (делегирование, отношение СОДЕРЖИТ, композицию, агрегирование и т.д.), а не наследование. При работе с моделью РЕАЛИЗОВАН ПОСРЕДСТВОМ всегда используйте выражение посредством включения, а не наследования.

В таком случае, что же мы можем сделать такого при наследовании, чего не в состоянии сделать при включении? Другими словами, для чего мы можем использовать закрытое наследование? Вот несколько причин, приведенных в порядке уменьшения их важности. Интересно, что последний пункт указывает одно полезное применение закрытого наследования.

- *Нам требуется замещение виртуальной функции.* Зачастую нам требуется такое замещение для того, чтобы изменить поведение используемого класса. Впрочем, иногда у нас не остается выбора: если используемый класс абстрактный, т.е. имеет как минимум одну чисто виртуальную функцию, которая не была замещена, мы обязаны использовать наследование и замещение этой функции, поскольку непосредственно создать объект такого класса невозможно.
- *Нам требуется доступ к защищенным членам.* Эта причина применима к защищенным функциям-членам[7] вообще и к защищенным конструкторам в частности.
- *Нам требуется создание используемого объекта до (или уничтожение его после) другого базового подобъекта.* Если требуется немного большее время жизни объекта, то решить этот вопрос можно только путем наследования. Это может оказаться необходимым, когда используемый класс обеспечивает какой-то вид блокировки, необходимой для работы критического раздела или транзакции базы данных, которые должны охватывать все время жизни базового подобъекта.
- *Нам требуется совместное использование общего виртуального базового класса или замещение конструктора виртуального базового класса.* Первая часть применима, когда использующий класс оказывается наследником того же виртуального базового класса, что и используемый класс. Если даже это не так, возможна вторая ситуация. Последний в иерархии производный класс отвечает за инициализацию всех виртуальных базовых классов, так что если нам требуется использование другого конструктора или конструктора с другими параметрами для виртуального базового класса, то мы должны использовать наследование.

[7] Я говорю о функциях-членах, поскольку вы никогда не напишете класс с открытыми или защищенными членами-переменными, не так ли?

- *Получение существенной выгоды от оптимизации пустого базового класса.* Иногда класс, который РЕАЛИЗОВАН ПОСРЕДСТВОМ, может вовсе не иметь членов данных, представляя собой набор функций. В этом случае, если использовать наследование вместо включения, можно получить определенную выгоду от оптимизации пустого базового класса. То есть, компилятор позволяет пустому базовому подобъекту занимать нулевое количество памяти, в то время как пустой объект-член должен занимать ненулевую память, даже если он и не содержит никаких данных.

```
class B { /* ... только функции, данных нет ... */ };
// Включение: приводит к некоторому перерасходу памяти
//
class D
{
    B b_; // b_ должен занимать по крайней мере один байт,
};        // несмотря на то, что реально это пустой класс

// Наследование: не приводит к перерасходу памяти
//
class D : private B
{    // Базовый подобъект B не требует для себя памяти
};
```

  Более подробно вопрос оптимизации пустых базовых классов рассмотрен в статье Натана Майерса (Nathan Myers) в *Dr. Dobb's Journal* [Myers97].

Позвольте добавить предупреждение для чрезмерно усердных читателей: в действительности не все компиляторы выполняют оптимизацию пустого базового класса. Но даже если она выполняется, то, вероятно, значительного выигрыша вы не получите (конечно, если только для вашей системы не требуется одновременного наличия десятков тысяч таких объектов. Введение излишней связи в результате использования наследования вместо включения оправданно, только если для вашего приложения очень важна экономия памяти, а вы твердо знаете, что ваш компилятор выполняет оптимизацию пустых базовых классов.

Имеется еще одна возможность, когда допустимо использование закрытого наследования, не соответствующая модели РЕАЛИЗОВАН ПОСРЕДСТВОМ.

- *Нам требуется "управляемый полиморфизм" — ЯВЛЯЕТСЯ из принципа подстановки Лисков — но только на определенном участке кода.* Открытое наследование, согласно принципу подстановки Лисков, всегда должно моделировать отношение ЯВЛЯЕТСЯ.[8] Закрытое наследование может выражать ограниченную форму этого отношения, хотя большинство людей отождествляет его только с открытым наследованием. Данный `class Derived: private Base` с точки зрения внешнего кода НЕ ЯВЛЯЕТСЯ `Base`. Конечно, он не может полиморфно использоваться в качестве `Base` из-за ограничений доступа, к которым приводит закрытое наследование. Однако *внутри* функций-членов и друзей `Derived` объект этого типа может полиморфно использоваться как объект типа `Base` (вы можете использовать указатель или ссылку на объект `Derived` там, где ожидается объект `Base`), поскольку члены и друзья обладают необходимыми правами доступа. Если вместо закрытого наследования вы используете защищенное, то отношение ЯВЛЯЕТСЯ будет видимым и для прочих производных классов в иерархии, что означает, что подклассы также смогут использовать полиморфизм.

[8] По адресу `http://www.gotw.ca/publications/xc++/om.htm` можно найти ряд статей, описывающих LSP.

На этом заканчивается список причин, которые я могу привести в защиту использования закрытого наследования.

Рассмотренный материал приводит нас к третьему вопросу.

3. Какую бы из версий вы предпочли — `MySet1` или `MySet2`?

Давайте проанализируем код из примера 1 и посмотрим, какие из приведенных выше критериев применимы в данном случае.

- `MyList` не имеет защищенных членов, так что нам не требуется наследование для доступа к ним.
- `MyList` не имеет виртуальных функций, так что нам не требуется наследование для их замещения.
- `MySet` не имеет других потенциальных базовых классов, так что объект `MyList` не обязан создаваться до или уничтожаться после базового подобъекта.
- `MyList` не имеет виртуальных базовых классов, совместное использование или замещение конструкторов которых могло бы потребоваться классу `MySet`.
- `MyList` не пуст, так что обоснование “оптимизация пустого класса” в данном случае неприменимо.
- `MySet` НЕ ЯВЛЯЕТСЯ `MyList`, в том числе внутри функций-членов или друзей `MySet`. Этот момент особенно интересен, поскольку указывает на (небольшой) недостаток наследования. Даже если бы один из остальных критериев оказался верен, так что мы бы использовали наследование, нам бы пришлось заботиться о том, чтобы случайно не использовать `MySet` полиморфно как `MyList`, — маловероятная, но вполне возможная ситуация, которая бы надолго поставила в тупик неопытного программиста.

Короче говоря, `MySet` не должен наследовать `MyList`. Использование наследования там, где включение не менее эффективно, только приводит к необоснованно сильным взаимосвязям и ненужным зависимостям. К сожалению, в реальной жизни довольно часто даже опытные программисты реализуют взаимоотношения, аналогичные показанным в примере, используя наследование.

Проницательные читатели заметят, что версия с наследованием все же имеет одно преимущество перед включением: при наследовании достаточно объявления `using`, чтобы использовать неизменную (унаследованную) функцию `Size`. В случае включения вы должны явно реализовать эту функцию.

Конечно, иногда наследование совершенно необходимо.

```
// Пример 2. Необходимо наследование
//
class Base
{
public:
    virtual int Func1();
protected:
    bool Func2();
private:
    bool Func3(); // Использует Func1
};
```

Если нам требуется заместить виртуальную функцию типа `Func1()` или нужен доступ к защищенному члену типа `Func2()`, наследование необходимо. Пример 2 показывает, для чего, кроме обеспечения полиморфизма, может потребоваться замещение виртуальной функции. В данном примере `Base` реализуется посредством `Func1()` (`Func3()` использует `Func1()` в своей реализации), так что единственный путь обес-

печения корректного поведения состоит в замещении `Func1()`. Но даже при необходимости наследования, как правильно его реализовать?

```
// Пример 2а
//
class Derived: private Base
{
public:
    int Func1();
    // Прочие функции. Некоторые из них
    // используют Base::Func2(), некоторые - нет
};
```

Данный код позволяет заместить `Base::Func1()`, что, конечно же, хорошо. Но, к сожалению, он также открывает доступ к `Base::Func2()` *всем членам класса* `Derived`, а это уже неприятно. Доступ к `Base::Func2()` может требоваться только нескольким, или и вовсе одной функции-члену `Derived`. Но при таком использовании наследования мы вынуждены делать все члены класса `Derived` зависящими от защищенного интерфейса `Base`.

Понятно, что наследование необходимо, но как получить только минимально необходимую зависимость? Для этого надо всего лишь немного видоизменить проект.

```
// Пример 2б
//
class DerivedImpl: private Base
{
public:
    int Func1();
    // Функции, использующие Func2
};

class Derived
{
    // Функции, не использующие Func2
private:
    DerivedImpl impl_;
};
```

Такой вариант гораздо лучше, поскольку он явно выделяет и инкапсулирует зависимости от `Base`. `Derived` непосредственно зависит только от открытого интерфейса `Base` и от открытого интерфейса `DerivedImpl`. Почему такой вариант гораздо лучше? В первую очередь потому, что он следует фундаментальному принципу “один класс, одна задача”. В примере 2*а* `Derived` отвечает как за создание `Base`, так и за собственную реализацию посредством `Base`. В примере 2*б* эти задачи отделены друг от друга и размещены в разных классах.

Поговорим теперь о включении. Включение обладает рядом преимуществ. Во-первых, оно легко позволяет иметь несколько экземпляров используемого класса. Если необходимо иметь несколько экземпляров базового класса в случае наследования, следует прибегнуть к той же идиоме, что и в примере 2*б*. Из базового следует вывести вспомогательный класс (типа `DerivedImpl`) для решения задачи наследования, а затем включить в окончательный класс требующееся количество экземпляров вспомогательного класса.

Во-вторых, наличие у используемого класса членов-данных придает дополнительную гибкость. Член может быть скрыт за указателем на реализацию[9] (в то время как определение базового класса всегда должно быть видимо) и легко преобразован в указатель при необходимости внесения изменений во время выполнения программы (в то время как иерархия наследования статична и фиксируется во время компиляции).

---

[9] См. задачи 4.1—4.5.

И наконец, еще один способ написания класса MySet2 из примера 1*б*, использующий более общий способ включения.

```
// Пример 1в. Обобщенное включение
//
template<class T, class Impl = MyList<T> >
class MySet3
{
public:
    bool   Add( const T& ); // Вызывает impl_.Insert()
    T      Get( size_t index ) const;
                               // Вызывает impl_.Access()
    size_t Size() const;    // Вызывает impl_.Size()
    // ...
private:
    Impl impl_;
};
```

Теперь вместо отношения РЕАЛИЗОВАН ПОСРЕДСТВОМ MyList<T> мы получили более гибкое отношение: MySet РЕАЛИЗУ*ЕМ* ПОСРЕДСТВОМ любого класса, который поддерживает функции Add, Get и другие, которые нам требуются. Стандартная библиотека C++ использует эту технологию в своих шаблонах stack и queue, которые по умолчанию РЕАЛИЗОВАНЫ ПОСРЕДСТВОМ deque, но при этом РЕАЛИЗУЕМЫ ПОСРЕДСТВОМ любого другого класса, который в состоянии предоставить необходимый сервис.

В частности, различные пользователи могут создать объект класса MySet с использованием реализаций с разными характеристиками производительности, например, если мне известно, что мой код будет выполнять больше вставок, чем поисков, я буду использовать реализацию, которая оптимизирована для быстрого выполнения вставок. Простота же использования класса при этом не теряется. При работе с примером 1*б* в коде клиента для создания объекта множества целых чисел используется MySet2<int>; та же запись может использоваться и при работе с примером 1*в*, так как MySet3<int> является ни чем иным, как синонимом MySet3<int,MyList<int> > в силу использования параметра шаблона по умолчанию.

Такую гибкость очень сложно получить при использовании наследования, в первую очередь потому, что наследование приводит к фиксации реализации на этапе проектирования. Пример 1*в* можно переписать с применением наследования от Impl, но такое решение приведет к появлению излишних зависимостей, и потому должно быть отвергнуто.

4. И наконец, приведите по возможности более полный список причин, по которым могло бы потребоваться открытое наследование.

Говоря об открытом наследовании, я хотел бы особо выделить одно правило, следование которому позволит вам избежать многих злоупотреблений: используйте открытое наследование *только* для моделирования истинного отношения ЯВЛЯЕТСЯ в соответствии с принципом подстановки Лисков.[10] Это означает, что объект открытого производного класса может использоваться в любом контексте вместо объекта базового класса и при этом гарантировать соблюдение исходной семантики. (Об одном редком исключении (вернее, расширении) мы говорили в задаче 1.3.)

Следование этому правилу позволяет, в частности, избежать двух распространенных ловушек.

- *Никогда не используйте открытое наследование там, где достаточно закрытого.* Открытое наследование никогда не должно применяться для моделирования

[10] По адресу http://www.gotw.ca/publications/xc++/om.htm можно найти ряд статей, описывающих LSP.

отношения РЕАЛИЗОВАН ПОСРЕДСТВОМ без истинного отношения ЯВЛЯЕТСЯ. Это может показаться очевидным, но я не раз замечал, что некоторые программисты используют открытое наследование просто по привычке. Хорошей такую привычку не назовешь, так же как и привычку использовать наследование (любого вида) там, где можно обойтись включением. Если излишние права доступа и более тесная связь не нужны, зачем их навязывать? Изо всех возможных путей выражения взаимоотношения классов выбирайте путь с наименьшей степенью взаимосвязи.

- *Никогда не используйте открытое наследование для реализации отношения "ЯВЛЯЕТСЯ ПОЧТИ".* Мне встречались программисты (и даже весьма опытные!), использующие открытое наследование и реализацию большинства замещенных виртуальных функций таким образом, чтобы сохранить семантику базового класса. Другими словами, в некоторых случаях использование объекта `Derived` вместо объекта `Base` не приводит к тому поведению, на которое рассчитывает клиент класса `Base`. В качестве примера можно привести наследование класса `Square` (квадрат) от класса `Rectangle` (прямоугольник), базирующееся на том, что "квадрат есть прямоугольник". Это может быть верно в математике, но это еще не означает, что то же верно и в случае классов. Например, у класса `Rectangle` может оказаться виртуальная функция `SetWidth(int)`. Естественно ожидать, что реализация этой функции в классе `Square` будет устанавливать и ширину, и высоту, чтобы квадрат оставался квадратом. Но ведь в общем случае такое поведение для класса `Rectangle` некорректно! Такое наследование — хороший пример открытого наследования, нарушающего принцип подстановки Лисков, поскольку порожденный класс изменяет семантику базового класса. Здесь нарушено ключевое правило открытого наследования: "Все замещающие функции-члены при этом не должны предъявлять повышенные требования к условиям своей работы и не должны иметь пониженную функциональность по сравнению с оригиналом".

Когда я вижу программистов, реализующих таким образом отношение "ЯВЛЯЕТСЯ ПОЧТИ", я обычно указываю им на то, что они, грубо говоря, нарываются на неприятности. В конце концов кто-то где-то когда-нибудь попытается полиморфно использовать объект и получит неожиданные результаты, не так ли? "Да все нормально, — ответил мне однажды такой программист. — Это всего лишь маленькая несовместимость, и я знаю, что никто не будет использовать объекты семейства `Base` таким опасным способом". Однако "быть немножко несовместимым" — это почти то же самое, что и "быть немножко беременной". У меня нет причин сомневаться в словах этого программиста, а именно в том, что никакой код в его системе не приведет к опасным последствиям. Но у меня нет причин считать, что никогда не наступит один прекрасный день, когда какой-то программист, поддерживающий данную программу, не внесет небольшое безобидное изменение, после чего долгие часы будет вынужден провести за анализом проектирования классов и многие дни — за решением этой проблемы.

Не ищите легкие пути. Просто скажите *нет*. Если что-то ведет себя не так, как `Base`, значит, это НЕ ЯВЛЯЕТСЯ `Base`, так что не используйте наследование и не делайте его `Base` насильственно.

Рекомендация

Всегда убеждайтесь, что открытое наследование, в соответствии с принципом подстановки Лисков, моделирует как отношение ЯВЛЯЕТСЯ, так и РАБОТАЕТ ПОДОБНО. Все замещающие функции-члены при этом не должны предъявлять повышенные требования к условиям своей работы и не должны иметь пониженную функциональность по сравнению с оригиналом.

Рекомендация

Никогда не используйте открытое наследование для повторного использования кода (базового класса); используйте открытое наследование только для того, чтобы быть повторно использованным кодом, полиморфно использующим базовые объекты.

Заключение

Разумно используйте наследование. Если вы можете выразить взаимоотношения классов путем включения/делегирования, вы должны предпочесть именно этот путь. Если вам требуется использовать наследование, но не моделировать отношение ЯВЛЯЕТСЯ, используйте закрытое наследование. Если комбинаторная мощь множественного наследования не является необходимой, используйте одиночное наследование. Большие иерархические структуры вообще и глубокие в особенности усложняют понимание и, соответственно, трудно поддерживаются. Наследование является решением времени проектирования.

Некоторым программистам кажется, что если нет наследования, то нет и объектной ориентированности, но это не так. Используйте простейшее из работающих решений, и ваш код будет более стабилен, легче сопровождаем и надежен.

Задача 3.6. Объектно-ориентированное программирование ***Сложность: 4***

Является ли C++ объектно-ориентированным языком? И да, и нет, вопреки распространенному мнению. Данная маленькая задача поможет вам понять это утверждение. Не спешите быстро дать ответ на вопрос и перейти к другим задачам — сначала немного подумайте.

"C++ представляет собой мощный язык программирования, поддерживающий множество передовых объектно-ориентированных конструкций, включая инкапсуляцию, обработку исключений, наследование, шаблоны, полиморфизм, строгую типизацию и модульность".

Обсудите это утверждение.

Решение

Цель данной задачи — заставить задуматься об основных (и даже отсутствующих) возможностях C++ и приблизить вас к реальности. В частности, я надеюсь, что размышления над этой задачей приведут вас к генерации идей и примеров, иллюстрирующих три основных положения.

1. *Не все согласны с тем, что именно обозначает "объектно-ориентированный".* Так что же такое объектная ориентированность? Даже сейчас, при широкой ее распространенности, если вы опросите 10 человек, то получите 15 ответов (впрочем, это немногим лучше, чем спросить совета у 10 адвокатов).

   Почти все согласятся, что наследование и полиморфизм являются объектно-ориентированными концепциями. Многие добавят сюда инкапсуляцию. Некоторые присовокупят обработку исключений. О шаблонах при этом вряд ли кто-то вспомнит. Мораль в том, что имеются различные мнения об отношении каждой конкретной возможности к объектно-ориентированному программированию, и каждая точка зрения имеет своих горячих защитников.

2. *C++ поддерживает множество парадигм.* C++ — не только объектно-ориентированный язык. Он поддерживает множество объектно-ориентированных возможностей, но не заставляет программиста использовать их. На C++ вы можете писать завершенные программы без использования ОО, и многие именно так и поступают.

   Наиболее важный вклад в стандартизацию C++ состоит в обеспечении строгой поддержки *сильного абстрагирования* для снижения сложности программного обеспечения [Martin95].[11] C++ не является исключительно объектно-ориентированным языком программирования. Он поддерживает несколько стилей программирования, включая как объектно-ориентированное, так и обобщенное программирование. Эти стили фундаментально важны, поскольку каждый из них обеспечивает гибкие способы организации кода посредством абстракций. Объектно-ориентированное программирование позволяет нам группировать состояние объекта вместе с функциями, управляющими им. Инкапсуляция и наследование обеспечивают возможность управления зависимостями и облегчают повторное использование кода. Обобщенное программирование представляет собой более современный стиль, который позволяет нам писать функции и классы, которые оперируют другими функциями и объектами неопределенных, не связанных и неизвестных типов, обеспечивая уникальный способ ослабления связей и зависимостей в программе. В настоящее время только некоторые языки обеспечивают поддержку обобщенности, но ни один не делает это так мощно, как C++. По сути современное обобщенное программирование стало возможно благодаря шаблонам C++.

   На сегодняшний день C++ обеспечивает множество мощных средств выражения абстракций, и получающаяся в результате гибкость является наиболее важным следствием стандартизации C++.

3. *Ни один язык не может быть абсолютно универсальным.* Сегодня я использую C++ как основной язык программирования; завтра я буду использовать тот язык, который будет в наибольшей степени подходить для решения моих задач. C++ не поддерживает понятие модулей в явном виде, в нем недостает некоторых других возможностей (например, сборки мусора), он обладает статической, но не обязательно “строгой” типизацией. Все языки имеют свои преимущества и свои недостатки. Просто выберите подходящий инструмент для работы над своими задачами, и избегайте соблазна стать слепым фанатиком конкретного языка программирования.

Задача 3.7. Множественное наследование *Сложность: 6*

В некоторых языках, например в SQL99, продолжается борьба по вопросу поддержки множественного и одиночного наследования. Данная задача предлагает рассмотреть этот вопрос и вам.

1. Что такое множественное наследование и какие дополнительные возможности или сложности привносит множественное наследование в C++?
2. Является ли множественное наследование необходимым? Если да, то приведите как можно больше ситуаций, доказывающих необходимость поддержки множественного наследования в языке. Если нет, аргументируйте, почему

[11] В этой книге содержится превосходное обсуждение того, почему одно из наиболее важных преимуществ объектно-ориентированного программирования состоит в том, что оно позволяет нам снизить сложность программного обеспечения путем управления взаимозависимостью кода.

одиночное наследование, возможно, в сочетании с интерфейсами в стиле Java, не уступает множественному и почему в языке не должно быть множественного наследования.

Решение

1. Что такое множественное наследование и какие дополнительные возможности или сложности привносит множественное наследование в C++?

Вкратце, множественное наследование означает возможность наследования более чем от одного прямого базового класса. Например:

```
class Derived : public Base1, private Base2
{
    // ...
};
```

Множественное наследование позволяет классу иметь в качестве предка (прямого или непрямого) несколько раз один и тот же базовый класс. Простейший пример такого наследования показан на рис. 3.1. Здесь `B` является непрямым предком `D` дважды — через класс `C1` и класс `C2`.

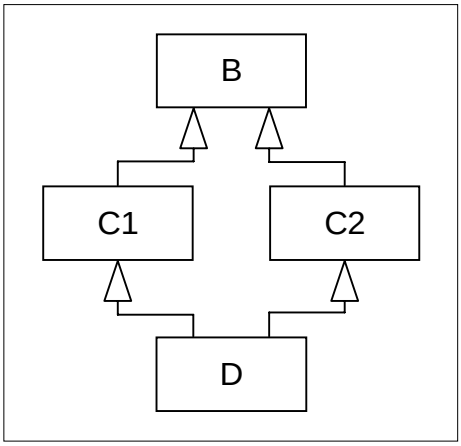


Рис. 3.1. Класс `B` дважды выступает в роли предка класса `D` (здесь показан случай виртуального наследования от `B`)

Эта ситуация приводит к необходимости использования другой возможности C++: виртуального наследования. Ключевым вопросом в данной ситуации является следующий: должен ли объект класса `D` иметь один или два подобъекта класса `B`? Если ответ — один, то класс `B` должен быть виртуальным базовым классом; если же требуется два подобъекта, то класс `B` должен быть обычным, не виртуальным базовым классом (эта ситуация показана на рис. 3.2).

Наконец, основная сложность виртуальных базовых классов состоит в том, что они должны быть непосредственно инициализированы в последнем производном классе иерархии. Дополнительную информацию по этому и другим аспектам множественного наследования можно найти в [Stroustrup00] или [Meyers97].

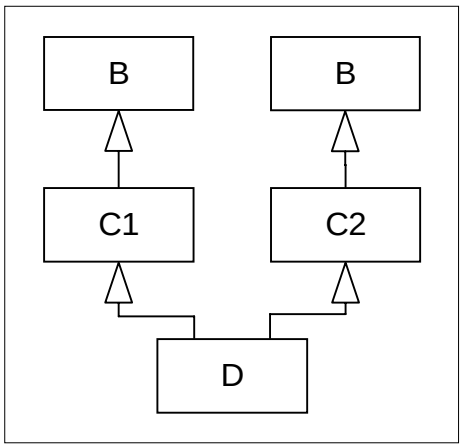


Рис. 3.2. Класс B дважды выступает в роли предка класса D (случай обычного наследования от B)

Рекомендация

Избегайте множественного наследования больше чем от одного непротокольного класса (под протокольным классом мы понимаем абстрактный базовый класс, состоящий исключительно из чисто виртуальных функций и не содержащий данных).

2. Является ли множественное наследование необходимым?

Коротко можно ответить так: ни одна возможность языка не является "необходимой", поскольку любая программа может быть написана на ассемблере или просто в машинных кодах. Однако так же как большинство программистов не пытаются реализовывать свой собственный механизм виртуальных функций на чистом C, так и отсутствие множественного наследования требует в ряде случаев использования тяжелых и трудоемких обходных путей.

Впрочем, множественное наследование у нас уже есть. Осталось выяснить, насколько множественное наследование Хорошая Вещь.[12]

Имеются люди, которые считают множественное наследование плохой идеей, они уверены, что его следует убрать из языка любой ценой. Конечно, бездумное употребление множественного наследования может привести к излишней сложности и взаимосвязи классов. Но хотя то же самое можно сказать и о бездумном применении наследования (см. задачу 3.5), вряд ли вы станете утверждать, что наследование следует убрать из языка. Да, конечно, любая программа может быть написана без использования множественного наследования, но по сути то же можно сказать и о наследовании вообще. В конце концов, теоретически можно написать программу и непосредственно в машинных кодах...

[12] Тема данной задачи, помимо прочего, была вызвана событиями на конференции по стандарту SQL в июне 1998 года, где множественное наследование было удалено из чернового варианта стандарта ANSI SQL99. Это было сделано, в первую очередь, из-за технических трудностей, связанных с предложенной спецификацией множественного наследования, а также для соответствия таким современным языкам, как Java, в которых множественного наследования нет. Было интересно побывать на этих заседаниях и послушать людей, горячо обсуждавших преимущества и недостатки множественного наследования, — это очень сильно напоминало давно прошедшие (впрочем, с завидной регулярностью возобновляющиеся и поныне) войны в группах новостей, посвященные множественному наследованию в C++.

Если да, то приведите как можно больше ситуаций, доказывающих необходимость поддержки множественного наследования в языке. Если нет, аргументируйте, почему одиночное наследование, возможно, в сочетании с интерфейсами в стиле Java, не уступает множественному и почему в языке не должно быть множественного наследования.

Итак, когда применение множественного наследования оправдано? Вкратце, когда оправдано каждое из индивидуальных наследований (в задаче 3.5 эти случаи перечислены достаточно подробно). Большинство реальных применений множественного наследования можно разделить на три категории.

1. *Комбинирование модулей или библиотек*. Многие классы изначально разрабатываются как базовые, т.е. для того, чтобы использовать их, вы должны наследовать от них. Естественно спросить: что делать, если вы хотите написать класс, который расширяет возможности двух библиотек, и каждая из них требует наследования от ее классов?

Когда вы оказываетесь в такой ситуации, обычно вы не в состоянии изменить код библиотеки, для того чтобы избежать одного из наследований. Возможно, вы купили эту библиотеку у стороннего производителя; возможно, она — результат работы другой команды программистов вашей же фирмы. В любом случае вы не только не можете изменить исходный код, но у вас вообще может его не быть! Если так, множественное наследование просто необходимо: другого (естественного) пути решения проблемы просто нет, и использование множественного наследования в этой ситуации вполне законно.

Я убедился на практике, что если вы программируете на C++, вы должны знать, как использовать множественное наследование для комбинирования библиотек сторонних производителей. Будете вы использовать его часто или нет — неважно; знать, как им пользоваться, вы должны в любом случае.

2. *Классы протоколов (интерфейсов)*. В C++ множественное наследование является лучшим и наиболее безопасным путем определения классов протоколов — т.е. классов, состоящих исключительно из виртуальных функций. Отсутствие членов данных в базовом классе полностью устраняет наиболее известные проблемы, связанные с использованием множественного наследования.

Интересно, что некоторые языки и языковые модели поддерживают этот тип множественного наследования посредством механизма, не использующего наследования. Примерами могут служить Java и COM. Строго говоря, в Java имеется множественное наследование, но наследование реализации ограничено одиночным наследованием. Класс Java может реализовать множественные “интерфейсы”, которые очень похожи на чисто абстрактные классы C++ без членов данных. COM не поддерживает концепцию наследования как таковую (хотя обычно при реализации объектов COM на C++ используется именно эта технология), но имеет понятие композиции интерфейсов, которые напоминают комбинацию интерфейсов Java и шаблонов C++.

3. *Простота (полиморфного) использования*. Применение наследования для того, чтобы позволить другому коду использовать производный объект там, где ожидается базовый, является одной из наиболее мощных концепций C++. В ряде случаев может оказаться очень полезным позволить использование одного и того же порожденного объекта вместо базовых объектов нескольких типов, и множественное наследование здесь незаменимо. Хорошим примером может служить проектирование классов исключений с использованием множественного наследования, описанное в [Stroustrup00, раздел 14.2.2].

В скобках заметим, что п. 3 в значительной степени перекрывается с пп. 1 и 2.

Вот о чем еще следует задуматься: не забывайте, что могут быть ситуации, когда наследование от двух различных базовых классов не является необходимым, но имеются различные причины для наследования от каждого из них по отдельности. Реализация отношения ЯВЛЯЕТСЯ в соответствии с LSP — не единственная причина для использования наследования. Например, классу может потребоваться закрытое наследование от класса `A` для получения доступа к его защищенным членам и в то же время — открытое наследование от класса `B` для полиморфной реализации его виртуальной функции.

Задача 3.8. Эмуляция множественного наследования *Сложность: 5*

Если использование множественного наследования невозможно, каким образом можно его эмулировать? Данная задача поможет вам понять, почему множественное наследование в C++ работает именно так, а не иначе. При решении задачи постарайтесь по возможности использовать максимально естественный синтаксис вызывающего кода.

Рассмотрим следующий пример.

```
class A
{
public:
    virtual ~A();
    string Name();
private:
    virtual string DoName();
};

class B1 : virtual public A
{
    string DoName();
};

class B2 : virtual public A
{
    string DoName();
};

A::~A() {}
string A::Name()    { return DoName(); }
string A::DoName()  { return "A"; }
string B1::DoName() { return "B1"; }
string B2::DoName() { return "B2"; }

class D : public B1, public B2
{
    string DoName() { return "D"; }
};
```

Покажите по возможности наилучший способ избежать множественного наследования, написав эквивалентный (или максимально приближенный к оригиналу) `class D` без использования множественного наследования. Каким образом можно достичь той же функциональности и применимости класса `D` при минимальном изменении синтаксиса вызывающего кода?

* * * * *

Можно начать решение с рассмотрения следующего тестового кода.

```
void f1( A&  x ) { cout << "f1:" << x.Name() << endl; }
void f2( B1& x ) { cout << "f2:" << x.Name() << endl; }
void f3( B2& x ) { cout << "f3:" << x.Name() << endl; }

void g1( A&  x ) { cout << "g1:" << x.Name() << endl; }
void g2( B1& x ) { cout << "g2:" << x.Name() << endl; }
```

```
void g3( B2& x ) { cout << "g3:" << x.Name() << endl; }

int main()
{
    D   d;
    B1* pb1 = &d;  // Преобразование D* -> B*
    B2* pb2 = &d;
    B1& rb1 = d;   // Преобразование D& -> B&
    B2& rb2 = d;

    f1( d );       // Полиморфизм
    f2( d );
    f3( d );

    g1( d );       // Срезка
    g2( d );
    g3( d );

                   // dynamic_cast/RTTI
    cout << ((dynamic_cast<D*>(pb1) != 0) ? "ok " : "bad " );
    cout << ((dynamic_cast<D*>(pb2) != 0) ? "ok " : "bad " );

    try
    {
        dynamic_cast<D&>(rb1);
        cout << "ok ";
    }
    catch(...)
    {
        cout << "bad ";
    }

    try
    {
        dynamic_cast<D&>(rb2);
        cout << "ok ";
    }
    catch(...)
    {
        cout << "bad ";
    }
}
```

Решение

```
class D : public B1, public B2
{
    string DoName() { return "D"; }
};
```

Покажите по возможности наилучший способ избежать множественного наследования, написав эквивалентный (или максимально приближенный к оригиналу) `class D` без использования множественного наследования. Каким образом можно достичь той же функциональности и применимости класса `D` при минимальном изменении синтаксиса вызывающего кода?

Имеется несколько стратегий, каждая со своими недостатками. Далее приведено решение, пожалуй, наиболее близкое к оригиналу.

```
class D : public B1
{
public:
    class D2 : public B2
```

```
    {
    public:
        void Set( D* d ) { d_ = d; }
    private:
        string DoName();
        D* d_;
    } d2_;

    D()                 { d2_.Set( this ); }

    D( const D& other ) : B1( other ), d2_( other.d2_ )
                        { d2_.Set( this ); }

    D& operator = ( const D& other )
                        {
                            B1::operator = ( other );
                            d2_ = other.d2_;
                            return *this;
                        }

    operator B2&()      { return d2_; }

    B2& AsB2()          { return d2_; }

private:
    string DoName()     { return "D"; }
};

string D::D2::DoName()  { return d_->DoName(); }
```

Перед тем как читать дальше, внимательно рассмотрите этот код и подумайте над предназначением каждого класса и функции.

Недостатки

Предложенное обходное решение выполняет по сути всю работу по реализации, автоматизации поведения и обеспечению используемости множественного наследования в той мере, в которой вы придерживаетесь строгой дисциплины написания тех частей, которые не являются полностью автоматизированными. В частности, данное решение имеет ряд недостатков, которые показывают, *что* именно не полностью автоматизируется.

- Наличие `operator B2&()` приводит к тому, что ссылки рассматриваются отдельно, не так, как указатели.
- Вызывающий код должен явно вызвать `D::AsB2()` для того, чтобы использовать `D` как `B2` (в тестовом коде это означает замену “`B2* pb2 = &d;`” инструкцией “`B2* pb2 = &d.AsB2();`”).
- Динамическое приведение `D*` к `B2*` не работает (теоретически возможно заставить его работать, переопределив с помощью директивы препроцессора вызов `dynamic_cast`, но это очень опасное решение).

Интересно заметить, что размещение объекта `D` в памяти такое же, каким должно быть и в случае множественного наследования. Это связано с тем, что мы пытаемся сымитировать множественное наследование, хотя и без всех тех “удобств”, которые предоставляет поддержка этой возможности, встроенная в язык программирования.

Обычно множественное наследование не требуется, но когда оно требуется, значит, оно *действительно необходимо.* Цель данной задачи — продемонстрировать, что наличие поддержки множественного наследования со стороны языка значительно проще и надежнее, чем использование обходных путей для достижения того же результата.

Задача 3.9. Множественное наследование и проблема сиамских близнецов

Сложность: 4

Замещение унаследованной виртуальной функции — дело несложное, но только до тех пор, пока вы не пытаетесь заместить виртуальную функцию, имеющую одну и ту же сигнатуру в двух базовых классах. Такое может случиться, даже если базовый класс не был разработан сторонним производителем! Так как же можно разделить такие "сиамские функции"?

Рассмотрим два следующих класса.

```
class BaseA
{
    virtual int ReadBuf( const char * );
    // ...
};

class BaseB
{
virtual int ReadBuf( const char * );
    // ...
};
```

Оба класса предназначены для использования в качестве базовых классов, но кроме этого они никак не связаны. Их функции `ReadBuf()` предназначены для выполнения различных задач, а сами классы разработаны разными сторонними производителями.

Покажите, каким образом следует написать открытый производный от классов `BaseA` и `BaseB` класс `Derived`, в котором независимо замещены обе функции `ReadBuf()`.

Решение

Цель данной задачи — показать возможные проблемы, возникающие при множественном наследовании, и эффективные пути их обхода. Итак, пусть мы используем две библиотеки различных разработчиков в одном и том же проекте. Разработчик А предоставил следующий базовый класс.

```
class BaseA
{
    virtual int ReadBuf( const char * );
    // ...
};
```

Предполагается, что вы будете использовать наследование от данного класса, вероятно, с замещением некоторых виртуальных функций, поскольку другие части библиотеки данного разработчика предполагают полиморфное использование объектов типа `BaseA`. Это обычная практика, в частности для случая расширяемых приложений, и ничего необычного или неверного здесь нет.

Ничего, пока вы не начнете использовать библиотеку разработчика В и не обнаружите в ней, к своему удивлению, следующее.

```
class BaseB
{
virtual int ReadBuf( const char * );
    // ...
};
```

"Ну, это просто совпадение", — подумаете вы. Но проблема не в том, что разработчик В предоставил класс, который требуется наследовать. Дело в том, что в пре-

доставленном разработчиком B базовом классе имеется в точности такая же виртуальная функция, как и в базовом классе разработчика A, однако выполняемые ими действия не имеют ничего общего.

Оба класса предназначены для использования в качестве базовых классов, но кроме этого они никак не связаны. Их функции `ReadBuf()` предназначены для выполнения различных задач, а сами классы разработаны разными сторонними производителями.

Покажите, каким образом следует написать открытый производный от классов `BaseA` и `BaseB` класс `Derived`, в котором независимо замещены обе функции `ReadBuf()`.

Проблема становится яснее, когда мы должны написать класс, который является наследником и `BaseA`, и `BaseB`, например, поскольку нам требуется объект, который бы мог полиморфно использоваться функциями из библиотек обоих разработчиков. Вот первая, наивная попытка решить проблему.

```
// Пример 1. Попытка №1, неверная
//
class Derived : public BaseA, public BaseB
{
    // ...
    int ReadBuf( const char* )
        // Замещает как BaseA::ReadBuf(),
        // так и BaseB::ReadBuf()
};
```

Здесь `Derived::ReadBuf()` замещает как `BaseA::ReadBuf()`, так и `BaseB::ReadBuf()`. Для того чтобы понять, почему в нашем случае этого недостаточно, рассмотрим следующий код.

```
// Пример 1a. Контрпример, доказывающий
//            неработоспособность примера 1
//
Derived d;
BaseA*  pba = d;
BaseB*  pbb = d;

pba->ReadBuf("sample buffer");
    // Вызов Derived::ReadBuf()

pbb->ReadBuf("sample buffer");
    // Вызов Derived::ReadBuf()
```

Видите, в чем проблема? `ReadBuf()` является виртуальной функцией в обоих интерфейсах и работает полиморфно, как мы и ожидали. Но независимо от того, какой из интерфейсов мы используем, вызывается *одна и та же* функция `Derived::ReadBuf()`. Но `BaseA::ReadBuf()` и `BaseB::ReadBuf()` имеют различную семантику и предназначены для решения совершенно разных, а не одной и той же задачи. Кроме того, у нас нет никакой возможности выяснить внутри функции `Derived::ReadBuf()`, была ли она вызвана через интерфейс `BaseA` или `BaseB` (если она вызвана через один из них), так что мы не можем использовать в функции конструкцию `if` для разделения функциональности в зависимости от того, как вызывается данная функция.

“Это наверняка надуманная проблема, которая в реальной жизни вряд ли когда-нибудь встретится”, — можете подумать вы и будете неправы. Например, Джон Кдллин (John Kdllin) из Microsoft сообщил о проблеме при создании класса, производного от интерфейсов COM `IOleObject` и `IConnectPoint` (рассматривая их как абстрактные базовые классы, состоящие только из открытых виртуальных функций), поскольку а) оба интерфейса содержали функцию-член, объявленную как `virtual HRESULT Unadvise(unsigned long);`, и б) обычно требуется замещение каждой из `Unadvise()` для выполнения своих специфичных действий.

Остановитесь на минутку и задумайтесь над этим примером. Как бы вы решили эту проблему? Имеется ли какой-либо способ раздельного замещения функций `ReadBuf` так, чтобы при вызове через интерфейсы `BaseA` или `BaseB` выполнялись корректные действия? Короче говоря, как нам разделить сиамских близнецов?

Как разделить сиамских близнецов

К счастью, имеется довольно простое и ясное решение. Проблема заключается в том, что две замещаемые функции имеют одно и то же имя и сигнатуру. Соответственно, решение должно заключаться в том, чтобы изменить как минимум сигнатуру одной функции, а простейший способ изменения сигнатуры — это изменение имени функции.

Каким же образом можно изменить имя функции? Само собой, посредством наследования! Все, что нам для этого надо, — это промежуточный класс, производный от базового, который объявляет новую виртуальную функцию и замещает унаследованную версию вызовом новой функции. Иерархия наследования при этом выглядит так, как показано на рис. 3.3.

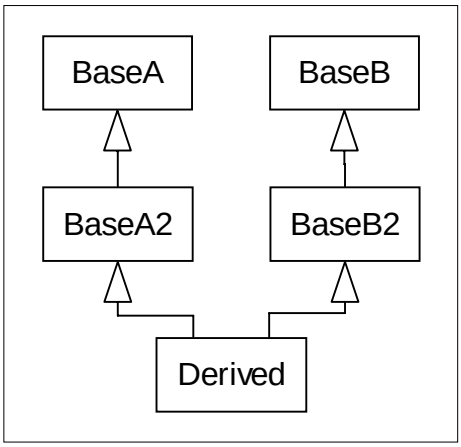


Рис. 3.3. Использование промежуточных классов для переименования унаследованных виртуальных функций

Код при этом будет выглядеть следующим образом.

```
// Пример 2. Попытка №2, корректная
//
class BaseA2 : public BaseA
{
public:
    virtual int BaseAReadBuf( const char* p ) = 0;
private:
    int ReadBuf( const char* p)   // Замещение наследованной
    {
        return BaseAReadBuf( p ); // Вызов новой функции
    }
};

class BaseB2 : public BaseB
{
public:
    virtual int BaseBReadBuf( const char* p ) = 0;
private:
```

```
    int ReadBuf( const char* p)   // Замещение наследованной
    {
        return BaseBReadBuf( p ); // Вызов новой функции
    }
};

class Derived : public BaseA2, public BaseB2
{
    /* ... */
public: // Или private: - в зависимости от того, должен ли
        // код быть доступен для непосредственного вызова
    int BaseAReadBuf( const char* );
        // Замещает BaseA::ReadBuf косвенно,
        // через BaseA2::BaseAReadBuf
    int BaseBReadBuf( const char* );
        // Замещает BaseB::ReadBuf косвенно,
        // через BaseB2::BaseBReadBuf
};
```

Классам `BaseA2` и `BaseB2` может потребоваться дублирование конструкторов `BaseA` и `BaseB`, чтобы `Derived` мог их вызвать. (Впрочем, чаще более простой путь состоит в том, чтобы сделать классы `BaseA2` и `BaseB2` виртуальными производными классами и, соответственно, обеспечить доступ `Derived` к конструкторам базовых классов.) `BaseA2` и `BaseB2` являются абстрактными классами, поэтому не требуется никакого дублирования иных функций или операторов `BaseA2` и `BaseB2` типа оператора присваивания.

Теперь все работает нормально.

```
// Пример 2а. Использование примера 2
//
Derived d;
BaseA*  pba = d;
BaseB*  pbb = d;

pba->ReadBuf("sample buffer");
    // Вызов Derived::BaseAReadBuf()

pbb->ReadBuf("sample buffer");
    // Вызов Derived::BaseBReadBuf()
```

Производные от `Derived` классы не должны замещать функцию `ReadBuf`. Если они сделают это, то отменят переименование функций в промежуточных классах.

Задача 3.10. (Не)чисто виртуальные функции *Сложность: 7*

Имеет ли смысл снабжать чисто виртуальную функцию телом?

1. Что такое чисто виртуальная функция? Приведите пример.
2. Для чего может понадобиться написание тела для чисто виртуальной функции? Приведите по возможности как можно больше причин для такого решения.

Решение

1. Что такое чисто виртуальная функция? Приведите пример.

Чисто виртуальная функция — это виртуальная функция, которая должна быть замещена в производном классе. Если класс содержит хотя бы одну незамещенную чисто виртуальную функцию, он является абстрактным классом, и объект такого класса не может быть создан.

```
// Пример 1
//
class AbstractClass
{
    // Объявление чисто виртуальной функции.
    // Теперь данный класс является абстрактным
    virtual void f( int ) = 0;
};

class StillAbstract : public AbstractClass
{
    // Не замещает f(int), поэтому
    // класс остается абстрактным
};

class Concrete : public StillAbstract
{
public:
    // Замещает f(int), что делает
    // данный класс конкретным
    void f( int ) { /* ... */ }
};

AbstractClass a;  // Ошибка - абстрактный класс
StillAbstract b;  // Ошибка - абстрактный класс
Concrete      c;  // Все в порядке, класс конкретный
```

2. Для чего может понадобиться написание тела для чисто виртуальной функции? Приведите по возможности как можно больше причин для такого решения.

Рассмотрим основные причины возникновения такой ситуации. Первая из приведенных причин достаточно широко распространена, вторая и третья возникают несколько реже, а четвертая просто представляет собой обходной путь, иногда используемый программистами, работающими со слабыми компиляторами.

1. Чисто виртуальный деструктор

Все деструкторы базовых классов должны быть либо виртуальными и открытыми, либо невиртуальными и закрытыми. Причина этого заключается в следующем. Вспомните, что мы должны избегать наследования от конкретных классов. Если данный базовый класс не должен быть конкретным, то, следовательно, открытый деструктор не является необходимым (так как не может быть создан объект данного класса). В результате могут возникнуть две ситуации. Либо мы позволяем полиморфное удаление посредством указателя на базовый класс, и тогда деструктор должен быть виртуальным и открытым, либо запрещаем его, и тогда деструктор должен быть невиртуальным и защищенным, чтобы предотвратить нежелательное использование производных классов. Более подробно об этом можно прочесть в [Sutter01].

Если класс должен быть абстрактным (вы хотите предупредить возможные попытки создания объектов данного класса), но в нем нет ни одной чисто виртуальной функции, и при этом есть открытый деструктор, сделать чисто виртуальным можно именно его.

```
// Пример 2a
//
// Файл b.h
//
class B
{
public:    /* "Начинка" класса */
    virtual ~B() = 0; // Чисто виртуальный деструктор
};
```

Конечно, деструктор производного класса должен неявно вызывать деструктор базового класса, так что этот деструктор следует определить, даже если это будет пустой деструктор.

```
// Пример 2а. Продолжение
//
// Файл b.cpp
//
B::~B() { /* Возможно, пустой */ }
```

Без такого определения вы сможете порождать другие классы из B, но толку от этого не будет никакого, поскольку создать объекты этих классов вы не сможете.

Рекомендация

Всегда делайте деструкторы базовых классов либо виртуальными и открытыми, либо невиртуальными и защищенными.

2. Определение поведения по умолчанию

Если производный класс не замещает обычную виртуальную функцию, то он просто наследует ее базовую версию. Если вы хотите обеспечить поведение по умолчанию, но при этом не дать производному классу возможности просто "молча" унаследовать виртуальную функцию, ее можно сделать чисто виртуальной, но с телом, позволив тем самым автору производного класса при необходимости вызывать ее.

```
// Пример 2б.
//
class B
{
protected:
    virtual bool f() = 0;
};

bool B::f()
{
    return true; // Определяем поведение по умолчанию
}

class D : public B
{
    bool f()
    {
        return B::f(); // Поведение по умолчанию
    }
};
```

В книге [Gamma95] шаблон состояния (State pattern) демонстрирует пример использования такой методики.

3. Обеспечение частичной функциональности

Очень часто бывает полезно обеспечить производный класс частичной функциональностью, которую ему просто останется завершить. Идея заключается в том, что производный класс выполняет реализацию базового класса как часть собственной.

```
// Пример 2в
//
class B
{
    // ...
protected virtual bool f() = 0;
```

```
};

bool B::f()
{
    // Некие действия общего назначения
}

class D : public B
{
    bool f()
    {
        // Сначала используем реализацию
        // базового класса...
        B::f();
        // ... и выполняем остальную
        // часть работы
    }
};
```

В книге [Gamma95] шаблон “Декоратор” (Decorator pattern) демонстрирует пример использования такой методики.

4. Обходной путь недостаточной диагностики компилятора

Бывают ситуации, когда вы случайно вызываете чисто виртуальную функцию (косвенно, из конструктора или деструктора). Конечно, корректно написанный код не сталкивается с такими проблемами, но никто из нас не совершенен, и такое вполне может случиться с кем угодно.

К сожалению, не все компиляторы[13] сообщают о возникновении этой проблемы. В результате через несколько часов отладки вы, найдя ошибку, возмущенно вопрошаете: и почему это компилятор не сообщил мне о такой простой вещи? (Ответ, кстати, прост: это одно из проявлений “неопределенного поведения”.)

Но вы можете сами защитить себя от таких проявлений “слабости” компилятора, обеспечив чисто виртуальные функции, которые никогда не должны быть вызваны, телами и поместив в них код, который даст вам знать о том, что произошло.

Например:

```
// Пример 2г
//
class B
{
public:
    bool f();
private:
    virtual bool do_f() = 0;
};

bool B::do_f() // Никогда не должна вызываться
{
    if ( PromptUser( "pure virtual B::f called -- "
                     "abort or ignore?" ) == Abort )
        DieDieDie();
}
```

В функции `DieDieDie()` выполняется нечто, что заставляет вашу систему перейти в режим отладки, сбросить дамп памяти или каким-то иным образом предоставить диагностирующую информацию. Вот несколько вариантов — выберите из них наиболее подходящий для вас.

---

[13] Конечно, технически перехватывать такие ошибки должна среда времени выполнения, но я говорю о компиляторе, поскольку именно компилятор должен позаботиться о коде, выполняющем соответствующую проверку во время выполнения программы.

```
void DieDieDie() // Испытанный C-способ: обращение
{                // по нулевому указателю
    memset(0,1,1);
}

void DieDieDie() // Еще один C-метод
{
    abort();
}

void DieDieDie() // Испытанный C++-способ: обращение
{                // по нулевому указателю
    *static_cast<char*>(0) = 0;
}

void DieDieDie() // C++-способ: вызов несуществующей
{                // функции по нулевому указателю
    static_cast<void(*)()>(0)();
}

void DieDieDie() // Раскручивание до catch(...)
{
    class LocalClass {};
    throw LocalClass();
}

void DieDieDie() // Для любителей стандартов
{
    throw std::logic_error();
}

void DieDieDie() throw() // Для любителей стандартов
{                        // с хорошими компиляторами
    throw 0;
}
```

Вы поняли идею? Подходите к вопросу творчески. Здесь приведены не самые быстрые способы вызвать крах программы, но главное не в этом, а в том, чтобы ваш отладчик помог вам быстро определить место возникновения неприятностей. Часто наиболее простым и надежным способом оказывается обращение по нулевому указателю.

Задача 3.11. Управляемый полиморфизм *Сложность: 3*

Отношение ЯВЛЯЕТСЯ — очень полезный инструмент в объектно-ориентированном моделировании. Однако иногда может потребоваться запретить полиморфное использование некоторых классов. В данной задаче приведен соответствующий пример и показано, как достичь требуемого эффекта.

Рассмотрим следующий код.

```
class Base
{
public:
    virtual void VirtFunc();
    // ...
};

class Derived : public Base
{
public:
    void VirtFunc();
    // ...
};

void SomeFunc( const Base& );
```

Имеются две другие функции. Цель заключается в том, чтобы позволить функции `f1()` полиморфно использовать объект `Derived` там, где ожидается объект `Base`, и запретить это всем другим функциям (включая `f2()`).

```
void f1()
{
    Derived d;
    SomeFunc( d ); // Все в порядке, работает
}

void f2()
{
    Derived d;
    SomeFunc( d ); // Мы хотим это предотвратить
}
```

Продемонстрируйте, как добиться такого результата.

Решение

Рассмотрим следующий код.

```
class Base
{
public:
    virtual void VirtFunc();
    // ...
};

class Derived : public Base
{
public:
    void VirtFunc();
    // ...
};

void SomeFunc( const Base& );
```

Причина, по которой любой код в состоянии полиморфно использовать объект `Derived` там, где ожидается объект `Base`, заключается в том, что класс `Derived` открыто наследует класс `Base` (здесь нет ничего удивительного).

Если же класс `Derived` наследует класс `Base` закрыто, то почти весь код не сможет полиморфно использовать объект `Derived` там, где ожидается объект `Base`. Причина оговорки *почти* в том, что код с доступом к закрытым частям `Derived` сможет полиморфно использовать объекты `Derived` вместо объектов `Base`. Обычно такой доступ имеют только функции-члены `Derived`, но тот же эффект дает и объявление функции как друга класса.

Итак, на поставленный вопрос имеется простой ответ.

Имеются две другие функции. Цель заключается в том, чтобы позволить функции `f1()` полиморфно использовать объект `Derived` там, где ожидается объект `Base`, и запретить это всем другим функциям (включая `f2()`).

```
void f1()
{
    Derived d;
    SomeFunc( d ); // Все в порядке, работает
}

void f2()
{
    Derived d;
```

```
    SomeFunc( d ); // Мы хотим это предотвратить
}
```

Продемонстрируйте, как добиться такого результата.
Для этого нам достаточно написать следующее.

```
class Derived : private Base
{
public:
    void VirtFunc();
    // ...
    friend void f1();
};
```

Этот код полностью решает поставленную задачу, хотя и дает функции `f1()` больший доступ к объектам `Derived`, чем в исходном варианте.

4

Брандмауэр и идиома скрытой реализации

Управление зависимостями между частями кода является далеко не самой маловажной частью разработки. Я считаю [Sutter98], что в этом плане сильные стороны C++ — поддержка двух мощных методов абстракции: объектно-ориентированного и обобщенного программирования. Оба эти метода представляют собой фундаментальный инструментарий, помогающий программисту в управлении зависимостями и, соответственно, сложностью проектов.

Задача 4.1. Минимизация зависимостей времени компиляции. Часть 1 *Сложность: 4*

Когда мы говорим о зависимостях, то обычно подразумеваем зависимости времени выполнения, типа взаимодействия классов. В данной задаче мы обратимся к анализу зависимостей времени компиляции. В качестве первого шага попытаемся найти и ликвидировать излишние заголовочные файлы.

Многие программисты часто используют гораздо больше заголовочных файлов, чем это необходимо на самом деле. К сожалению, такой подход существенно увеличивает время построения проекта, в особенности когда часто используемые заголовочные файлы включают слишком много других заголовочных файлов.

Какие из директив `#include` могут быть удалены из следующего заголовочного файла без каких-либо отрицательных последствий? Кроме удаления и изменения директив `#include`, вы не вправе вносить в текст никакие другие изменения. Обратите также внимание на важность комментариев.

```
//  x.h: исходный заголовочный файл
//
#include <iostream>
#include <ostream>
#include <list>

// Ни один из классов A, B, C, D и E не является шаблоном
// Только классы A и C имеют виртуальные функции
#include "a.h"  // class A
#include "b.h"  // class B
#include "c.h"  // class C
#include "d.h"  // class D
#include "e.h"  // class E

class X : public A, private B
{
```

```
public:
    X(const C&);
    B  f( int, char* );
    C  f( int, C );
    C& g( B );
    E  h( E )
    virtual std::ostream& print( std::ostream& ) const;
private:
    std::list<C> clist_;
    D            d_;
};

inline std::ostream& operator<<( std::ostream& os,
                                 const X& x )
{
    return x.print( os );
}
```

Решение

Из первых двух стандартных заголовочных файлов, упомянутых в `x.h`, один может быть немедленно удален за ненадобностью, а второй — заменен меньшим заголовочным файлом.

1. Удалите `#include <iostream>`.

Многие программисты включают `#include <iostream>` исключительно по привычке, как только видят любое упоминание потоков в тексте программы. Класс `X` использует потоки, это так, но ему вовсе не требуется все упомянутое в файле `iostream`. Максимум, что ему необходимо, — это включение одного лишь `ostream`, да и этого слишком много.

Рекомендация

Никогда не включайте в программу излишние заголовочные файлы.

2. Замените `ostream` на `iosfwd`.

Для работы нам требуется знать только параметры и возвращаемые типы, так что вместо полного определения `ostream` достаточно только предварительного объявления этого класса.

Раньше мы могли бы просто заменить “`#include <ostream>`” строкой “`class ostream;`”, поскольку `ostream` представлял собой класс и не располагался в пространстве имен `std`. Теперь же такая замена некорректна по двум причинам.

1. `ostream` теперь находится в пространстве имен `std`, а программисту запрещается объявлять что-либо, находящееся в этом пространстве имен.
2. `ostream` в настоящее время представляет собой синоним шаблона, а именно шаблона `basic_ostream<char>`. Дать предварительное объявление шаблона `basic_ostream` нельзя, поскольку реализации библиотеки могут, например, добавлять свои собственные параметры шаблона (помимо требуемых стандартом), о которых ваш код, естественно, не осведомлен. Кстати, это одна из основных причин, по которой программисту запрещается давать собственные объявления чего-либо, находящегося в пространстве имен `std`.

Однако не все потеряно. Стандартная библиотека предупредительно обеспечивает нас заголовочным файлом `iosfwd`, который содержит предварительные объявления

всех шаблонов потоков (включая basic_iostream) и стандартных определений typedef этих шаблонов (включая ostream). Так что все, что нам надо, — это заменить директиву #include <ostream> директивой #include <iosfwd>.

Рекомендация

Там, где достаточно предварительного определения потока, предпочтительно использовать директиву #include <iosfwd>.

Кстати, если, узнав о существовании заголовочного файла iosfwd, вы сделаете вывод, что должны существовать аналогичные файлы для других шаблонов, типа stringfwd или listfwd, то вы ошибетесь. Заголовочный файл iosfwd был создан с целью обратной совместимости, чтобы избежать неработоспособности кода, использующего старую, не шаблонную подсистему потоков.

Итак, все просто. Мы...

"Минутку! — слышу я голос кого-то из читателей. — Не так быстро! Наш operator<< реально использует объект ostream, так что нам будет недостаточно упоминания типов параметров или возвращаемого типа — нам требуется полное определение ostream, так?"

Нет, не так. Хотя вопрос вполне разумный. Рассмотрим еще раз интересующую нас функцию.

```
inline std::ostream& operator<<( std::ostream& os,
                                 const X& x )
{
    return x.print( os );
}
```

Здесь ostream& упоминается как тип параметра и как возвращаемый тип (большинство программистов знают, что для этого определение класса не требуется) и передается в качестве параметра другой функции (многие программисты *не знают*, что и в этом случае определение класса не требуется). До тех пор пока мы работаем со ссылкой ostream&, полное определение ostream необходимым не является. Конечно, нам потребуется полное определение, если мы попытаемся вызвать любую функцию-член, но здесь ничего подобного не происходит.

1. Замените "#include "e.h"" предварительным определением "class E;".

Класс E упоминается только в качестве типа параметра и возвращаемого типа, так что его определение нам не требуется.

Рекомендация

Никогда не используйте директиву #include там, где достаточно предварительного объявления.

Задача 4.2. Минимизация зависимостей времени компиляции. Часть 2 — *Сложность: 6*

Теперь, когда удалены все излишние заголовки, пришло время для второго захода: каким образом можно ограничить зависимости внутреннего представления класса?

Далее приведен заголовочный файл из задачи 4.1 после первого преобразования. Какие еще директивы #include можно удалить, если внести некоторые изменения, и какие именно изменения требуются?

Мы можем вносить только такие изменения в класс X, при которых базовые классы X и его открытый интерфейс остаются без изменений; любой код, который использует X в настоящий момент, не должен требовать изменений, ограничиваясь простой перекомпиляцией.

```
//  x.h: без лишних заголовочных файлов
//
#include <iosfwd>
#include <list>

// Ни один из классов A, B, C, D и E не является шаблоном
// Только классы A и C имеют виртуальные функции
#include "a.h"  // class A
#include "b.h"  // class B
#include "c.h"  // class C
#include "d.h"  // class D

class E;

class X : public A, private B
{
public:
    X(const C&);
    B  f( int, char* );
    C  f( int, C );
    C& g( B );
    E  h( E )
    virtual std::ostream& print( std::ostream& ) const;
private:
    std::list<C> clist_;
    D            d_;
};

inline std::ostream& operator<<( std::ostream& os,
                                 const X& x )
{
    return x.print( os );
}
```

Решение

Давайте рассмотрим, чего мы не можем сделать.

- Мы не можем удалить a.h и b.h, поскольку X является наследником как A, так и B. Для того чтобы компилятор мог получить размер объекта X, информацию о виртуальных функциях и прочую существенную информацию, требуется полное определение базовых классов.
- Мы не можем удалить list, c.h и d.h, поскольку list<C> и D представляют собой типы членов данных X. Хотя C не является ни базовым классом, ни членом, он используется для настройки члена list, а большинство современных компиляторов требуют полного определения C в случае настройки list<C>. (Стандарт не требует полного определения в этой ситуации, так что в будущем можно ожидать снятия этого ограничения.)

А теперь поговорим о красоте скрытой реализации.

C++ позволяет нам легко инкапсулировать закрытые части класса от неавторизованного *доступа*. К сожалению, поскольку подход с использованием заголовочных файлов унаследован от C, для инскапсуляции *зависимостей* закрытых частей класса требуется бо́льшая работа. “Но, — скажете вы, — ведь суть инкапсуляции и состоит в

том, что клиентский код не должен ничего знать о деталях реализации закрытых частей класса, не так ли?" Да, так, и клиентский код в C++ не должен ничего знать о закрытых частях класса (да и если это не друг класса, то он просто не получит доступ к ним), но поскольку эти закрытые части видны в заголовочном файле, код клиента вынужденно зависит от всех упомянутых там типов.

Каким же образом можно изолировать клиента от деталей реализации? Один из хороших способов — использование специального вида идиомы handle/body [Coplien92] (которую я называю идиомой Pimpl[1] из-за удобства произношения "указатель `pimpl_`") в качестве брандмауэра компиляции [Lakos96, Meyers98, Meyers99, Murray93].

Скрытая реализация представляет собой не что иное, как непрозрачный указатель (указатель на предварительно объявленный, но не определенный вспомогательный класс), использующийся для сокрытия защищенных членов класса. То есть вместо

```
// Файл x.h
class X
{
    // Открытые и защищенные члены
private:
    // Закрытые члены. При внесении изменений
    // в эту часть весь клиентский код должен
    // быть перекомпилирован.
};
```

мы пишем

```
// Файл x.h
class X
{
    // Открытые и защищенные члены
private:
    struct XImpl;
    XImpl* pimpl_; // Указатель на предварительно
                   // объявленный класс
};

// Файл x.cpp
struct X::XImpl
{
    // Закрытые члены; внесение изменений
    // не требует перекомпиляции кода клиентов
};
```

Для каждого объекта `X` динамически выделяется объект `XImpl`. Мы по сути обрезаем большой кусок объекта и оставляем на этом месте только маленький кусочек — указатель на реализацию.

Основные преимущества идиомы скрытой реализации обусловлены тем, что она позволяет разорвать зависимости времени компиляции.

- Типы, упоминающиеся только в реализации класса, больше не должны быть определены в коде клиента, что позволяет устранить лишние директивы `#include` и повысить скорость компиляции.
- Реализация класса может быть легко изменена — вы можете свободно добавлять или удалять закрытые члены без перекомпиляции клиентского кода.

Платой за эти преимущества является снижение производительности.

---

[1] Pimpl означает "указатель на реализацию" (**p**ointer to **impl**ementation), но использование в качестве перевода термина "скрытая реализация" в данном случае отражает суть идиомы — сокрытие деталей реализации класса. — *Прим. перев.*

- Каждое создание и уничтожение объекта сопровождается выделением и освобождением памяти.
- Каждое обращение к скрытому члену требует как минимум одного уровня косвенности. (Уровней косвенности может оказаться и несколько, например, если обращение к скрытому члену использует указатель для вызова функции в видимом классе.)

Мы вернемся к этим и другим вопросам использования скрытой реализации немного позже, а пока продолжим решение нашей задачи. В нашем случае три заголовочных файла нужны только потому, что соответствующие классы встречаются при описании закрытых членов X. Если реструктуризировать X с использованием идиомы скрытой реализации, мы тут же сможем внести в код ряд дальнейших упрощений.

Итак, используем идиому Pimpl для сокрытия деталей реализации X.

```
#include <list>

#include "c.h"  // class C
#include "d.h"  // class D
```

Один из этих заголовочных файлов (c.h) может быть заменен предварительным объявлением (поскольку C упоминается в качестве типа параметра и возвращаемого типа), в то время как остальные два могут быть просто удалены.

Рекомендация

Для широко используемых классов предпочтительно использовать идиому сокрытия реализации (брандмауэр компиляции, Pimpl) для сокрытия деталей реализации. Для хранения закрытых членов (как переменных состояния, так и функций-членов) используйте непрозрачный указатель (указатель на объявленный, но не определенный класс), объявленный как "struct XxxxImpl pimpl_;". Например: "class Map { private: struct MapImpl* pimpl_; };".*

После внесения дополнительных изменений заголовочный файл выглядит следующим образом.

```
// x.h: Использование Pimpl
//
#include <iosfwd>
#include "a.h" // class A (имеет виртуальные функции)
#include "b.h" // class B (виртуальных функций нет)
class C;
class E;
class X : public A, private B
{
public:
    X( const C& );
    B  f( int, char* );
    C  f( int, C );
    C& g( B );
    E  h( E );
    virtual std::ostream& print( std::ostream& ) const;
private:
    struct XImpl;
    XImpl* pimpl_; // Непрозрачный указатель на
                   // предварительно объявленный класс
};
inline std::ostream& operator<<( std::ostream& os,
                                 const X& x )
{
    return x.print(os);
}
```

Детали реализации находятся в файле реализации X, который не виден клиентскому коду, который, соответственно, никак не зависит от этого файла.

```
// Файл реализации x.cpp
//
struct X::XImpl
{
    std::list<C> clist_;
    D            d_;
};
```

Итак, у нас осталось только три заголовочных файла, что само по себе является существенным усовершенствованием первоначального кода. А нельзя ли еще больше изменить класс X и добиться дальнейшего усовершенствования полученного кода? Ответ на это вы найдете в задаче 4.3.

Задача 4.3. Минимизация зависимостей времени компиляции. Часть 3 *Сложность: 7*

Теперь, когда удалены все излишние заголовочные файлы и устранены все излишние зависимости во внутреннем представлении класса, остались ли какие-либо возможности для дальнейшего разъединения классов? Ответ возвращает нас к базовым принципам проектирования классов.

Верные принципу не останавливаться на достигнутом, ответьте на вопрос: каким образом можно продолжить уменьшение количества зависимостей? Какой заголовочный файл может быть удален следующим и какие изменения в X для этого следует сделать?

На этот раз вы можете вносить любые изменения в X, лишь бы открытый интерфейс при этом оставался неизменным и для работы клиентского кода было достаточно простой перекомпиляции. Вновь обратите особое внимание на комментарии в приведенном тексте.

```
// x.h: После использования Pimpl для
//      сокрытия деталей реализации
//
#include <iosfwd>
#include "a.h" // class A (имеет виртуальные функции)
#include "b.h" // class B (виртуальных функций нет)
class C;
class E;
class X : public A, private B
{
public:
    X( const C& );
    B  f( int, char* );
    C  f( int, C );
    C& g( B );
    E  h( E );
    virtual std::ostream& print( std::ostream& ) const;
private:
    struct XImpl;
    XImpl* pimpl_; // Непрозрачный указатель на
                   // предварительно объявленный класс
};
inline std::ostream& operator<<( std::ostream& os,
                                 const X& x )
{
    return x.print(os);
}
```

```
// Файл реализации x.cpp
//
struct X::XImpl
{
    std::list<C> clist_;
    D            d_;
};
```

Решение

Согласно моему опыту, множество программистов, работающих на C++, все еще полагают, что без наследования нет объектно-ориентированного программирования и используют наследование везде, где только можно. Этому вопросу посвящена задача 3.5, главный вывод которой заключается в том, что наследование (включающее отношение ЯВЛЯЕТСЯ, но не ограничивающееся им) — существенно более сильное отношение классов, чем СОДЕРЖИТ или ИСПОЛЬЗУЕТ. Таким образом, при работе с зависимостями следует отдавать предпочтение включению.

В нашем случае `X` открыто наследует `A` и закрыто — `B`. Вспомним, что открытое наследование всегда должно моделировать отношение ЯВЛЯЕТСЯ и удовлетворять принципу подстановки Лисков.[2] В данном случае `X` ЯВЛЯЕТСЯ `A` и в этом нет ничего некорректного, так что оставим открытое наследование от `A` в покое.

Но не обратили ли вы внимания на одну интересную деталь, связанную с `B`?

Дело в том, что класс `B` является закрытым базовым классом для `X`, но не имеет виртуальных функций. Но ведь обычно для того, чтобы использовать закрытое наследование, а не включение, имеется только одна причина — получить доступ к защищенным членам, что в большинстве случаев означает “заместить виртуальную функцию”.[3] Как мы знаем, виртуальных функций в классе B нет, так что, вероятно, нет и причины использовать наследование (разве что классу `X` требуется доступ к каким-либо защищенным функциям или данным в `B`, но мы предполагаем, что в нашем случае такой необходимости нет). Соответственно, от наследования можно отказаться в пользу включения, а значит, и удалить из текста ненужную теперь директиву “`#include "b.h"`”.

Поскольку объект-член `B` должен быть закрытым (в конце концов, он — всего лишь деталь реализации), его следует скрыть за указателем `pimpl_`.

Рекомендация

Никогда не используйте наследование там, где достаточно включения.

Итак, предельно упрощенный в смысле заголовочных файлов код выглядит следующим образом.

```
// x.h: После удаления ненужного наследования
//
#include <iosfwd>
#include "a.h" // class A (имеет виртуальные функции)
class B;
class C;
class E;
class X : public A
{
```

[2] Материал на эту тему можно найти по адресу `http://www.gotw.ca/publications/xc++/om.htm` и в [Martin95].

[3] Конечно, бывают и другие возможные причины для выбора наследования, но эти ситуации крайне редки. Подробное изложение данного вопроса можно найти в [Sutter98a, Sutter99].

```
public:
    X( const C& );
    B  f( int, char* );
    C  f( int, C );
    C& g( B );
    E  h( E );
    virtual std::ostream& print( std::ostream& ) const;
private:
    struct XImpl;
    XImpl* pimpl_; // Сюда включен объект B
};
inline std::ostream& operator<<( std::ostream& os,
                                 const X& x )
{
    return x.print(os);
}
```

После трех последовательных усовершенствований файл `x.h` использует имена других классов, но из восьми включаемых заголовочных файлов осталось только два. Неплохой результат!

Задача 4.4. Брандмауэры компиляции *Сложность: 6*

Использование скрытой реализации (идиомы Pimpl) существенно снижает взаимозависимости кода и время построения программ. Но что именно следует скрывать за указателем `pimpl_` и как наиболее безопасно использовать его?

В C++ при любых изменениях в определении класса (даже при изменениях в закрытых членах) требуется перекомпиляция всех пользователей этого класса. Для снижения этой зависимости обычной методикой является использование непрозрачного указателя для сокрытия некоторых деталей реализации.

```
class X
{
public:
    /* ... Открытые члены ... */
protected:
    /* ... Защищенные члены? ... */
private:
    /* ... Закрытые члены? ... */
    struct XImpl;
    XImpl* pimpl_; // Непрозрачный указатель на
                   // предварительно объявленный класс
};
```

Ответьте на следующие вопросы.

1. Что следует внести в `XImpl`? Имеется четыре распространенные стратегии.
   - Разместить в `XImpl` все закрытые данные (но не функции).
   - Разместить в `XImpl` все закрытые члены.
   - Разместить в `XImpl` все закрытые и защищенные члены.
   - Сделать `XImpl` классом, полностью повторяющим класс `X`, которым он должен был бы быть, и оставить в классе `X` только открытый интерфейс, просто перенаправляющий вызовы функций классу `XImpl`.

   Каковы преимущества и недостатки каждой из стратегий? Как выбрать среди них наиболее подходящую?
2. Требуется ли наличие в `XImpl` обратного указателя на объект `X`?

Решение

Класс `X` использует вариант идиомы “handle/body” [Coplien92], которая применяется в первую очередь для счетчиков ссылок разделяемой реализации, но может использоваться и для более общих целей сокрытия реализации. Для удобства далее я буду называть `X` “видимым классом”, а `XImpl` — Pimpl-классом.

Большим преимуществом этой идиомы является разрыв зависимостей времени компиляции. Во-первых, ускоряется компиляция и сборка программы, поскольку использование сокрытия реализации, как мы видели в задачах 4.2 и 4.3, позволяет устранить излишние директивы `#include`. В своей практике мне приходилось работать над проектами, в которых применение сокрытия реализации позволяло сократить время компиляции вдвое. Во-вторых, при этом возможно свободное изменение части класса, скрытой за указателем `pimpl_`, так, например, здесь можно свободно добавлять или удалять члены без необходимости перекомпилировать клиентский код.

1. Что следует внести в `XImpl`?

Вариант 1 (оценка: 6/10): разместить в `XImpl` все закрытые данные (но не функции). Это хорошее начало, поскольку теперь нам достаточно предварительного объявления классов, использующихся только в качестве членов данных (мы можем обойтись без директивы `#include` и полного определения класса, что автоматически приведет к зависимости от этого класса клиентского кода).

Вариант 2 (оценка: 10/10): разместить в `XImpl` все закрытые невиртуальные члены. В настоящее время я (почти) всегда поступаю именно таким образом. В конце концов закрытые члены в C++ — это те члены, о которых клиентский код не должен ничего знать вообще. Однако следует сделать ряд предостережений.

- Вы не можете скрывать виртуальные функции-члены, даже если это закрытые функции. Если виртуальная функция замещает виртуальную функцию, унаследованную от базового класса, то она обязана находиться в классе-наследнике. Если же виртуальная функция не является унаследованной, то она все равно должна оставаться в видимом классе, чтобы производные классы могли ее заместить.

  Обычно виртуальные функции являются закрытыми, за исключением ситуаций, когда функция производного класса вызывает функцию базового класса — тогда виртуальная функция должна быть защищенной.
- Если функция в Pimpl-классе в свою очередь использует функции видимого класса, то ей может потребоваться “обратный указатель” на видимый объект, что приводит к дополнительному уровню косвенности (по соглашению такой обратный указатель обычно носит имя `self_`).
- Зачастую наилучшим компромиссом является использование варианта 2, с дополнительным размещением в `XImpl` тех не закрытых функций, которые должны вызываться закрытыми.

Рекомендация

Для широко используемых классов предпочтительно использовать идиомы сокрытия реализации (брандмауэра компиляции, Pimpl) для сокрытия деталей реализации. Для хранения закрытых членов (как переменных состояния, так и функций-членов) используйте непрозрачный указатель (указатель на объявленный, но не определенный класс), объявленный как “`struct XxxxImpl* pimpl_;`”.

Вариант 3 (оценка: 0/10): разместить в `XImpl` все закрытые и защищенные члены. Распространение идиомы Pimpl на защищенные члены является ошибкой. Защищенные члены никогда не должны переноситься в Pimpl-класс. Они существуют для того, чтобы быть видимыми и используемыми производными классами, так что ничего хорошего от того, что они будут скрыты от производных классов, не получится.

Вариант 4 (оценка: 10/10 в некоторых ситуациях): сделать `XImpl` классом, полностью повторяющим класс `X`, которым он должен был бы быть, и оставить в классе `X` только открытый интерфейс, просто перенаправляющий вызовы функций классу `XImpl`. Это неплохое решение в некоторых случаях, обладающее тем преимуществом, что при этом не требуется обратный указатель на видимый объект (все сервисы доступны внутри Pimpl-класса). Основной недостаток данного метода в том, что обычно он делает видимый класс бесполезным с точки зрения наследования.

2. Требуется ли наличие в `XImpl` обратного указателя на объект `X`?

Требуется ли наличие в Pimpl-классе обратного указателя на видимый объект? Ответ — иногда, к сожалению, да. В конце концов, мы делаем не что иное, как (несколько искусственное) разделение каждого объекта на две части с целью скрыть одну из них.

Рассмотрим вызов функции видимого класса. Обычно при этом для завершения запроса требуется обращение к каким-то данным или функциям скрытой части. Это просто и понятно. Однако, на первый взгляд, не столь очевидно, что функции скрытой части могут вызывать функции, находящиеся в видимом классе, обычно потому, что это открытые или виртуальные функции. Один способ минимизации количества таких вызовов состоит в разумном использовании описанного ранее варианта 4, т.е. в реализации варианта 2 и добавлении в Pimpl-класс тех не закрытых функций, которые используются закрытыми.

Задача 4.5. Идиома “Fast Pimpl” *Сложность:6*

Иногда хочется схитрить во имя “уменьшения зависимостей” или “эффективности”, но не всегда это приводит к хорошим результатам. В данной задаче рассматривается превосходная идиома для безопасного достижения обеих целей одновременно.

Стандартные вызовы `malloc` и `new` относительно дорогостоящи.[4] В приведенном ниже коде изначально в классе `Y` имелся член данных типа `X`.

```
// Попытка №1
//
// Файл y.h
#include "x.h"
class Y
{
    /* ... */
    X x_;
};

// Файл y.cpp
Y::Y() {}
```

Данное объявление класса `Y` требует, чтобы класс `X` был видимым (из `x.h`). Чтобы избежать этого, программист пытается переделать код следующим образом.

```
// Попытка №2
//
// Файл y.h
class X;
class Y
```

[4] По сравнению с другими типичными операциями, типа вызова функции.

```
{
    /* ... */
    X* px_;
};

// Файл y.cpp
Y::Y() : px_(new X ) {}
Y::~Y() { delete px_; px_ = 0; }
```

Таким образом удается скрыть X, но из-за частого использования Y и накладных расходов на динамическое распределение памяти это решение приводит к снижению производительности программы.

Тогда наш программист находит решение, которое не требует включения x.h в y.h (и даже не требует предварительного объявления класса X) и позволяет избежать неэффективности из-за динамического распределения.

```
// Попытка №3
//
// Файл y.h
class X;
class Y
{
    /* ... */
    static const size_t sizeofx = /* Некоторое значение */;
    char x_[sizeofx];
};

// Файл y.cpp
#include "x.h"
Y::Y()
{
    assert( sizeofx >= sizeof(X) );
    new(&x_[0]) X;
}

Y::~Y()
{
    (reinterpret_cast<X*>(&x_[0]))->~X();
}
```

1. Каковы накладные расходы памяти при использовании идиомы Pimpl?
2. Каково влияние идиомы Pimpl на производительность?
3. Рассмотрите код из третьей попытки. Не могли бы вы предложить лучший способ избежать накладных расходов?

Примечание: просмотрите еще раз задачу 4.4.

Решение

1. Каковы накладные расходы памяти при использовании идиомы Pimpl?

В случае использования идиомы Pimpl нам требуется память как минимум для одного дополнительного указателя (а при необходимости размещения обратного указателя в XImpl — двух) в каждом объекте X. Обычно это приводит к добавлению как минимум 4 (или 8) байт, но может потребовать 14 байт и более — в зависимости от требований выравнивания. Например, попробуйте скомпилировать следующую программу, используя ваш любимый компилятор.

```
struct X { char c; struct XImpl; XImpl* pimpl_; };
struct X::XImpl { char c; };
int main()
```

```
{
    cout << sizeof(X::XImpl) << endl
         << sizeof(X) << endl;
}
```

Многие популярные компиляторы, использующие 32-битовые указатели, создадут программу, вывод которой будет следующим.

```
1
8
```

В этих компиляторах накладные расходы, вызванные дополнительным указателем, составляют не 4, а 7 байт. Почему? Потому что платформа, на которой работает компилятор, требует, чтобы указатель размещался выровненным по 4-байтовой границе, иначе при обращении к нему резко уменьшится производительность. Зная это, компилятор выделяет для выравнивания 3 байта неиспользуемого пространства внутри каждого объекта `X`, так что стоимость добавления указателя оказывается равной 7 байтам. Если требуется и обратный указатель, то общие накладные расходы достигнут 14 байт на 32-битовой машине и 30 байт на 64-битовой.

Каким образом можно избежать этих накладных расходов? Говоря коротко — избежать их невозможно, но можно их минимизировать.

Имеется один совершенно безрассудный путь для устранения накладных расходов, который вы не должны никогда использовать (и даже говорить кому-либо, что слышали о нем от меня), и корректный, но непереносимый путь их минимизации. Обсуждение пути устранения накладных расходов, как слишком некорректного, вынесено во врезку “Безответственная оптимизация и ее вред”.

Тогда, и только тогда, когда указанное небольшое различие в выделяемой памяти действительно играет важную роль в вашей программе, вы можете воспользоваться корректным, но непереносимым путем, использовав специфичные для данного компилятора директивы `#pragma`. Во многих компиляторах допускается возможность отмены выравнивания по умолчанию для данного класса (как это делается в конкретном компиляторе, вы узнаете из соответствующей документации). Если ваша целевая платформа рекомендует, но не заставляет использовать выравнивание, а компилятор поддерживает возможность отмены выравнивания по умолчанию, вы сможете избежать перерасхода памяти за счет (возможно, очень небольшого) уменьшения производительности, поскольку использование невыровненных указателей несколько менее эффективно. Но перед тем как приступать к такого рода действиям, всегда помните главное правило — *сначала сделай решение правильным, а уж потом быстрым.* Никогда не приступайте к оптимизации — ни в плане памяти, ни в плане скорости работы, — пока ваш профайлер и прочий инструментарий не подскажут вам, что вы должны этим заняться.

2. Каково влияние использования идиомы Pimpl на производительность?

Использование идиомы Pimpl приводит к снижению производительности по двум основным причинам. Первая из них состоит в том, что при каждом создании и уничтожении объекта `X` происходит выделение и освобождение памяти для объекта `XImpl`, что является относительно дорогой операцией.[5] Вторая причина заключается в том, что каждое обращение к члену в Pimpl-классе требует как минимум одного уровня косвенности; если же скрытый член использует обратный указатель для вызова функции из видимого класса, уровень косвенности увеличивается.

Каким образом можно обойти эти накладные расходы? Путем использования идиомы Fast Pimpl, о котором рассказывается далее (имеется еще один способ, которым лучше никогда не пользоваться, — см. врезку “Безответственная оптимизация и ее вред”).

---

[5] По сравнению с другими типичными операциями C++, типа вызова функции. Замечу, что я говорю о стоимости использования распределителя памяти общего назначения, обычно вызываемого посредством встроенного `::operator new()` или `malloc()`.

3. Рассмотрите код из третьей попытки.

Вкратце можно сказать одно: так поступать не надо. C++ не поддерживает непрозрачные типы непосредственно, и приведенный код — непереносимая попытка (я бы даже сказал — “хак”) обойти это ограничение. Почти наверняка программист хотел получить нечто иное, а именно — идиому Fast Pimpl.

Вторая часть третьего вопроса звучит так: **не могли бы вы предложить лучший способ избежать накладных расходов?**

Главное снижение производительности связано с использованием динамического выделения памяти для Pimpl-объекта. Вообще говоря, корректный путь обеспечить высокую производительность выделения памяти для определенного класса — это обеспечить его специфичным для данного класса оператором `new()` и использовать распределитель памяти фиксированного размера, поскольку такой распределитель может быть сделан существенно более производительным, чем распределитель общего назначения.

```
// Файл x.h
class X
{
    /* ... */
    struct XImpl;
    XImpl* pimpl_;
};

// Файл x.cpp
#include "x.h"
struct X::XImpl
{
    /* ... */
    static void* operator new( size_t )   { /* ... */ }
    static void  operator delete( void* ) { /* ... */ }
};
X::X() : pimpl_( new XImpl ) {}
X::~X() { delete pimpl_; pimpl_ = 0; }
```

“Ага! — готовы сказать вы. — Вот мы и нашли священную чашу Грааля — идиому Fast Pimpl!” Да, но подождите минутку и подумайте, как работает этот код и во что обойдется его использование.

Уверен, что в вашем любимом учебнике по программированию или языку C++ вы найдете детальное описание того, как создать эффективные функции выделения и освобождения памяти фиксированного размера, так что здесь я не буду останавливаться на этом вопросе, а поговорю об используемости данного метода. Один способ состоит в размещении функций распределения и освобождения в обобщенном шаблоне распределения памяти фиксированного размера примерно таким образом.

```
template<size_t S>
class FixedAllocator
{
public:
    void* Allocate( /* Запрашиваемый объем памяти
                       всегда равен S */ );
    void  Deallocate( void* );
private:
    /* Реализация с использованием статических членов */
};
```

Поскольку закрытые детали скорее всего используют статические члены, могут возникнуть проблемы при вызове `Deallocate` из деструкторов статических объектов. Вероятно, более безопасный способ — использование синглтона, который управляет отдельным списком для каждого запрашиваемого размера (либо, в качестве компромисса, поддерживает списки для наборов размеров, например, один список для блоков размером от 0 до 8 байт, второй — от 9 до 16 и т.д.).

```
class FixedAllocator
{
public:
    static FixedAllocator& Instance();
    void* Allocate( size_t );
    void  Deallocate( void* );
private:
    /* Реализация с использованием синглтона,
       обычно с более простыми, чем в приведенном
       ранее шаблонном варианте, статическими
       членами */
};
```

Давайте добавим вспомогательный базовый класс для инкапсуляции вызовов (при этом производные классы наследуют базовые операторы).

```
struct FastArenaObject
{
    static void* operator new( size_t s )
    {
        return FixedAllocator::Instance()->Allocate(s);
    }
    static void operator delete( void* p )
    {
        FixedAllocator::Instance()->Deallocate(p);
    }
};
```

Теперь мы можем легко написать любое количество объектов Fast Pimpl.

```
// Требуется, чтобы это был объект Fast Pimpl?
// Это очень просто - достаточно наследования...
struct X::XImpl : FastArenaObject
{
    /* Закрытые члены */
}
```

Применяя эту технологию к исходной задаче, мы получим вариант попытки □2 из условия задачи.

```
// Файл y.h

class X;
class Y
{
    /* ... */
    X* px_;
};

// Файл y.cpp

#include "x.h" // X наследует FastArenaObject
Y::Y() : px_( new X ) {}
Y::~Y() { delete px_; px_ = 0; }
```

Однако будьте осторожны и не используйте эту идиому бездумно где только можно. Да, она дает увеличение скорости, но не следует забывать о том, какой ценой это делается. Работа с отдельными списками памяти для объектов различного размера обычно приводит к неэффективному использованию памяти в силу большей, чем обычно, ее фрагментации.

И последнее напоминание: как и в случае любой другой оптимизации, использовать идиому Pimpl вообще и Fast Pimpl в частности следует только тогда, когда профилирование программы и опыт программиста доказывают реальную необходимость повышения производительности в вашей ситуации.

Рекомендация

Избегайте использования встроенных функций и настройки кода, пока необходимость этого не будет доказана путем профилирования программы.

Безответственная оптимизация и ее вред

Приведенное далее решение, казалось бы, позволяет избежать излишних накладных расходов при использовании идиомы Pimpl как с точки зрения используемой памяти, так и с точки зрения производительности. Иногда такой метод даже рекомендуют использовать, хотя совершенно зря.

Это совершенно бездумный, небезопасный и вредный метод. Вообще не следовало бы упоминать о нем, но мне приходилось видеть людей, которые поступали следующим образом.

```
// Файл x.h
class X
{
    /* ... */
    static const size_t sizeofximpl
                        = /* Некоторое значение */
    char pimpl_[sizeofximpl];
}:

// Файл реализации x.cpp
#include "x.h"
X::X()
{
    assert( sizeofximpl >= sizeof( XImpl ) );
    new( &pimpl_[0]) XImpl;
}

X::~X()
{
    (reinterpret_cast<XImpl*>(&pimpl_[0]))->~XImpl();
}
```

НЕ ДЕЛАЙТЕ ЭТОГО! Да, используя этот метод, вы сохраняете память — в количестве, достаточном для размещения целого указателя.[6] Да, вы избегаете снижения производительности — в коде нет ни одного вызова `new` или `malloc`. Может даже случиться так, что этот метод заработает при использовании текущей версии вашего конкретного компилятора.

Но этот метод абсолютно непереносим и может испортить всю вашу программу, даже если сначала покажется, что он отлично работает. Тому есть несколько причин.

1. *Выравнивание.* Любая память, выделяемая посредством вызова `new` или `malloc`, гарантированно оказывается корректно выровненной для объекта любого типа, но на буферы, не выделенные динамически, эта гарантия не распространяется.

```
char* buf1 = (char*)malloc( sizeof(Y) );
char* buf2 = new char[ sizeof(Y) ];
char buf3[ sizeof(Y) ];
new (buf1) Y;      // OK, buf1 выделен динамически
new (buf2) Y;      // OK, buf2 выделен динамически
new (&buf3[0]) Y;  // Ошибка, buf3 может быть не выровнен
(reinterpret_cast<Y*>(buf1))->~Y();      // OK
```

[6] При использовании этого метода Pimpl-класс оказывается полностью скрыт, но очевидно, что клиентский код должен быть перекомпилирован при изменении `sizeofximpl`.

```
(reinterpret_cast<Y*>(buf2))->~Y();       // OK
(reinterpret_cast<Y*>(&buf3[0]))->~Y();   // Ошибка
```

Чтобы не оставалось неясностей: я не рекомендую использовать не только третий, но и первые два метода тоже. Я просто указываю корректность первых двух вариантов с точки зрения выравнивания, но никоим образом не призываю использовать их для получения Pimpl без динамического выделения и не говорю о корректности с этой точки зрения.[7]

2. *Хрупкость.* Автор `X` должен быть чрезвычайно осторожен при работе с обычными функциями класса. Например, `X` не должен использовать оператор присваивания по умолчанию и должен либо запретить присваивание вовсе, либо снабдить класс собственным оператором присваивания. (Написать безопасный `X::operator=()` не так сложно, так что я оставлю это занятие читателям в качестве упражнения. Не забудьте только о безопасности исключений в этом операторе и в `X::~X`.[8] Когда вы выполните это упражнение, я думаю, вы согласитесь, что мороки с ним гораздо больше, чем кажется на первый взгляд.)
3. *Стоимость сопровождения.* При увеличении `sizeof(XImpl)` и превышении значения `sizeofximpl` программист должен не забыть увеличить последнее, что усложняет сопровождение программы. Конечно, можно заранее выбрать заведомо большое значение для `sizeofximpl`, но это приведет к неэффективному использованию памяти (см. п. 4).
4. *Неэффективность.* При `sizeofximpl > sizeof(XImpl)` наблюдается бесполезный перерасход памяти. Его минимизация приводит к увеличению сложности сопровождения кода (см. п. 3).
5. *Хакерство.* Из приведенного кода очевидно, что программист пытается сделать “нечто необычное”. Из моего опыта “необычное” почти всегда означает “хак”. Так что когда вы видите нечто подобное — размещение объекта в массиве символов или реализацию присваивания путем явного вызова деструктора с последующим размещением объекта (см. задачу 9.4), — вам следует решительно сказать “Нет!”.

Основной вывод — C++ непосредственно не поддерживает непрозрачные указатели, и попытки обойти это ограничение к добру не приводят.

[7] Ну хорошо, я признаюсь: есть способ (хотя и не очень переносимый, но вполне безопасный) обеспечить размещение Pimpl-класса подобным образом. Этот способ включает создание структуры `max_align`, которая гарантирует максимальную степень выравнивания, и определение Pimpl-члена как `union { max_align dummy; char pimpl_[sizeofximpl]; };` — это гарантирует достаточное выравнивание. Детальную информацию вы можете найти, осуществив поиск “`max_align`” в Web или на DejaNews. Но я еще раз повторяю, что крайне не рекомендую идти по этому скользкому пути, так как применение `max_align` решает только один вопрос и оставляет нерешенными остальные четыре. И не говорите, что вас не предупреждали!

[8] Для чего обратитесь к задачам 2.1—2.10.

5

Пространства и поиск имен

Задача 5.1. Поиск имен и принцип интерфейса. Часть 1 *Сложность: 9.5*

Когда вы вызываете функцию — какую именно функцию вы вызываете?

Какие функции вызываются в приведенном ниже примере? Почему?

```
namespace A
{
    struct X;
    struct Y;
    void f( int );
    void g( X );
}

namespace B
{
    void f( int i )
    {
        f( i ); // Какая f()?
    }

    void g( A::X x )
    {
        g( x ); // Какая g()?
    }

    void h( A::Y y )
    {
        h( y ); // Какая h()?
    }
}
```

Решение

Два из трех случаев очевидны, но третий требует хорошего знания правил поиска имен в C++, в частности поиска Кёнига (Koenig lookup).

Начнем с простого.

```
namespace A
{
    struct X;
    struct Y;
    void f( int );
    void g( X );
```

```
}

namespace B
{
    void f( int i )
    {
        f( i ); // Какая f()?
    }
```

Здесь `f()` вызывает саму себя, приводя к бесконечной рекурсии. Причина этого в том, что единственной видимой функцией `f()` является сама `B::f()`.

Имеется еще одна функция с сигнатурой `f(int)`, находящаяся в пространстве имен `A`. Если бы в пространстве имен `B` была инструкция "`using namespace A;`" или "`using A::f;`", то функция `A::f(int)` была бы видима в качестве кандидата при поиске `f(int)`, и при вызове `f(i)` возникла бы неоднозначность — выбрать функцию `A::f(int)` или `B::f(int)`. Однако поскольку `A::f(int)` не входит в область видимости, в качестве кандидата рассматривается только функция `B::f(int)`, и никакой неоднозначности не возникает.

А вот теперь рассмотрим хитрый вызов.

```
void g( A::X x )
{
    g( x ); // Какая g()?
}
```

Здесь возникает неоднозначность между выбором `A::g(X)` и `B::g(X)`. Программист должен явно указать, какому именно пространству имен принадлежит вызываемая функция.

Вы можете удивиться, почему это так. Ведь в пространстве имен `B` нет инструкции `using`, которая бы вносила `A::g(X)` в область видимости, так что видимой должна быть только функция `B::g(X)`, не так ли? Да, это было бы так, не будь в C++ одного дополнительного правила поиска имен.

> Поиск Кёнига[1] (упрощенный): *если вы передаете функции аргумент типа класса (в нашем случае `x` типа `A::X`), то при поиске функции компилятором помимо прочих рассматриваются имена в пространстве имен (в нашем случае `A`), содержащем тип аргумента.*

На самом деле правило немного сложнее, но суть его именно такова. Вот пример из стандарта.

```
namespace NS
{
    class T { };
    void f(T);
}
NS::T parm;
int main()
{
    f(parm);  // Вызов NS::f
}
```

Я не хочу углубляться в причины возникновения этого правила (если хотите, попробуйте взять пример из стандарта, заменить `NS` на `std`, `T` на `string`, а `f` — оператором `<<` и рассмотреть, что получится в результате). Пока достаточно знать, что поиск Кёнига — это верное решение, и вам следует знать, как оно работает, поскольку иногда оно может повлиять на разрабатываемый вами код.

Вернемся к последней функции из условия задачи.

---

[1] Названо по имени Эндрю Кёнига (Andrew Koenig), члена комитета по стандарту C++, благодаря которому это правило вошло в стандарт C++. См. также [Koenig97].

```
    void h( A::Y y )
    {
        h( y ); // Какая h()?
    }
}
```

В данном случае не имеется другой функции `h(A::Y)`, так что функция `h()` вызывает сама себя, приводя к бесконечной рекурсии.

Хотя в сигнатуре `B::h()` и упоминается тип из пространства имен `A`, это никак не влияет на поиск функции, поскольку в пространстве имен `A` нет функции с соответствующим именем и сигнатурой `h(A::Y)`.

Итак, что же все это означает? Каковы результаты решения поставленной задачи?

Вкратце: на код в пространстве имен `B` влияют функции, объявленные в совершенно отдельном пространстве имен `A`, несмотря на то, что все, что сделано в пространстве имен `B`, — это упомянут тип из пространства имен `A` и не имеется ни одной инструкции `using`.

Таким образом, пространства имен не настолько независимы друг от друга, как полагают многие программисты. Но не спешите отказываться от их использования — пространства имен действительно во многом независимы, а цель данной задачи состояла в том, чтобы показать один из (редких) случаев, когда пространства имен оказываются недостаточно герметично разделены — да и не должны быть в данном случае, как показывает следующая задача.

Задача 5.2. Поиск имен и принцип интерфейса. Часть 2 *Сложность: 9*

Что содержится в классе, т.е. что является "частью" класса и его интерфейса?

Вспомним традиционное определение класса.

> *Класс описывает множество данных вместе с функциями, оперирующими этими данными.*

Вопрос в том, какие функции являются частью класса или образуют его интерфейс?

Указание 1. Понятно, что нестатические функции-члены тесно связаны с классом, а открытые нестатические функции-члены образуют часть интерфейса класса. А что можно сказать о статических функциях-членах, о свободных функциях?

Указание 2. Еще раз вернитесь к результатам решения задачи 5.1.

Решение

Начнем наше решение с обманчиво простого вопроса: **какие функции являются частью класса или образуют его интерфейс?**

А вот более сложные вопросы.

- Как ответ на этот вопрос согласуется с объектно-ориентированным программированием в стиле C?
- Как ответ на этот вопрос согласуется с поиском Кёнига? С примером Майерса? (И поиск Кёнига, и пример Майерса будут описаны далее.)
- Как это влияет на анализ зависимостей классов и проектирование объектных моделей?

Вернемся еще раз к тому, что же такое класс.

Класс описывает множество данных вместе с функциями, оперирующими этими данными.

Программисты часто неверно трактуют это определение, говоря: “Класс — это то, что находится в его определении, т.е. члены данных и функции-члены”. Но это определение отличается от приведенного выше, поскольку в нем под словом *функции* подразумеваются только *функции-члены*. Рассмотрим следующий фрагмент.

```
//*** Пример 1a
class X { /* ... */ };
/* ... */
void f( const X& );
```

Вопрос в том, *является ли f частью X?* Некоторые автоматически говорят “Нет”, поскольку `f` не является функцией-членом (или, что то же самое, является свободной функцией). Другие же понимают, что если весь код из примера 1*a* находится в одном заголовочном файле, то он немногим отличается от следующего кода.

```
//*** Пример 1б
class X
{
    /* ... */
public:
    void f() const;
};
```

Поразмыслите немного над сказанным. Не считая прав доступа,[2] функция `f` оказывается той же, что и в исходном коде, получая указатель/ссылку на `X` посредством неявного параметра `this`. Таким образом, если весь код из примера 1*a* располагается в одном заголовочном файле, становится очевидно, что, несмотря на то, что `f` не член `X`, эта функция тесно связана с `X`. В следующем разделе я точно определю их отношение.

Однако если `X` и `f` находятся в разных заголовочных файлах, то `f` в этом случае — просто некоторая старая функция-клиент, не являющаяся частью `X` (даже если `f` предназначена для расширения `X`). Мы регулярно пишем функции, типами параметров которых являются типы из некоторых библиотечных заголовочных файлов, но совершенно ясно, что наши функции частью библиотеки не являются.

С учетом приведенного примера я предлагаю следующий принцип интерфейса (Interface Principle).

Для класса X все функции, включая свободные, которые

- *“упоминают” X и*
- *“поставляются с” X,*

являются логической частью X, поскольку они образуют часть интерфейса X.

Каждая функция-член по определению является частью `X`.

- Каждая функция-член должна “упоминать” `X` (нестатические функции-члены неявно получают параметр типа `X* const` или `const X* const`; статические функции-члены находятся в области видимости `X`)[3].
- Каждая функция-член должна “поставляться с” `X` (в определении `X`).

Применение принципа интерфейса к примеру 1*a* приводит к тому же результату, что и проведенный ранее анализ. Очевидно, что `f` упоминает `X`. Если, к тому же, `f` поставляется с `X` (например, в одном заголовочном файле и/или в одном пространстве

---

[2] Которые могут остаться неизменными, если исходная функция `f` была объявлена как друг класса.

[3] Согласно разделу 9.3.2.1 стандарта, `this` может иметь тип `X*`, `const X*`, `const volatile X*`, `volatile X*`. То, что к типу указателя часто ошибочно добавляют лишний `const`, связано с тем, что в выражениях `this` не может быть lvalue. — *Прим. ред.*

имен[4]), то, в соответствии с принципом интерфейса, f является логической частью X, поскольку образует часть интерфейса X.

Таким образом, принцип интерфейса можно считать хорошим критерием для определения того, что именно является частью класса. Вас все еще смущает трактовка свободной функции как части класса? Давайте рассмотрим пример с другим именем функции f.

```
//*** Пример 1в
class X { /* ... */ };
/* ... */
ostream& operator<<( ostream&, const X& );
```

В данном случае обоснование принципа интерфейса совершенно очевидно, поскольку мы знаем, как работает данная свободная функция. Если operator<< "поставляется с" X (например, в одном и том же заголовочном файле и/или пространстве имен), то operator<< является логической частью X, поскольку образует часть его интерфейса. Это имеет смысл, несмотря на то что функция не является членом, поскольку обычно авторы класса обеспечивают наличие оператора <<. Если же оператор << представлен клиентским кодом, а не автором класса X, то этот оператор не является частью X, поскольку не "поставляется с" X.[5]

В очередной раз вернемся к традиционному определению класса.

> *Класс описывает множество данных вместе с функциями, оперирующими этими данными.*

Это определение абсолютно точное; все дело в том, что в нем ничего не говорится о том, являются ли эти функции членами или нет.

Когда я говорил о том, что означает "поставляется с", я упоминал термины C++, например пространства имен. Соответственно, встает вопрос о том, является ли принцип интерфейса специфичным для C++ или это общий принцип объектно-ориентированного программирования, который может быть применен и в других языках?

Рассмотрим хорошо знакомый пример из другого (не объектно-ориентированного) языка программирования.

```
/*** Пример 2а ***/
struct _iobuf { /* ...здесь располагаются данные... */ };
typedef struct _iobuf FILE;
FILE* fopen ( const char* filename,
              const char* mode );
int   fclose( FILE* stream );
int   fseek ( FILE* stream,
              long  offset,
              int   origin );
long  ftell ( FILE* stream );
      /* и т.д. */
```

Это стандартная технология с использованием дескриптора для записи объектно-ориентированного кода в языке, в котором нет классов. Вы предоставляете структуру, которая хранит данные объекта, и функции (обязательно нечлены), которые получают в качестве параметра указатель на эту структуру. Это свободные функции для создания объекта (fopen), уничтожения (fclose) и работы с данными (fseek, ftell и т.д.).

Такой способ имеет свои недостатки (например, при этом приходится полагаться на то, что программист не станет работать с данными непосредственно), но это вполне объектно-ориентированный метод — ведь, в конце концов, класс представляет со-

---

[4] Детально отношения между пространствами имен будут рассмотрены нами чуть позже.

[5] Подобие функций-членов и свободных функций еще более сильно в случае некоторых других перегружаемых операторов. Например, когда вы пишете a+b, в зависимости от типов a и b может вызываться как a.operator+(b), так и operator+(a,b).

бой “множество данных вместе с функциями, оперирующими этими данными”. В нашем случае все функции вынуждены быть свободными, но при этом они остаются частью интерфейса `FILE`.

Теперь рассмотрим “обычный” способ, как переписать пример 2*а* на языке, поддерживающем классы.

```
//*** Пример 2б
class FILE
{
public:
    FILE( const char* filename,
          const char* mode );
    ~FILE();
    int  fseek( long offset, int origin );
    long ftell();
        /* и т.д. */
private:
    /* ...здесь располагаются данные... */
};
```

Теперь параметр `FILE*` становится неявным параметром `this`, и становится совершенно очевидно, что `fseek` является частью `FILE`, так же, как и в примере 2*а*, несмотря на то, что там эта функция была свободна. Мы даже можем сделать некоторые функции членами, а некоторые — нет.

```
//*** Пример 2в
class FILE
{
public:
    FILE( const char* filename,
          const char* mode );
    ~FILE();
    long ftell();
        /* и т.д. */
private:
    /* ...здесь располагаются данные... */
};
int fseek( FILE* stream,
           long offset,
           int origin );
```

В действительности не имеет значения, являются ли функции членами или нет. До тех пор пока они “упоминают” `FILE` и “поставляются с” `FILE`, они являются частью `FILE`. В примере 2*а* все функции были свободными, поскольку C не поддерживает функции-члены. Более того, даже в C++ часть функций интерфейса класса может (или должна) быть свободными. Так, оператор `<<` не может быть членом, поскольку требует в качестве левого аргумента объект потока, а оператор `+` не должен быть членом, чтобы обеспечить возможность преобразования типа левого аргумента.

Поиск Кёнига

Принцип интерфейса предстанет перед вами в новом свете, когда вы поймете, что он делает по сути то же, что и поиск К□нига. Я приведу два примера для иллюстрации и определения поиска К□нига, а в следующем разделе использую пример Майерса, чтобы показать непосредственную связь поиска с принципом интерфейса.

Воспользуемся тем же примером из стандарта C++, что и в задаче 5.1.

```
//*** Пример 3а
namespace NS
{
    class T { };
    void f(T);
```

```
}
NS::T parm;
int main()
{
    f(parm);  // Вызов NS::f
}
```

“Очевидно”, что программист не должен явно записывать `NS::f(parm)`, поскольку “очевидно”, что `f(parm)` означает `NS::f(parm)`, не так ли? Но то, что очевидно для нас, не всегда очевидно для компилятора, особенно если учесть, что в коде нет ни одной инструкции `using`, которая бы могла внести `f` в область видимости. Поиск К□нига позволяет компилятору корректно работать с таким фрагментом.

Вот как работает компилятор. Вспомним, что поиск имени означает, что когда вы записываете вызов наподобие `f(parm)`, компилятор должен вычислить, какую именно функцию с именем `f` вы имеете в виду (с учетом перегрузки и областей видимости у вас может быть несколько функций с таким именем). Поиск К□нига гласит, что если вы передаете функции аргумент типа класса (в нашем случае `parm` типа `NS::T`), то при поиске функции компилятор просматривает не только все “обычные места”, как, например, локальную область видимости, но и пространство имен (в нашем случае `NS`), которое содержит тип аргумента.[6] Поэтому пример *3а* вполне работоспособен: передаваемый функции `f` параметр имеет тип `T`, определенный в пространстве имен `NS`, так что компилятор ищет `f` в том числе и в этом пространстве имен.

То, что мы не должны указывать полное имя функции `f`, является несомненным преимуществом, поскольку бывают ситуации, когда невозможно легко указать полное имя функции.

```
//*** Пример 3б
#include <iostream>
#include <string>   // Здесь объявлена функция
                    // std::operator<< для типа string
int main()
{
    std::string hello = "Hello, world";
    std::cout << hello; // Вызывается std::operator<<
}
```

В этой ситуации, не будь поиска К□нига, компилятор не смог бы найти `operator<<`, поскольку последний является свободной функцией, известной только как часть пакета `string`. Указывать полное имя данного оператора вместо естественного его использования было бы просто издевательством. Нам бы пришлось либо писать “`std::operator<<(std::cout,hello)`”, что выглядит исключительно безобразно; либо использовать инструкцию “`using std::operator<<;`”, что при наличии большого количества используемых операторов раздражает и утомляет; либо использовать одну инструкцию “`using namespace std;`”, что сводит на нет большинство преимуществ использования пространств имен. Надеюсь, после этого вам стала совершенно очевидна важность поиска К□нига.

Словом, если вы поставляете класс и свободную функцию, которая упоминает этот класс,[7] в одном и том же пространстве имен, компилятор обеспечивает их тесную взаимосвязь.[8]

---

[6] На самом деле имеются некоторые тонкости, но суть изложена вполне корректно.

[7] Как угодно — по значению, ссылке, как указатель.

[8] Эта взаимосвязь остается менее строгой, чем взаимосвязь между классом и одной из его функций-членов. См. врезку *Насколько сильным является взаимоотношение “быть частью”* далее в этой задаче.

Пример Майерса

Сначала рассмотрим (немного) упрощенный пример.

```
//*** Пример 4а
namespace NS        // Обычно эта часть
{                   // находится в некотором
    class T { };    // заголовочном файле T.h
}

void f( NS::T );
int main()
{
    NS::T parm;
    f( parm );      // Вызов глобальной f
}
```

В пространстве имен NS содержится тип T, а вне пространства имен имеется функция f, получающая параметр этого типа. Все в порядке, травка зеленеет, солнышко блестит...

Но в один прекрасный день автор NS предупредительно добавляет в пространство имен функцию f.

```
//*** Пример 4б
namespace NS        // Обычно эта часть
{                   // находится в некотором
    class T { };    // заголовочном файле T.h
    void f( T );    // <-- Новая функция
}

void f( NS::T );
int main()
{
    NS::T parm;
    f( parm );      // Неоднозначность: вызов
                    // NS::f или глобальной f?
}
```

Итак, добавление функции в область видимости пространства имен портит код вне этого пространства, при том что клиент нигде не использовал инструкцию using для внесения имен из пространства NS в область видимости! Но и это еще не все — Натан Майерс (Nathan Myers) привел интересный пример совместного использования пространств имен и поиска К□нига.

```
//*** Пример Майерса: "До"
namespace A
{
    class X { };
}

namespace B
{
    void f( A::X );
    void g( A::X parm )
    {
        f(parm);    // Вызов B::f
    }
}
```

Все в порядке, все отлично работает. Пока в один прекрасный день автор A не добавляет в пространство имен функцию f.

```
//*** Пример Майерса: "После"
namespace A
{
```

```
    class X { };
    void f( X );   // <-- Новая функция
}

namespace B
{
    void f( A::X );
    void g( A::X parm )
    {
        f(parm);   // Вызов A::f или B::f?
    }
}
```

Итак, что же произошло? Назначение пространств имен состоит в избежании коллизий имен, не так ли? И при этом добавление имени в одном пространстве имен приводит к нарушению работоспособности кода в совершенно другом пространстве имен. Создается впечатление, что код в пространстве имен B оказывается нарушенным в связи с тем, что в нем упоминается тип из пространства имен A.

Однако на самом деле никакой проблемы нет, и код в пространстве имен B остается работоспособным. В действительности происходит *в точности* то, что и должно произойти.[9] Если в том же пространстве имен, где находится X, имеется функция f(X), то, согласно принципу интерфейса, f является частью интерфейса X. Не имеет ни малейшего значения, что f является свободной функцией. Для того чтобы было понятно, что эта функция логически является частью X, дадим ей другое имя.

```
//*** Преобразованный пример Майерса: "После"
namespace A
{
    class X { };
    ostream& operator<<( ostream&, const X& );
}

namespace B
{
    ostream& operator<<( ostream&, const A::X& );
    void g( A::X parm )
    {
        cout << parm;   // A::operator<< или
    }                   // B::operator<< ?
}
```

Насколько сильным является взаимоотношение “быть частью”

Хотя принцип интерфейса и утверждает, что как функции-члены, так и свободные функции могут быть логически частью класса, это не означает, что члены и нечлены эквивалентны. Например, функции-члены автоматически получают полный доступ ко внутреннему представлению класса, в то время как свободные функции могут получить такой доступ, только будучи друзьями класса. Функции-члены в C++ (в частности, при поиске имен) рассматриваются как более сильно связанные с классом, чем свободные функции.

```
//*** НЕ пример Майерса
namespace A
{
    class X { };
    void f( X );
}
class B    // <-- Класс, не пространство имен!
```

[9] Этот пример был приведен на конференции в Морристауне в 1997 году и подтолкнул меня к размышлениям на эту тему.

```
{
    void f( A::X );
    void g( A::X parm )
    {
        f(parm); // Вызов B::f
    }
};
```

Когда мы рассматриваем класс B, а не пространство имен B, никакой неоднозначности не возникает. Когда компилятор находит функцию-член с именем f, он больше не пытается использовать поиск К□нига среди свободных функций.

Таким образом, даже если функция в соответствии с принципом интерфейса является частью класса, то все равно как с точки зрения прав доступа, так и с точки зрения правил поиска имен функция-член сильнее связана с классом, чем свободная функция.

Если в клиентском коде содержится функция, которая упоминает X, а ее сигнатура соответствует сигнатуре функции из пространства имен, содержащего X, то вызов такой функции *должен* быть неоднозначным. Программист *должен* явно указать в пространстве имен B, какую именно функцию он имеет в виду — свою собственную или поставляемую вместе с X. Это именно то поведение, которое следует ожидать в соответствии с принципом интерфейса.

Для класса X все функции, включая свободные, которые

- *"упоминают" X и*
- *"поставляются с" X,*

являются логической частью X, поскольку они образуют часть интерфейса X.

Пример Майерса всего лишь показывает, что пространства имен не настолько независимы друг от друга, как думают многие программисты. И тем не менее пространства имен независимы в достаточной степени для того, чтобы вполне соответствовать своему предназначению.

Нет ничего удивительного в том, что принцип интерфейса работает в точности так же, как и поиск К□нига. Дело в том, что поиск К□нига работает именно так как раз в соответствии с принципом интерфейса.

Задача 5.3. Поиск имен и принцип интерфейса. Часть 3 *Сложность: 5*

Потратим несколько минут и рассмотрим влияние принципа интерфейса на проектирование программ. Вернемся ненадолго к классическому вопросу проектирования: какой способ написания оператора << наилучший?

Имеется два основных способа создания оператора << для класса: как свободной функции, использующей только обычный интерфейс класса (возможно, с непосредственным доступом к закрытым частям класса при объявлении функции как друга класса), либо как свободной функции, вызывающей виртуальную вспомогательную функцию, являющуюся членом класса.

Какой метод лучше?

Решение

Если вас удивило название задачи (*Поиск имен и принцип интерфейса. Часть 3*), то скоро ваше удивление пройдет. Дело в том, что сейчас мы рассмотрим применение принципа интерфейса, о котором столько говорили в предыдущей задаче.

От чего зависит класс

“Что такое класс?” — вопрос не только философский. Это фундаментальный практический вопрос, без верного ответа на который мы не сможем правильно проанализировать зависимости класса.

Для того чтобы продемонстрировать это, рассмотрим (на первый взгляд кажущуюся никак не связанной с этим вопросом) задачу определения наилучшего способа написания оператора << для класса. Имеются два основных метода, каждый со своими преимуществами и недостатками, которые я и проанализирую.

Вот первый способ.

```
//*** Пример 1а - не виртуальный
class X
{
    /* ostream в классе не упоминается */
};

ostream& operator<<( ostream& o, const X& x )
{
    /* Код вывода X в поток */
    return o;
}
```

А вот — второй.

```
//*** Пример 1б - виртуальный
class X
{
    /* ... */
public:
    virtual ostream& print( ostream& ) const
};
ostream& X::print( ostream& ) const
{
    /* Код вывода X в поток */
    return o;
}
ostream& operator<<( ostream& o, const X& x )
{
    return x.print( o );
}
```

Предположим, что в обоих случаях класс и объявление функции находятся в одном и том же заголовочном файле и/или пространстве имен. Какой из этих способов вы выберете? В чем заключаются недостатки каждого из них? Опытные программисты на C++ анализируют перечисленные методы следующим образом.

- Преимущество примера 1*а* в меньшем количестве зависимостей класса. Поскольку ни одна функция-член не упоминает `ostream`, `X` не зависит (*по крайней мере так это выглядит*) от `ostream`. Кроме того, при этом нет накладных расходов на вызов виртуальной функции.
- Преимущество примера 1*б* заключается в том, что в поток корректно выводятся все производные от `X` классы, даже при передаче оператору << ссылки `X&`.

Таков традиционный анализ предложенных методов. Увы, этот анализ ошибочен, и, вооруженные принципом интерфейса, мы можем сказать, почему: первое преимущество примера 1*б* не более чем фантом, как сказано в скобках курсивом при его описании.

1. В соответствии с принципом интерфейса, поскольку `operator<<` упоминает `X` как в первом, так и во втором случае, и в обоих случаях поставляется с `X`, `operator<<` является логической частью `X`.

2. В обоих случаях `operator<<` упоминает `ostream`, так что `operator<<` зависит от `ostream`.
3. Поскольку в обоих случаях `operator<<` является логической частью `X` и зависит от `ostream`, следовательно, `X` также в обоих случаях зависит от `ostream`.

Итак, основное преимущество первого метода оказалось мыльным пузырем. Как при использовании первого, так и второго метода `X` остается зависимым от `ostream`. Если, как обычно, и `X`, и `operator<<` находятся в одном и том же заголовочном файле, то в обоих случаях как модули реализации `X`, так и клиентские модули, использующие `X`, физически зависят от `ostream` и требуют для успешной компиляции как минимум его предварительного объявления.

Все преимущества первого метода свелись к отсутствию накладных расходов на вызов виртуальной функции. Согласитесь, что прийти к правильному выводу о реальных зависимостях (и, соответственно, реальных преимуществах и недостатках рассматриваемых методов) без применения принципа интерфейса было бы не так легко.

Как видите, как и подразумевает принцип интерфейса, не всегда полезно отличать функции-члены и свободные функции, и в особенности это проявляется при анализе зависимостей.

Некоторые интересные результаты

В целом, если `A` и `B` являются классами, а `f(A,B)` — свободной функцией, то выполняется следующее.

- Если `A` и `f` поставляются вместе, то `f` является частью `A`, и `A` зависит от `B`.
- Если `B` и `f` поставляются вместе, то `f` является частью `B`, и `B` зависит от `A`.
- Если `A`, `B` и `f` поставляются вместе, то `f` является частью `A` и `B`, так что `A` и `B` взаимозависимы. До сих пор такие выводы делались на интуитивном уровне: если автор библиотеки поставляет два класса и операцию, которая использует их оба, то, вероятно, оба класса и функция предназначены для совместного использования. Теперь же наличие принципа интерфейса дает нам возможность более точно сформулировать эту взаимозависимость.

И наконец, действительно интересный случай. В целом, если `A` и `B` являются классами и `A::g(B)` — функция-член `A`, то справедливо следующее.

- Очевидно, что существование `A::g(B)` обуславливает зависимость `A` от `B`. Здесь нет никаких сюрпризов.
- Если `A` и `B` поставляются вместе, то, конечно, вместе поставляются `B` и `A::g(B)`. Таким образом, поскольку `A::g(B)` “упоминает” `B` и “поставляется с” `B`, то, в соответствии с принципом интерфейса, из этого следует, что `A::g(B)` является частью `B`. А поскольку `A::g(B)` неявно использует параметр `A*`, то `B` зависит от `A`. Поскольку `A` также зависит от `B`, это означает, что `A` и `B` взаимозависимы.

На первый взгляд кажется странным рассматривать функцию-член одного класса как часть другого класса, но это справедливо только в том случае, когда `A` и `B` *поставляются вместе*. В этом случае (например, когда `A` и `B` располагаются в одном заголовочном файле) и `A` упоминает `B` в функции-члене наподобие рассматриваемой, внутренний голос говорит нам, что, вероятно, классы `A` и `B` взаимосвязаны. Эта взаимосвязанность и факт совместной поставки и взаимодействия классов означает, что а) они предназначены для совместного использования и б) изменения одного из них влияют на другой.

Проблема в том, что, если не прислушиваться ко внутреннему голосу, увидеть взаимозависимость `A` и `B` достаточно сложно. Однако теперь можно продемонстрировать эту взаимозависимость как прямое следствие принципа интерфейса.

Заметим, что, в отличие от классов, пространствам имен не обязательно быть объявленным одновременно и что означает “поставляется вместе”, зависит от того, какие именно части пространства имен оказываются видимыми.

```
//*** Пример 2
//    Файл a.h
namespace N { class B; }  // Предварительное объявление
namespace N { class A; }  // Предварительное объявление
class N::A { public: void g(B); };

//    Файл b.h
namespace N { class B { /* ... */ }; }
```

Клиенты A включают в свои программы заголовочный файл a.h, так что для них и A, и B поставляются вместе и являются взаимозависимыми. Клиенты же B включают заголовочный файл b.h, так что с их точки зрения классы A и B не поставляются вместе.

В заключение мне бы хотелось предложить вам три вывода из серии задач *Поиск имен и принцип интерфейса.*

1. Принцип интерфейса: для класса X все функции, включая свободные, которые “упоминают” X и “поставляются с” X, являются логической частью X, поскольку они образуют часть интерфейса X.
2. Таким образом, логически частью класса могут быть как функции-члены, *так и свободные* функции. Однако функция-член связана с классом в большей степени, чем свободная функция.
3. Трактовать в принципе интерфейса “поставляется с” можно как “находится в том же заголовочном файле и/или пространстве имен”. Если функция находится в том же заголовочном файле, что и класс, с точки зрения зависимостей она является частью класса. Если же функция находится с классом в одном пространстве имен, то она является частью класса в смысле использования объекта и поиска имен.

Задача 5.4. Поиск имен и принцип интерфейса. Часть 4 *Сложность: 9*

В завершение данной главы рассмотрим влияние принципа интерфейса на поиск имен. Сможете ли вы указать скрытые в приведенном коде проблемы?

1. Что такое сокрытие имен? Покажите, каким образом оно может влиять на видимость имен базовых классов в производных классах.
2. Будет ли корректно скомпилирован приведенный далее фрагмент кода? Постарайтесь по возможности дать наиболее полный ответ. Постарайтесь найти и прояснить все имеющиеся неясности.

```
//*** Пример 2. Скомпилируется ли данный код?
//
//    Заголовочный файл некоторой библиотеки
namespace N { class C { }; }
int operator+( int i, N::C ) { return i+1; }

// Использование
#include <numeric>
int main()
{
    N::C a[10];
    std::accumulate(a, a+10, 0);
}
```

Решение

1. **Что такое сокрытие имен? Покажите, каким образом оно может влиять на видимость имен базовых классов в производных классах.**

Сокрытие имен

Рассмотрим следующий пример.

```
//  Пример 1а. Сокрытие имени из базового класса
//
struct B
{
    int f( int );
    int f( double );
    int g( int );
};

struct D : public B
{
private:
    int g( std::string, bool );
};

D   d;
int i;
d.f(i);  // Вызов B::f(int)
d.g(i);  // Ошибка: g получает 2 аргумента
```

Большинство читателей книги наверняка сталкивались с таким сокрытием имен, хотя для многих новичков в программировании на C++ тот факт, что последняя строка не компилируется, оказывается неприятным (и непонятным) сюрпризом. Вкратце, при объявлении функции с именем `g` в производном классе `D`, автоматически скрываются все функции с тем же именем во всех непосредственных и опосредованных базовых классах. Не имеет значения "очевидность" того, что `D::g` не может быть вызываемой функцией (кстати, `D::g` не только имеет неподходящую сигнатуру, но еще и является закрытой функцией, которая не может быть вызвана вне класса), поскольку функция `B::g` скрыта и не рассматривается при поиске имен.

Для того чтобы понять, что происходит, рассмотрим подробнее действия компилятора, встретившего вызов `d.g(i)`. Сначала он просматривает непосредственную область видимости (в данном случае область видимости `D`) и составляет список всех функций с именем `g`, которые может найти (независимо от их доступности или количества параметров). И *только если компилятор не находит ни одной такой функции*, он продолжает поиск в охватывающей области видимости (в нашем случае — в области видимости `B`). Так происходит до тех пор, пока в конце концов будут просмотрены все области видимости, но функция с таким именем окажется не найденной, либо пока не будет обнаружена область видимости, содержащая по крайней мере одну функцию с искомым именем. Если найдена область видимости, содержащая хотя бы одну искомую функцию, компилятор прекращает поиск и работает с найденными кандидатами, выполняя разрешение перегрузки и применяя правила доступа.

Имеется ряд весьма основательных причин для работы именно таким образом.[10] Можно рассмотреть один предельный случай — когда точно соответствующей вызову глобальной функции следует предпочесть функцию-член с неточным соответствием, требующим некоторого преобразования типов.

Обход нежелательного сокрытия имен

Само собой, выйти из неприятной ситуации, показанной в примере 1*а*, можно, причем даже несколькими путями. Первый путь состоит в том, что в вызывающем коде точно указывается область видимости, в которой должен осуществляться поиск.

```
//  Пример 1б. Поиск имени в базовом классе
//
D   d;
int i;
d.f(i);      // Вызов B::f(int)
d.B::g(i);  // Вызов  B::g(int)
```

Второй путь более удобен и состоит в том, что разработчик класса `D` делает функцию `B::g` видимой с помощью объявления `using`. Это позволяет компилятору при поиске имен и последующем разрешении перегрузки рассматривать `B::g` в одной области видимости с `D::g`.

```
//  Пример 1в. Открытие имени из базового класса
//
struct D : public B
{
    using B::g;
private:
    int g( std::string, bool );
};
```

Каждый из приведенных способов успешно решает проблему сокрытия имен из примера 1*а*.

Пространства имен и принцип интерфейса

Пространства имен следует использовать разумно. Если вы помещаете класс в пространство имен, следует убедиться, что все вспомогательные функции и операторы также размещены в том же пространстве имен. Если это не так, в своем коде вы можете обнаружить неожиданные и нежелательные эффекты.

Следующая простая программа основана на коде, присланном мне читателем Дарином Адлером (Darin Adler). В ней представлен класс `C` в пространстве имен `N` и операция над этим классом. Обратите внимание: `operator+()` находится в глобальном пространстве имен, а не в пространстве `N`. Имеет ли это значение? Корректен ли приведенный код?

---

[10] Например, можно предложить вариант продолжения поиска во внешней области видимости в случае, если ни одна из найденных функций во внутренней области видимости не может быть использована. Однако в ряде случаев этот метод может привести к неожиданным результатам (рассмотрим случай, когда во внешней области видимости имеется функция, точно соответствующая количеству и типу передаваемых параметров, а во внутренней области видимости — функция, требующая небольшого преобразования типов параметров). Еще один вариант заключается в составлении списка всех функций с требуемым именем во всех областях видимости, а затем выполнении разрешения перегрузки по всем областям видимости. Но, увы, здесь тоже есть свои ловушки (функции-члену должно быть отдано предпочтение перед глобальной функцией, чтобы не получить возможную неоднозначность).

2. **Будет ли корректно скомпилирован приведенный далее фрагмент кода? Постарайтесь по возможности дать наиболее полный ответ. Постарайтесь найти и прояснить все имеющиеся неясности.**

```
//*** Пример 2. Скомпилируется ли данный код?
//
//    Заголовочный файл некоторой библиотеки
namespace N { class C { }; }
int operator+( int i, N::C ) { return i+1; }

// Использование
#include <numeric>
int main()
{
    N::C a[10];
    std::accumulate(a, a+10, 0);
}
```

Перед тем как читать дальше, остановитесь и подумайте над моей подсказкой: будет ли компилироваться приведенная программа?[11] Переносима ли она?

Сокрытие имен во вложенных пространствах имен

На первый взгляд пример 2 выглядит совершенно корректно, так что правильный ответ вас должен удивить: этот код может скомпилироваться, а может и нет. Это целиком зависит от используемой вами реализации языка. Лично мне знакомы как удовлетворяющие стандарту реализации, которые компилируют этот код, так и в не меньшей степени удовлетворяющие стандарту реализации, которые отказываются скомпилировать приведенный фрагмент. Усаживайтесь поудобнее, и я расскажу вам, почему это так...

Ключом к пониманию ответа является понимание того, что компилятор делает внутри `std::accunulate`. Этот шаблон выглядит примерно так.

```
namespace std
{
    template<class Iter, class T>
    inline T accumulate( Iter first,
                         Iter last,
                         T    value )
    {
        while( first != last )
        {
            value = value + *first; //1
            ++first;
        }
        return value;
    }
}
```

Код в примере 2 вызывает `std::accumulate<N::C*,int>`. Каким образом в строке `//1` компилятор интерпретирует выражение `value + *first`? Он начнет искать `operator+()`, который получает в качестве параметров `int` и `N::C` (или параметры, которые могут быть преобразованы в `int` и `N::C`). Но как раз такой `operator+(int,N::C)` имеется в глобальной области видимости! Отлично, значит, все в порядке, не так ли?

---

[11] Если вы думаете, что потенциальные проблемы переносимости зависят от того, использует ли реализация `std::accumulate()` оператор `operator+(int,N::C)` или `operator+(N::C,int)`, то вы ошибаетесь, проблема не в этом. Стандарт гласит, что будет использоваться первый из перечисленных операторов, так что в примере приведен оператор с правильной сигнатурой.

Проблема в том, что компилятор может увидеть `operator+(int,N::C)` в глобальной области видимости, но может и не увидеть, в зависимости от того, какие другие функции будут объявлены в пространстве `std` к моменту настройки `std::accumulate<N::C*,int>`.

Чтобы понять, почему это так, следует учесть, что сокрытие имен, которое рассматривалось нами на примере производных классов, относится к любым вложенным областям видимости, включая пространства имен, а также выяснить, *где* компилятор начинает поиск подходящего `operator+()`. (Здесь я повторю мое пояснение, которое я уже приводил при решении данной задачи, просто заменив некоторые названия.) Сначала он просматривает непосредственную область видимости, в данном случае область видимости пространства имен `std`, и составляет список всех функций с именем `operator+()`, которые может найти (независимо от их доступности или количества параметров). И *только если компилятор не находит ни одной такой функции*, он продолжает поиск в охватывающей области видимости (в нашем случае — в области видимости, охватывающей пространство имен `std`, т.е. в глобальной области видимости) — и так до тех пор, пока в конце концов будут просмотрены все области видимости, но функция с таким именем окажется не найденной, либо пока не будет обнаружена область видимости, содержащая по крайней мере одну функцию с искомым именем. Если область видимости с по меньшей мере одной функцией-кандидатом найдена, компилятор прекращает поиск и работает с найденными кандидатами, выполняя разрешение перегрузки и применяя правила доступа.

Будет ли скомпилирован пример 2, целиком зависит от следующих условий: а) включает ли стандартный заголовочный файл `numeric` объявление `operator+()` (любого оператора +, неважно, подходящего ли по количеству и типу параметров и доступного для вызова или нет), или б) включает ли стандартный заголовочный файл `numeric` другой заголовочный файл с объявлением `operator+()`. В отличие от стандарта C, стандарт C++ не определяет, какие заголовочные файлы включают друг друга, так что при включении заголовочного файла `numeric` в некоторых реализациях возможно, например, включение заголовочного файла `iterator`, в котором определен ряд функций `operator+()`. Мне встречались разные компиляторы C++ — как те, которые не компилировали пример 2, так и те, которые его успешно компилировали (но могли перестать это делать при включении, например, директивы `#include<vector>`).

Шутки компиляторов

Достаточно неприятно, когда компилятор не в состоянии найти нужную функцию из-за того, что при поиске ему встречается другой `operator+()`. Однако обычно этот оператор, находящийся в стандартном заголовочном файле, является шаблоном, а в таких случаях компилятор генерирует совершенно неудобочитаемые сообщения об ошибках, что еще неприятнее. Вот как сообщает о случившемся при компиляции примера 2 одна популярная реализация C++ (в которой, кстати говоря, заголовочный файл `numeric` включает заголовочный файл `iterator`).

```
error C2784: 'class std::reverse_iterator<'template-
parameter-1','template-parameter-2','template-parameter
-3','template-parameter-4','template-parameter-5'>
__cdecl std::operator+(template-parameter-5,const class
std::reverse_iterator<'template-parameter-1','template-
parameter-2','template-parameter-3','template-parameter-
4','template-parameter-5'>&)' : could not deduce template
argument for 'template-parameter-5' from 'int'

error C2677: binary '+' : no global operator defined which
takes type 'class N::C' (or there is no acceptable
conversion)
```

Уфф!... А теперь представьте смятение программиста, прочитавшего такое сообщение...

- Первое сообщение совершенно неудобочитаемо. Компилятор просто жалуется (настолько ясно, насколько он в состоянии это сделать), что он нашел `operator+()`, но не в состоянии использовать его так, как требуется. Это сообщение мало чем может помочь слабому программисту. “Не понял, — скажет такой программист, почесывая затылок, — при чем здесь `reverse_iterator`, если в моей программе его и близко нет?”
- Второе сообщение вообще представляет собой вопиющую ложь (хотя, наверное, можно если не оправдать, то понять автора данной реализации: это сообщение в большинстве случаев было вполне корректно до того, как начали широко использоваться пространства имен). Оно достаточно близко к корректному сообщению “*не найден* оператор, который...”, но и это сообщение мало чем может помочь неопытному программисту. “Что?! — возмутится такой программист. — А что же тогда такое `int operator+(int i,N::C)`, как не глобальный оператор, принимающий параметр типа `class N::C`?!”

Как видите, обычному программисту нелегко даже просто выяснить, что же именно произошло. А ведь всего этого можно легко избежать...

Решение

Когда мы столкнулись с аналогичной проблемой в ситуации с сокрытием базового имени в производном классе, у нас было два возможных решения: явно указать вызываемую функцию (пример 1*б*) или использовать объявление `using` для внесения требуемой функции в требуемую область видимости (пример 1*в*). В данном случае ни одно из этих решений не применимо. Первое из них хотя и возможно,[12] но чересчур многого требует от программиста; второе попросту невозможно.

Реальное же решение состоит в размещении нашего `operator+()` там, где ему и следует находиться: в пространстве имен `N`.

```
// Пример 2б. Решение
//
// В некотором заголовочном файле библиотеки
namespace N
{
    class C {};
    int operator+(int i, N::C) { return i+1; }
}

// Использование
#include <numeric>
int main()
{
    N::C a[10];
    std::accumulate(a, a+10, 0); // OK
}
```

Этот код переносим и будет корректно скомпилирован всеми компиляторами, удовлетворяющими стандарту, независимо от того, что именно определено в пространстве имен `std` или каком-либо ином. Теперь наш `operator+()` находится в том же пространстве имен, что и его второй параметр, и когда компилятор пытается разрешить вызов оператора “+” в `std::accumulate`, в соответствии с поиском К□нига он способен найти необходимый `operator+()`. Вспомним, что поиск К□нига гласит, что в дополнение к просмотру всех обычных областей видимости компилятор должен также просмот-

[12] Требуя от программиста использования версии `std::accumulate`, которая принимает в качестве параметра предикат, и явного его указания всякий раз при вызове функции... неплохой способ потерять клиентов!

реть все области видимости типов параметров функции. `N::C` находится в пространстве имен `N`, следовательно, компилятор осуществляет поиск необходимого оператора в этом пространстве имен, где и находит искомое (вне зависимости от того, сколько операторов `operator+()` оказываются разбросанными по пространству имен `std`).

Итак, проблема состоит в том, что пример 2 нарушил принцип интерфейса.

> *Для класса `X` все функции, включая свободные, которые*
>
> - *"упоминают" `X` и*
> - *"поставляются с" `X`,*
>
> *являются логической частью `X`, поскольку они образуют часть интерфейса `X`.*

Если операция, представленная свободной функцией (и в особенности оператором), упоминает класс и предназначена для использования в качестве части интерфейса класса, то она должна поставляться с классом, что помимо прочего означает, что она должна находиться в том же пространстве имен, что и класс. Проблема в примере 2 возникла постольку, поскольку мы написали класс `C` и *поместили часть его интерфейса в другое пространство имен*. Класс и его интерфейс, невзирая ни на что, должны всегда располагаться вместе, и в этом заключается простейший способ избежать многих сложных проблем поиска имен позже, когда другие программисты будут использовать ваш класс.

Используйте пространства имен разумно. Либо размещайте класс полностью в одном пространстве имен, включая сущности, которые неискушенному глазу представляются не являющимися частью класса, например, свободные функции, упоминающие класс (не забывайте о принципе интерфейса!), либо не помещайте класс в пространство имен вообще. Ваши пользователи при выполнении этого правила будут вам только благодарны.

Рекомендация

Разумно используйте пространства имен. Если вы помещаете класс в пространство имен, убедитесь, что вы разместили в том же пространстве все вспомогательные функции и операторы. Если не сделать этого, в своем коде вы можете обнаружить самые неожиданные и нежелательные сюрпризы.

6

Управление памятью и ресурсами

Интеллектуальные указатели и `std::auto_ptr` — ключевой инструментарий в нашем распоряжении для безопасного и эффективного управления памятью и другими ресурсами, включая такие объекты реального мира, как дисковые файлы или блокировки транзакций баз данных. Каким образом избежать ловушек при использовании этих инструментов? Можно ли использовать `auto_ptr` в качестве члена класса? О чем следует не забывать при его использовании? Можно ли избежать ряда проблем, создав свой специализированный интеллектуальный указатель?

Задача 6.1. Управление памятью. Часть 1 *Сложность: 3*

Насколько хорошо вы знакомы с памятью? Какие различные области памяти существуют?

C++ имеет несколько различных областей памяти, в которых хранятся объекты и значения, не являющиеся объектами, и каждая из этих областей обладает своими характеристиками.

Перечислите все области, которые вы знаете, а также приведите их характеристики производительности и опишите время жизни хранящихся в них объектов.

Пример: в стеке хранятся автоматические переменные, как встроенных типов, так и объекты классов.

Решение

В приведенной далее табл. 6.1 перечислены различные области памяти программ на языке C++. Некоторые из перечисленных имен (например, “куча”) как таковые в стандарте отсутствуют; в частности, “куча” и “свободная память” представляют собой удобные обозначения для того, чтобы различать два вида динамически выделяемой памяти.

Таблица 6.1. Области памяти C++

Область памяти	Характеристика и время жизни объектов
Константные данные (Const Data)	В этой области памяти хранятся строки и другие данные, чьи значения известны во время компиляции. В этой области не могут находиться объекты классов. Все данные из этой области доступны в течение всего времени жизни программы. Кроме того, все данные в этой области памяти доступны только для чтения, и результат попытки их изменения в процессе работы программы не определен. В частности, это связано с тем, что формат хранения этих данных может быть оптимизирован конкретной реализацией языка. Например, компилятор может оптимизировать хранение строк и разместить их в перекрывающихся объектах
Стек (Stack)	В стеке хранятся автоматические переменные. Объекты конструируются немедленно в точке определения и уничтожаются немедленно в конце области видимости, так что у программиста нет никакой возможности непосредственно работать с выделенным, но неинициализированным пространством стека (исключая преднамеренные манипуляции с использованием явного вызова деструкторов и размещающего оператора `new`). Выделение стековой памяти обычно происходит гораздо быстрее, чем выделение динамической памяти, поскольку каждое выделение стековой памяти включает только увеличение указателя стека, без какой-либо более сложной системы управления памятью
Свободная (динамическая) память (Free Store)	Свободная память представляет собой одну из двух областей динамической памяти, выделяемую/освобождаемую посредством операторов `new`/`delete`. Время жизни объектов может быть меньшим, чем время нахождения памяти в выделенном состоянии. То есть объектам в этой области может быть выделена память без немедленной инициализации, и объекты могут быть уничтожены без немедленного освобождения выделенной им памяти. Во время между выделением памяти и до инициализации объекта доступ и управление этой памятью может осуществляться посредством указателя типа `void*`, но обращение ни к каким нестатическим членам или функциям-членам невозможно, так же как и получение их адресов или попытки работы с ними любыми иными методами
Куча (Heap)	Куча представляет собой вторую область динамической памяти, выделяемую/освобождаемую посредством функций `malloc()`/`free()` и их вариантов. Заметим, что хотя стандартные операторы `new` и `delete` в конкретных компиляторах могут быть реализованы посредством `malloc()` и `free()`, куча не идентична свободной памяти, и выделенная одним способом память не может быть безопасно освобождена другим способом. Память, выделенная в куче, может использоваться для объектов классов с помощью размещающего оператора `new` и явного вызова деструктора. При таком ее использовании применимы все замечания о времени жизни объектов в свободной памяти

Окончание табл. 6.1

Область памяти	Характеристика и время жизни объектов
Глобальная/статтическая (Global/Static)	Глобальные или статические переменные и объекты получают память при запуске программы, но могут быть не инициализированы до начала ее выполнения. Например, статические переменные в функции инициализируются только при первом прохождении их определения выполняющейся программой. Порядок инициализации глобальных переменных в разных единицах трансляции не определен, поэтому для управления зависимостями между глобальными объектами (включая статические члены классов) требуется принятие специальных мер. Как всегда, доступ к этой памяти и управление ею может осуществляться посредством указателя типа `void*`, но обращение к каким-либо нестатическим членам или функциям-членам вне действительного времени жизни объекта некорректно

Достаточно важно различать кучу и свободную память, поскольку стандарт умышленно оставляет вопрос о взаимоотношении этих областей памяти открытым. Например, при освобождении памяти посредством `::operator delete()` последнее замечание в разделе 18.4.1.1 стандарта C++ гласит следующее.

> *“Не определено, при каких условиях часть или вся освобожденная таким образом память будет выделена последующим вызовом `operator new` или любой из функций `calloc`, `malloc` или `realloc`, объявленным в `<cstdlib>`”.*

Кроме того, не определено, должны ли операторы `new/delete` быть реализованы посредством `malloc/free` в конкретном компиляторе. Однако `malloc/free` *не* должны быть реализованы посредством `new/delete` в соответствии с разделами 20.4.6.3-4 стандарта.

> *“Функции `calloc()`, `malloc()` и `realloc()` не должны пытаться выделить память, используя вызов `::operator new()`”.*
>
> *“Функция `free()` не должна пытаться освободить память, используя вызов `::operator delete()`”.*

Куча и свободная память ведут себя по-разному, обращение к ним также осуществляется по-разному, таким образом, и использовать их вы также должны по-разному.

Рекомендация

Следует четко различать пять различных типов памяти: стек (автоматические переменные); свободная память (`new/delete`); куча (`malloc/free`); глобальная область (статические, глобальные переменные, переменные области видимости файла и т.п.); постоянные данные (строки и т.п.).

Рекомендация

Предпочтительно использовать свободную память (`new/delete`); использования кучи (`malloc/free`) следует избегать.

Задача 6.2. Управление памятью. Часть 2 — *Сложность: 6*

Пробовали ли вы использовать управление памятью в своих классах (или даже заменить глобальные `new` и `delete` собственными операторами)? Перед тем как делать это, обдумайте хорошенько предложенную задачу.

В предлагаемых далее фрагментах кода классы самостоятельно управляют памятью. Укажите по возможности все связанные с памятью ошибки, а также ответьте на дополнительные вопросы.

1. Рассмотрим следующий код.

```
class B
{
public:
    virtual ~B();
    void operator delete  ( void*, size_t ) throw();
    void operator delete[]( void*, size_t ) throw();
    void f( void*, size_t ) throw();
};
class D : public B
{
public:
    void operator delete  ( void* ) throw();
    void operator delete[]( void* ) throw();
};
```

   Почему операторы `delete` в классе `B` имеют вторые параметры, а в классе `D` — нет? Какие способы совершенствования объявлений функций вы видите?

2. Приведенный далее фрагмент кода является продолжением предыдущего. Какой из операторов `delete()` вызывается в каждом из выражений? Почему, и с какими параметрами?

```
D* pd1 = new D;
delete pd1;
B* pb1 = new D;
delete pb1;
D* pd2 = new D[10];
delete[] pd2;
B* pb2 = new D[10];
delete[] pb2;
```

3. Корректны ли два следующих присваивания?

```
B b;
typedef void (B::*PMF)( void*, size_t);
PMF p1 = &B::f;
PMF p2 = &B::operator delete;
```

4. Имеются ли в следующем коде ошибки, связанные с управлением памятью?

```
class X
{
public:
    void* operator new( size_t s, int ) throw( bad_alloc )
    {
        return ::operator new(s);
    }
};
class SharedMemory
{
public:
    static void* Allocate( size_t s )
    {
        return OsSpecificSharedMemAllocation( s );
    }
    static void  Deallocate( void* p, int i = 0 )
    {
        OsSpecificSharedMemDeallocation( p, i );
```

```
    }
};
class Y
{
public:
    void * operator new( size_t s, SharedMemory& m )
                    throw( bad_alloc )
    {
        return m.Allocate( s );
    }
    void  operator delete( void* p,
                           SharedMemory& m,
                           int i ) throw()
    {
        m.Deallocate( p, i );
    }
};
void operator delete( void* p ) throw()
{
    SharedMemory::Deallocate( p );
}
void operator delete( void* p,
                      std::nothrow_t& ) throw()
{
    SharedMemory::Deallocate( p );
}
```

Решение

1. **Рассмотрим следующий код.**

```
class B
{
public:
    virtual ~B();
    void operator delete  ( void*, size_t ) throw();
    void operator delete[]( void*, size_t ) throw();
    void f( void*, size_t ) throw();
};
class D : public B
{
public:
    void operator delete  ( void* ) throw();
    void operator delete[]( void* ) throw();
};
```

Почему операторы `delete` в классе `B` имеют вторые параметры, а в классе `D` — нет?

Ответ прост: это дело вкуса, и не более того. Оба объявления описывают обычные функции освобождения памяти (см. раздел 3.7.3.2/2 стандарта C++).

Однако в этом фрагменте скрыта ошибка, связанная с управлением памятью. Оба класса имеют операторы `delete()` и `delete[]()`, но не имеют соответствующих операторов `new()` и `new[]()`. Это очень опасно, поскольку операторы `new()` и `new[]()` по умолчанию могут некорректно работать с операторами `delete()` и `delete[]()` класса (представьте, например, что может произойти, если в производном классе будут определены собственные операторы `new()` или `new[]()`).

Рекомендация

В классе всегда должны быть либо определены как операторы `new` (`new[]`), так и операторы `delete` (`delete[]`), либо не определен ни один из них.

Теперь рассмотрим вторую часть вопроса — **какие способы совершенствования объявлений функций вы видите?**

Как `operator new()`, так и `operator delete()` всегда являются статическими функциями, даже если они не объявлены как `static`. Хотя C++ не заставляет вас явно использовать описание `static` при объявлении этих операторов, лучше все же сделать это, поскольку это будет служить напоминанием как вам самим при написании кода, так и программистам, которые будут в дальнейшем осуществлять его поддержку.

Рекомендация

Всегда явно объявляйте `operator new()` и `operator delete()` как статические функции. Нестатическими функциями-членами они быть не могут.

2. Приведенный далее фрагмент кода является продолжением предыдущего. Какой из операторов `delete()` вызывается в каждом из выражений? Почему, и с какими параметрами?

```
D* pd1 = new D;
delete pd1;
```

Здесь вызывается `D::operator delete(void*)`.

```
B* pb1 = new D;
delete pb1;
```

Здесь также вызывается `D::operator delete(void*)`. Поскольку деструктор `B` виртуальный, очевидно, что вызывается деструктор `D`; однако виртуальность деструктора `B` неявно означает также, что должен вызываться `D::operator delete()`, несмотря на то, что он не является (да и не может быть) виртуальным.

В качестве отступления для тех, кто интересуется, каким образом компиляторы реализуют такие вещи. Обычно использующийся метод состоит в том, что каждому деструктору сопоставляется невидимый флаг "следует ли при уничтожении объекта освобождать память?" (который равен `false` при уничтожении автоматического и `true` — при уничтожении динамического объекта). Последнее, что делает сгенерированный деструктор, — проверяет этот флаг, и, если его значение равно `true`, вызывает корректный `operator delete()`.[1] Такая методика автоматически гарантирует корректное поведение, а именно — "виртуальное" поведение оператора `delete`, несмотря на то, что он является статической функцией, которая не может быть виртуальной.

```
D* pd2 = new D[10];
delete[] pd2;
```

Здесь вызывается `D::operator delete[](void*)`.

```
B* pb2 = new D[10];
delete[] pb2;
```

Поведение этого выражения не определено. Язык требует, чтобы статический тип указателя, передаваемого `operator delete[]()`, соответствовал его динамическому типу. Дополнительную информацию по этому вопросу можно найти в [Meyers99].

[1] Разработка метода уничтожения массива остается читателю в качестве задания.

Рекомендация

Никогда не работайте с массивами полиморфно.

Рекомендация

Вместо массивов лучше использовать `vector<>` или `deque<>`.

3. Корректны ли два следующих присваивания?

```
B b;
typedef void (B::*PMF)( void*, size_t);
PMF p1 = &B::f;
PMF p2 = &B::operator delete;
```

Первое присваивание вполне корректно — мы просто присваиваем адрес функции-члена указателю на функцию-член.

Второе присваивание некорректно, поскольку `void operator delete (void*, size_t) throw()` является статической функцией-членом `B`, несмотря на то, что это не указано в ее объявлении явно. Вспомните, что `operator new()` и `operator delete()` всегда являются статическими функциями класса, несмотря на способ их объявления (в связи с чем хорошей практикой является явное указание `static` при объявлении этих операторов).

Рекомендация

Всегда явно объявляйте `operator new()` и `operator delete()` как статические функции. Нестатическими функциями-членами они быть не могут.

4. Имеются ли в следующем коде ошибки, связанные с управлением памятью?

Да, в каждом случае.

```
class X
{
public:
    void* operator new( size_t s, int ) throw( bad_alloc )
    {
        return ::operator new(s);
    }
};
```

Такой код склонен к утечкам памяти, поскольку не существует соответствующего размещающего `delete`. То же относится и к следующему фрагменту.

```
class SharedMemory
{
public:
    static void* Allocate( size_t s )
    {
        return OsSpecificSharedMemAllocation( s );
    }
    static void  Deallocate( void* p, int i = 0 )
    {
        OsSpecificSharedMemDeallocation( p, i );
    }
};
class Y
{
public:
    void * operator new( size_t s, SharedMemory& m )
```

```
                    throw( bad_alloc )
    {
        return m.Allocate( s );
    }
```

Такой код может привести к утечке памяти, поскольку оператор `delete` с соответствующей сигнатурой отсутствует. Если в процессе создания объекта, располагающегося в выделенной данной функцией памяти, будет сгенерировано исключение, эта память не будет корректно освобождена.

```
SharedMemory shared;
...
new (shared) Y; // При генерации исключения в Y::Y()
                // происходит утечка памяти
```

Кроме того, любая память, выделенная этим оператором `new()`, не может быть корректно освобождена в связи с тем, что в классе нет обычного оператора `delete()`. Это означает, что при освобождении памяти будет использоваться оператор `delete` базового класса или глобальный оператор (что почти всегда приводит к неприятностям, если, конечно, вы не замените все эти операторы своими).

```
    void  operator delete( void* p,
                           SharedMemory& m,
                           int i ) throw()
    {
        m.Deallocate( p, i );
    }
};
```

Этот оператор никогда не может быть вызван, а потому совершенно бесполезен.

```
void operator delete( void* p ) throw()
{
    SharedMemory::Deallocate( p );
}
```

Это серьезнейшая ошибка, так как замещенный глобальный оператор `delete()` освобождает память, выделенную оператором `new()`, а не вызовом `SharedMemory::Allocate()`. Лучшее, на что вы можете при этом надеяться, — быстрый крах с выводом дампа памяти.

```
void operator delete( void* p,
                      std::nothrow_t& ) throw()
{
    SharedMemory::Deallocate( p );
}
```

Здесь применимо то же замечание, что и в предыдущем случае, хотя данная ситуация имеет свои особенности. Этот оператор будет вызван только тогда, когда в выражении типа “`new(nothrow) T`” будет сгенерировано исключение в конструкторе `T`; таким образом, будет сделана попытка освободить посредством вызова `SharedMemory::Deallocate()` память, которая не была выделена в результате вызова `SharedMemory::Allocate()`.

Рекомендация

В классе всегда должны быть либо определены как операторы `new` (`new[]`), так и операторы `delete` (`delete[]`), либо не определен ни один из них.

Если вы сумели верно ответить на все поставленные в задаче вопросы, вы можете смело считать себя экспертом в области управления памятью в C++.

Задача 6.3. Применение auto_ptr. Часть 1 *Сложность: 8*

В этой задаче вы познакомитесь с тем, как эффективно и безопасно использовать стандартный шаблон `auto_ptr`.

Прежде всего мне бы хотелось высказать признательность Биллу Гиббонсу (Bill Gibbons), Грегу Колвину (Greg Colvin), Стиву Рамсби (Steve Rumsby) и другим, кто приложил так много усилий для усовершенствования `auto_ptr`.

Прокомментируйте приведенный далее код. Что в нем хорошо, что плохо, безопасен ли он, корректен ли?

```
auto_ptr<T> source()
{
    return auto_ptr<T>( new T(1) );
}
void sink( auto_ptr<T> pt ) { }
void f()
{
    auto_ptr<T> a( source() );
    sink( source() );
    sink( auto_ptr<T>( new T(1) ) );
    vector< auto_ptr<T> > v;
    v.push_back( auto_ptr<T>( new T(3) ) );
    v.push_back( auto_ptr<T>( new T(4) ) );
    v.push_back( auto_ptr<T>( new T(1) ) );
    v.push_back( a );
    v.push_back( auto_ptr<T>( new T(2) ) );
    sort( v.begin(), v.end() );
    cout << a->Value();
}
class C
{
public:    /* ... */
protected: /* ... */
private:   /* ... */
    auto_ptr<CImpl> pimpl_;
};
```

Решение

Большинство программистов слышали о стандартном интеллектуальном указателе `auto_ptr`, но далеко не все постоянно применяют его. Это досадно, так как `auto_ptr` в состоянии решить многие из распространенных проблем C++, и его разумное использование приводит к повышению качества кода. Из решения данной задачи вы узнаете, как корректно использовать `auto_ptr`, чтобы сделать ваш код более безопасным, и как при этом избежать подстерегающих вас опасностей и ловушек.

Почему “auto”?

`auto_ptr` — всего лишь один из большого множества возможных интеллектуальных указателей. Многие коммерческие библиотеки предоставляют свои, зачастую более интеллектуальные указатели с широким спектром функций — от счетчиков ссылок до обеспечения различного рода прокси-сервисов. Рассматривайте `auto_ptr` как один из интеллектуальных указателей — простой интеллектуальный указатель общего назначения, хотя и не обладающий всеми “рюшечками и финтифлюшечками” специализированных или высокопроизводительных интеллектуальных указателей, но способный к качественному решению обычных повседневно встречающихся задач.

Функция auto_ptr состоит во владении динамически выделенным объектом и выполнении автоматического его освобождения, когда объект становится не нужен. Вот простой пример кода, который небезопасен, если не использовать auto_ptr.

```
// Пример 1а. Исходный код
//
void f()
{
    T * pt( new T );
    /* ... */
    delete pt;
}
```

Большинство из нас пишет такой код чуть ли не ежедневно. Если f() — трехстрочная функция, которая не делает ничего исключительного, такой код может быть вполне приемлем. Но если f() не выполнит инструкцию delete, например, в связи с наличием инструкции return до delete или с генерацией исключения при выполнении тела функции, динамически выделенный объект не будет уничтожен, и мы получим классическую утечку памяти.

Простейший путь сделать пример 1*а* безопасным — “завернуть” указатель в интеллектуальный указателеподобный объект, который владеет указателем, а при уничтожении автоматически удаляет объект, на который тот указывает. Поскольку такой интеллектуальный указатель просто используется как автоматический объект (а значит, автоматически уничтожающийся при выходе из области видимости), его вполне обоснованно назвали “auto-указатель”.

```
// Пример 1б. Безопасный код
//
void f()
{
    auto_ptr<T> pt( new T );
    /* ... */
}  // деструктор pt вызывается при выходе из области
   // видимости, и объект удаляется автоматически
```

Теперь код застрахован от утечки объекта T, независимо от того, осуществляется ли выход из функции нормально или посредством исключения, поскольку деструктор pt будет вызван в любом случае при свертке стека. Освобождение ресурсов выполняется автоматически.

Использование auto_ptr не сложнее использования встроенного указателя, а для возврата владения ресурсом всегда можно воспользоваться вызовом функции release().

```
// Пример 2. Использование auto_ptr
//
void g()
{
    T* pt1 = new T;
    // Мы владеем выделенным объектом
    // Передаем владение
    auto_ptr<T> pt2( pt1 );
    // Используем auto_ptr так же,
    // как и обычный указатель
    *pt2 = 12;       // То же, что и *pt1 = 12;
    pt2->SomeFunc(); // То же, что и pt1->SomeFunc()
    // Используем get() для получения значения указателя
    assert( pt1 == pt2.get() );
    // Используем release() для возврата владения
    T* pt3 = pt2.release();
    // Удаляем объект самостоятельно, так как auto_ptr
    // больше объектом не владеет.
}   // pt2 не владеет указателем, так что он не будет
    // пытаться удалить его (нет двойного удаления).
```

И наконец, мы можем использовать функцию reset() для передачи во владение объекту auto_ptr другого указателя. Если в момент вызова reset() объект auto_ptr уже владеет другим указателем, сначала этот указатель будет удален. Таким образом вызов reset() представляет собой практически то же, как если бы объект auto_ptr был уничтожен, а затем создан заново с владением новым объектом.

```
// Пример 3. Использование reset()
//
void h()
{
    auto_ptr<T> pt( new T(1) );
    pt.reset( new T(2) );
    // Сначала происходит удаление объекта,
    // выделенного посредством "new T(1)"
}   // pt выходит из области видимости и
    // удаляет второй объект T
```

Упаковка членов-указателей

Аналогично, auto_ptr может использоваться и для безопасного хранения членов-указателей. Рассмотрим следующий пример, в котором используется идиома Pimpl.[2]

```
// Пример 4а. Типичное использование Pimpl
//

// Файл c.h
//
class C
{
public:
    C();
    ~C();
    /* ... */
private:
    struct CImpl; // Предварительное объявление
    CImpl* pimpl_;
};

// Файл c.cpp
//
struct C::CImpl { /* ... */ };
C::C() : pimpl_( new CImpl ) { }
C::~C() { delete pimpl_; }
```

Короче говоря, закрытые детали C выделены в отдельный объект реализации, который скрыт за непрозрачным указателем. Идея заключается в том, что конструктор C отвечает за выделение закрытого вспомогательного объекта, который содержит скрытое внутреннее представление класса, а деструктор C отвечает за его освобождение. Использование auto_ptr позволяет использовать более легкий путь.

```
// Пример 4б. Идиома Pimpl с использованием auto_ptr
//

// Файл c.h
//
class C
{
public:
    C();
    ~C();
    /* ... */
private:
```

[2] См. задачи 4.1—4.5.

```
    struct CImpl; // Предварительное объявление
    auto_ptr<CImpl> pimpl_;
    C& operator = ( const C& );
    C( const C& );
};

// Файл c.cpp
//
struct C::CImpl { /* ... */ };
C::C() : pimpl_( new CImpl ) { }
C::~C() { }
```

Теперь деструктору не надо беспокоиться об удалении указателя `pimpl_`, поскольку `auto_ptr` сделает это автоматически. Кроме того, конструктору `C::C()` придется выполнять меньше работы по обнаружению и восстановлению ошибок в конструкторе, поскольку в этом случае `pimpl_` будет освобожден автоматически. Таким образом, использование `auto_ptr` легче, чем непосредственная работа с указателем. Позже мы еще вернемся к этому примеру.

Владение, источники и стоки

Сам по себе `auto_ptr` довольно удобен, но станет еще лучше, если вы научитесь правильно его применять. Неплохим использованием `auto_ptr` является применение его для передачи информации в функцию и из нее, т.е. в качестве параметров и возвращаемых значений функции.

Для того чтобы понять, почему это так, рассмотрим сначала, что происходит при копировании `auto_ptr`. `auto_ptr` владеет объектом, храня указатель на него, причем одновременно владельцем объекта может быть только один объект. Когда вы копируете `auto_ptr`, то автоматически передаете владение от исходного к целевому `auto_ptr`. Если целевой `auto_ptr` уже владеет объектом, то сначала этот объект освобождается. После копирования указателем владеет только целевой `auto_ptr`, который и освободит его в надлежащее время, а исходный `auto_ptr` возвращается в нулевое состояние и больше не может использоваться в качестве ссылки на хранимый объект.

```
// Пример 5. Передача владения от одного
//           auto_ptr к другому
//
void f()
{
    auto_ptr<T> pt1( new T );
    auto_ptr<T> pt2;
    pt1->DoSomething();    // OK
    pt2 = pt1;             // Теперь указателем
                           // владеет pt2, а не pt1
    pt2->DoSomething();    // OK
} // При выходе за пределы области видимости деструктор
  // pt2 удаляет указатель; деструктор pt1 не делает ничего
```

Не забывайте, однако, что использовать `auto_ptr`, не владеющий указателем, нельзя.

```
// Пример 6. Работать с не владеющим
//           указателем auto_ptr нельзя
//
void f()
{
    auto_ptr<T> pt1( new T );
    auto_ptr<T> pt2;
    pt2 = pt1;             // Теперь указателем
                           // владеет pt2, а не pt1
    pt1->DoSomething();    // Ошибка: обращение по
                           // нулевому указателю
}
```

Помня об этом, приступим к рассмотрению работы auto_ptr с источниками и стоками. Источником (source) является функция или другая операция, которая создает новый ресурс, а затем передает владение этим ресурсом. Сток (sink) — это функция, выполняющая обратное действие, а именно — получающая владение существующим объектом (и обычно освобождающая его).

Так мы подошли к первой части кода из условия задачи.

```
auto_ptr<T> source()
{
    return auto_ptr<T>( new T(1) );
}
void sink( auto_ptr<T> pt ) { }
```

Здесь все корректно и безопасно. Обратите внимание на элегантность решения.

1. source() создает новый объект и возвращает его вызывающей функции совершенно безопасным способом, позволяя ей получить владение данным указателем. Даже если вызывающая функция проигнорирует возвращаемое значение, созданный объект будет всегда безопасно удален.

   Это достаточно важная идиома, так как иногда возвращение результата в "оболочке" типа auto_ptr оказывается единственным способом сделать функцию строго безопасной в плане исключений (см. задачу 2.12).

2. sink() получает в качестве параметра auto_ptr по значению и, таким образом, принимает владение ним. По выполнении sink() выполняется удаление вышедшего за пределы видимости локального объекта auto_ptr и, соответственно, указателя, которым он владеет (если только в функции sink() не происходит передача владения кому-либо). Поскольку функция sink() ничего не делает со своим параметром, вызов "sink(pt);" представляет собой то же, что и "pt.reset(0);".

Следующая часть кода показывает source() и sink() в действии.

```
void f()
{
    auto_ptr<T> a( source() );
```

Здесь все также корректно и безопасно. Функция f() принимает владение указателем, полученным от source() и (игнорируя некоторые проблемы в f(), с которыми мы встретимся чуть позже) автоматически удаляет его при выходе автоматической переменной из области видимости. Именно таким образом и происходит передача по значению auto_ptr, предназначенного для работы.

```
sink( source() );
```

Тут тоже все законно и безопасно. С учетом приведенных определений функций source() и sink() этот код по сути сводится к "delete new T(1);". Вопрос в том, зачем это может понадобиться? Представьте, что source() и sink() — не столь тривиальные функции, как показано в примере, и все встанет на свои места.

```
sink( auto_ptr<T>( new T(1) ) );
```

И тут все законно и безопасно. Это другой способ записи "delete new T(1);", но эта идиома может оказаться очень полезной, если sink() — нетривиальная пользовательская функция, получающая владение указываемым объектом.

Но будьте осторожны: никогда не используйте auto_ptr каким-либо иным способом, кроме описанных. Мне приходилось видеть, как программисты используют auto_ptr другими способами, как любой другой объект. Но проблема в том, что auto_ptr *не* похож на другие объекты. Дело в том, что *копии auto_ptr НЕ эквивалентны*.

Это приводит к важным эффектам при попытке использования auto_ptr с обобщенным кодом, в котором делаются копии, но при этом не учитывается возможность их неэквивалентности (в конце концов копии других классов практически всегда эквивалентны). Рассмотрим следующий код (который я регулярно привожу в группах новостей, посвященных C++).

```
vector< auto_ptr<T> > v;
```

Этот код некорректен и небезопасен! Осторожно — этот путь выстлан благими намерениями!

Небезопасно помещать auto_ptr в стандартные контейнеры. Некоторые программисты возражают, говоря, что их компиляторы и библиотеки успешно компилируют данный код, а другие утверждают, что видели именно этот пример как рекомендованный в документации некоторого популярного компилятора. Не слушайте их.

Проблема в том, что auto_ptr недостаточно удовлетворяет требованиям, предъявляемым к помещаемым в контейнеры типам, поскольку копии auto_ptr не эквивалентны. Прежде всего, ничего не гарантирует, что вектор не будет создавать "лишние" внутренние копии некоторых содержащихся в нем объектов. Но погодите, сейчас все окажется еще хуже...

```
v.push_back( auto_ptr<T>( new T(3) ) );
v.push_back( auto_ptr<T>( new T(4) ) );
v.push_back( auto_ptr<T>( new T(1) ) );
v.push_back( a );
```

(Заметим, что копирование a в контейнер v означает, что объект a больше не владеет хранимым указателем. Мы поговорим об этом чуть позже.)

```
v.push_back( auto_ptr<T>( new T(2) ) );
sort( v.begin(), v.end() );
```

Неверно. Небезопасно. Когда вы вызываете обобщенную функцию, которая копирует элементы, — наподобие того, как это делает sort(), — эти функции должны полагаться на эквивалентность копий. Как минимум одна популярная реализация сортировки получает копию "разделительного" элемента, и если заставить работать ее с auto_ptr, то получится следующее: она копирует разделительный объект auto_ptr (таким образом передавая владение указателем временному объекту), затем выполняет остальную часть работы (копируя при этом ничем не владеющий объект auto_ptr), а по окончании работы уничтожает временный объект. И мы получаем кучу проблем, так как минимум один объект auto_ptr в последовательности больше не владеет указателем, а тот указатель, которым он владел ранее, оказывается уничтоженным.

Комитет по стандартизации делает все, что можно, чтобы помочь программистам. Стандартный auto_ptr разработан так, чтобы помешать вам использовать его при работе со стандартными контейнерами (или по крайней мере с наиболее естественными реализациями стандартной библиотеки). Для этого комитетом использован следующий способ: конструктор копирования auto_ptr и оператор копирующего присваивания получают ссылку на неконстантный объект. Большинство же реализаций функций insert() стандартных контейнеров получает ссылку на константный объект, а следовательно, с auto_ptr работать не будут.

Работа с невладеющим auto_ptr

```
    // После копирования a в другой auto_ptr
    cout << a->Value();
}
```

(Мы предполагаем, что объект a был скопирован, но его указатель не был удален объектом vector или функцией sort.) Копирование auto_ptr не только передает

владение, но и сбрасывает значение указателя auto_ptr в нулевое. Это делается специально с той целью, чтобы помешать работе с невладеющим auto_ptr. Использование невладеющего auto_ptr некорректно, а попытка разыменования нулевого указателя имеет неопределенное поведение (в большинстве систем, как правило, сброс дампа памяти или генерация исключения обращения к памяти).

Так мы подошли к последнему использованию auto_ptr.

Оболочка указателей-членов

```
class C
{
public:    /* ... */
protected: /* ... */
private:   /* ... */
    auto_ptr<CImpl> pimpl_;
};
```

Использование auto_ptr для инкапсуляции указателей-членов работает так же, как и примеры в начале этой задачи, с тем отличием, что использование auto_ptr в функции уберегает нас от необходимости освобождения ресурсов вручную при выходе из функции, а здесь auto_ptr позволяет нам не беспокоиться об освобождении ресурсов в деструкторе класса C.

Конечно, и здесь не обходится без неприятностей. Так же, как и при использовании простого указателя, вам необходимо снабдить класс собственными деструктором, конструктором копирования и оператором копирующего присваивания (даже если вы запретите их использование, разместив в закрытом разделе описания и не дав их определений), поскольку предоставляемые по умолчанию функции в данном случае работают некорректно.

auto_ptr и безопасность исключений

Иногда auto_ptr необходим при написании безопасного с точки зрения исключений кода. Рассмотрим следующую функцию.

```
// Безопасна ли данная функция?
//
String f()
{
    String result;
    result = "some value";
    cout << "some output";
    return result;
}
```

Эта функция выполняет два видимых действия: она производит некоторый вывод и возвращает значение типа String. Детальное рассмотрение вопросов безопасности исключений выходит за рамки данной задачи, но цель, которую мы преследуем, состоит в обеспечении гарантии строгой безопасности. Функция при этом должна работать атомарно — даже при наличии исключений должны быть либо полностью выполнены все действия, либо не выполнено ни одно из них.

Хотя приведенный код и достаточно близок к строгой безопасности, в нем есть одна маленькая червоточинка, которая становится понятна из следующего фрагмента.

```
String theName;
theName = f();
```

Здесь вызываются конструктор копирования String, поскольку результат функции возвращается по значению, и оператор копирующего присваивания, так как результат требуется скопировать в переменную theName. Если какое-либо из этих копирований

даст сбой, получится, что функция f() полностью выполнила все свои действия, но их результат оказывается утерян.

Может, нам стоит поступить каким-то иным образом, — например, позволить функции получать в качестве параметра неконстантную ссылку на String и помещать в нее возвращаемое значение?

```
// Лучше?
//
void f( String& result )
{
    cout << "some output";
    result = "some value";
}
```

Этот способ выглядит лучше, но это только видимость, поскольку в результате присваивания может быть сгенерировано исключение, и тогда одно из действий функции будет завершено, а второе — нет. Итак, этот способ тоже не дает нам строгой гарантии безопасности.

Один из способов решения проблемы состоит в возврате динамически выделенного объекта String, но еще лучшее решение — сделать еще один шаг и вернуть указатель, обернутый в auto_ptr.

```
// Наконец-то верно!
//
auto_ptr<String> f()
{
    auto_ptr<String> result = new String;
    *result = "some value";
    cout << "some output";
    return result;
    // Передача владения не генерирует исключений
}
```

Это в определенной мере хитрость, поскольку мы сначала выполняем всю работу по созданию возвращаемого результата (второе действие), оставляя невыполненным только возврат значения (при котором гарантированно не может быть сгенерировано исключение) пользователю и выполняя его только в случае успешного завершения второго действия (вывода сообщения). Таким образом, если вывод сообщения прошел успешно, нам не о чем беспокоиться: возвращаемое значение не менее успешно будет передано вызываемой функции и в любом случае будет корректно освобождено: если вызывающая функция принимает возвращаемое значение, она получает владение указателем и он будет освобожден в свое время; если вызывающая функция не принимает значение, например, попросту игнорируя его, то указатель будет освобожден при уничтожении временного объекта auto_ptr. Какова же цена строгой безопасности? Как это часто случается, за нее приходится платить (обычно небольшим) снижением эффективности — в нашем случае из-за дополнительного динамического выделения памяти. Но в споре эффективности и корректности обычно всегда побеждает последняя!

Старайтесь использовать интеллектуальные указатели типа auto_ptr в своей повседневной работе. auto_ptr позволяет решить многие распространенные проблемы и делает ваш код более безопасным и надежным, в особенности в смысле предупреждения утечек ресурсов и гарантии строгой безопасности. В силу своей стандартности auto_ptr переносим и будет корректно работать везде, где бы вы ни компилировали ваш код.

Идиома const auto_ptr

Теперь, когда мы преодолели все трудности, рассмотрим один интересный способ работы с auto_ptr. Помимо прочих преимуществ auto_ptr, следует упомянуть то, что const auto_ptr никогда не теряет владения. Копирование такого объекта невоз-

можно, и все, что вы в состоянии с ним сделать, — это разыменовать его посредством операторов `operator*()` или `operator->()` (или получить значение содержащегося в нем указателя, вызвав `get()`). Таким образом мы получаем идиому для выражения `auto_ptr`, который никогда не может потерять владение.

```
const auto_ptr<T> pt1( new T );
    // pt1 не может быть скопирован и не
    // может потерять владение указателем
auto_ptr<T> pt2( pt1 );   // Некорректно
auto_ptr<T> pt3;
pt3 = pt1;                // Некорректно
pt1.release();            // Некорректно
pt1.reset( new T );       // Некорректно
```

Вот то, что я называю `const`. Если вы хотите объявить, что `auto_ptr` не может быть изменен и сам позаботится об освобождении того, чем владеет, — объявите его как `const`. Идиома `const auto_ptr` очень полезна и достаточно распространена, так что вам не следует забывать о ней.

Задача 6.4. Применение auto_ptr. Часть 2 ***Сложность: 5***

В этой задаче рассматриваются наиболее распространенные ловушки при использовании `auto_ptr`. Сможете ли вы обнаружить проблему и разрешить ее?

1. Рассмотрим следующую функцию, иллюстрирующую распространенную ошибку при использовании `auto_ptr`.

```
template<typename T>
void f( size_t n )
{
    auto_ptr<T> p1( new T );
    auto_ptr<T> p2( new T[n] );

    // ... Остальной код ...
}
```

   Что в этом коде неверно? Поясните.

2. Каким образом можно решить эту проблему? Рассмотрите по возможности большее количество вариантов, включая шаблон проектирования Адаптер (Adapter), альтернативы проблематичной конструкции и альтернативы `auto_ptr`.

Решение

Несовместимость массивов и auto_ptr

1. **Рассмотрим следующую функцию, иллюстрирующую распространенную ошибку при использовании `auto_ptr`.**

```
template<typename T>
void f( size_t n )
{
    auto_ptr<T> p1( new T );
    auto_ptr<T> p2( new T[n] );

    // ... Остальной код ...
}
```

   Что в этом коде неверно? Поясните.

Каждый delete должен соответствовать виду своего new. Если вы используете new для одного объекта, то должны использовать и delete для одного объекта; если вы используете new[], то, соответственно, должны использовать delete[]. Поступать иначе, значит, получить неопределенное поведение программы, как показано в следующем немного модифицированном коде.

```
T* p1 = new T;
// delete[] p1;      // Ошибка
delete p1;           // OK - так поступает auto_ptr

T* p2 = new T[10];
delete[] p2;         // OK
// delete p2;        // Ошибка - так поступает auto_ptr
```

Проблема заключается в том, что auto_ptr предполагает, что указатель, которым он владеет, должен указывать на одиночный объект, так что для его освобождения он всегда вызывает delete, а не delete[]. Это означает, что при работе с p1 и p2 первый указатель будет освобожден корректно, а второй — нет.

Что именно происходит при использовании некорректного вида оператора delete, зависит от вашего компилятора. Лучшее, чего вы можете ожидать, это утечка ресурсов. Более типичный результат — повреждение целостности памяти, быстро приводящее к сбросу дампа памяти. Чтобы увидеть это в действии, попробуйте скомпилировать следующую программу вашим любимым компилятором.

```
#include <iostream>
#include <memory>
#include <string>
using namespace std;

int c = 0;

class X
{
public:
    X() : s( "1234567890" ) { ++c; };
    ~X()                    { --c; };
    string s;
};

template<typename T>
void f( size_t n )
{
    {
        auto_ptr<T> p1( new T );
        auto_ptr<T> p2( new T[n] );
    }
    cout << c << " "; // Выводим количество имеющихся
                      // в настоящий момент объектов
}

int main()
{
    while( true )
    {
        f<X>( 100 );
    }
}
```

Обычно эта программа либо аварийно завершается, либо выводит все большее количество потерянных объектов X (кстати, попробуйте параллельно с работой программы отслеживать в другом окне общее количество используемой системной памяти — это поможет вам оценить степень утечки, приводящую к краху программы).

Массивы нулевой длины

Что произойдет, если параметром `f()` будет ноль (например, при вызове `f<int>(0)`)? Тогда второе выражение с `new` превратится в `new T[0]`, и программисты часто задаются вопросом: *насколько это корректно? можно ли использовать массив нулевой длины?*

Ответ — да. Массивы нулевой длины разрешены и совершенно корректны. Результат `new T[0]` — указатель на массив из нуля элементов, и он ведет себя в точности так же, как и любой другой результат вызова `new T[n]`, включая тот факт, что вы не можете обращаться к элементам массива за пределами `n` элементов. В случае массива нулевой длины мы просто не можем обращаться ни к одному из элементов массива — по той простой причине, что их вообще нет.

Раздел 5.3.4.7 стандарта гласит следующее.

> *Когда значение выражения в непосредственном описании `new` равно нулю, вызываемая функция выделяет массив без элементов. Выражение `new` возвращает при этом ненулевой указатель. [Примечание: при вызове библиотечной функции выделения памяти возвращаемый указатель не совпадает ни с одним из указателей на любые другие объекты.]*

"Но если мы не можем ничего делать с массивами нулевой длины (кроме как получить их адрес), — можете удивиться вы, — то зачем они вообще разрешены?" Одной важной причиной является то, что тем самым упрощается написание кода, выполняющего динамическое выделение массивов. Например, функция `f()`, рассмотренная выше, стала бы более сложной, если бы требовалась проверка значения параметра `n` перед вызовом `new T[n]`.

Конечно, возвращаясь к основной задаче, все сказанное не означает, что `auto_ptr` может владеть массивом нулевой длины — проблема удаления массива остается актуальной и в этом случае.

2. Каким образом можно решить эту проблему? Рассмотрите по возможности большее количество вариантов, включая шаблон проектирования Адаптер (Adapter), альтернативы проблематичной конструкции и альтернативы `auto_ptr`.

Имеется ряд вариантов, одни из них лучше, другие хуже. Далее перечислены четыре из них.

Вариант 1. Создание auto_array

Имеются разные способы этого решения.

Вариант 1а. Наследование от auto_ptr (оценка: 0/10)

Плохое решение. Например, вы должны воспроизвести семантику владения и вспомогательных классов. На это способны только настоящие гуру, но они никогда не станут этим заниматься, так как есть и более простые пути.

Преимущества: никаких.

Недостатки: слишком много, чтобы их перечислять.

Вариант 1б. Клонирование кода auto_ptr (оценка: 8/10)

Идея заключается в том, чтобы взять код `auto_ptr` из библиотеки вашей реализации C++, клонировать его (переименовав в `auto_array` или что-то подобное) и заменить инструкции `delete` инструкциями `delete[]`.

Преимущества.

а. *Легко реализуется.* Нам не требуется кодировать собственный `auto_array`, и вся семантика `auto_ptr` сохраняется автоматически — что помогает избежать неприятных сюрпризов для программистов, хорошо знакомых с `auto_ptr`.

б. *Нет дополнительных накладных расходов ни в плане памяти, ни в плане эффективности.*

Недостатки.

а. *Сложность поддержки.* Вероятно, вы захотите поддерживать синхронизацию `auto_ptr` и вашего `auto_array` при обновлении или смене вашего компилятора или библиотеки.

Вариант 2. Использование шаблона проектирования Adapter (оценка: 7.10)

Этот вариант родился в результате дискуссии с Генриком Нордбергом (Henrik Nordberg). Первая реакция Генрика на данную проблему — удивление — почему бы не использовать адаптер для того, чтобы заставить корректно работать стандартный `auto_ptr` вместо того, чтобы переписывать его. Эта идея имеет ряд реальных преимуществ и заслуживает анализа, невзирая на определенные недостатки.

Итак, идея состоит в том, что вместо

```
auto_ptr<T> p2( new T[n] );
```

мы пишем

```
auto_ptr< ArrDelAdapter<T> >
    p2( new ArrDelAdapter<T>( new T[n] ));
```

где `ArrDelAdapter` имеет конструктор, который получает `T*`, и деструктор, вызывающий `delete[]` для этого указателя.

```
template<typename T>
class ArrDelAdapter
{
public:
    ArrDelAdapter( T* p ) : p_(p) {}
    ~ArrDelAdapter() { delete[] p_; }
    // операторы типа "->" и "T*"
    // и другие вспомогательные функции
private:
    T* p_;
};
```

Поскольку имеется только один объект `ArrDelAdapter<T>`, оператор удаления одного объекта `delete` в деструкторе `~auto_ptr` работает корректно, а так как деструктор `~ArrDelAdapter<T>` вызывает `delete[]` для выделенного массива, то наша проблема оказывается решенной.

Пожалуй, это не самое элегантное решение, но зато оно избавляет нас от необходимости писать собственный шаблон `auto_array`.

Преимущества.

а. *Простота реализации.* Нам не требуется писать свой шаблон `auto_array`. Кроме того, мы автоматически сохраняем всю семантику `auto_ptr`, что позволяет избежать сюрпризов для программистов, поддерживающих данную программу и знакомых с `auto_ptr`.

Недостатки.

а. *Трудно читаемо.* Это решение и в самом деле несколько многословно...

б. *(Возможно) трудно используемо.* Весь код в `f()`, который использует `p2`, потребует внесения синтаксических изменений, которые делают код более громоздким из-за излишних косвенных обращений.

в. *Приводит к излишним накладным расходам.* Этот код требует дополнительной памяти для хранения адаптера массива и дополнительного времени, поскольку происходит как минимум двойное выделение памяти, а при каждом обращении к массиву требуется дополнительное косвенное обращение.

Несмотря на все указанные недостатки, мне бы хотелось порекомендовать читателям не забывать о существовании этого шаблона проектирования в своей работе. Адаптеры широко применимы и являются одним из тех шаблонов проектирования, с которыми должен быть знаком каждый программист.

Рекомендация

Не забывайте о существовании шаблонов проектирования.

И последнее замечание по поводу второго варианта. Подумайте о том, так ли уж сильно запись

```
auto_ptr< ArrDelAdapter<T> >
    p2( new ArrDelAdapter<T>( new T[n] ));
```

отличается от

```
auto_ptr< vector<T> > p2( new vector<T>(n) );
```

и ответьте на вопрос: что же вы выигрываете при динамическом выделении вектора по сравнению с простым использованием “`vector p2(n);`”? А после этого обратитесь к четвертому варианту решения проблемы.

Вариант 3. Замена auto_ptr обработкой исключений (оценка: 1/10)

Функция `f()` использует `auto_ptr` для автоматического освобождения выделенной памяти и, вероятно, для обеспечения безопасности исключений. Вместо этого мы можем отказаться от использования `auto_ptr` для `p2` и выполнить обработку ошибок самостоятельно, т.е. вместо кода

```
auto_ptr<T> p2( new T[n] );
//
// ... Прочий код ...
//
```

мы пишем нечто типа

```
T* p2( new T[n] );
try {
    //
    // ... Прочий код ...
    //
}
delete[] p2;
```

Преимущества.

а. *Простота использования.* Такое решение мало влияет на использование `p2` в коде, указанном как “Прочий код”; вероятно, все, что потребуется, — это удалить все `.get()`.

б. *Нет ни значительного расхода памяти, ни снижения эффективности.*

Недостатки.

а. *Сложность реализации.* Такое решение, вероятно, потребует гораздо больших изменений в коде, чем показано выше. Причина этого в том, что, в то время как `auto_ptr` для `p1` автоматически освободит ресурс (независимо от того, каким образом осуществится выход из функции), для освобождения `p2` мы должны написать код освобождения так, чтобы он работал на каждом из возможных путей выхода из функции. Рассмотрите, например, насколько усложнится решение, если "Прочий код" содержит ряд инструкций ветвления, которые могут завершаться инструкциями `return`.

б. *Ненадежность.* См. пункт *а*: сможем ли мы разместить код освобождения ресурсов на всех возможных путях выполнения?

в. *Неудобочитаемость.* См. пункт *а*: дополнительная логика освобождения ресурса запутывает нормальную логику функции.

Вариант 4. Использование vector вместо массива (оценка: 9.5/10)

Большинство встречающихся нам проблем связано с использованием массивов в стиле C. Так что если можно (а можно практически всегда), лучше вместо них использовать `vector`. В конце концов, главная причина наличия `vector` в стандартной библиотеке — обеспечить безопасную и простую в использовании замену массивам в стиле C!

Так что вместо

```
auto_ptr<T> p2( new T[n] );
```

мы просто используем

```
vector<T> p2(n);
```

Преимущества.

а. *Простота реализации.* Нам не надо писать собственный `auto_array`.

б. *Удобочитаемость.* Программисты, знакомые со стандартными контейнерами (а такими теперь должны быть все!), легко поймут код с использованием `vector`.

в. *Надежность.* При избавлении от непосредственного управления памятью наш код (обычно) становится проще и надежнее. Нам не надо работать с буфером объектов `T` — мы просто работаем с объектом `vector<T>`.

г. *Нет ни значительного расхода памяти, ни снижения эффективности.*

Недостатки.

а. *Синтаксические изменения.* Весь код в `f()`, использующий `p2`, должен подвергнуться синтаксическим изменениям. Впрочем, эти изменения довольно просты, в особенности по сравнению с изменениями, требующимися в варианте 2.

б. *Изменения применимости (иногда).* Вы не можете настроить никакой стандартный контейнер (включая вектор) объектов `T`, если эти объекты не имеют конструктора копирования и оператора копирующего присваивания. Большинство типов имеет эти функциональные возможности, но в противном случае описанное решение неприменимо.

Заметим, что передача или возврат `vector` по значению приводит к большему количеству работы, чем передача или возврат `auto_ptr`. Однако к этому сравнению надо подходить с осторожностью в силу его некорректности: если вы хотите получить тот же эффект, передавайте `vector` по ссылке или по указателю.

Рекомендация

Предпочитайте применение `vector` использованию массивов в стиле C.

Задача 6.5. Интеллектуальные указатели-члены. Часть 1 *Сложность: 5*

Большинство программистов на C++ знают, что при работе с указателями-членами следует принимать специальные меры. А что делать с классами, членами которых являются `auto_ptr`? Можно ли обезопасить себя и своих пользователей, разработав интеллектуальный указатель специального назначения для использования в качестве члена класса?

1. Рассмотрим следующий класс.

```
// Пример 1
//
class X1
{
    // ...
private:
    Y* y_;
};
```

Почему автор класса не может использовать генерируемые компилятором деструктор, конструктор копирования и копирующее присваивание, если объект `X1` владеет объектом `Y`, на который указывает указатель?

2. Каковы преимущества и недостатки следующего подхода?

```
// Пример 2
//
class X2
{
    // ...
private:
    auto_ptr<Y> y_;
};
```

Решение

Проблема указателей-членов

1. **Рассмотрим следующий класс.**

```
// Пример 1
//
class X1
{
    // ...
private:
    Y* y_;
};
```

Почему автор класса не может использовать генерируемые компилятором деструктор, конструктор копирования и копирующее присваивание, если объект `X1` владеет объектом `Y`, на который указывает указатель?

Если объект X1 владеет объектом Y, на который указывает указатель, то генерируемые компилятором функции работают некорректно. Для начала заметим, что одна функция (вероятно, конструктор X1) должна создать объект Y, а другая (скорее всего, деструктор X1::~X1()) должна его уничтожить.

```
// Пример 1а. Семантика владения
//
{
    X1 a; // Выделение нового объекта Y и
          // получение указателя на него
    // ...
}   // Объект a выходит из области видимости и
    // уничтожается, удаляя при этом объект Y
```

Использование конструктора копирования по умолчанию приводит к появлению нескольких объектов X1, указывающих на один и тот же объект Y, в результате чего происходят такие странные вещи, как изменение одним объектом состояния другого или многократное удаление одного указателя.

```
// Пример 1б. Совместное использование и
//            двойное удаление
{
    X1 a;      // Выделение нового объекта Y

    X1 b( a ); // b указывает на тот же объект Y, что и a

    // ... работа с a и b приводит к изменению
    // одного и того же объекта Y

}   // Объект b выходит из области видимости и
    // уничтожается, удаляя при этом объект Y...
    // Но то же самое делает и объект a!
```

Использование почленного копирующего присваивания по умолчанию приводит к появлению нескольких объектов X1, указывающих на один и тот же объект Y, в результате чего происходят такие же странные вещи, как и при использовании конструктора копирования по умолчанию, а кроме того, при этом наблюдается утечка, связанная с наличием объектов, которые некому удалить.

```
// Пример 1в. Совместное использование,
//            двойное удаление и утечка
{
    X1 a;      // Выделение нового объекта Y
    X1 b;      // Выделение нового объекта Y

    b = a;     // b указывает на тот же объект Y, что и a;
               // нет ни одного указателя на ранее
               // принадлежавший b объект Y

    // ... работа с a и b приводит к изменению
    // одного и того же объекта Y

}   // Объект b выходит из области видимости и
    // уничтожается, удаляя при этом объект Y...
    // Но то же самое делает и объект a!
    // Объект Y, созданный при конструировании
    // объекта b, остается не удаленным
```

Во многих случаях, чтобы упростить освобождение объектов, мы “заворачиваем” обычный указатель в управляющий объект, который владеет данным указателем. Улучшит ли ситуацию использование вместо указателя на Y такого управляющего объекта?

Использование auto_ptr как члена класса

2. Каковы преимущества и недостатки следующего подхода?

```
// Пример 2
//
class X2
{
    // ...
private:
    auto_ptr<Y> y_;
};
```

Такой подход имеет определенные преимущества, хотя и не решает описанную ранее проблему, связанную с некорректной работой генерируемых по умолчанию функций (просто при этом функции работают неправильно, но по-другому).

Во-первых, если X2 имеет пользовательские конструкторы, то их проще сделать безопасными, поскольку при генерации исключения в конструкторе auto_ptr выполнит освобождение захваченного ресурса автоматически. Автор X2, тем не менее, должен самостоятельно создать новый объект Y и передать владение им объекту-члену auto_ptr.

Во-вторых, автоматически генерируемый деструктор теперь работает совершенно корректно. При выходе объекта X2 за пределы области видимости и его уничтожении деструктор auto_ptr<Y> автоматически выполняет освобождение объекта Y, которым владеет. Но даже в этом случае есть один маленький изъян: если вы используете генерируемый компилятором деструктор, он будет определен в каждом модуле трансляции, использующем X2. Это в свою очередь означает, что определение Y должно быть видимо всем, кто использует X2. Это мешает, например, использованию идиомы скрытой реализации. Так что использовать автоматически генерируемый деструктор лучше только тогда, когда полное определение Y поставляется вместе с X2 (например, если файл x2.h включает y.h).

```
// Пример 2а. Y должен быть определен
//
{
    X2 a;  // Выделяет новый объект Y

    // ...

}   // Объект a выходит из области видимости и
    // уничтожается, удаляя при этом объект Y.
    // Это возможно только в случае, когда
    // доступно полное определение Y.
```

Если вы не хотите предоставлять определение Y, вам следует явно написать деструктор X2 — даже если это пустой деструктор.

В-третьих, автоматически генерируемый конструктор копирования больше не приводит к двойному удалению, описанному в примере 1*б*. Это хорошая новость. Новость похуже: автоматически генерируемый конструктор копирования приводит к другой проблеме — воровству. Создаваемый объект X2 похищает у копируемого объекта всю информацию об объекте Y.

```
// Пример 2б. Кража указателя
//
{
    X2 a;      // Выделяет новый объект Y

    X2 b( a ); // b получает от объекта a всю
               // информацию об объекте Y, a
               // объекту a остается нулевой auto_ptr

    // Если объект a попытается воспользоваться своим
    // членом y_, возникнут проблемы. В лучшем случае
```

```
        // это будет немедленный крах программы; в не столь
        // удачном случае вы получите мигрирующий
        // трудноуловимый сбой в работе программы.
    }
```

Единственным утешением может служить то, что автоматически генерируемый конструктор копирования выдает некоторое честное предупреждение о возможности не совсем корректного поведения. Почему? Потому что его сигнатура — `X2::X2(X2&)`. Обратите внимание: он получает в качестве параметра ссылку на неконстантный объект (именно поэтому работает конструктор копирования `auto_ptr`). Это, конечно, не настоящая защита от неприятностей, но по крайней мере вы не сможете скопировать таким образом объект, описанный как `const X2`.

И наконец, автоматически генерируемый оператор копирующего присваивания больше не приводит ни к проблеме двойного удаления, ни к проблеме утечки ресурсов, описанных в примере 1*в*. Это хорошие новости, которые, конечно же, сопровождаются плохими: при использовании автоматически генерируемого оператора копирующего присваивания остается актуальной проблема воровства указателей.

```
    // Пример 2в. Кража указателя
    //
    {
        X2 a;       // Выделяет новый объект Y
        X2 b;       // Выделяет новый объект Y

        b = a;      // b удаляет собственный объект Y,
                    // получает от объекта a всю
                    // информацию об его объекте Y, a
                    // объекту a остается нулевой auto_ptr

        // Как и в примере 2б, любые попытки объекта a
        // воспользоваться своим членом y_ приводят к сбоям
```

Как и в предыдущем случае, небольшой намек на возможные неприятности подается сигнатурой автоматически генерируемого оператора: `X2& X2::operator=(X2&)`, что соответствует объявлению (правда, мелким шрифтом, а не кричащим заголовком), что операнд может быть модифицирован.

Короче говоря, применение членов-объектов `auto_ptr` дает определенные преимущества, в частности, автоматизацию освобождения ресурсов конструкторами и деструкторами, но при этом не решает нашу исходную задачу — мы все равно должны писать собственный конструктор копирования и оператор копирующего присваивания (или запретить их использование, если копирование для данного класса не имеет смысла). Более приемлемое решение состоит в разработке чего-то более специализированного, чем `auto_ptr`.

Вариации на тему ValuePtr

Текущая и следующая задачи представляют собой одну мини-серию, цель которой — постепенно разработать и усовершенствовать шаблон `ValuePtr`, который в большей степени, чем `auto_ptr`, подходит для описанного выше применения.

Примечание по спецификации исключений: по причинам, которые мне не хочется здесь перечислять, спецификации исключений не так полезны, как кажется. Тем не менее важно знать, какие исключения могут генерироваться функцией, в особенности знать, что некоторая функция не генерирует исключений вообще, соответствуя гарантии отсутствия исключений. В следующей задаче для документирования поведения функций вам не потребуются спецификации исключений, поскольку я заявляю следующее.

> *Во всех представленных версиях `ValuePtr<T>` все функции-члены обеспечивают гарантию отсутствия исключений, кроме возможных исключений, генерируе-*

мых конструктором `T`, при создании копий `ValuePtr<U>` (где `U` может являться `T`) во время конструирования или присваивания объекта.

Задача 6.6. Интеллектуальные указатели-члены. Часть 2 ***Сложность: 6***

Можно ли обезопасить себя и своих пользователей, разработав интеллектуальный указатель специального назначения для работы в качестве члена класса?

1. Напишите шаблон `ValuePtr`, пригодный для использования следующим образом

```
// Пример 1
//
class X
{
    // ...
private:
    ValuePtr<Y> y_;
};
```

и удовлетворяющий следующим различным условиям.

а. Копирование и присваивание `ValuePtr` не разрешено.

б. Копирование и присваивание `ValuePtr` разрешено и имеет семантику создания копии принадлежащего `ValuePtr` объекта `Y` с использованием конструктора копирования `Y`.

в. Копирование и присваивание `ValuePtr` разрешено и имеет семантику создания копии принадлежащего `ValuePtr` объекта `Y` с использованием виртуального метода `Y::Clone()` при наличии такового или конструктора копирования `Y` в противном случае.

Решение

Простейшее решение

1. **Напишите шаблон `ValuePtr`, пригодный для использования следующим образом**

```
// Пример 1
//
class X
{
    // ...
private:
    ValuePtr<Y> y_;
};
```

Мы рассмотрим три разных случая. Во всех трех случаях остается актуальным преимущество автоматического освобождения ресурса в случае сбоя в конструкторе `X`, что позволяет написать безопасный и лишенный утечек памяти конструктор `X::X()`, затратив меньше усилий.[3] Кроме того, во всех трех случаях на деструктор накладывается следующее ограничение: либо определению `X` должно сопутствовать определение `Y`, либо следует явно обеспечить наличие деструктора `X`, даже если он пуст.

[3] Можно сделать так, что `ValuePtr` будет самостоятельно создавать объект `Y`, которым владеет, однако для ясности я опускаю эту возможность (да и семантика `ValuePtr<Y>` будет при этом настолько схожа с `Y`, что поневоле возникнет вопрос: почему бы нам попросту не использовать обычный член `Y`?).

... и удовлетворяющий различным следующим условиям.

а. Копирование и присваивание `ValuePtr` не разрешено.

В этом случае у нас не очень много работы.

```
// Пример 2а. ValuePtr без копирования и присваивания
//
template<typename T>
class ValuePtr
{
public:
    explicit ValuePtr(T* p = 0 ) : p_( p ) {}
    ~ValuePtr() { delete p_; }
```

Конечно, нам требуется возможность обращения к указателю, так что к определению `ValuePtr` требуется добавить код, аналогичный имеющемуся в `auto_ptr`.

```
T& operator*()  const { return *p_; }
T* operator->() const { return p_; }
```

Что еще может нам понадобиться? В большинстве интеллектуальных указателей имеет смысл обеспечить такие дополнительные возможности, как имеющиеся в `auto_ptr` функции `reset()` и `release()`, которые позволяют сменить объект, которым владеет `ValuePtr`. Сначала может показаться, что наличие таких функций — удачная идея. Рассмотрим примеры 2 и 3 из задачи 2.18, где имеется член-указатель скрытой реализации и требуется разработать безопасный оператор присваивания. Для этого нам требовалась возможность поменять местами два `ValuePtr` без копирования объектов, которыми они владеют. Однако использование для этой цели функций наподобие `reset()` и `release()` — не совсем верный путь. Эти функции позволяют слишком много, так что хотя с их помощью вы можете сделать все необходимое для обмена и обеспечения безопасности исключений, они все же открывают возможности некорректного использования, лежащего за пределами предназначения `ValuePtr`.

Так что лучше ограничиться следующим минимально необходимым набором.

```
    void Swap( ValuePtr& other ) { swap( p_, other.p_ );}

private:
    T* p_;

    // Копирование не разрешено
    ValuePtr( const ValuePtr& );
    ValuePtr& operator=( const ValuePtr& );
};
```

Мы получаем владение указателем, а позже удаляем его. Мы обрабатываем случай нулевого указателя, а копирование и присваивание специально запрещены путем их объявления как `private` и отсутствием их определений. Конструктор объявлен как `explicit`, что позволяет избежать неявного преобразования типов, которое никогда не требуется для `ValuePtr` в силу его специализированного предназначения.

Эту часть задачи мы решили достаточно просто, но дальше будет сложнее.

Копирующий конструктор и копирующее присваивание

б. Копирование и присваивание `ValuePtr` разрешено и имеет семантику создания копии принадлежащего `ValuePtr` объекта `Y` с использованием конструктора копирования `Y`.

Вот один из вариантов кода, который удовлетворяет поставленным требованиям, но не настолько универсальный, каким мог бы быть. Это по сути тот же код, что и в примере 2*а*, но с определенными конструктором копирования и оператором присваивания.

```
// Пример 2б. ValuePtr с копированием и присваиванием
//            Вариант 1
template<typename T>
class ValuePtr
{
public:
    explicit ValuePtr(T* p = 0) : p_( p ) {}
    ~ValuePtr() { delete p_; }
    T& operator*() const { return *p_; }
    T* operator->() const { return p_; }

    void Swap(ValuePtr& other) { swap(p_, other.p_); }

    // ---------- Новый код ------------------
    ValuePtr( const ValuePtr& other )
        : p_( other.p_ ? new T( *other.p_ ) : 0 ) {}
```

Обратите внимание на важность проверки на равенство нулю указателя объекта `other`. Поскольку оператор присваивания реализован посредством конструктора копирования, в нем такая проверка не нужна.

```
    ValuePtr& operator=( const ValuePtr& other )
    {
        ValuePtr temp( other );
        Swap( temp );
        return *this;
    }
    // ---------- Конец нового кода ----------

private:
    T* p_;
};
```

Этот код удовлетворяет поставленному условию, поскольку при предполагаемом использовании этого шаблона не предусматривается копирование или присваивание объектов `ValuePtr`, которые управляют отличным от `T` типом указателей. Если это условие нас устраивает, то такое решение вполне корректно и работоспособно. Но при разработке любого класса мы должны подумать о расширении его возможностей, если, конечно, это не требует слишком большой дополнительной работы и может обеспечить новые полезные возможности для будущих пользователей. В то же время лучшее — враг хорошего, так что слишком увлекаться и разрабатывать чересчур сложное решение простой задачи не следует.

“Шаблонное” копирование

Рассмотрим следующий вопрос: что если в будущем мы захотим позволить присваивание между разными типами `ValuePtr`? То есть захотим позволить присваивать объект `ValuePtr<X>` объекту `ValuePtr<Y>`, если `X` можно преобразовать в `Y`. Достичь этого можно достаточно просто. Нам нужно лишь продублировать конструктор копирования и оператор копирующего присваивания, добавив к ним “`template<typename U>`” и изменив тип параметра на `ValuePtr<U>&`.

```
// Пример 2в. ValuePtr с копированием и присваиванием
//            Вариант 2
template<typename T>
class ValuePtr
{
public:
    explicit ValuePtr(T* p = 0) : p_( p ) {}
    ~ValuePtr() { delete p_; }
    T& operator*() const { return *p_; }
    T* operator->() const { return p_; }
```

```
    void Swap(ValuePtr& other) { swap(p_, other.p_); }

    ValuePtr( const ValuePtr& other )
        : p_( other.p_ ? new T( *other.p_ ) : 0 ) {}

    ValuePtr& operator=( const ValuePtr& other )
    {
        ValuePtr temp( other );
        Swap( temp );
        return *this;
    }

    // ---------- Новый код ------------------
    template<typename U>
    ValuePtr( const ValuePtr<U>& other )
        : p_( other.p_ ? new T( *other.p_ ) : 0 ) {}

    template<typename U>
    ValuePtr& operator=( const ValuePtr<U>& other )
    {
        ValuePtr temp( other );
        Swap( temp );
        return *this;
    }

private:
    template<typename U> friend class ValuePtr;
    // ---------- Конец нового кода ----------

    T* p_;
};
```

Заметили ли вы ловушку, которую нам удалось благополучно избежать? Мы должны определить нешаблонные конструктор копирования и оператор копирующего присваивания для того, чтобы воспрепятствовать автоматической их генерации (шаблонный конструктор не может быть конструктором копирования, а шаблонный оператор присваивания — оператором копирующего присваивания). Более подробно об этом сказано в задаче 1.5.

Имеется еще одна маленькая тонкость, которая, я бы сказал, вообще находится вне нашей компетенции, как авторов `ValuePtr`. Дело в том, что при использовании как шаблонного, так и нешаблонного копирования исходный объект `other` может хранить указатель на производный тип — и в этом случае мы получаем срезку.

```
class A {};
class B : public A {};
class C : public B {};

ValuePtr<A> a1( new B );
ValuePtr<B> b1( new C );

// Вызов конструктора копирования со срезкой
ValuePtr<A> a2( a1 );

// Вызов шаблонного конструктора со срезкой
ValuePtr<A> a3( b1 );

// Вызов копирующего присваивания со срезкой
a2 = a1;

// Вызов шаблонного присваивания со срезкой
a3 = b1;
```

Я заострил на этом ваше внимание, поскольку это те моменты, которые нельзя забывать указывать в документации (лучше всего в разделе “Этого делать не следует”)

для того, чтобы предупредить пользователей. Предотвратить такое некорректное использование ValuePtr путем каких-то изменений в коде мы не в состоянии.

Так какое же решение верное — 2*б* или 2*в*? Оба решения хороши, и решение должно приниматься на основе вашего собственного опыта как компромисс между максимальной повторной используемостью и простотой. Я думаю, что сторонники краткости и простоты выберут решение 2*б*, как минимально отвечающее поставленным требованиям. Но я могу представить ситуацию, в которой ValuePtr находится в библиотеке, написанной одной группой программистов и использующейся рядом других команд, — а в этом случае решение 2*в* может сэкономить общие усилия путем повышения уровня повторного использования и предупреждения “изобретения велосипеда”.

Расширяемость с использованием свойств

Но что если Y имеет виртуальный метод Clone()? Из примера 1 задачи 6.5 может показаться, что X всегда сам создает свой объект Y, но с тем же успехом он может получить его от фабрики классов или из выражения new для некоторого производного типа. Как мы уже видели, объект Y, которым владеет ValuePtr, может на самом деле иметь тип, производный от Y, и копирование его как Y будет приводить к срезке в лучшем случае или приводить к неработоспособности в худшем. Обычно в таких ситуациях в классе Y имеется специальная виртуальная функция-член Clone(), которая обеспечивает выполнение полноценного копирования даже без знания полного типа указываемого объекта.

Что если кому-то понадобится использовать ValuePtr для хранения такого объекта, который может быть скопирован в результате использования специальной функции а не конструктором копирования? В этом и заключается наш последний вопрос.

в. Копирование и присваивание ValuePtr разрешено и имеет семантику создания копии принадлежащего ValuePtr объекта Y с использованием виртуального метода Y::Clone() при наличии такового или конструктора копирования Y в противном случае.

В шаблоне ValuePtr нам неизвестно заранее, какой именно тип представлен параметром T. Соответственно, нам неизвестно, есть ли в этом типе виртуальная функция Clone(). Следовательно, мы не знаем, каким образом следует копировать его. Или знаем?

Одно из решений состоит в применении широко распространенной в стандартной библиотеке C++ технологии, — а именно классов-свойств (чтобы освежить свои знания о свойствах, обратитесь к задаче 1.11). Для реализации такого подхода мы сначала несколько изменим пример 2*в*, удалив из него некоторую избыточность. Как вы увидите, и конструктор копирования, и шаблонный конструктор проверяют исходный указатель на равенство нулю, так что мы размещаем всю эту работу в одном месте и получаем функцию CreateFrom(), которая и создает новый объект T.

```
// Пример 2г. ValuePtr с копированием и присваиванием.
//            (Пример 2в с изменениями)
template<typename T>
class ValuePtr
{
public:
    explicit ValuePtr(T* p = 0) : p_( p ) {}
    ~ValuePtr() { delete p_; }
    T& operator*() const { return *p_; }
    T* operator->() const { return p_; }

    void Swap(ValuePtr& other) { swap(p_, other.p_); }

    ValuePtr( const ValuePtr& other )
```

```
        : p_( CreateFrom( *other.p_ ) ) {}  // изменено

    ValuePtr& operator=( const ValuePtr& other )
    {
        ValuePtr temp( other );
        Swap( temp );
        return *this;
    }

    template<typename U>
    ValuePtr( const ValuePtr<U>& other )
        : p_( CreateFrom( *other.p_ ) ) {}  // изменено

    template<typename U>
    ValuePtr& operator=( const ValuePtr<U>& other )
    {
        ValuePtr temp( other );
        Swap( temp );
        return *this;
    }

private:
    // ---------- Новый код ------------------
    template<typename U>
    T* CreateFrom( const U* p ) const
    {
        return p ? new T( *p ) : 0;
    }
    // ---------- Конец нового кода ----------

    template<typename U> friend class ValuePtr;

    T* p_;
};
```

Теперь `CreateFrom()` дает нам возможность инкапсуляции знаний о различных путях копирования `T`.

Применение свойств

Теперь мы можем применить технологию свойств. Заметим, что представленный метод не единственный, существуют и другие способы применения свойств. Мною выбрано использование единого шаблона класса свойств с единственной статической функцией-членом `Clone()`, которая вызывается тогда, когда необходимо выполнение клонирования, и которая может рассматриваться как адаптер. Этот метод следует, например, стилю `char_traits`, когда `basic_string` делегирует работу по определению стратегии обработки символов классу свойств. (Альтернативой, например, может быть класс свойств, который предоставляет различные инструкции `typedef` и другие вспомогательные средства, так что шаблон `ValuePtr` может определить, что именно он должен делать, но эти действия он выполняет самостоятельно. Мне не нравится этот подход, поскольку он представляет собой по сути излишнее разделение областей ответственности. Почему, собственно, что именно надо делать, выясняется в одном месте, а делается — в совершенно другом?)

```
template<typename T>
class ValuePtr
{
    // ...
    template<typename U>
    T* CreateFrom( const U* p ) const
    {
        return p ? VPTraits<U>::Clone( p ) : 0;
```

```
    }
};
```

VPTraits должен быть шаблоном, который выполняет работу по клонированию (для чего реализация функции Clone() использует конструктор копирования U). Добавлю два небольших замечания. Поскольку ValuePtr берет на себя ответственность за проверку указателя на равенство нулю, VPTraits::Clone() этим заниматься не надо; и если T и U — различные типы, то эта функция будет скомпилирована только в том случае, когда U* может быть преобразовано в T* (при обработке полиморфных случаев, когда T является базовым, а U — производным классом).

```
template<typename T>
class VPTraits
{
public:
    static T* Clone( const T* p ) { return new T( *p ); }
};
```

Для любого класса Y, который не должен использовать конструктор копирования, шаблон VPTraits можно специализировать следующим образом. Пусть, например, класс Y имеет виртуальную функцию Y* CloneMe(), а класс Z — виртуальную функцию void CopyTo(Z&). Тогда VPTraits специализируется так, чтобы для клонирования использовались данные функции.

```
template<>
class VPTraits<Y>
{
public:
    static Y* Clone( const Y* p )
        { return p->CloneMe(); }
};

template<>
class VPTraits<Z>
{
public:
    static Z* Clone( const Z* p )
        { Z* z = new Z; p->CopyTo(*z); return z; }
};
```

Этот способ гораздо лучше, поскольку дизайн закрыт для изменений, но открыт для расширения, и будет работать даже в случае, если, например, в будущем изменится сигнатура CloneMe(). Функция Clone() необходима только для того, чтобы создать объект таким образом, который требуется для данного класса, и единственный ее видимый результат — указатель на новый объект. Это еще один аргумент в пользу строгой инкапсуляции.

Все, что потребуется автору нового класса Y для обеспечения его корректной работы с ValuePtr, — это обеспечить корректную специализацию VPTraits<Y>. Если же класс Y не имеет специализированной функции типа Clone(), то не придется делать даже этого, так как ValuePtr можно использовать и без специализации VPTraits.

И последнее: поскольку VPTraits содержит только одну статическую функцию, почему мы использовали именно шаблон класса, а не просто шаблон функции? Основная причина этого — в инкапсуляции (в частности, для лучшего управления именами) и расширяемости. Мы хотим избежать засорения глобального пространства имен свободными функциями. Шаблон функции можно поместить в область видимости пространства имен, которому принадлежит ValuePtr, но это свяжет нашу функцию с ValuePtr сильнее, чем с любым другим кодом в этом пространстве имен. Да, сегодня нам нужна только функция Clone(), но кто может поручиться, что завтра нам не понадобится какая-то другая функция? Если дополнительные свойства также окажутся функциями, запутанность только усилится. Но что если дополнительное

свойство должно будет быть инструкцией typedef или даже классом? Инкапсулировать такое разнообразие может только класс свойств — VPTraits.

ValuePtr со свойствами

Итак, вот код ValuePtr, который использует свойство клонирования и удовлетворяет всем предъявляемым к нему требованиям.[4] Заметим, что все изменения по сравнению с примером 2*г* представлены одной измененной строкой и одним новым шаблоном с единственной функцией. Это именно то, что я называю минимальным воздействием с максимальным результатом.

```
// Пример 2д. ValuePtr с копированием и присваиванием.
//            и настраиваемостью с использованием свойств
// ---------- Новый код ------------------
template<typename T>
class VPTraits
{
    static T* Clone( const T* p ) { return new T( *p ); }
};
// ---------- Конец нового кода ----------

template<typename T>
class ValuePtr
{
public:
    explicit ValuePtr(T* p = 0) : p_( p ) {}
    ~ValuePtr() { delete p_; }
    T& operator*() const { return *p_; }
    T* operator->() const { return p_; }

    void Swap(ValuePtr& other) { swap(p_, other.p_); }

    ValuePtr( const ValuePtr& other )
        : p_( CreateFrom( *other.p_ ) ) {}

    ValuePtr& operator=( const ValuePtr& other )
    {
        ValuePtr temp( other );
        Swap( temp );
        return *this;
    }

    template<typename U>
    ValuePtr( const ValuePtr<U>& other )
        : p_( CreateFrom( *other.p_ ) ) {}

    template<typename U>
    ValuePtr& operator=( const ValuePtr<U>& other )
    {
        ValuePtr temp( other );
        Swap( temp );
```

[4] Альтернативой может быть передача класса свойств как дополнительного параметра шаблона ValuePtr, который имеет значение по умолчанию VPTraits<T>, как это сделано в шаблоне std::basic_string.

```
template<typename T, typename Traits = VPTraits<T> >
class ValuePtr { /* ... */ };
```

Таким образом, пользователь может иметь в одной программе разные объекты ValuePtr<X>, отличающиеся способом копирования. Эта дополнительная гибкость не имеет большого смысла в нашем конкретном случае, и я так не поступаю; однако не забывайте и об этой возможности.

```
        return *this;
    }

private:

    template<typename U>
    T* CreateFrom( const U* p ) const
    {
        // ---------- Новый код ------------------
        return p ? VPTraits<U>::Clone( p ) : 0;
        // ---------- Конец нового кода ----------
    }

    template<typename U> friend class ValuePtr;

    T* p_;
};
```

Пример использования

Вот пример типичной реализации основных функций класса, использующего `ValuePtr` (многие детали опущены).

```
// Пример 3. Пример использования ValuePtr
//
class X
{
public:
    X() : y_( new Y( /* ... */ ) ) {}

    ~X() {}

    X( const X& other ) : y_(new Y(*(other.y_))) {}

    void Swap( X& other ) { y_.Swap(other.y_); }

    X& operator=( const X& other )
    {
        X temp( other );
        Swap( temp );
        return *this;
    }

private:
    ValuePtr<Y> y_;
};
```

Резюме

Главный вывод, который следует сделать из решения этой задачи: не забывайте о расширяемости при проектировании.

Рекомендация

По умолчанию предпочтительно проектирование с учетом расширяемости.

Стараясь избежать сложных решений простых задач, не забывайте и о долгосрочном использовании ваших решений. Его можно отвергнуть, но забывать о нем не следует — небольшие усилия сейчас могут обернуться большой экономией времени и сил в будущем.

7

Оптимизация и производительность

Эффективность всегда была важна для программиста. C и C++ традиционно предназначаются для создания эффективных программ и стремятся неукоснительно следовать принципу, известному как принцип нулевой стоимости — “вы не должны платить за то, чего не заказывали”.

В данном разделе книги рассматривается несколько ключевых вопросов оптимизации C++. Когда и как вы должны оптимизировать свой код? Что на самом деле делает `inline`? Почему оптимизация бывает способна ухудшить код? И (что самое интересное) как изменятся ответы на эти вопросы при работе в многопоточной среде? В конце концов, нас интересует не теоретическая, а практическая сторона дела, и хотя стандарт C++ ничего не говорит о потоках, все больше и больше программистов ежедневно работают над многопоточными приложениями, так что ответ на последний вопрос для них — насущная необходимость.

Задача 7.1. `inline` ***Сложность: 4***

Вопреки распространенному мнению, ключевое слово `inline` — отнюдь не панацея. Это всего лишь полезный инструмент, если правильно его применять. Когда именно вы должны это делать?

1. Что делает ключевое слово `inline`?
2. Увеличивает ли эффективность описание функции как `inline`?
3. Когда и какие функции следует описывать как `inline`?

Решение

1. Что делает ключевое слово `inline`?

Описание функции как `inline` указывает компилятору, что это встроенная функция и что он может размещать копию кода функции непосредственно там, где этот код используется. Когда компилятор поступает таким образом, код для вызова функции не генерируется.

2. Увеличивает ли эффективность описание функции как `inline`?

Не обязательно.

Во-первых, если вы попытаетесь ответить на этот вопрос, не определив, что именно вы хотите оптимизировать, то попадете в классическую ловушку. Первым вопро-

сом должен быть следующий: что мы подразумеваем под эффективностью? Имеем ли мы в виду размер программы, использование памяти, время выполнения, скорость разработки, время построения или что-то, не попавшее в этот список?

Во-вторых, вопреки распространенному мнению, объявление функции как `inline` может как улучшить, так и ухудшить все перечисленные аспекты эффективности.

а. *Размер программы.* Многие программисты считают, что использование `inline` приводит к увеличению размера программ, поскольку вместо одной копии кода функции компилятор создает его в каждом месте использования этой функции. Это мнение справедливо довольно часто, но не всегда. Если размер функции меньше, чем размер генерируемого компилятором кода для выполнения вызова функции, использование `inline` приводит к снижению размера программы.

б. *Использование памяти.* Обычно использование `inline` не влияет на использование памяти, не считая влияния размера программы (см. предыдущий пункт).

в. *Время выполнения.* Многие программисты считают, что использование встроенных функций снижает время выполнения программы, поскольку устраняет накладные расходы на вызов функции; кроме того, поскольку код `inline`-функции встроен в код вызывающей функции, для оптимизации открываются бо́льшие возможности. Это может быть справедливо, но не всегда. Если только функция не вызывается очень часто, видимого повышения скорости работы программы не имеется. Более того, вполне может произойти обратное: `inline`-функция увеличивает размер вызывающей функции, что приводит к снижению ее локализации, и, например, если внутренний цикл вызывающей программы больше не помещается в кэше процессора, то общая скорость выполнения может заметно снизиться. Следует, однако, заметить, что большинство реальных программ ориентировано не на использование процессора, а на работу с устройствами ввода-вывода, которые и оказываются самым узким местом в плане времени выполнения.

г. *Скорость разработки*, *время построения.* Чтобы встроенные функции были максимально полезны, их код должен быть видим вызывающим функциям, что означает, что вызывающие функции зависят от встроенных функций. Зависимость от деталей внутренней реализации другого модуля увеличивает практическую взаимосвязь модулей (это не означает усиление теоретической взаимосвязи, так как вызывающая функция никак не использует внутреннюю реализацию вызываемой функции). Обычно при изменении функции перекомпилировать вызывающую ее функцию не требуется — достаточно только перекомпоновать программу. Однако при изменении встроенной функции требуется перекомпиляция всех вызывающих функций. Еще одно влияние на скорость разработки связано с отладкой встроенных функций, так как пошаговое их прохождение и управление точками останова в них существенно сложнее для большинства отладчиков.

   Единственный случай применения встроенных функций, который можно рассматривать как оптимизацию скорости разработки, — это встроенные функции доступа, позволяющие избежать открытости членов данных. Применение для этого встроенных функций — хороший стиль кодирования, обеспечивающий лучшую изоляцию модулей.

И последнее: когда вы пытаетесь повысить эффективность программы, всегда начинайте с рассмотрения используемых вами алгоритмов и структур данных. Таким образом можно повысить производительность программ на порядки, в то время как оптимизация процесса (в частности, применение встроенных функций) в общем случае (подчеркиваю — в общем случае) приводит к гораздо меньшим результатам.

Скажи оптимизации “Нет”

3. Когда и какие функции следует описывать как `inline`?

Ответ на этот вопрос такой же, как и на другие вопросы, связанные с оптимизацией: только тогда, когда необходимость этого будет доказана профайлером, и ни минутой раньше. Из этого правила есть несколько исключений, когда вы используете встроенные функции невзирая ни на что, например для пустых функций, которые останутся пустыми, или при написании не экспортируемых шаблонов.

Рекомендация

Первое правило оптимизации: не оптимизируйте.
Второе правило оптимизации: если вы все же решите оптимизировать, см. правило первое.

Основной вывод — за использование встроенных функций приходится платить, хотя бы ценой повышения взаимосвязей модулей, и вы не должны платить до тех пор, пока не убедитесь в том, что в результате получите достаточную выгоду.

“Но я всегда могу сказать, где узкое место в моем коде”, — можете возразить вы. Вы не одиноки, и большинство программистов именно так и считает время от времени, но они ошибаются. Иногда может повезти, и программист может правильно указать узкое место, но в большинстве случаев он ошибается. Обычно точно указать узкое место может только профайлер.

И еще одно практическое замечание: профайлеры обычно гораздо хуже способны указать, какие встроенные функции *не* следует делать таковыми, чем какие функции следует сделать встроенными.

Вычислительные задачи

Задачи некоторых программистов заключаются в написании небольшого тесно связанного библиотечного кода, такого как современные научные и инженерные числовые библиотеки, в которых использование встроенных функций вполне оправданно. Но даже разработчики такого программного обеспечения должны использовать `inline`-функции благоразумно и только при окончательной настройке кода, а не при первоначальной. Заметим, что написание модуля и сравнение производительности при включенной встроенности функций и при отключенной — не лучшая идея, так как “все включено” или “все отключено” — очень грубый способ сравнения, который может дать нам только усредненную информацию, и не в состоянии сказать, какие именно функции стоит делать встроенными, а какие — нет. Так что даже при написании такого тесно связанного небольшого кода для выяснения, какие функции стоит (и стоит ли) делать встроенными, лучше всего воспользоваться профайлером.

Функции доступа

Есть программисты, которые доказывают необходимость того, чтобы однострочные функции доступа (типа “`X& Y::f(){ return myX_; }`”) были признаны исключением из правил и считались встроенными автоматически. Я понимаю их доводы, но здесь также следует быть осторожным. Как минимум любой встроенный код усиливает взаимосвязь. Так что кроме случаев, когда вы заранее твердо знаете, что встраивание приведет к улучшению кода, стоит все же отложить принятие такого решения до проведения профилирования. Тогда вы будете точно знать, что именно вы делаете, почему, и чем это для вас обернется.

Рекомендация

Избегайте тонкой настройки производительности вообще и использования встроенности в частности до тех пор, пока необходимость этого не будет доказана с использованием профайлера.

Задача 7.2. Отложенная оптимизация. Часть 1 *Сложность: 2*

Копирование при записи (именуемое также "отложенным копированием" ("lazy copy")[1]) представляет собой распространенную технологию оптимизации, использующую счетчики ссылок. Знаете ли вы, как правильно реализовать ее? В этой задаче мы рассмотрим простейший строковый класс, не использующий счетчиков ссылок, а в оставшейся части этой мини-серии задач мы рассмотрим, к каким последствиям может привести добавление к классу семантики копирования при записи.

Рассмотрим следующий упрощенный класс `String`. (Примечание: он не предназначается для использования в качестве полнофункционального строкового класса и служит только в качестве примера, так что в нем отсутствуют многие возможности реального строкового класса. В частности, в нем отсутствует `operator=()`, код которого по сути тот же, что и у конструктора копирования.)

```
namespace Original
{
    class String
    {
    public:
        String();                // Пустая строка
        ~String();               // Очистка буфера
        String( const String& ); // Копирование
        void Append( char );     // Добавление символа

        // ... operator=() и другие функции опущены ...

    private:
        char*   buf_;            // Выделенный буфер
        size_t  len_;            // размер буфера
        size_t  used_;           // Количество символов
    };
}
```

Это простой класс, в котором не используется никакая оптимизация. При копировании `Original::String` новый объект немедленно выделяет свой собственный буфер, и вы получаете два совершенно различных объекта.

Ваша задача — реализовать `Original::String`.

[1] В литературе используется также перевод "ленивое копирование", но в данном случае использован термин "отложенное копирование", тем более что его смысл ближе ко второму названию данной технологии — "копирование при записи", т.е. по сути копирования, отложенного до осуществления записи. — *Прим. перев.*

Решение

Реализация Original::String

Вот простейшая реализация. Написать конструктор и деструктор по умолчанию очень просто.

```
namespace Original {

    String::String() : buf_(0), len_(0), used_(0) {}

    String::~String() { delete[] buf_; }
```

Теперь для реализации копирующего присваивания мы просто выделяем новый буфер и используем стандартный алгоритм `copy()` для копирования содержимого исходной строки.

```
String::String( const String& other )
    : buf_(new char[other.len_]),
      len_(other.len_),
      used_(other.used_)
{
    copy( other.buf_, other.buf_ + used_, buf_ );
}
```

Далее для ясности кода реализована вспомогательная функция `Reserve()`, которая может понадобиться при написании и других функций, помимо `Append()`. `Reserve()` гарантирует, что наш внутренний буфер имеет размер как минимум `n` символов и выделяет дополнительное пространство с использованием стратегии экспоненциального роста.

```
    void String::Reserve( size_t n )
    {
        if (len_ < n )
        {
            size_t newlen = max( len_ * 1.5, n );
            char*  newbuf = new char[ newlen ];
            copy( buf_, buf_ + used_, newbuf );

            delete[] buf_;   // Вся работа сделана,
            buf_ = newbuf;   // надо только получить
            len_ = newlen;   // владение
        }
    }
```

И наконец, `Append()` убеждается, что в строке имеется необходимое пространство и добавляет к строке новый символ.

```
    void String::Append( char c )
    {
        Reserve( used_ + 1 );
        buf_[ used_++ ] = c;
    }
}
```

Отступление от темы: какая стратегия увеличения буфера лучше

Когда размер строки превышает выделенный ей буфер, необходимо выделить новый буфер большего размера. Ключевым вопросом в этом случае является вопрос: какого размера должен быть новый буфер? Заметим, это далеко не праздный вопрос, и анализ, кото-

рый мы проведем, применим и к другим структурам и контейнерам, которые используют выделяемые буфера, — например, к стандартному контейнеру `vector`.

Имеется несколько распространенных стратегий роста буфера. Я опишу каждую из них в терминах количества необходимых выделений и среднего количества копирований каждого символа для строки с конечной длиной N.

а. *Точный рост*. При этой стратегии новый буфер имеет в точности необходимый для текущей операции размер. Например, добавление одного символа, а затем еще одного приводит к двум выделениям буфера — сначала для существующей строки и первого нового символа, а потом — для получившейся после первого добавления символа строки и второго добавляемого символа.

- Преимущество: нет расхода памяти впустую.
- Недостаток: низкая производительность. Эта стратегия требует $O(N)$ выделений памяти и в среднем $O(N)$ копирований на символ, с высоким значением коэффициента в худшем случае (так же, как и в следующей стратегии на шаге 1). Величина коэффициента зависит от кода пользователя, и разработчик `String` влиять на нее не может.

б. *Рост с фиксированным шагом*. Новый буфер больше текущего на некоторое фиксированное количество байт. Например, 64-байтовый шаг означает, что все буфера строк будут иметь размеры, кратные 64 байтам.

- Преимущество: малый расход памяти впустую. Количество неиспользуемой памяти буфера ограничено размером шага и не изменяется с изменением длины строки.
- Недостаток: невысокая производительность. Эта стратегия, так же, как и предыдущая, требует $O(N)$ выделений и в среднем $O(N)$ копирований на символ, т.е. и количество выделений, и количество копирований линейно растут с ростом строки. Однако в этом случае управление коэффициентом линейной зависимости находится в руках разработчика `String`.

в. *Экспоненциальный рост*. Новый буфер больше прежнего на размер буфера, умноженный на множитель F. Например, при $F = 0.5$ добавление одного символа к заполненной строке длиной 100 байт приведет к выделению нового буфера размером 150 байт.

- Преимущество: высокая производительность. Эта стратегия требует $O(\log N)$ выделений памяти и $O(1)$ копирований на символ, т.е. количество выделений памяти растет как логарифм длины строки, а среднее количество копирований данного символа представляет собой *константу*.
- Недостаток: расход части памяти впустую. Количество неиспользованной памяти в буфере всегда строго меньше, чем $N \times F$ байт; средний непроизводительный расход памяти равен $O(N)$.

Подытожим приведенное описание стратегий в виде следующей таблицы.

Стратегия роста	**Количество выделений памяти**	**Количество Копирований символов**	**Непроизводительные расходы памяти**
Точная	$O(N)$ с большим множителем	$O(N)$ с большим множителем	Нет
Фиксированный шаг	$O(N)$	$O(N)$	$O(1)$
Экспоненциальный рост	$O(\log N)$	$O(1)$	$O(N)$

Вообще говоря, наилучшей в смысле производительности является стратегия экспоненциального роста. Рассмотрим изначально пустую строку и увеличим ее до длины 1200 символов. Рост с фиксированным шагом требует $O(N)$ выделений памяти и в среднем копирует каждый символ $O(N)$ раз (в нашем случае при 32-байтовом шаге требуется 38 выделений памяти и в среднем 19 копирований каждого символа). Экспоненциальный рост требует $O(\log N)$ выделений памяти и только $O(1)$ копирований каждого символа. В нашем случае при использовании множителя 1.5 потребуется 10 выделений памяти и в среднем два копирования каждого символа.

	1200 символов		**12000 символов**	
	Фиксированный шаг (32 байта)	Экспоненциальный рост (1.5×)	Фиксированный шаг (32 байта)	Экспоненциальный рост (1.5×)
Количество выделений памяти	38	10	380	16
Среднее количество копирований символов	19	2	190	2

Результат может оказаться неожиданным. Дополнительную информацию по этому вопросу можно найти в колонке Эндрю Книга (Andrew Koenig) в сентябрьском 1998 года номере *JOOP* (*Journal of Object-Oriented Programming*). Книг также показал, почему (опять-таки, в общем случае) наилучший множитель не равен 2, а, вероятно, близок к 1.5. Им также показано, что среднее количество копирований символов в случае экспоненциального роста не зависит от длины строки.

Задача 7.3. Отложенная оптимизация. Часть 2 *Сложность: 3*

Копирование при записи представляет собой распространенный вид оптимизации. Знаете ли вы, как его реализовать?

`Original::String` из задачи 7.2 неплох, но иногда предпринимается копирование строки, использование ее без каких-либо попыток модификации, а затем уничтожение.

“Обидно, — может сказать разработчик `Original::String` сам себе, — что я всегда выполняю работу по выделению нового буфера (недешевая операция), даже когда все, что надо пользователю, — это прочесть данные из новой строки и уничтожить ее. Ведь с тем же успехом можно было бы позволить двум объектам ссылаться на один и тот же буфер и копировать его только при необходимости, когда объект намерен изменить его содержимое. Таким образом, если пользователь не модифицирует копию строки, мы избавимся от выполнения никому не нужной работы!”

С улыбкой на лице и решимостью в глазах разработчик проектирует класс `Original::String`, использующий реализацию копирования при записи (именуемого также “отложенным копированием”) посредством счетчика ссылок.

```
namespace Optimized
{
    class StringBuf
    {
    public:
        StringBuf();              // Пустая строка
        ~StringBuf();             // Очистка буфера
        void Reserve( size_t n); // Обеспечение len >= n
```

```
        char*    buf;            // Выделенный буфер
        size_t   len;            // размер буфера
        size_t   used;           // Количество символов
        unsigned refs;           // Счетчик ссылок

    private:
        // Копирование не разрешено...
        StringBuf( const StringBuf& );
        StringBuf& operator=( const StringBuf& );
    };

    class String
    {
    public:
        String();                // Пустая строка
        ~String();               // Уменьшение счетчика,
                                 // очистка буфера при
                                 // нулевом счетчике
        String( const String& ); // Указывает на тот же
                                 // буфер и увеличивает
                                 // счетчик ссылок
        void Append( char );     // Добавление символа

        // ... operator=() и другие функции опущены ...

    private:
        StringBuf* data_;
    };
}
```

Ваша задача: реализовать `Optimized::StringBuf` и `Optimized::String`.[2] Для упрощения логики можно добавить закрытую вспомогательную функцию `String::AboutToModify()`.

Решение

Сначала рассмотрим класс `StringBuf`. Заметим, что почленное копирование и присваивание (обеспечиваемые по умолчанию) для `StringBuf` смысла не имеют, так что мы запрещаем обе операции (объявляя их как `private` и не определяя их).

`Optimized::StringBuf` выполняет часть работы `Original::String`. Конструктор по умолчанию и деструктор делают именно то, что от них и ожидается.

```
namespace::Optimized
{
    StringBuf::StringBuf()
        : buf(0), len(0), used(0), refs(1) {}

    StringBuf::~StringBuf() { delete[] buf; }
```

Функция `Reserve()` очень похожа на эту функцию в классе `Original::String`.

```
    void StringBuf::Reserve( size_t n )
    {
        if (len < n )
        {
            size_t newlen = max( len*1.5, n );
            char*  newbuf = new char[ newlen ];
            copy( buf, buf+used, newbuf );
```

[2] Между прочим, данная задача иллюстрирует еще одно использование пространств имен — более ясное представление кода. Пространства имен позволяют сделать код более понятным. Согласитесь, что использование длинных имен классов наподобие `OptimizedString` или `OriginalString` сделало бы код менее понятным и удобочитаемым.

```
            delete[] buf;  // Вся работа сделана,
            buf = newbuf;  // передаем владение
            len = newlen;
        }
    }
```

Это все, что касается `StringBuf`. Теперь перейдем к собственно `String`. Проще всего реализуется конструктор по умолчанию.

```
String::String() : data_( new StringBuf ) {}
```

В деструкторе мы должны не забыть о счетчике ссылок, так как могут иметься и другие объекты, которые используют то же представление `StringBuf`. Если другие объекты `String` используют тот же `StringBuf`, что и наш объект, нам следует просто уменьшить счетчик ссылок, не изменяя объект `StringBuf`; но если наш объект — последний его пользователь, то нам следует позаботиться об удалении буфера.

```
String::~String()
{
    if ( --data_->refs < 1 ) // Последний
    {
        delete data_;
    }
}
```

Единственное другое место, где мы изменяем счетчик ссылок, — это конструктор копирования, который реализуется с использованием семантики отложенного копирования. В нем мы указываем на буфер копируемой строки и увеличиваем его счетчик ссылок, указывая на наше присутствие. Это так называемое “мелкое” копирование; глубокое копирование будет выполнено при необходимости внесения изменений в одну из строк, совместно использующих один и тот же буфер.

```
String::String( const String& other )
    : data_( other.data_ )
{
    ++data_->refs;
}
```

Для ясности я реализовал дополнительную вспомогательную функцию `AboutToModify()`, которая будет нужна как при работе `Append()`, так и для реализации в будущем других функций, изменяющих строку. `AboutToModify()` гарантирует, что мы работаем с копией строки, которую не использует никакой другой объект, выполняя отложенное глубокое копирование, если до сих пор оно не было осуществлено. Для удобства функция получает в качестве параметра необходимый минимальный размер буфера и при необходимости выполняет повторное выделение памяти.

```
void String::AboutToModify( size_t n )
{
    if ( data_->refs > 1 )
    {
        auto_ptr<StringBuf> newdata( new StringBuf );
        newdata->Reserve( max( data_->len, n ) );
        copy( data_->buf, data_->buf + data_->used,
              newdata->buf );
        newdata->used = data_->used;

        --data_->refs;              // Вся работа сделана,
        data_ = newdata.release(); // передаем владение
    }
    else
    {
        data_->Reserve( n );
    }
}
```

Теперь, когда вся работа сделана, функция `Append()` имеет очень простой вид. Как и ранее, она просто объявляет о своем намерении изменить строку, тем самым гарантируя, что буфер безраздельно принадлежит текущему объекту и в нем достаточно места для дополнительного символа. После этого выполняется само изменение строки.

```
    void String::Append( char c )
    {
        AboutToModify( data_->used + 1 );
        data_->buf[ data_->used++ ] = c;
    }
}
```

Вот и все. Далее мы несколько расширим интерфейс класса и посмотрим, что из этого получится.

Задача 7.4. Отложенная оптимизация. Часть 3 *Сложность: 6*

В этой части мини-серии мы рассмотрим влияние ссылок и итераторов на копируемую при записи строку. Сможете ли вы распознать возникающие проблемы?

Рассмотрим класс `Optimized::String` из предыдущей задачи, но с двумя дополнительными функциями: `Length()` и `operator[]()`.

```
namespace Optimized
{
    class StringBuf
    {
    public:
        StringBuf();              // Пустая строка
        ~StringBuf();             // Очистка буфера
        void Reserve( size_t n); // Обеспечение len >= n

        char*    buf;             // Выделенный буфер
        size_t   len;             // размер буфера
        size_t   used;            // Количество символов
        unsigned refs;            // Счетчик ссылок

    private:
        // Копирование не разрешено...
        StringBuf( const StringBuf& );
        StringBuf& operator=( const StringBuf& );
    };

    class String
    {
    public:
        String();                 // Пустая строка
        ~String();                // Уменьшение счетчика,
                                  // очистка буфера при
                                  // нулевом счетчике
        String( const String& ); // Указывает на тот же
                                  // буфер и увеличивает
                                  // счетчик ссылок
        void   Append( char );    // Добавление символа

        size_t Length() const;    // Длина строки

        char& operator[](size_t);// Доступ к элементу
        const char operator[](size_t) const;

        // ... operator=() и другие функции опущены ...

    private:
        StringBuf* data_;
    };
}
```

Эти функции позволят писать код наподобие следующего.

```
if ( s.Length() > 0 )
{
    cout << s[0];
    s[0] = 'a';
}
```

Реализуйте новые члены `Optimized::String`. Следует ли изменять другие функции-члены из-за появления новых? Поясните.

Решение

Рассмотрим класс `Optimized::String` из предыдущей задачи, но с двумя дополнительными функциями: `Length()` и `operator[]()`.

Цель этой задачи — продемонстрировать, почему дополнительная функциональность наподобие `operator[]()` изменяет семантику `Optimized::String` настолько, что это влияет на другие части класса. Но сначала подумаем о первой части задания: **реализуйте новые члены `Optimized::String`**.

Функция `Length()` проще всего.

```
namespace Optimized
{
    size_t String::Length() const
    {
        return data_->used;
    }
```

Однако `operator[]()` не так прост, как может показаться на первый взгляд. Я хочу, чтобы вы обратили внимание на то, что действие этого оператора (неконстантной версии, возвращающей ссылку) не отличается от действия обычного итератора для стандартного класса `string`. Любые реализации `std::basic_string` с отложенным копированием должны учитывать вопросы, рассматриваемые нами в данной задаче.

operator[] для разделяемых строк

Вот простейшая первая попытка.

```
// плохо. Попытка №1
//
char& String::operator[]( size_t n )
{
    return data_->buf[n];
}

const char String::operator[]( size_t n ) const
{
    return data_->buf[n];
}
```

Эта попытка неудачна. Рассмотрим следующий фрагмент.

```
// Пример 1. Почему не годится попытка №1
//
void f( const Optimized::String& s )
{
    Optimized::String& s2( s ); // Получаем копию строки
    s2[0] = 'x';                // Изменяется и s2, и s!
}
```

Обратите внимание на то, что произошло: изменились не просто две строки там, где мы хотели изменить только одну, — произошло изменение строки, описанной как `const`! Так что нам ничего не остается, как вернуться к `operator[]()` и сделать еще одну попытку. Первое, что приходит на ум, — все наши неприятности связаны с наличием буфера, используемого несколькими объектами одновременно. Соответственно, неопытный программист тут же найдет решение, заключающееся в вызове `AboutToModify()` для гарантии того, что буфер будет использоваться только текущим объектом.

```
// плохо. Попытка №2.
//
char& String::operator[]( size_t n )
{
    AboutToModify( data_->len );
    return data_->buf[n];
}

const char String::operator[]( size_t n ) const
{
    // Здесь проверка совместного использования не нужна
    return data_->buf[n];
}
```

Это уже лучше, хотя все еще недостаточно хорошо. Достаточно несколько изменить порядок действий из примера 1, и мы получим тот же результат, что и ранее.

```
// Пример 2. Почему не годится попытка №2
//
void f( Optimized::String& s )
{
    char& rc = s[0];            // Ссылка на первый символ
    Optimized::String& s2( s ); // Получаем копию строки
    rc = 'x';                   // Изменяется и s2, и s!
}
```

В данном случае проблема заключается в том, что ссылку мы получаем в тот момент, когда буфер используется только одной строкой, но затем этот буфер используют уже две строки, и одно обновление посредством полученной ранее ссылки приводит к изменению видимого состояния двух объектов `String`.

Неразделяемая строка

При вызове неконстантного оператора `[]`, кроме получения единолично используемого буфера `StringBuf`, нам требуется пометить строку как “неразделяемую” на тот случай, если пользователь запомнит ссылку и воспользуется ею позже.

Маркировка строки как “неразделяемой навсегда” будет работать, но на самом деле это несколько излишне. На самом деле нам достаточно маркировать строку как “неразделяемую некоторое время”. Чтобы понять, что я имею в виду, обратите внимание на то, что после операций, изменяющих строку, возвращаемая оператором `[]` ссылка становится недействительной. Так что код наподобие приведенного

```
// Пример 3. Почему после изменения строки ссылка
//           становится недействительной
//
void f( Optimized::String& s )
{
    char& rc = s[0];
    s.Append( 'i' );
    rc = 'x';  // Ошибка: буфер может быть перемещен
}              // в связи с выделением памяти
```

должен быть документирован как некорректный, независимо от использования копирования при записи. Короче говоря, операция, изменяющая строку, делает недействи-

тельными все ссылки на ее символы, поскольку вы не знаете, перемещается ли при этом память, отведенная для строки, или нет (перемещение с точки зрения вызывающего кода невидимо).

Это приводит к тому, что в примере 2 `rc` станет недействительным после выполнения очередной изменяющей содержимое строки операции над `s`. Таким образом, вместо того, чтобы пометить строку как "неразделяемую навсегда", достаточно использовать метку "неразделяемая до очередной изменяющей операции", после которой все ссылки внутрь строки все равно станут недействительными.

Следует ли изменять другие функции-члены из-за появления новых? Поясните.

Как мы могли увидеть, ответ на этот вопрос положительный.

Начнем с того, что нам требуется иметь возможность запоминать состояние данной строки в смысле возможности совместного ее использования. Проще всего использовать для этого логическое значение в качестве флага, но можно обойтись и без него. Мы можем закодировать "неразделяемое состояние" в счетчике ссылок, использовав для этого значение счетчика ссылок, равное наибольшему возможному значению соответствующего типа. Кроме того, следует добавить параметр-флаг к функции `AboutToModify()`, который указывает, следует ли помечать строку как неразделяемую.

```
// ХОРОШО. Попытка № 3
// Для удобства добавлен статический член и
// соответствующим образом изменена функция
// AboutToModify(). Кроме того, теперь имеется
// конструктор копирования StringBuf

    const size_t String::Unshareable =
                    numeric_limits<size_t>::max();

    StringBuf::StringBuf( const StringBuf& other, size_t n )
        : buf(0), len(0), used(0), refs(1)
    {
        Reserve( max( other.len, n ) );
        copy( other.buf, other.buf + other.used, buf );
        used = other.used;
    }

    void String::AboutToModify(
                size_t n,
                bool markUnshareable /* = false */ )
    {
        if ( data_->refs > 1 && data_->refs != Unshareable )
        {
            StringBuf* newdata = new StringBuf( *data_, n );
            --data_->refs;          // Вся работа сделана,
            data_ = newdata;        // передаем владение
        }
        else
        {
            data_->Reserve( n );
        }
        data_->refs = markUnshareable ? Unshareable : 1;
    }

    char& String::operator[]( size_t n )
    {
        AboutToModify( data_->len, true );
        return data_->buf[n];
    }

    const char String::operator[]( size_t n ) const
    {
        return data_->buf[n];
    }
```

Заметим, что все прочие вызовы `AboutToModify()` продолжают работать так же, как и ранее. Обратите также внимание и на изменения в конструкторе копирования `String`, связанные с возможной пометкой объекта как неразделяемого.

```
String::String( const String& other )
// По возможности используем копирование при записи;
// в противном случае - немедленное глубокое копирование
{
    if ( other.data_->refs != Unshareable )
    {
        data_ = other.data_;
        ++data_->refs;
    }
    else
    {
        data_ = new StringBuf( *other.data_ );
    }
}
```

Небольшая поправка требуется и в деструкторе `String`.

```
String::~String()
{
    if ( data_->refs == Unshareable ||
         --data_->refs < 1 )
    {
        delete data_;
    }
}
```

Остальные функции работают так же, как и ранее.

```
String::String() : data_( new StringBuf ) {}

void String::Append( char c )
{
    AboutToModify( data_->used + 1 );
    data_->buf[ data_->used++ ] = c;
}
```

Вот и все... в однопоточной среде. В последней задаче мини-серии мы рассмотрим влияние многопоточности на копирование при записи.

Резюме

Здесь приведен окончательный код классов `StringBuf` и `String`. Я несколько изменил функцию `StringBuf::Reserve()` — теперь размер нового буфера всегда будет кратен 4. Это сделано с целью повышения эффективности, так как многие распространенные операционные системы все равно не распределяют память меньшими блоками. Кроме того, для малых строк этот код несколько быстрее, чем исходная версия (в первоначальном варианте до того, как заработает экспоненциальный рост, будут последовательно выделены буфера размером 1, 2, 3, 4 и 6 байт; исправленный вариант сразу выделит 4-байтовый буфер, а за ним — 8-байтовый).

```
namespace Optimized
{
    class StringBuf
    {
    public:
        StringBuf();                // Пустая строка
        ~StringBuf();               // Очистка буфера
        StringBuf( const StringBuf& other, size_t n = 0 );
                                    // Копирование буфера
```

```
    void Reserve( size_t n); // Обеспечение len >= n

    char*    buf;            // Выделенный буфер
    size_t   len;            // размер буфера
    size_t   used;           // Количество символов
    unsigned refs;           // Счетчик ссылок

private:
    // Копирование не разрешено...
    StringBuf& operator=( const StringBuf& );
};

class String
{
public:
    String();                // Пустая строка
    ~String();               // Уменьшение счетчика,
                             // очистка буфера при
                             // нулевом счетчике
    String( const String& ); // Указывает на тот же
                             // буфер и увеличивает
                             // счетчик ссылок
    void   Append( char );   // Добавление символа

    size_t Length() const;   // Длина строки

    char& operator[](size_t);// Доступ к элементу
    const char operator[](size_t) const;

    // ... operator=() и другие функции опущены ...

private:
    void AboutToModify( size_t n,
                        bool bUnshareable = false );
                      // Отложенное копирование,
                      // обеспечение len >= n
                      // и метка неразделяемости
    static const size_t Unshareable;
                      // Значение флага неразделяемости
    StringBuf* data_;
};

StringBuf::StringBuf()
    : buf(0), len(0), used(0), refs(1) {}

StringBuf::~StringBuf() { delete[] buf; }

StringBuf::StringBuf( const StringBuf& other, size_t n )
    : buf(0), len(0), used(0), refs(1)
{
    Reserve( max( other.len, n ) );
    copy( other.buf, other.buf + other.used, buf );
    used = other.used;
}

void StringBuf::Reserve( size_t n )
{
    if (len < n )
    {
        // Округляем новое значение до ближайшего
        // кратного 4 байтам
        size_t needed = max<size_t>( len*1.5, n );
        size_t newlen = needed ?
                        4 * ((needed-1)/4 + 1) : 0;
        char*  newbuf = newlen ? new char[ newlen ] : 0;
```

```
        if ( buf )
        {
            copy( buf, buf+used, newbuf );
        }

        delete[] buf;  // Вся работа сделана,
        buf = newbuf;  // передаем владение
        len = newlen;
    }
}

const size_t String::Unshareable =
                     numeric_limits<size_t>::max();

String::String() : data_( new StringBuf ) {}

String::~String()
{
    if ( data_->refs == Unshareable ||
         --data_->refs < 1 )
    {
        delete data_;
    }
}

String::String( const String& other )
// По возможности используем копирование при записи;
// в противном случае - немедленное глубокое копирование
{
    if ( other.data_->refs != Unshareable )
    {
        data_ = other.data_;
        ++data_->refs;
    }
    else
    {
        data_ = new StringBuf( *other.data_ );
    }
}

void String::AboutToModify(
             size_t n,
             bool markUnshareable /* = false */ )
{
    if ( data_->refs > 1 && data_->refs != Unshareable )
    {
        StringBuf* newdata = new StringBuf( *data_, n );
        --data_->refs;           // Вся работа сделана,
        data_ = newdata;         // передаем владение
    }
    else
    {
        data_->Reserve( n );
    }
    data_->refs = markUnshareable ? Unshareable : 1;
}

void String::Append( char c )
{
    AboutToModify( data_->used + 1 );
    data_->buf[ data_->used++ ] = c;
}

size_t String::Length() const
{
    return data_->used;
```

```
    }

    char& String::operator[]( size_t n )
    {
        AboutToModify( data_->len, true );
        return data_->buf[n];
    }

    const char String::operator[]( size_t n ) const
    {
        return data_->buf[n];
    }

}
```

Задача 7.5. Отложенная оптимизация. Часть 4 ***Сложность: 8***

В последней задаче мини-серии рассматривается влияние безопасности потоков[3] на копирование при записи. В действительности ли отложенное копирование является оптимизацией? Ответ может вас удивить.

1. Почему `Optimized::String` из предыдущей задачи небезопасна в смысле потоков? Приведите примеры.
2. Покажите, как сделать `Optimized::String` безопасным:
   а) в предположении атомарности операций получения, установки и сравнения целых чисел;
   б) без этого предположения.
3. Обсудите, как все это повлияет на производительность.

Решение

Вопросы безопасности потоков

Стандарт C++ ничего не говорит о потоках. В отличие от Java, C++ не имеет встроенной поддержки потоков и не пытается решить вопросы безопасности потоков посредством языка. Так почему же мы рассматриваем этот вопрос в нашей книге? Да просто потому, что все больше и больше программистов пишут многопоточные программы, и обсуждение копирования при записи было бы неполным без рассмотрения вопросов безопасности потоков.

Первые классы для работы с потоками появились практически сразу же после появления C++, и в настоящее время они входят во многие коммерческие библиотеки.[4] Вероятно, в своей повседневной практике вы тоже используете либо такие библиотеки, либо пишете собственные оболочки вокруг сервисов, предоставляемых вашей операционной системой.

1. **Почему `Optimized::String` из предыдущей задачи небезопасна в смысле потоков? Приведите примеры.**

Код, использующий объект, должен при необходимости обеспечить последовательность обращений к этому объекту. Например, если некоторый объект `String` мо-

[3] Далее в этой главе под безопасностью (потоков), если не оговорено иное, будет пониматься код, безопасный в смысле потоков, т.е. код, корректно работающий в многопоточной среде. — *Прим. перев.*

[4] См., например, `http://www.gotw.ca/publications/mxc++/ace.htm`, `http://www.gotw.ca/publications/mxc++/omniorb.htm`.

жет быть модифицирован двумя различными потоками, то в обязанности объекта не входит защита от некорректного изменения двумя потоками одновременно — об этом должны позаботиться сами потоки.

Код из предыдущей задачи имеет две проблемы. Во-первых, реализация копирования при записи разработана для сокрытия того факта, что два различных видимых объекта `String` могут совместно использовать одно общее скрытое состояние. Следовательно, за то, чтобы вызывающий код не мог изменить объект `String`, представление которого используется совместно с другими объектами, должен отвечать сам класс `String`. Приведенный ранее код класса поступает именно так, выполняя глубокое копирование (отделяя представления разных объектов) при попытке вызывающего кода модифицировать объект. Этот код хорошо работает в целом.

К сожалению, это приводит нас ко второй проблеме, заключающейся в небезопасности кода, выполняющего отделение представления разных объектов. Представим себе два объекта `String` — `s1` и `s2` — в следующей ситуации.

а. `s1` и `s2` совместно используют одно и то же представление объектов (все в порядке, класс `String` создан именно для этого).

б. Поток 1 пытается изменить `s1` (все правильно, поскольку поток 1 знает, что больше никто не пытается изменить объект `s1`).

в. Поток 2 пытается изменить `s2` (все правильно, поскольку поток 2 знает, что больше никто не пытается изменить объект `s2`).

г. Но они делают это одновременно, а это уже ошибка!

Проблема в последнем пункте. Объекты `s1` и `s2` пытаются одновременно отделить свои представления от представления другого объекта, а выполняющий это действие код небезопасен. В частности, рассмотрим самую первую строку кода функции `String::AboutToModify()`.

```
void String::AboutToModify(
             size_t n,
             bool   markUnshareable /* = false */
     )
{
    if ( data_->refs > 1 && data_->refs != Unshareable )
    {
        /* ... и т.д. ... */
```

Эта инструкция `if` не является безопасной. Например, даже вычисление `data_->refs > 1` может не быть атомарным. Если так, вполне возможно, что поток 1 пытается вычислить значение `data_->refs > 1` в тот момент, когда поток 2 обновляет значение `refs`, и в этом случае значение, прочитанное из `data_->refs`, может оказаться любым — в том числе не являющимся ни исходным, ни обновленным значением. Проблема заключается в том, что `String` не следует основному требованию безопасности, гласящему, что код, использующий объект, должен позаботиться в случае необходимости о последовательном доступе к нему. В нашем случае `String` должен побеспокоиться о том, чтобы никакие два потока не могли использовать одно и то же значение `refs` небезопасным образом в один и тот же момент времени. Традиционно это достигается за счет последовательного обращения к `StringBuf` (или только к его члену `refs`) с использованием взаимоисключений или семафоров. В нашем случае гарантированно атомарной должна быть как минимум операция сравнения целых чисел.

Рассмотренный материал приводит нас ко второму вопросу. Даже если чтение и обновление `refs` атомарны, даже если мы работаем на однопроцессорной машине, так что вместо истинной параллельности выполнения потоки чередуются, все равно инструкция `if` содержит две части. Проблема заключается в том, что поток, выпол-

няющий приведенный выше код, может быть прерван после вычисления первой части выражения, но до вычисления второй, как показано ниже.

Поток 1	Поток 2
Вход в `s1.AboutToModify()`. Вычисление выражения `data_->refs > 1` (истинно, так как `data_->refs` равно 2)	
Переключение	*контекста*
	Вход в `s2.AboutToModify()`. Выполнение до конца, включая уменьшение `data_->refs` до 1. **Выход из** `s2.AboutToModify()`
Переключение	*контекста*
Вычисление `data_->refs != Unshareable` (истинно, так как `data_->refs` равно 1) и вход в блок отделения представлений разных объектов (где представление клонируется, а значение `data_->refs` уменьшается до 0). Получаем утечку памяти, так как исходный `StringBuf` теперь не принадлежит ни `s1`, ни `s2` и не может быть удален. **Выход из** `s1.AboutToModify()`	

Теперь приступим к решению возникающих проблем.

Защита строк с отложенным копированием

При работе над последующим материалом следует не забывать, что наша цель не состоит в том, чтобы защитить `Optimized::String` от любого некорректного использования. В конце концов, код, использующий объект `String`, отвечает за использование объекта разными потоками и, соответственно, за обеспечение корректности параллельной работы потоков с объектом. Проблема в том, что поскольку совместное использование объекта скрыто, вызывающий код *не в состоянии* выполнять свои функции — ведь ему неизвестно, являются ли два объекта различными или они только кажутся таковыми, а в действительности представляют собой по сути один и тот же объект. Наша цель — обеспечить безопасность `String`, достаточную для того, чтобы код, использующий объекты `String`, мог положиться на то, что разные объекты в действительности различны и, таким образом, пользоваться обычными средствами обеспечения безопасности.

2. **Покажите, как сделать `Optimized::String` безопасным:**
   а) **в предположении атомарности операций получения, установки и сравнения целых чисел;**
   б) **без этого предположения.**

Я начну с п. *б* в связи с его большей обобщенностью. Все, что нам надо, — это устройство управления блокировками, типа взаимоисключений.[5] При его использовании решение задачи становится несложным — при работе нам просто надо блокировать доступ к счетчику ссылок.

[5] Если вы используете Win32, вместо взаимоисключений лучше использовать существенно более эффективные “критические разделы”, оставив взаимоисключения для тех случаев, когда их использование совершенно необходимо.

Перед тем как приступить к решению поставленной задачи, нам надо добавить член Mutex в Optimized::StringBuf.

```
namespace Optimized
{
    class StringBuf
    {
    public:
        StringBuf();              // Пустая строка
        ~StringBuf();             // Очистка буфера
        StringBuf( const StringBuf& other, size_t n = 0 );
                                  // Копирование буфера

        void Reserve( size_t n); // Обеспечение len >= n

        char*    buf;             // Выделенный буфер
        size_t   len;             // размер буфера
        size_t   used;            // Количество символов
        unsigned refs;            // Счетчик ссылок

        Mutex    m;       // Обеспечивает последовательную
                          // работу с данным объектом

    private:
        // Копирование не разрешено...
        StringBuf& operator=( const StringBuf& );
    };
```

Единственная функция, которая вынужденно работает с двумя объектами StringBuf одновременно, — это конструктор копирования. String вызывает конструктор копирования StringBuf только в двух местах — в собственном конструкторе копирования и в AboutToModify(). Заметим, что обеспечить последовательный доступ требуется только к счетчику ссылок, поскольку по определению String никак не изменяет совместно используемый буфер StringBuf.

Конструктор по умолчанию в блокировках не нуждается.

```
String::String() : data_( new StringBuf ) {}
```

Деструктору блокировка нужна на время опроса и обновления значения счетчика refs.

```
String::~String()
{
    bool bDelete = false;
    data_->m.Lock();     //-----------------------
    if ( data_->refs == Unshareable ||
         --data_->refs < 1 )
    {
        bDelete = true;
    }
    data_->m.Unlock();   //-----------------------
    if ( bDelete )
    {
        delete data_;
    }
}
```

В случае конструктора копирования String заметим, что мы можем считать, что никакие другие буфера во время этой операции не будут модифицироваться или перемещаться, поскольку за последовательность обращений к видимым объектам отвечает вызывающий код. Но мы должны как и ранее обеспечить последовательный доступ к счетчику ссылок.

```
String::String( const String& other )
{
    bool bSharedIt = false;
```

```
    other.data_->m.Lock();    //------------------
    if ( other.data_->refs != Unshareable )
    {
        bShareIt = true;
        data_ = other.data_;
        ++data_->refs;
    }
    other.data_->m.Unlock();  //------------------
    if ( !bShareIt )
    {
        data_ = new StringBuf( *other.data_ );
    }
}
```

Как видите, сделать конструктор копирования String безопасным не так уж трудно. Выполненная нами работа подводит нас к функции AboutToModify(), где решение проблем безопасности очень похоже. Заметим, однако, что этот код требует блокирования на время всей операции глубокого копирования (в действительности блокировка строго необходима только при получении значения refs и обновлении его в конце, но мы заблокируем доступ на все время глубокого копирования).

```
void String::AboutToModify(
    size_t n, bool markUnshareable /* = false */ )
{
    data_->m.Lock();         //-----------------------
    if ( data_->refs > 1 && data_->refs != Unshareable )
    {
        StringBuf* newdata = new StringBuf( *data_, n );
        --data_->refs;      // Вся работа сделана,
        data_->m.Unlock();  //---------------------
        data_ = newdata;    // передаем владение
    }
    else
    {
        data_->m.Unlock();  //---------------------
        data_->Reserve( n );
    }
    data_->refs = markUnshareable ? Unshareable : 1;
}
```

Ни одну другую функцию изменять не следует. Append() и operator[]() не нуждаются в блокировках, поскольку по завершении AboutToModify() гарантируется, что мы единолично используем свой буфер. Функция Length() корректно работает по определению — при работе с собственным буфером изменить его величину used просто некому, а при работе с совместно используемым буфером перед тем, как модифицировать буфер, другой поток получит его копию и, следовательно, изменить значение used, возвращаемое функцией Length(), он не в состоянии.

```
void String::Append( char c )
{
    AboutToModify( data_->used + 1 );
    data_->buf[ data_->used++ ] = c;
}

size_t String::Length() const
{
    return data_->used;
}

char& String::operator[]( size_t n )
{
    AboutToModify( data_->len, true );
    return data_->buf[n];
}
```

```
    const char String::operator[]( size_t n ) const
    {
        return data_->buf[n];
    }

}
```

Обратите внимание на интересную особенность: мы должны блокировать доступ других потоков только на время работы со счетчиком ссылок `refs`.

А теперь, вооруженные этим наблюдением и общим решением, ответим на первую часть вопроса —

а) в предположении атомарности операций получения, установки и сравнения целых чисел.

Некоторые операционные системы обеспечивают нас такого рода функциями.

Примечание: эти функции обычно значительно более эффективны, чем примитивы синхронизации общего назначения, типа взаимоисключений. Однако неверно было бы сказать, что мы можем использовать атомарные операции с целыми числами “вместо блокировок”.

Далее приведена безопасная реализация `String` в предположении наличия трех безопасных функций `IntAtomicGet()`, `IntAtomicDecrement()` и `IntAtomicIncrement()`. По сути это решение является таким же, как и предыдущее, но последовательность доступа к `refs` обеспечивается атомарными операциями с целыми числами.

```
namespace Optimized
{
    String::String() : data_( new StringBuf ) {}

    String::~String()
    {
        if ( IntAtomicGet( data_->refs ) == Unshareable ||
             InAtomicDecrement( --data_->refs ) < 1 )
        {
            delete data_;
        }
    }

    String::String( const String& other )
    {
        if ( IntAtomicGet( other.data_->refs ) !=
             Unshareable )
        {
            data_ = other.data_;
            IntAtomicIncrement( ++data_->refs );
        }
        else
        {
            data_ = new StringBuf( *other.data_ );
        }
    }

    void String::AboutToModify(
         size_t n,
         bool markUnshareable /* = false */ )
    {
        int refs = IntAtomicGet( data_->refs );
        if ( refs > 1 && refs != Unshareable )
        {
            StringBuf* newdata = new StringBuf( *data_, n );
            if ( IntAtomicDecrement( --data_->refs ) < 1 )
            // Два потока пытаются сделать это одновременно
            {
```

```
                delete newdata;
            }
            else
            {
                                        // Вся работа сделана,
                data_ = newdata;        // передаем владение
            }
        }
        else
        {
            data_->Reserve( n );
        }
        data_->refs = markUnshareable ? Unshareable : 1;
    }

    void String::Append( char c )
    {
        AboutToModify( data_->used + 1 );
        data_->buf[ data_->used++ ] = c;
    }

    size_t String::Length() const
    {
        return data_->used;
    }

    char& String::operator[]( size_t n )
    {
        AboutToModify( data_->len, true );
        return data_->buf[n];
    }

    const char String::operator[]( size_t n ) const
    {
        return data_->buf[n];
    }
}
```

3. Обсудите, как все это повлияет на производительность.

При отсутствии атомарных целочисленных операций копирование при записи обычно приводит к значительной потере производительности. Даже при наличии атомарных операций копирование при записи делает операции со строкой примерно на 50% длиннее даже в однопоточных программах.

В целом отложенная запись — не самая лучшая идея для многопоточной среды. Это связано с тем, что вызывающий код не знает, не используют ли скрыто два различных объекта одно и то же представление, так что класс `String` вынужден предпринимать дополнительные усилия по обеспечению последовательного доступа. В зависимости от наличия более эффективных способов обеспечения последовательного доступа (например, атомарных целочисленных операций) влияние на производительность в многопоточной среде колеблется от умеренного до сильного.

Некоторые эмпирические результаты

Для наглядности я протестировал шесть основных вариантов реализации класса `String`.

Название[6]	Описание (усовершенствованные версии ранее приведенного кода)

[6] Аббревиатура COW означает “copy on write” — копирование при записи. — *Прим. перев.*

`Plain`	Строки, не использующие копирования при записи
`COW_Unsafe`	`Plain` + COW, небезопасная версия
`COW_AtomicInt`	`Plain` + COW, безопасная версия
`COW_AtomicInt2`	`COW_Atomic_Int` + `StringBuf` в том же буфере, что и данные
`COW_CritSec`	`Plain` + COW, безопасная версия (критические разделы Win32)
`COW_Mutex`	`Plain` + COW, безопасная версия (взаимоисключения Win32) (`COW_CritSec` с заменой критических разделов взаимоисключениями Win32)

Я добавил также седьмую версию для измерения результата применения оптимизированного распределения памяти вместо оптимизирующего копирования.

`Plain_FastAlloc`	Plain + оптимизированное распределение памяти

Основное внимание я уделил сравнению версий `Plain` и `COW_AtomicInt`, последняя из которых в общем случае является наиболее эффективной безопасной реализацией COW. Были получены следующие результаты.

1. Для всех модифицирующих операций `COW_AtomicInt` всегда работает хуже, чем `Plain`. Это естественный и ожидаемый результат.
2. COW должен давать эффект при наличии большого количества неизменяемых копий. Но для средней длины строки в 50 символов характерно следующее.
   а) При неизменности 33% копий и однократном изменении остальных `COW_AtomicInt` работает медленнее `Plain`.
   б) При неизменности 50% копий и трехкратном изменении остальных `COW_AtomicInt` работает медленнее `Plain`.

   Второй результат для многих может показаться удивительным — то, что `COW_AtomicInt` работает медленнее, когда в системе в целом операций копирования больше, чем операций изменения. Заметим, что в обоих случаях традиционное небезопасное копирование при записи работает более эффективно, чем `Plain`. Это говорит о том, что метод копирования при записи может служить в качестве оптимизации в чисто однопоточной среде, но гораздо реже применим в многопоточной среде.
3. Утверждение, что основное преимущество отложенного копирования — возможность избежать излишнего распределения памяти, не более чем миф. Основное преимущество метода для длинных строк — возможность избежать излишнего копирования символов строки.
4. Оптимизированное распределение (не отложенное копирование!) во всех случаях представляет собой оптимизацию скорости работы (но влечет за собой дополнительный расход памяти).

Вот, вероятно, наиболее важный вывод из проведенных эмпирических исследований.

> *Большая часть преимуществ отложенного копирования для небольших строк может быть достигнута без применения технологии копирования при записи, путем использования более эффективного распределения памяти. (Само собой, вы можете применять более эффективное распределение памяти наряду с отложенным копированием.)*

И два последних вопроса.

Во-первых, почему я тестировал такой неэффективный метод, как `COW_CritSec`? Дело в том, что как минимум одна популярная коммерческая реализация `basic_string` использовала этот метод до 2000 года (наверняка использует его и по сей день, я просто не видел

более свежих исходных текстов), несмотря на все его недостатки. Проверяйте, что и как именно делается в используемых вами библиотеках, поскольку если библиотека разработана для возможного многопоточного использования, вы будете платить за это эффективностью ваших программ все время, даже если эти программы будут однопоточными.

Во-вторых, что можно сказать о `COW_AtomicInt2`? Это тот же метод `COW_AtomicInt`, только вместо выделения `StringBuf` с последующим отдельным выделением буфера данных в этом случае и `StringBuf`, и данные располагаются в одном выделенном блоке памяти. Все прочие варианты `COW_*` используют для `StringBuf` быстрое распределение памяти, и цель `COW_AtomicInt2` — продемонстрировать, что я рассмотрел и эту возможность. В действительности из-за более сложной логики `COW_AtomicInt2` оказывается немного медленнее `COW_AtomicInt`.

Кроме того, я тестировал относительную производительность различных целых операций (инкремент `int`, инкремент `volatile int`, а также инкремент `int` с использованием атомарных целочисленных операций Win32), чтобы убедиться, что результаты `COW_AtomicInt` не были искажены из-за плохой реализации или накладных расходов на вызовы функций.

Резюме

Вот несколько выводов из приведенного материала.

1. Это всего лишь небольшой тест, не претендующий на полноту и истину в последней инстанции. Любые исправления и предложения только приветствуются. Я просто привел некоторые показавшиеся мне интересными результаты, не разбираясь в скомпилированном машинном коде и не делая попыток определить влияние кэширования или тому подобных эффектов.
2. Даже при использовании атомарных целочисленных операций нельзя говорить, что нам не нужны блокировки. Упомянутые операции обеспечивают последовательность доступа и приводят к наличию накладных расходов.
3. Все выполняемые тесты — однопоточные. Накладные расходы, связанные с обеспечением безопасности отложенного копирования, проявляются в однопоточных программах так же, как и в многопоточных. Как выяснилось, копирование при записи может быть оптимизирующей технологией только в случаях, когда нам не надо беспокоиться о вопросах многопоточной безопасности; в противном случае снижение производительности за счет безопасности слишком велико. Более того, в [Murray93] показано, что эта технология даже без учета безопасности потоков дает преимущества только в отдельных случаях.

8

Свободные функции и макросы

Хотя другие разделы этой книги посвящены обобщенному и объектно-ориентированному программированию и их поддержке в C++, нам стоит поговорить и о добрых старых свободных функциях и даже макросах, унаследованных (воспользуемся этим объектно-ориентированным термином) от языка программирования C.

Вы работали раньше с языком программирования, поддерживающим вложенные функции? Вы удивитесь, но в C++ можно добиться того же эффекта, — о том, как это сделать, вы узнаете из задачи 8.2. Вас интересует, всегда ли использование макросов препроцессора — зло, которого следует избегать, или у них тоже есть свое место под солнцем C++? Обратитесь за ответом на этот и другие вопросы к задачам 8.3 и 8.4.

Но сначала обратимся к очень интересному вопросу (который вовсе не теоретическая абстракция и встречается в реальных задачах): как нам написать функцию, которая возвращает указатель на саму себя?

Задача 8.1. Рекурсивные объявления ***Сложность: 6***

Можете ли вы написать функцию, возвращающую указатель на саму себя? Если да, то для чего она может вам пригодиться?

1. Что такое указатель на функцию? Каким образом его можно использовать?
2. Предположим, что можно написать функцию, которая возвращает указатель на саму себя. Такая функция может одинаково хорошо возвращать указатель на любую функцию с той же сигнатурой, что и у нее самой. Когда такая возможность может оказаться полезной? Поясните свой ответ.
3. Можно ли написать функцию, которая возвращает указатель на саму себя? Она должна использоваться следующим естественным путем.

```
// Пример 3
//
// FuncPtr представляет собой синоним указателя
// на функцию с той же сигнатурой, что и у f()
//
FuncPtr p = f();    // Выполняется f()
(*p)();             // Выполняется f()
```

Если это возможно, продемонстрируйте, как. Если нет, объясните, почему.

Решение

Указатели на функции

1. Что такое указатель на функцию? Каким образом его можно использовать?

Так же как указатель на объект позволяет вам динамически указывать на объект данного типа, указатель на функцию позволяет вам динамически указывать на функцию с данной сигнатурой. Например:

```
// Пример 1
//

// Создаем typedef с именем FPDoubleInt для
// функции с параметром double, возвращающим int
//
typedef int (*FPDoubleInt)( double );

// Используем его
//
int f( double )  { /* ... */ }
int g( double )  { /* ... */ }
int h( double )  { /* ... */ }

FPDoubleInt fp;
fp = f;
fp( 1.1 );   // Вызов f()
fp = g;
fp( 2.2 );   // Вызов g()
fp = h;
fp( 3.14 );  // Вызов h()
```

При правильном применении указатели на функции обеспечивают высокую гибкость времени выполнения. Например, стандартная функция `qsort()` получает указатель на функцию сравнения с заданной сигнатурой, что позволяет вызывающему коду расширить возможности `qsort()` путем использования пользовательских функций сравнения.

Знакомство с конечными автоматами

2. Предположим, что можно написать функцию, которая возвращает указатель на саму себя. Такая функция может одинаково хорошо возвращать указатель на любую функцию с той же сигнатурой, что и у нее самой. Когда такая возможность может оказаться полезной? Поясните свой ответ.

На ум приходит множество вариантов, но наиболее распространенным, пожалуй, является реализация конечного автомата.

Вкратце конечный автомат состоит из множества возможных состояний и множества допустимых переходов между этими состояниями. Например, на рис. 8.1 показан пример простого конечного автомата.

Когда мы находимся в состоянии `Start`, при получении ввода “`a`” мы переходим в состояние `S2`, а при получении ввода “`be`” — в состояние `Stop`. Находясь в состоянии `S2`, мы переходим в состояние `Stop` при получении ввода “`see`”; любой другой ввод в этом состоянии некорректен. Данный конечный автомат имеет только два корректных входных потока: “`be`” и “`asee`”.

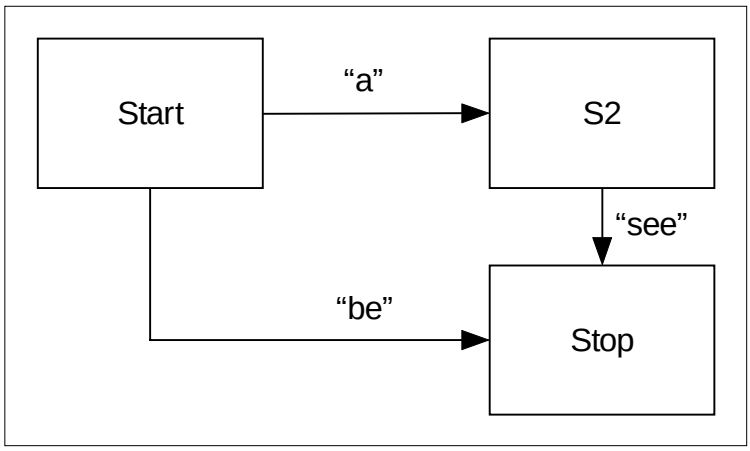


Рис. 8.1. Простой конечный автомат

Для реализации конечного автомата иногда достаточно сделать каждое состояние функцией. Все функции состояний имеют одну и ту же сигнатуру, и каждая из них возвращает указатель на функцию (состояние), вызываемую следующей. Вот предельно упрощенный код, иллюстрирующий эту идею.

```
// Пример 2
//
StatePtr Start( const string& input );
StatePtr S2   ( const string& input );
StatePtr Stop ( const string& input );
StatePtr Error( const string& input ); // Состояние ошибки

StatePtr Start( const string& input )
{
    if ( input == "a" )
    {
        return S2;
    }
    else if ( input == "be" )
    {
        return Stop;
    }
    else
    {
        return Error;
    }
}
```

О конечных автоматах и их применении имеется немало литературы, так что при желании получить дополнительную информацию по этой теме совсем нетрудно; что же касается нашей книги, то конечные автоматы выходят за рамки ее тематики, а потому более подробно не рассматриваются.

В приведенном выше коде не сказано, что такое `StatePtr`, и вовсе не очевидно, каким образом его можно определить. По сути, это и есть наш последний вопрос в этой задаче.

Как функции вернуть указатель на себя?

3. Можно ли написать функцию, которая возвращает указатель на саму себя? Она должна использоваться следующим естественным путем.

```
// Пример 3
//
// FuncPtr представляет собой синоним указателя
// на функцию с той же сигнатурой, что и у f()
//
FuncPtr p = f();    // Выполняется f()
(*p)();             // Выполняется f()
```

Если это возможно, продемонстрируйте, как. Если нет, объясните, почему.

Да, возможно, но это не так просто.

Например, вы могли бы попытаться использовать инструкцию `typedef` непосредственно — примерно так.

```
// Пример 3а. Наивная (и неверная) попытка
//
typedef FuncPtr (*FuncPtr); // Ошибка
```

Увы, такое рекурсивное определение посредством `typedef` не разрешено. Другие программисты пытаются обойти это путем использования приведения типа к `void*`.

```
// Пример 3б. Нестандартно и непереносимо
//
typedef void* (*FuncPtr)();

void* f() { return (void*)f; } // Приведение к void*

FuncPtr p = (FuncPtr)(f());    // Приведение к FuncPtr
p();
```

Этот код не является решением, поскольку не удовлетворяет поставленным условиям. И, что еще хуже, он небезопасен, поскольку преднамеренно нарушает корректность работы системы типов. Кроме того, он заставляет всех пользователей заниматься постоянным приведением типов. Ну, и самое неприятное — пример 3*б* не поддерживается стандартом. Хотя и гарантируется, что `void*` имеет размер, достаточный для хранения указателя на любой объект, в отношении указателя на функцию такой гарантии в стандарте нет. На некоторых платформах указатель на функцию имеет больший размер, чем указатель на объект. Таким образом, если даже код наподобие приведенного в примере 3*б* компилируется вашим текущим компилятором, это не гарантирует, что он будет работать с другим компилятором или даже с новой версией текущего.

Конечно, обойти это отклонение от стандарта можно путем приведения к другому типу указателя на функцию вместо приведения к `void*`.

```
// Пример 3в. Стандартно и переносимо,
//            но все равно это хак...
//
typedef void (*VoidFuncPtr)();
typedef VoidFuncPtr (*FuncPtr)();

VoidFuncPtr f() { return (VoidFuncPtr)f; }
                  // Приведение к VoidFuncPtr

FuncPtr p = (FuncPtr)f();
                  // Приведение VoidFuncPtr
p();
```

Технически этот код корректен, но имеет все те же недостатки, что и код из примера *3б* (за исключением одного — нестандартности). Он разрушает систему типов, заставляет пользователя постоянно использовать приведение типов и, само собой, нарушает поставленные в задаче условия. Таким образом, мы не можем считать этот код решением.

Неужели мы не придумаем ничего лучше?

Корректное и переносимое решение

К счастью, наша задача имеет корректное и переносимое решение. Как это часто происходит, нам поможет дополнительный уровень косвенности в виде прокси-класса, который получает требующийся нам тип указателя в качестве параметра конструктора и содержит оператор преобразования к данному типу.

```
// Пример 3г. Корректное решение
//
class FuncPtr_;
typedef FuncPtr_ (*FuncPtr)();

class FuncPtr_
{
public:
    FuncPtr_( FuncPtr p ) : p_( p ) {}
    operator FuncPtr() { return p_; }
private:
    FuncPtr p_;
};
```

Теперь мы можем естественным образом объявить, определить и использовать `f()`.

```
FuncPtr_ f() { return f; } // Естественный синтаксис

int main()
{
    FuncPtr p = f();       // Естественный синтаксис
    p();
}
```

У этого решения три основных преимущества.

1. Это корректное решение поставленной задачи. Кроме того, оно безопасно в смысле работы с типами и переносимо.
2. Его механизм прозрачен для программиста. Решение позволяет использовать естественный синтаксис вызова функции и возврата из нее.
3. Это решение, вероятно, не приводит к накладным расходам. В современных компиляторах прокси-класс должен быть встроенным и полностью оптимизироваться.

Эпилог

Обычно прокси-классы специального назначения так и хочется обобщить в шаблон общего назначения. Увы, непосредственно использовать в данном случае шаблон невозможно, поскольку при этом инструкция `typedef` должна выглядеть примерно следующим образом:

```
typedef Holder<FuncPtr> (*FuncPtr)();
```

что недопустимо.

Задача 8.2. Имитация вложенных функций ***Сложность: 5***

В C++ есть вложенные классы, но нет вложенных функций. В каких случаях вложенные функции могут оказаться полезными и как сымитировать их в C++?

1. Что такое вложенный класс? Почему он может быть полезен?
2. Что такое локальный класс? Почему он может быть полезен?
3. C++ не поддерживает вложенные функции, так что мы не можем написать код наподобие следующего.

```
// Пример 3
//
int f( int i )
{
    int j = i * 2;

    int g( int k ) // Не разрешено в C++
    {
        return j + k;
    }

    j += 4;

    return g( 3 );
}
```

Продемонстрируйте, каким образом можно добиться того же эффекта в стандартном C++, и покажите, как обобщить полученное решение.

Решение

Вложенные и локальные классы

C++ предоставляет множество полезных инструментов для сокрытия информации и управления зависимостями. Мы поговорим о вложенных и локальных классах, так что не будем уделять чрезмерное внимание синтаксису и семантике, а вместо этого рассмотрим, каким образом эти возможности могут использоваться для написания надежного и легко сопровождаемого кода.

1. Что такое вложенный класс? Почему он может быть полезен?

Вложенный класс — это класс, расположенный в области видимости другого класса, например:

```
// Пример 1. Вложенный класс
//
class OuterClass
{
    /* public, protected или private: */
    class NestedClass
    {
        // ...
    };
    // ...
};
```

Вложенные классы применяются для организации кода и управления доступом и зависимостями. Вложенные классы так же подчиняются правилам доступа, как и прочие части класса. Так, если в примере 1 `NestedClass` объявлен как `public`, то любой внешний код

может обратиться к нему как к `OuterClass::NestedClass`. Часто вложенные классы содержат детали реализации, и тогда они должны быть объявлены как `private`. Если в примере 1 `NestedClass` объявлен как `private`, то использовать его могут только члены и друзья `OuterClass`.

Заметим, что вы не можете достичь того же результата с помощью одних лишь пространств имен, поскольку пространства имен только группируют имена по различным областям и не предоставляют возможности управления доступом, обеспечиваемой классами. Так что если вам необходимо обеспечить управление доступом к классу, один из способов сделать это — вложить его в другой класс.

2. Что такое локальный класс? Почему он может быть полезен?

Локальный класс — это класс, определенный в области видимости функции (любой функции, как свободной, так и функции-члена), например:

```
// Пример 2. Локальный класс
//
int f()
{
    class LocalClass
    {
        // ...
    };
    // ...
};
```

Аналогично вложенным классам, локальные классы могут пригодиться для управления зависимостями кода. В примере 2 о локальном классе `LocalClass` знает только код из функции `f()`, и только он может использовать данный класс. Это может быть полезно, когда, например, `LocalClass` представляет собой детали внутренней реализации `f()`, которые не должны быть известны другому коду.

Локальный класс можно использовать в большинстве ситуаций, где можно использовать нелокальный класс, однако имеется одно важное ограничение, о котором не следует забывать. Локальный или неименованный классы нельзя использовать в качестве параметров шаблонов. Вот что говорится в стандарте C++ [C++98], раздел 14.3.9.

> *Локальный тип, тип без связи, неименованный тип или тип, составленный из перечисленных, не могут использоваться в качестве параметра шаблона. [Пример:*
>
> ```
> template<class T>
> class X { /* ... */ }
> void f()
> {
>  struct S { /* ... */ };
>  X<S> x3;  // Ошибка: локальный тип использован
>  // в качестве аргумента шаблона
>  X<S*> x4; // Ошибка: указатель на локальный тип
>  // использован в качестве аргумента шаблона
> }
> ```
>
> *-- конец примера].*

И вложенные, и локальные классы представляют собой один из множества инструментов C++, предназначенных для сокрытия информации и управления зависимостями.

Вложенные функции

Некоторые языки программирования (но не C++) позволяют использовать вложенные функции, которые имеют много схожего с вложенными классами. Вложенная функция определяется внутри другой (объемлющей) функции и обладает следующими свойствами.

- Вложенная функция имеет доступ к переменным охватывающей функции.
- Вложенная функция локальна по отношению к охватывающей функции, т.е. она не может быть вызвана из других мест, если только охватывающая функция не возвращает вам указатель на вложенную функцию.

Так же как вложенные классы могут применяться для управления видимостью классов, так и вложенные функции позволяют управлять видимостью функций.

C++ не поддерживает вложенность функций. Но нельзя ли каким-либо образом получить тот же эффект в стандартном C++?

3. C++ не поддерживает вложенные функции, так что мы не можем написать код наподобие следующего.

```
// Пример 3
//
int f( int i )
{
    int j = i * 2;

    int g( int k ) // Не разрешено в C++
    {
        return j + k;
    }

    j += 4;

    return g( 3 );
}
```

Продемонстрируйте, каким образом можно добиться того же эффекта в стандартном C++, и покажите, как обобщить полученное решение.

Да, это возможно при небольшой реорганизации кода и минимальных ограничениях. Основная идея состоит в превращении функции в объект-функцию. Рассмотрение этого решения послужит также прекрасной иллюстрацией мощи объектов-функций.

Первая попытка

Решение задачи, поставленной в третьем вопросе, большинство программистов начинают с написания примерно следующего кода.

```
// Пример 3a. Наивная попытка использования
//            "локального объекта-функции"
//
int f( int i )
{
    int j = i*2;

    class g_
    {
    public:
        int operator()( int k )
        {
            return j+k; // Ошибка: переменная j недоступна
        }
    } g;

    j += 4;
    return g( 3 );
}
```

В примере 3*а* идея состоит в “оборачивании” функции в локальный класс и вызове ее посредством объекта-функции.

Идея неплохая, но не работающая по одной простой причине: объект локального класса не имеет доступа к переменным охватывающей функции.

“Ну хорошо, — скажете вы, — но почему бы нам не использовать указатели или ссылки локального класса для обращения к переменным функции?” Обычно именно такой подход и используется при второй попытке решения поставленной задачи.

```
// Пример 3б. Наивная попытка использования "локального
//            объекта-функции" и ссылок на переменные
//            (сложно и ненадежно)
//
int f( int i )
{
    int j = i*2;

    class g_
    {
    public:
        g_( int& j ) : j_( j ) {}

        int operator()( int k )
        {
            return j_+k; // Обращение к j по ссылке
        }
    private:
        int& j_;
    } g( j );

    j += 4;
    return g( 3 );
}
```

Ну что ж, я должен принять такое “решение”, но с большой натяжкой. Это решение ненадежно, трудно расширяемо и вполне обоснованно может рассматриваться как “хак”. Например, рассмотрим, что требует от нас любое добавление новой переменной.

а) Добавить переменную.

б) Добавить соответствующую закрытую ссылку в `g_`.

в) Добавить соответствующий параметр в конструктор `g_`.

г) Добавить соответствующую инициализацию в `g_::g_()`.

Не так уж легко поддерживать такой код, в особенности при наличии нескольких локальных функций... Не можете ли вы предложить решение получше?

Улучшенное решение

А что если разместить переменные в самом локальном классе?

```
// Пример 3в. Усовершенствованное решение
//
int f( int i )
{
    class g_
    {
    public:
        int j;

        int operator()( int k )
        {
            return j+k;
```

```
        }
    } g;

    g.j = i*2;
    g.j += 4;
    return g( 3 );
}
```

Это существенный шаг вперед. Теперь классу g_ не требуются члены данных для работы с локальными переменными; не нужен и конструктор; да и сам код выглядит намного естественнее. Заметим также, что этот способ расширяем для любого количества локальных функций, так что попробуем написать код с тремя локальными функциями x(), y() и z() и, раз уж мы занялись переписыванием кода, почему бы нам не переименовать класс g_ с тем, чтобы его имя несло бо́льшую информацию о его предназначении?

```
// Пример 3г. Почти окончательное решение
//
int f( int i )
{
    // Определяем локальный класс-"обертку"
    // вокруг локальных данных и функций
    class local_
    {
    public:
        int j;

        // Локальные функции
        int g( int k )
        {
            return j+k;
        }
        void x() { /* ... */ }
        void y() { /* ... */ }
        void z() { /* ... */ }
    } local;

    local.j = i*2;
    local.j += 4;

    local.x();
    local.y();
    local.z();

    return local.g( 3 );
}
```

Тем не менее данное решение не идеально, так как мы должны инициализировать j с использованием способа, отличного от инициализации по умолчанию. В первоначальном варианте мы инициализировали j значением i*2; теперь же нам надо создать j, а затем присвоить ему значение, что не совсем одно и то же и может привести к трудностям при использовании более сложных типов.

Окончательное решение

Если мы не намерены, например, получать указатель на функцию f, то нам не надо, чтобы это была истинная функция, а значит, мы можем сделать следующий шаг и сделать объектом-функцией саму f и одним махом решить все проблемы.

```
// Пример 3д. Окончательное решение
//
class f
{
```

```
    int retval;   // Возвращаемое значение
    int j;
    int g( int k ) { return j + k; }
    void x() { /* ... */ }
    void y() { /* ... */ }
    void z() { /* ... */ }

public:
    f( int i )    // Исходная функция - теперь конструктор
        : j( i*2 )
    {
        j += 4;
        x();
        y();
        z();
        retval = g( 3 );
    }

    operator int() const //Возврат результата функции
    {
        return retval;
    }
};
```

Приведенный код для краткости показан как встроенный, но все закрытые члены могут быть скрыты посредством идиомы закрытой реализации, позволяя полностью отделить интерфейс от реализации.

Заметим, что этот подход легко распространим и на функции-члены. Предположим, например, что `f()` — функция-член, и мы хотим использовать локальную функцию `g()` в `f()` следующим образом.

```
// Пример 4. Это не корректный код C++, но он
//           иллюстрирует нашу задачу: локальная
//           функция внутри функции-члена
//
class C
{
    int data_;

public:
    int f( int i )
    {
        // Гипотетическая вложенная функция
        int g( int i ) { return data_ + i; }

        return g( data_ + i*2 );
    }
};
```

Решить поставленную задачу можно путем преобразования `f()` в класс, как показано в примере 3*д*, с тем отличием, что в примере 3*д* класс находился в глобальной области видимости, а в нашем случае это — вложенный класс с доступом посредством вспомогательной функции.

```
// Пример 4а. Корректное решение для случая
//            функции-члена
//
class C
{
    int data_;
    friend class C_f;
public:
    int f( int i );
};

class C_f
```

```
{
    C* self;
    int retval;
    int g( int i ) { return self->data_ + i; }

public:
    C_f( C* c, int i ) : self( c )
    {
        retval = g( self->data_ + i*2 );
    }

    operator int() const { return retval; }
};

int C::f( int i ) { return C_f( this, i ); }
```

Резюме

Подход, проиллюстрированный в примерах 3*д* и 4*а*, эмулирует большинство возможностей локальных функций и легко расширяем до любого количества локальных функций. Основной недостаток данного подхода состоит в том, что все переменные должны быть определены в начале “функции”, так что мы не можем объявлять переменные поближе к месту их первого использования. Конечно, это более трудный способ, чем написание обычной локальной функции, но ведь он представляет собой обходной путь для получения возможностей, не имеющихся в языке программирования. В конечном счете этот способ не так уж плох и демонстрирует преимущества объектов-функций над обычными функциями.

Цель данной задачи — не только получить локальные функции в C++. Как обычно, цель состоит в том, чтобы определить конкретную проблему проектирования, рассмотреть различные пути ее решения и выбрать из них наиболее подходящее. Попутно мы поэкспериментировали с различными возможностями C++ и еще раз убедились в полезности объектов-функций.

При проектировании ваших программ старайтесь находить наиболее простые и элегантные решения. Некоторые из приведенных здесь решений вполне работоспособны, но они не должны появляться в профессиональном коде. Для этого они слишком сложны, трудны для понимания и, соответственно, их тяжело сопровождать.

Рекомендация

Предпочитайте ясность. Избегайте сложных решений. Остерегайтесь запутанности.

Более простое проектирование дает более простой и легче тестируемый код. Избегайте сложных решений, которые почти всегда чреваты ненадежностью, трудностями в понимании и сопровождении. Но, стараясь избегать чрезмерной сложности, тем не менее, не забывайте, что очень часто незначительное время, потраченное на раздумья и кодирование сейчас, при проектировании, может сэкономить массу времени впоследствии. Учитесь находить компромиссы!

Задача 8.3. Макросы препроцессора — *Сложность: 4*

Макросы препроцессора представляют собой часть C++, унаследованную от C. В данной задаче рассматриваются ситуации, когда макросы применимы в современном C++.

Почему программисты на C++, несмотря на наличие таких мощных возможностей языка, как перегрузка и шаблоны, все равно используют директивы препроцессора типа “`#define`”?

Решение

Возможности C++ зачастую (но не всегда!) позволяют избежать использования директивы `#define`. Например, предпочтительнее использовать “`const int c = 42`” вместо “`#define c 42`”, поскольку, кроме прочего, первое объявление обеспечивает безопасность типов и позволяет избежать редактирования кода препроцессором.

Тем не менее имеется несколько основательных причин для использования директивы `#define`.

1. Защита заголовков

Это распространенный способ защиты от многократного включения заголовочного файла.

```
#ifndef MYPROG_X_H
#define MYPROG_X_H

// Содержимое заголовочного файла x.h

#endif
```

2. Доступ к возможностям препроцессора

Очень часто в код диагностики желательно внести такую информацию, как номера строк или время компиляции. Простейший способ сделать это — использовать предопределенные стандартные макросы, такие как `__FILE__`, `__LINE__`, `__DATE__` и `__TIME__`. По тем же и другим причинам часто удобно использовать операторы препроцессора `#` и `##` для получения строк и объединения токенов.

3. Выбор кода времени компиляции

Хотя я вовсе не фанат препроцессорной магии, имеются вещи, которые невозможно выразить так же хорошо (если можно вообще), как с использованием директив препроцессора.

а) Отладочный код

Иногда желательно построить программу с рядом дополнительных фрагментов кода (обычно это отладочный код), а потом этот отладочный код становится не нужен.

```
void f()
{
#ifdef MY_DEBUG
    cerr << "Вывод отладочного сообщения" << endl;
#endif
    // Остальной код f()
}
```

Взгляните внимательнее на приведенный код. По сути здесь содержатся две различные программы. Когда мы компилируем код при определенном макросе `MY_DEBUG`, мы получаем код, в котором имеется вывод в поток `cerr`, и компилятор проверяет синтаксис и семантику соответствующей строки. При компиляции с неопределенным макросом `MY_DEBUG` мы получим другую программу, в которой нет вывода в поток `cerr`, и компилятор не проверяет корректность исключенного кода.

Обычно в таких ситуациях вместо `#define` лучше использовать условное выражение.

```
void f()
{
    if ( MY_DEBUG )
    {
        cerr << "Вывод отладочного сообщения" << endl;
    }
    // Остальной код f()
}
```

Таким образом мы получаем только одну компилируемую и тестируемую программу, в которой компилятор может проверить весь код; что же касается кода, недостижимого при MY_DEBUG равном false, то компилятор легко удалит его из собранной программы при оптимизации.

б) Код, зависящий от платформы

Обычно работать с кодом, зависящим от платформы, лучше с использованием шаблона проектирования Factory, поскольку он обеспечивает лучшую организацию кода и гибкость времени выполнения. Иногда, однако, отличий не так уж много, и использование препроцессора для выбора того или иного кода вполне оправданно.

в) Различные представления данных

Типичным примером может служить модуль, который определяет список кодов ошибок, которые для внешнего пользователя видны как обычное перечисление enum с комментариями, а внутри модуля должны храниться в виде ассоциативного множества для упрощения поиска.

```
// Для внешнего пользователя
//
enum Error
{
    ERR_OK = 0,             // Ошибки нет
    ERR_INVALID_PARAM = 1,  // <описание>
    ...
};

// Для внутреннего использования
//
map<Error, const char*> lookup;
lookup.insert( make_pair( ERR_OK, (const char*)"Ошибки нет") );
lookup.insert( make_pair( ERR_INVALID_PARAM, (const char*)"<описание>") );
...
```

Хотелось бы иметь оба представления, обойдясь при этом единственным определением пар “код ошибки/сообщение”. Используя макросы, мы можем записать список ошибок следующим образом, создавая необходимую структуру во время компиляции.

```
ERR_ENTRY( ERR_OK,            0, "Ошибки нет" ),
ERR_ENTRY( ERR_INVALID_PARAM, 1, "<описание>" ),
...
```

Реализация ERR_ENTRY и связанных с ним макросов остаются читателю в качестве дополнительного задания.

Здесь приведены три типичных примера использования препроцессора, но на самом деле их значительно больше. Достаточно сказать, что, хотя использования препроцессора C и следует по возможности избегать, все же иногда при разумном его использовании можно сделать код C++ и проще, и безопаснее.

Рекомендация

Избегайте использования макросов препроцессора, за исключением следующих ситуаций.

- *Защита директивы `#include`.*
- *Условная компиляция для обеспечения переносимости или отладки в `.cpp`-файлах (но не в `.h`-файлах!).*
- *Использование директив `#pragma` для отключения предупреждений, о которых точно известно, что их можно игнорировать. Такие директивы `#pragma` должны находиться только внутри директив условной компиляции, обеспечивающих переносимость, с тем, чтобы избежать возможных предупреждений компилятора о наличии неизвестной директивы.*

Для того чтобы узнать, что думает о препроцессоре автор C++, обратитесь к разделу 18 книги [Stroustrup94].

Задача 8.4. `#Definition` ***Сложность: 4***

Что могут макросы и чего они не могут?

1. Покажите, как написать простой макрос препроцессора `max()`, который принимает два аргумента и находит больший из них с использованием обычного оператора сравнения <. Какие типичные ловушки встречаются при написании такого макроса?
2. Чего не может создать макрос препроцессора и почему?

Решение

Типичные ошибки при работе с макросами

1. **Покажите, как написать простой макрос препроцессора `max()`, который принимает два аргумента и находит больший из них с использованием обычного оператора сравнения <. Какие типичные ловушки встречаются при написании такого макроса?**

При применении макросов в дополнение к некоторым недостаткам имеются четыре основные ловушки. Сначала мы поговорим именно о них.

1. Не забывайте о скобках вокруг аргументов

```
// Пример 1a.
//
#define max(a,b) a < b ? b : a
```

Здесь проблема заключается в том, что параметры макроса используются без скобок. Макросы выполняют буквальное замещение, так что отсутствие скобок может привести к неожиданным результатам. Например,

```
max( i += 3, j )
```

превращается в

```
i+= 3 < j ? j : i += 3
```

что с учетом приоритетов операторов и правил языка для компилятора означает

```
i+= ((3 < j) ? j : i += 3)
```

а для разработчика это означает долгую работу с отладчиком...

2. Не забывайте о скобках вокруг всего выражения

Исправив первую ошибку, мы тут же получим вторую.

```
//Пример 1б.
//
#define max(a,b) (a) < (b) ? (b) : (a)
```

Теперь проблема заключается в том, что в скобки не заключено все выражение. Следовательно, мы вновь можем получить странные результаты. Например,

```
k = max( i, j ) + 42;
```

раскрывается в

```
k = (i) < (j) ? (j) : (i) + 42;
```

что в соответствии с приоритетами операторов означает

```
k = (((i) < (j)) ? (j) : ((i) + 42));
```

Если `i >= j`, `k` присваивается значение `i+42`, как и предполагалось. Но если `i < j`, то `k` получает значение `j`.

Исправить ситуацию можно с помощью скобок, в которые следует заключить выражение целиком. Но и это не спасает нас от следующей проблемы.

3. Многократное вычисление аргумента

Рассмотрим, что произойдет, если одно из выражений имеет побочное действие.

```
// Пример 1в.
//
#define max(a,b) ((a) < (b) ? (b) : (a))

max( ++i, j )
```

В случае, когда `++i >= j`, значение `i` увеличивается дважды (и вряд ли это тот результат, которого добивался программист).

```
((++i) < (j) ? (j) : (++i))
```

Аналогично, `max( f(), pi )` раскрывается до

```
((f()) < (pi) ? (pi) : (f()))
```

так что если `f() >= pi`, то `f()` выполняется дважды, что как минимум неэффективно и почти наверняка неверно.

При том, что первые две проблемы мы смогли решить, проблему многократного вычисления аргументов в рамках макросов решить невозможно.

4. Конфликт имен

И наконец, макросы не подозревают об областях видимости (вообще говоря, они не подозревают ни о чем — см. `http://www.gotw.ca/gotw/063.htm`). Они выполняют буквальную подстановку независимо от того, где именно находится замещаемый текст. Это означает, что если мы используем макросы, то должны быть очень внимательны при назначении им имен. В частности, наибольшая проблема при использовании макроса `max()` состоит в весьма вероятном столкновении имен со стандартной шаблонной функцией `max()`.

```
// Пример 1г.
//
#define max(a,b) ((a) < (b) ? (b) : (a))

#include <algorithm>
```

Проблема заключается в том, что внутри заголовочного файла `algorithm` имеется примерно следующее определение.

```
template<typename T> const T&
max(const T& a, const T& b);
```

Увы, препроцессор услужливо превратит это объявление в нечто несуразное.

```
template<typename T> const T&
((const T& a) < (const T& b) ? (const T& b) : (const T& a));
```

Если вы думаете, что легко избавитесь от этих неприятностей, располагая определения макросов после всех директив `#include` (что неплохо делать в любом случае), просто представьте себе, что произойдет с вашим кодом, если в нем найдутся переменные (или что-то другое) с именем макроса (в нашем случае — `max`).

Если вы пишете макрос, попытайтесь дать ему необычное имя, которое вряд ли совпадет с другими используемыми вами именами.

Прочие неприятности

Вот несколько важных вещей, на которые неспособны макросы.

5. Макрос не может быть рекурсивным

Мы можем написать рекурсивную функцию, но невозможно написать рекурсивный макрос. Стандарт ([C++98], раздел 16.3.4.2) по этому поводу гласит следующее.

> *Если в процессе сканирования текущего списка замещений (не включая остальной части файла, обрабатываемого препроцессором) обнаруживается имя замещаемого макроса, оно не замещается. Кроме того, если в любых вложенных замещениях встречается имя замещаемого макроса, оно также не замещается. Данные токены, представляющие собой имя незамещенного макроса, становятся недоступны для замещения, даже если позже будет выполняться новый просмотр, в контексте которого это замещение возможно.*

6. Макросы не имеют адресов

В C++ возможно получение указателя на любую свободную функцию или функцию-член (например, для использования в качестве предиката), но невозможно получение указателя на макрос, поскольку макрос не имеет адреса. Почему это так, должно быть очевидно — макрос не является кодом. Сам по себе макрос не существует, поскольку он представляет собой не что иное, как правило замены текста.

7. Макросы трудно отлаживать

Мало того, что макросы изменяют код до того, как его увидит компилятор, но они еще и не позволяют пошагового прохождения в процессе отладки.

Чего не могут макросы

Имеется ряд причин для использования макросов (см. задачу 8.3), но есть и ограничения на их применение. Это приводит нас к последнему вопросу.

2. Чего не может создать макрос препроцессора и почему?

В стандарте раздел 2.1 определяет фазы трансляции. Обработка директив препроцессора и раскрытия макросов происходят в фазе 4. Таким образом в соответствующем стандарту компиляторе невозможно применение макроса для создания чего-либо из приведенного списка.

- Триграфы (замещаемые в первой фазе).
- Универсальные имена символов (`\uXXXX`, замещаемые в первой фазе).
- Символ продолжения строки (\ в конце строки, замещаемый во второй фазе).
- Комментарий (замещаемый в третьей фазе).
- Другой макрос или директива препроцессора (раскрываемые и выполняемые в четвертой фазе).
- Изменения в символе (например, `'x'`) или строке (например, `"Hello"`) посредством наличия имен макросов внутри строки.

В одной статье[1] был приведен способ создания комментария с помощью макросов.

```
#define COMMENT SLASH(/)
#define SLASH(s) /##s
```

Этот метод нестандартен и непереносим, хотя и работает в некоторых популярных компиляторах. Однако это говорит всего лишь о том, что эти компиляторы некорректно реализуют фазы трансляции.

[1] Timperley M. *A C/C++ Comment Macro*, (C/C++ Users Journal, 19(1), January 2001.

9

Ловушки, ошибки и антиидиомы

Задача 9.1. Тождественность объектов ***Сложность: 5***

"Кто я?" В данной задаче выясняется вопрос о том, указывают ли два указателя на один и тот же объект.

Проверка "`this != &other`", показанная ниже, представляет собой распространенный способ защиты от присваивания самому себе. Является ли данное условие необходимым и/или достаточным? Почему да или почему нет? Если нет, каким образом можно исправить ситуацию?

```
T& T::operator=( const T& other )
{
    if (this != &other )
    {
        // ...
    }
    return *this
}
```

Решение

Краткий ответ: в большинстве случаев без такой проверки присваивание самому себе обрабатывается некорректно. Хотя такая проверка и не в состоянии защитить от всех некорректных использований данного кода, на практике этот метод работает вполне хорошо, до тех пор пока используется для оптимизации кода. В прошлом высказывались определенные предположения о влиянии множественного наследования на корректность данной проверки, но это не более чем ложная тревога.

Безопасность исключений

Если оператор `T::operator=()` написан с использованием идиомы создания временного объекта и обмена состояний (см. раздел "Элегантное копирующее присваивание" задачи 2.6), то он безопасен с точки зрения исключений, а кроме того, не нуждается в проверке на присваивание самому себе. Поскольку обычно мы предпочитаем использовать именно эту технологию, проверка на присваивание самому себе нам не нужна, не так ли?

Рекомендация

Никогда не пишите оператор присваивания таким образом, чтобы для корректной работы он должен был полагаться на проверку присваивания самому себе. Оператор присваивания, использующий идиому создания временного объекта и обмена, автоматически является строго безопасным в смысле исключений и в смысле присваивания самому себе.

И да, и нет. Отсутствие проверки присваивания самому себе имеет две потенциальные причины снижения эффективности.

- Если вы можете проверить присваивание самому себе, то вы в состоянии оптимизировать код присваивания.
- Зачастую обеспечение безопасности кода в смысле исключений делает его менее эффективным.

На практике, если присваивание самому себе встречается достаточно часто (на самом деле в большинстве программ это большая редкость) и в результате скорость работы программы резко снижается из-за излишней работы во время такого присваивания (что еще большая редкость в реальной жизни), то тогда вы можете использовать проверку на присваивание самому себе.

Рекомендация

Можно использовать проверку на присваивание самому себе для оптимизации, заключающейся в устранении ненужной работы.

Перегрузка операторов

Поскольку классы могут иметь свой собственный `operator&()`, предлагаемая проверка может иметь совершенно иной смысл. Впрочем, разработчик оператора присваивания должен знать о переопределенном операторе & (но, с другой стороны, базовый класс также может иметь переопределенный оператор &).

Заметим, что наличие у класса `T::operator!=()` не имеет никакого отношения к сравнению `this != &other`. Дело в том, что вы не можете написать `operator!=()`, параметрами которого являются два указателя на `T`, поскольку как минимум один параметр обязан иметь тип класса.

Постскриптум 1

В качестве небольшого развлечения — фрагмент кода, который программисты ухитряются написать исключительно из лучших побуждений.

```
T::T( const T& other )
{
    if (this != &other )
    {
        // ...
    }
}
```

Удалось ли вам понять с первого раза, в чем тут соль?[1]

Постскриптум 2

Заметим, что имеются и другие случаи, когда сравнение указателей на самом деле выполняет не те функции, которые, как интуитивно кажется, оно должно выполнять. Вот пара примеров.

- Сравнение указателей в строковые литералы не определено. Причина этого в том, что для оптимизации стандарт явным образом разрешает компилятору хранить строковые литералы в перекрывающихся областях памяти. По этой причине может оказаться, что указатели внутрь двух разных строковых литералов при сравнении оказываются одинаковыми.
- Вообще говоря, вы не можете сравнивать произвольные указатели с использованием встроенных операций <, <=, >, и >=, хотя в конкретных ситуациях результат таких операций определен (например, таким образом можно сравнивать указатели на объекты из одного массива). Стандартная библиотека обходит это ограничение, используя для упорядочения указателей библиотечную шаблонную функцию `less<>` и аналогичные ей. Это необходимо, чтобы вы могли создать, скажем, ассоциативный контейнер `map` с ключевыми значениями, представляющими собой указатели, например `map<T*,U>`, который представляет собой не что иное, как ассоциативный контейнер `map` с параметром шаблона по умолчанию, т.е. `map<T*,U,less<T*> >`.

Задача 9.2. Автоматические преобразования *Сложность: 4*

Автоматические преобразования от одного типа к другому могут быть исключительно удобны. Эта задача показывает, почему автоматические преобразования могут оказаться исключительно опасными.

Стандартный строковый класс `string` не содержит неявного преобразования к типу `const char*`. Не упущение ли это?

Зачастую было бы весьма удобно иметь возможность обращения к `string` как к `const char*` языка C. Конечно, `string` имеет функцию-член `c_str()`, которая позволяет обратиться к строке как к символьному массиву.

---

[1] Проверка на присваивание самому себе в конструкторе лишена смысла, поскольку объект `other` не может быть тем же объектом, что и наш объект, — ведь наш объект все еще находится в процессе создания! Когда я впервые написал эту шутку, я (совершенно справедливо) считал, что такой тест не имеет никакого смысла. Тем не менее этот код попадает в категорию “защиты от дурака”. Мой друг (и проницательный читатель) Джим Хайслоп (Jim Hyslop) привел (технически некорректный) код с использованием размещающего оператора `new`, который придает смысл нашей проверке (конечно, если бы этот код был технически корректен).

```
T t;
new (&t) T(t); // Некорректно, но придает смысл проверке
```

Просто непонятно, радоваться ли, что люди по достоинству оценили мою шутку, или беспокоиться о том, что они тут же попытались найти способ воплотить шутку в жизнь...

Приведенный далее код также некорректен в C++: хотя он и корректен синтаксически (соответствующий стандарту компилятор должен без проблем его скомпилировать), но его поведение не определено (тот же соответствующий стандарту компилятор может вполне законно вставить здесь код, форматирующий ваш винчестер). Если бы этот код был корректен, он бы также придавал смысл описанной проверке.

```
T t = t; // Некорректно, но придает смысл проверке
```

```
string s1("hello"), s2("world");
strcmp( s1, s2 );                  // 1 - ошибка
strcmp( s1.c_str(), s2.c_str() ); // 2 - OK
```

Конечно, было бы неплохо иметь возможность работать со строками так, как показано в строке 1, но это невозможно из-за отсутствия автоматического преобразования `string` в `const char*`. Инструкция в строке 2 работает корректно, но она длиннее предыдущей в связи с необходимостью явного вызова `c_str()`.

Основной вопрос данной задачи: не лучше ли было бы иметь возможность использовать инструкции наподобие приведенной в строке 1?

Решение

Итак, главный вопрос: **стандартный строковый класс `string` не содержит неявного преобразования к типу `const char*`. Не упущение ли это?**

Ответ: нет. И тому есть важные причины.

Почти всегда следует избегать автоматического преобразования типов, как путем предоставления оператора преобразования типа, так и посредством наличия не-`explicit` конструктора с одним аргументом.[2] Основные причины небезопасности неявных преобразований типов следующие.

- Неявные преобразования могут мешать разрешению перегрузки.
- Неявные преобразования могут привести к компиляции неверного кода без каких-либо сообщений об ошибках.

Если тип `string` имеет автоматическое преобразование к `const char*`, такое преобразование может быть неявно вызвано везде, где компилятор сочтет это необходимым. В результате оказывается очень просто написать код, который не только правильно выглядит, но и компилируется без сообщений об ошибках, но при этом работает совершенно не так, как требуется. Вот один маленький пример.

```
string s1, s2, s3;
s1 = s2 - s3;      // Опечатка: должен был быть знак "+"
```

Вычитание не имеет никакого смысла и должно привести к выводу сообщения об ошибке. Если тип `string` может быть неявно преобразован к типу `const char*`, то этот код будет скомпилирован без сообщений об ошибках (компилятор преобразует обе строки к указателям `const char*`, а затем вычислит их разность).

Рекомендация

Избегайте написания операторов преобразования типов и не-`explicit` конструкторов.

Задача 9.3. Времена жизни объектов. Часть 1 — *Сложность: 5*

"Быть иль не быть..." Когда в действительности существует объект? В данной задаче выясняется, когда можно безопасно работать с объектами.

Рассмотрите следующий фрагмент кода.

```
void f()
{
    T t(1);
```

[2] Данное решение рассматривает обычные проблемы неявных преобразований, но в случае с классом `string` имеются и другие причины для отказа от неявного преобразования в `const char*`, о которых можно прочесть в [Koenig97] и [Stroustrup94].

```
    T& rt = t;
    // --- №1: Некоторые действия с t или rt
    t.~T();
    new (&t) T(2);
    // --- №2: Некоторые действия с t или rt
}   // Уничтожение t
```

Является ли код в блоке №2 безопасным и/или корректным? Поясните свой ответ.

Решение

Да, код в блоке №2 безопасен и корректен (если вы доберетесь до него). Однако в целом данная функция небезопасна.

Стандарт C++ явно разрешает использование данного кода. Ссылка `rt` не становится недействительной из-за уничтожения объекта и его повторного конструирования. (Конечно, вы не можете использовать `t` и `rt` между вызовами `t.~T()` и размещающего оператора `new`, поскольку в это время объект не существует. Кроме того, мы полагаем, что `T::operator&()` не перегружен и возвращает адрес объекта.)

Причина, по которой мы сказали, что код в блоке №2 безопасен "если вы доберетесь до него", заключается в том, что функция `f()` в целом может не быть безопасна в смысле исключений. Если конструктор `T` может сгенерировать исключение при вызове `new(&t) T(2)`, то `f()` не безопасна. Рассмотрим, почему. Если `T(2)` генерирует исключение, то в области памяти `t` новый объект не создается, но при выходе из функции все равно вызывается деструктор `T::~T()` (поскольку `t` — автоматическая переменная) и, таким образом, `t` уничтожается дважды, что влечет за собой непредсказуемые последствия.

Рекомендация

Всегда старайтесь писать безопасный в смысле исключений код. Структурируйте код таким образом, чтобы даже при генерации исключений корректно освобождались все ресурсы, а данные находились в согласованном состоянии.

Вообще использования этого трюка следует избегать, и не только по причине небезопасности в плане исключений. Применение данной методики в функции-члене способно привести к опасным последствиям (даже при игнорировании вопросов безопасности исключений).

```
// Сумеете ли вы обнаружить неприятности?
//
void T::DestroyAndReconstruct( int i )
{
    this->~T();
    new(this) T(i);
}
```

Безопасен ли данный метод? (Напомню: сейчас мы не рассматриваем вопросы безопасности исключений.) Вообще говоря, нет. Рассмотрим следующий код.

```
class U : public T { /* ... */ };
void f()
{
    /*AAA*/  t(1);
    /*BBB*/& rt = t;
    // --- №1: Некоторые действия с t или rt
    t.DestroyAndReconstruct(2);
    // --- №2: Некоторые действия с t или rt
}   // Уничтожение t
```

Если `/*AAA*/` представляет собой `T`, то код №2 остается работоспособным, даже если `/*BBB*/` не является `T` (это может быть базовый класс `T`).

Но если `/*AAA*/` — это `U`, то не имеет значения, что такое `/*BBB*/`. Вероятно, наилучшее, на что вы можете надеяться в этом случае — это немедленный сброс дампа памяти, поскольку вызов `t.DestroyAndReconstruct()` "срезает" объект. Срезка происходит потому, что `t.DestroyAndReconstruct()` замещает исходный объект объектом другого типа, а именно `T` на `U`. Даже если вы готовы написать непереносимый код, все равно у вас нет возможности узнать, позволяет ли схема размещения объекта `T` в памяти использовать его на том же месте памяти, где был объект `U`. Шансы, что это невозможно, достаточно велики.

Рекомендация

Избегайте "темных закутков" языка программирования. Среди методов, эффективно решающих поставленную задачу, выбирайте простейший.

В этой задаче мы рассмотрели базовые вопросы безопасности и срезки при использовании уничтожения объекта и построения нового на старом месте. Дальнейшее развитие этой темы — в задаче 9.4.

Задача 9.4. Времена жизни объектов. Часть 2 *Сложность: 6*

В данной задаче мы рассматриваем часто рекомендуемую идиому, которая нередко чревата ошибками.

Рассмотрите следующую "антиидиому".

```
T& T::operator=( const T& other )
{
    if ( this != &other )
    {
        this->~T();
        new(this) T(other);
    }
    return *this;
}
```

1. Какую цель преследует данный код? Исправьте все дефекты в данной версии.
2. Безопасна ли данная идиома даже после исправления всех дефектов? Поясните. Если нет, каким способом программист может достичь желаемого результата?

Решение

Эту идиому[3] часто рекомендуют, и она даже приведена в стандарте C++ (см. врезку "Пример-предупреждение"). Однако использование этой идиомы приводит ко многим проблемам, так что лучше все же без нее обойтись.

К счастью, как мы вскоре увидим, имеется корректный способ достижения поставленной цели.

[3] Здесь я не рассматриваю патологические случаи типа перегрузки оператора `T::operator&()` таким образом, что он не возвращает значение указателя `this`. См. также задачу 9.1.

Пример-предупреждение

В стандарте C++ данный пример призван продемонстрировать правила времени жизни объектов, и он не рекомендуется для практического использования. В связи с тем, что этот пример достаточно интересен, он приводится здесь полностью.

```
[Пример:
    struct C {
            int i;
            void f();
            const C& operator=( const C& );
    };
    const C& C::operator=( const C& other)
    {
        if ( this != &other )
        {
            this->~C();  // Время жизни '*this' закончилось
            new (this) C(other);
                         // Создан новый объект типа C
            f();         // OK
        }
        return *this;
    }
    C c1;
    C c2;
    c1 = c2; // OK
    c1.f();  // OK, c1 ссылается на новый объект типа C
--конец примера]
```

Как еще одно доказательство того, что данный метод не рекомендуется использовать на практике, заметим, что оператор `C::operator=()` возвращает `const C&`, а не просто `C&`, что мешает использованию этих объектов в контейнерах стандартной библиотеки.

В стандартах кодирования PeerDirect написано следующее.

- Объявляйте копирующее присваивание как `T& T::operator=(const T&)`.
- Не возвращайте `const T&`. Хотя, с одной стороны, это и предотвращает использование конструкций типа `(a=b)=c`, с другой стороны, такой тип возвращаемого значения не позволяет размещать объекты типа `T` в контейнерах стандартной библиотеки, поскольку она требует, чтобы присваивание возвращало обычную ссылку `T&` (см. также [Cline95] и [Murray93]).

Вернемся к нашим вопросам.

1. Какую цель преследует данный код?

Данная идиома выражает копирующее присваивание посредством конструктора копирования. Таким образом, применение данной идиомы должно гарантировать, что копирующее присваивание и конструктор копирования выполняют одни и те же действия, и уберечь программиста от необходимости писать один и тот же код в двух различных местах.

Это благородная цель. В конечном счете программирование становится проще, когда не надо писать один и тот же код дважды, да и при изменении `T` (например, добавлении новой переменной-члена), обновив одну функцию, вам не придется тут же вносить добавления в другую.

Эта идиома представляется особенно практичной при наличии виртуальных базовых классов с членами данных, присваивание которых в противном случае может оказаться некорректным (или, в лучшем случае, неоднократно выполненным). Впрочем, на самом деле это не совсем так, поскольку в виртуальных базовых классах не рекомендуется иметь члены данных (см. [Meyers98], [Meyers99] и [Barton94]). Кроме того, наличие виртуальных

базовых классов говорит о том, что наш класс предназначен для наследования, при котором использование данной идиомы, как мы увидим, слишком опасно.

Вторая часть первого вопроса звучит так: **исправьте все дефекты в данной версии**. Приведенный код имеет один дефект, который можно исправить, и ряд других, исправлению не поддающихся.

Проблема 1. Срезка объектов

Код "`this->~T(); new(this) T(other);`" работает неверно, если `T` — базовый класс с виртуальным деструктором. При вызове объектом производного класса выполнение данного кода приведет к уничтожению объекта производного класса и замене его объектом `T`. Это почти гарантированно приведет к неработоспособности кода, который после этого попытается использовать данный объект (эта проблема упоминалась в задаче 9.3).

В частности, это приводит к тому, что жизнь авторов производных классов превращается в сплошной кошмар. Вспомним, что обычно операторы присваивания производного класса реализуются на основе присваивания базового класса следующим образом.

```
Derived& Derived::operator=( const Derived& other )
{
    Base::operator=( other );
    // ... Присваивание членов Derived ...
    return *this;
}
```

В нашем случае этот код превращается в следующий.

```
class U : T { /* ... */ };
U& operator=( const T& other )
{
    T::operator=( other );
    // ... Присваивание членов U ...
    //    ... Стоп! Но ведь U у нас больше нет!
    return *this; // Та же история...
}
```

Как и указывалось, вызов `T::operator=()` приводит к неработоспособности следующего за ним кода (как присваивания членов `U`, так и инструкции `return`). Все это может проявиться в виде странных и трудноуловимых ошибок времени выполнения.

Для исправления ситуации можно использовать несколько вариантов возможных действий.

- Не следует ли в копирующем присваивании `U` непосредственно вызывать `this->T::~T()` с последующим размещающим `new` базового подобъекта `T`? При этом получится, что замещенным окажется только базовый подобъект `T` (вместо срезки и некорректного преобразования в `T` производного объекта).
- Но почему бы не пойти на шаг дальше и не следовать идиоме копирующего конструктора `T` в копирующем конструкторе `U`? Это решение, хотя и несколько лучше предыдущего, отлично иллюстрирует второй недостаток данной идиомы. Если один класс использует эту идиому, то ее должны использовать и все производные классы (что плохо по целому ряду причин, не последней из которых является та, что генерируемый по умолчанию оператор копирующего присваивания не использует данную идиому, и многие программисты, которые выполняют наследование без добавления членов данных, даже не думают о разработке собственной версии `operator=()`, полностью полагаясь на оператор копирующего присваивания, сгенерированный компилятором).

Увы, оба вышеперечисленных варианта просто устраняют очевидную опасность и заменяют ее менее очевидной, которая мешает жить авторам производных классов.

Рекомендация

Избегайте "темных закутков" языка программирования. Среди методов, эффективно решающих поставленную задачу, выбирайте простейший.

Рекомендация

Избегайте написания излишне сжатого кода, даже если при его написании вам совершенно ясно, как он работает.

Обратимся теперь ко второму вопросу.

2. Безопасна ли данная идиома даже после исправления всех дефектов? Поясните.

Нет. Ни одна из перечисленных далее проблем не может быть решена без отказа от использования данной идиомы.

Проблема 2. Небезопасность в смысле исключений

Инструкция `new` вызывает конструктор копирования `T`. Если этот конструктор может генерировать исключения (а многие классы сообщают об ошибках в конструкторе путем генерации исключений), то функция в целом не является безопасной, поскольку может быть ситуация, когда уничтоженный объект не заменяется новым.

Аналогично срезке, это приводит к неработоспособности последующего кода, который попытается использовать данный объект. Еще неприятнее то, что программа при этом, вероятно, попытается уничтожить один и тот же объект дважды, поскольку внешний код не знает о том, что деструктор объекта уже был вызван (см. задачу 9.3).

Рекомендация

Всегда старайтесь писать безопасный в смысле исключений код. Структурируйте код таким образом, чтобы даже при генерации исключений корректно освобождались все ресурсы, а данные находились в согласованном состоянии.

Проблема 3. Изменение обычного времени жизни объекта

Данная идиома делает неработоспособным код, полагающийся на обычное время жизни объекта. В частности, она мешает нормальной работе классов, которые используют идиому "захват ресурса при инициализации". В общем случае она мешает или влияет на работу любого класса, конструктор или деструктор которого имеют побочные эффекты.

Например, если `T` (или любой базовый класс `T`) использует блокировку или начало транзакции базы данных в своем конструкторе и производит разблокирование или завершение транзакции в деструкторе? Тогда в процессе присваивания блокировка/транзакция будет некорректно завершена и начата заново, что скорее всего приведет к некорректной работе как класса, так и его клиентов. Помимо `T` и его базовых классов, некорректно будут работать и все порожденные классы, которые опираются на обычную семантику времени жизни объектов.

Некоторые скажут: “Я никогда не использую блокировку/деблокирование в конструкторе/деструкторе!” В ответ на это я только спрошу: “Правда? И откуда вы знаете, что этого не делает ни один из базовых классов — непосредственных или опосредованных?” Вы далеко не всегда имеете возможность узнать об этом и никогда не должны полагаться на то, что ваши базовые классы будут корректно вести себя при некорректных играх с временем жизни объектов.

Фундаментальная проблема заключается в том, что эта идиома извращает смысл конструирования и деструкции, которые в точности соответствуют началу и концу времени жизни объекта (когда объект, как правило, захватывает и, соответственно, освобождает ресурсы). Конструктор и деструктор не предназначаются для изменения значения объекта (и на самом деле они не меняют его — они по сути уничтожают старый объект и создают новый, что, хотя со стороны и выглядит как изменение значения, на самом деле таковым не является).

Проблема 4. Некорректная работа производных классов

При решении проблемы 1 путем использования вызова “`this->~T::T();`” данная идиома замещает только “часть `T`” (базовый подобъект `T`) внутри производного объекта. С одной стороны, это должно беспокоить нас, поскольку нарушает обычную гарантию C++ о том, что время жизни базового подобъекта полностью включает время жизни производного объекта (т.е. подобъект базового класса всегда создается до и уничтожается после подобъекта производного класса). На практике многие производные классы могут нормально реагировать на замену своей базовой части, но реакция некоторых может оказаться не такой спокойной. В частности, любой производный класс, отвечающий за состояние базового класса, может начать некорректно работать при изменении его базовой части без соответствующего уведомления производного класса.

Проблема 5. this != &other

Данная идиома полностью основана на проверке `this != &other` (если вы сомневаетесь в этом, рассмотрите, что произойдет в случае присваивания самому себе без такой проверки).

Проблема в данном случае та же, что и рассматривавшаяся в задаче 9.1: любое копирующее присваивание, написанное таким образом, что оно *должно* выполнять проверку на присваивание самому себе, вероятно, не является строго безопасным в смысле исключений.[4]

Данная идиома скрывает и другие потенциальные опасности, которые могут оказывать влияние на код клиента и/или производные классы (такие, например, как поведение при наличии виртуальных операторов присваивания), но и уже рассмотренного материала достаточно, чтобы продемонстрировать наличие серьезных проблем при использовании этой идиомы.

Нам осталось рассмотреть последнюю часть второго вопроса данной задачи: **если нет, каким способом программист может достичь желаемого результата?**

Выражение копирующего присваивания посредством конструктора копирования — неплохая идея, но правильный способ состоит в использовании канонического вида

[4] Нет ничего плохого в том, что вы используете проверку `this != &other` для оптимизации случая присваивания самому себе. Но ваш оператор копирующего присваивания должен быть написан таким образом, что должен корректно обрабатывать случай присваивания самому себе и при отсутствии такой проверки. Детальное рассмотрение аргументов “за” и “против” использования этой проверки с целью оптимизации лежит за пределами “области видимости” данной задачи.

строго безопасного копирующего присваивания. Обеспечьте наличие функции-члена `Swap()`, которая гарантирует отсутствие генерации исключений и просто атомарно обменивает “внутренности” объектов, а затем запишите оператор копирующего присваивания следующим образом.

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

Этот метод также реализует копирующее присваивание посредством конструктора копирования, но делает это корректным и безопасным способом. При этом нет никаких “игр” со временем жизни объектов и не требуется проверка на присваивание объекта самому себе.

Дополнительную информацию о каноническом виде и разных уровнях безопасности исключений можно получить в задачах 2.1—2.10 и 9.3.

Итак, описанная в данной задаче идиома переполнена неприятностями, часто просто ошибочна и делает жизнь авторов производных классов сплошным кошмаром.[5]

Рекомендация

Предпочтительно обеспечить класс функцией-членом `Swap()`, не генерирующей исключений, и реализовать копирующее присваивание посредством конструктора копирования следующим образом.

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

Никогда не пользуйтесь реализацией копирующего присваивания посредством конструктора копирования с использованием явного вызова деструктора с последующим размещающим `new`.

Несколько слов в пользу простоты

Мой главный совет? Не выпендриваться! Рассматривайте все новые возможности C++ как заряженный автомат со снятым предохранителем в переполненном людьми помещении. Никогда не используйте его только потому, что он красиво выглядит. Подождите, пока вы не сможете оценить все последствия применения новых возможностей. Пишите то, что вы знаете, и всегда знайте, что вы пишете.

Красота губит как минимум по двум причинам (конечно, когда она становится самоцелью, а не следствием хорошо продуманного проекта). Во-первых, она непереносима. Вычурный код может хорошо работать на одном компиляторе и заставить другой прийти в изумление. Или, хуже того, другой компилятор вежливо скомпилирует то, что вы ему подсунули, но результаты работы скомпилированной программы станут для вас неприятной неожиданностью. Простейший пример — очень многие компиляторы не поддерживают аргументы шаблонов по умолчанию

[5] Правду говоря, ни одно представленное решение не работает с классами, среди членов которого имеются переменные-ссылки. Однако это не ограничение решений, а прямой результат того факта, что классы с членами-ссылками не должны присваиваться. Если требуется изменение объекта, на который указывает ссылка, вместо нее следует использовать указатель.

или специализации шаблонов, так что код, который основан на их использовании, на сегодняшний день можно считать непереносимым. Другой пример — многие компиляторы расходятся во мнениях о том, как следует обрабатывать множественные активные исключения.

Во-вторых, вычурный код трудно поддерживать. Никогда не забывайте о том, что кому-то придется разбираться в том, что вы написали. То, что вам сегодня кажется красивым фокусом, для читающего ваш код завтра может выглядеть чем-то вроде последствий взрыва на макаронной фабрике. И не забывайте, что этим читающим (и поддерживающим!) ваш код можете оказаться вы сами. Например, некая графическая библиотека использует то, что в свое время было новшеством — перегрузку операторов. При использовании этой библиотеки клиент добавляет управляющий элемент к окну простым прибавлением.

```
Window  w( /* ... */ );
Control c( /* ... */ );
w + c;
```

Таким образом оказывается нарушенным правило, гласящее, что семантика перегруженных операторов должна соответствовать семантике операторов встроенных типов и типов стандартной библиотеки.

Пишите то, что вы знаете. Можно сознательно попытаться ограничить 90% вашего кода применением только тех средств языка, которые вы знаете настолько хорошо, что способны использовать их даже во сне (неужели вам никогда не снится программирование на C++?), и только в оставшихся 10% использовать новые возможности для получения опыта работы с ними. Если вы новичок в C++, это поначалу приведет вас к использованию C++ как улучшенного C, но в этой ситуации нет ничего страшного или неверного.

Знайте то, что вы пишете. Вы должны ясно представлять себе все последствия работы вашего кода. Не пренебрегайте учением, ибо ученье — свет, а неученых — тьма. Почаще читайте книги наподобие этой, не забывайте о специализированных журналах типа *C/C++ User's Journal* или *Dr. Dobb's Journal*. Заглядывайте в группы новостей, посвященные C++, такие как *comp.lang.c++.moderated*, *comp.std.c++* или *fido7.su.c_cpp*. Для того чтобы постоянно оставаться на одном уровне, надо прилагать немалые усилия, — что же говорить об усилиях, требующихся для профессионального роста?

В качестве иллюстрации важности знания того, что вы делаете, можно привести только что рассмотренную идиому реализации копирующего присваивания посредством конструктора копирования с использованием явного вызова деструктора с последующим размещающим `new`. Этот вопрос регулярно поднимается в группах новостей, и сторонники данного метода преследуют благие намерения — избежать дублирования кода. Но знают ли они, что пишут? Вряд ли, исходя из того, сколько неприятных последствий использования данной идиомы перечислено в решении данной задачи. В самой же цели нет ничего некорректного, но достичь ее можно и другим путем, — например, создав закрытую функцию-член, вызываемую как из конструктора копирования, так и из оператора копирующего присваивания.

Словом, избегайте "темных закутков" языка программирования и помните, что красота многогранна. Пишите то, что знаете, и знайте то, что пишете, и все будет в порядке.

10

Понемногу обо всем

Задача 10.1. Инициализация. Часть 1 *Сложность: 3*

Эта задача демонстрирует важность понимания того, что вы пишете. Здесь приведены четыре строки простейшего кода, которые, несмотря на малое различие в синтаксисе, означают разные вещи.

В чем состоит различие (если таковое имеется) между следующими строками (`T` здесь обозначает произвольный класс)?

```
T t;
T t();
T t(u);
T t = u;
```

Решение

Эта головоломка демонстрирует различие между тремя способами инициализации: по умолчанию, непосредственной и инициализации копированием. Здесь имеется и один ложный пример, который не является инициализацией вовсе. Итак, рассмотрим их по очереди.

```
T t;
```

Это — *инициализация по умолчанию*. Этот код объявляет переменную `t` типа `T`, которая инициализируется с использованием конструктора по умолчанию `T::T()`.

```
T t();
```

Ловушка. На первый взгляд эта строка представляет собой еще одно объявление переменной, но на самом деле это объявление функции с именем `t`, не имеющей параметров и возвращающей объект `T` по значению. (Если это вам непонятно, рассмотрите объявление `int f();`, которое, совершенно очевидно, является объявлением функции).

Некоторые программисты предлагают писать “`auto T t();`”, чтобы путем использования ключевого слова `auto` показать, что мы хотим объявить переменную с именем `t` типа `T`. Позвольте сказать пару слов по этому поводу. Такое объявление не будет работать с компилятором, соответствующим стандарту, так как компилятор будет рассматривать эту строку как объявление функции и отвергнет ее в силу неприменимости описания `auto` в данном случае. К тому же, даже если бы такое описание работало, его вряд ли стоило бы применять, поскольку есть куда более простое описание переменной `t` типа `T`, конструируемой по умолчанию, — а именно “`T t;`”. Никогда не следует писать код более сложный, чем это необходимо для решения конкретной задачи.

```
T t(u);
```

Если u не является именем типа, то данная строка представляет собой объявление переменной с непосредственной инициализацией (direct initialization). Переменная t инициализируется непосредственно значением u путем вызова T::T(u). Если u — имя типа, мы получаем случай, описанный выше.

```
T t = u;
```

Здесь мы имеем дело с инициализацией копированием. Переменная t инициализируется с использованием конструктора копирования T, возможно, после вызова другой функции.

Распространенная ошибка

Данная инициализация, несмотря на наличие в строке символа "=", всегда является именно инициализацией и никогда — присваиванием, так что при этом никогда не вызывается `T::operator=()`. Это всего лишь пережиток синтаксиса C, а не операция присваивания.

Вот семантика данного присваивания.

- Если u представляет собой переменную типа T, то это объявление эквивалентно объявлению "`T t(u);`" и просто вызывает конструктор копирования T.
- Если u имеет некоторый другой тип, то данное объявление имеет смысл "`T t(T(u));`", т.е. u сначала преобразуется во временный объект типа T, после чего вызывается конструктор копирования из этого временного объекта. Заметим, однако, что в данном случае компилятору позволено опустить излишние копирования и преобразовать данную инициализацию в непосредственную (т.е. привести ее к виду "`T t(u);`"). Даже если ваш компилятор поступает таким образом, должен быть доступен соответствующий конструктор копирования. Однако если конструктор копирования имеет побочные действия, то вы можете не получить ожидаемый результат, так как он будет зависеть от того, выполнит ли компилятор оптимизацию инициализации или нет.

Рекомендация

Предпочтительно использовать инициализацию вида "`T t(u);`", а не "`T t=u;`". Первый метод работает везде, где работает второй, и имеет ряд преимуществ, в частности он допускает использование нескольких параметров.

Задача 10.2. Инициализация. Часть 2 — *Сложность: 3*

В чем состоит различие между непосредственной инициализацией и инициализацией копированием и когда они должны использоваться?

1. В чем состоит различие между непосредственной инициализацией и инициализацией копированием?
2. В каких из следующих случаев используется непосредственная инициализация, а в каких — инициализация копированием?

```
class T : public S
{
public:
    T() : S(1),    // Инициализация базового класса
          x(2) {}  // Инициализация члена
```

```
    X x;
};

T f( T t )          // Передача аргумента функции
{
    return t;       // Возврат значения
}

S  s;
T  t;
S& r = t;

reinterpret_cast<S&>(t);  // Выполнение reinterpret_cast
dynamic_cast<T&>(r);      // Выполнение dynamic_cast
const_cast<const T&>(t);  // Выполнение const_cast
static_cast<S>(t);        // Выполнение static_cast

try
{
    throw T();            // Генерация исключения
}
catch( T t )              // Обработка исключения
{
}

f( T(s) );                // Преобразование типа
S a[3] = { 1, 2, 3 };     // Инициализаторы в скобках
S* p = new S(4);          // Выражение new
```

Решение

1. В чем состоит различие между непосредственной инициализацией и инициализацией копированием?

Непосредственная инициализация означает, что объект инициализируется с использованием одного (возможно, преобразующего тип) конструктора и эквивалентна записи “`T t(u);`”:

```
U u;
T t1(u);    // Вызов T::T( U& ) или его аналога
```

Копирующая инициализация означает, что объект инициализируется с использованием конструктора копирования (после вызова при необходимости преобразования типа, определенного пользователем) и эквивалентна записи “`T t = u;`”:

```
T t2 = t1;  // Тот же тип – вызов T::T(T&) или аналога
T t3 = u;   // Другой тип – вызов T::T(T(u)),
            // T::T(u.operator T()) или аналога
```

Отступление: причина, по которой упоминается “аналог”, заключается в том, что конструкторы копирования и преобразования типа могут принимать нечто, отличающееся от обычной ссылки (ссылка, например, может быть описана как `const` или `volatile`); кроме того, конструктор и оператор преобразования типа могут принимать или возвращать объект, отличный от ссылки. В дополнение к сказанному конструкторы копирования и преобразования типа могут иметь дополнительные аргументы по умолчанию.

Заметим, что в последнем случае (“`T t3 = u;`”) компилятор может как вызвать определенное пользователем преобразование типов (для создания временного объекта) и конструктор копирования `T` (для создания `t3` из временного объекта), так и по-

строить t3 непосредственно из u (что эквивалентно записи “T t3(u);”). В любом случае неоптимизированный код должен быть корректен. В частности, конструктор копирования должен быть доступен, даже если вызов осуществляется оптимизированным путем.

В стандарте C++ возможности компилятора по устранению временных объектов по сравнению с “достандартными” временами несколько ограничены. Более детально об этом говорится в задачах 10.1 и 10.6.

Рекомендация

Предпочтительно использовать инициализацию вида “T t(u);”, а не “T t=u;”.

2. В каких из следующих случаев используется непосредственная инициализация, а в каких — инициализация копированием?

Для большинства случаев ответ можно найти в разделе 8.5 стандарта C++. Имеется также три случая, когда инициализация отсутствует вовсе. Сможете ли вы верно их указать?

```
class T : public S
{
public:
    T() : S(1),     // Инициализация базового класса
          x(2) {}   // Инициализация члена
    X x;
};
```

При инициализации базового класса и члена используется непосредственная инициализация.

```
T f( T t )          // Передача аргумента функции
{
    return t;       // Возврат значения
}
```

При передаче и возврате значения используется инициализация копированием.

```
S  s;
T  t;
S& r = t;

reinterpret_cast<S&>(t);  // Выполнение reinterpret_cast
dynamic_cast<T&>(r);      // Выполнение dynamic_cast
const_cast<const T&>(t);  // Выполнение const_cast
```

В этих случаях нет инициализации нового объекта — здесь происходит только создание ссылок.

```
static_cast<S>(t);        // Выполнение static_cast
```

При приведении static_cast используется непосредственная инициализация.

```
try
{
    throw T();            // Генерация исключения
}
catch( T t )              // Обработка исключения
{
}
```

При генерации и перехвате объекта исключения используется инициализация копированием.

Заметим, что в данном конкретном коде имеются три объекта T и осуществляется два копирования. Копия объекта генерируемого исключения создается при генерации

последнего; вторая копия в данном случае создается в связи с тем, что обработчик получает объект исключения по значению.

Вообще говоря, следует использовать перехват исключений по ссылке, а не по значению, чтобы избежать создания излишних копий и устранить потенциальную срезку объектов.

```
f( T(s) );                  // Преобразование типа
```

Это преобразование типов в стиле функции использует непосредственную инициализацию.

```
S a[3] = { 1, 2, 3 };       // Инициализаторы в скобках
```

Инициализаторы в скобках используют инициализацию копированием.

```
S* p = new S(4);            // Выражение new
```

И наконец, `new` использует непосредственную инициализацию.

Задача 10.3. Корректность `const` *Сложность: 6*

`const` представляет собой мощный инструмент для написания безопасного кода. Используйте `const` везде, где можно, но не более того. В задаче показан ряд очевидных (и не совсем) мест, где следует (или не следует) использовать `const`.

Не следует комментировать или изменять структуру приведенной далее программы — ее цель сугубо иллюстративная. Просто добавьте или удалите `const` там, где это стоит сделать (допустимы минимальные изменения и использование необходимых ключевых слов).

Вопрос на засыпку: в каких местах результаты работы программы неопределенные (а где программа просто не компилируется) из-за ошибок, связанных с `const`?

```
class Polygon
{
public:
    Polygon() : area_(-1) {}
    void AddPoint( const Point pt ) { InvalidateArea();
                                      points_.push_back(pt); }
    Point GetPoint( const int i )   { return points_[i]; }
    int   GetNumPoints()            { return points_.size(); }
    double GetArea()
    {
        if (area_ < 0 )  // Если пока что не вычислена
        {
            CalcArea();  // Вычисляем
        }
        return area_;
    }
private:
    void InvalidateArea() { area_ = -1; }
    void CalcArea()
    {
        area_ = 0;
        vector<Point>::iterator i;
        for( i = points_.begin(); i != points_.end(); ++i )
            area_ += /* некоторое вычисление */;
    }
    vector<Point> points_;
    double        area_;
};

Polygon operator+( Polygon& lhs, Polygon& rhs )
{
    Polygon ret = lhs;
    int last = rhs.GetNumPoints();
```

```
    for( int i = 0; i < last; ++i ) // Конкатенация
    {
        ret.AddPoint( rhs.GetPoint(i) );
    }
    return ret;
}

void f( const Polygon& poly )
{
    const_cast<Polygon&>(poly).AddPoint( Point(0,0) );
}

void g( Polygon& const rPoly )
        { rPoly.AddPoint( Point(1,1) ); }

void h( Polygon* const rPoly )
        { rPoly->AddPoint( Point(2,2) ); }

int main()
{
    Polygon poly;
    const Polygon cpoly;
    f(poly);
    f(cpoly);
    g(poly);
    h(&poly);
}
```

Решение

Когда я формулировал данную задачу, я обнаружил, что большинство программистов относятся к проблеме `const` как к очень простой, и не придают ей особого значения. Однако на самом деле здесь немало своих тонкостей, и эта задача заслуживает не меньшего внимания, чем другие.

```
class Polygon
{
public:
    Polygon() : area_(-1) {}
    void AddPoint( const Point pt ) { InvalidateArea();
                                      points_.push_back(pt); }
```

1. Объект `Point` передается по значению, так что в объявлении его как `const` мало толку. Фактически с точки зрения компилятора сигнатура функции с параметром, передаваемым по значению, не зависит от наличия перед параметром ключевого слова `const`. Например:

```
int f( int );
int f( const int );    // Повторное объявление f(int).
                       // Здесь нет перегрузки; это
                       // одна и та же функция.

int g( int& );
int g( const int& );   // Эта функция отлична от g(int&).
                       // Функция g перегружена.
```

Рекомендация

Избегайте передачи параметров по значению как `const` в объявлении функций. Если вы не хотите, чтобы эти параметры могли изменяться, используйте для них описание `const` в определении функции.

```
Point GetPoint( const int i )   { return points_[i]; }
```

2. То же замечание относительно типа параметра. Обычно передача `const` по значению не имеет смысла и в лучшем случае запутывает программиста.
3. Поскольку эта функция не изменяет состояния объекта, она должна быть константной функцией-членом.
4. Если ваша функция возвращает не встроенный тип (такой как `int` или `long`), то обычно по значению должны возвращаться константные объекты.[1] Это обеспечит генерацию компилятором сообщения об ошибке в случае попытки изменения временного объекта со стороны вызывающей функцией.

   Например, пользователь может написать нечто вроде "`poly.GetPoint(i) = Point(2,2);`". Чтобы этот код работал так, как задумывал его автор, следует использовать возврат по ссылке. Что касается описания самой функции как `const`, то для этого есть еще одна причина, о которой мы узнаем чуть позже: функция `GetPoint()` должна быть применима к объектам типа `const Polygon` в `operator+()`.

Рекомендация

При возврате из функции объектов не встроенных типов по значению желательно возвращать константные объекты.

```
int   GetNumPoints()            { return points_.size(); }
```

5. Данная функция должна быть описана как `const`.

   Кстати, данная функция — хороший пример того, где не следует использовать возврат по значению константного объекта. Возврат типа "`const int`" в данном случае некорректен, потому что а) ничего не дает вам, поскольку возвращаемый `int` представляет собой rvalue и, соответственно, не может быть изменен; и б) возврат "`const int`" может препятствовать настройке шаблонов и только запутывать программиста.

```
double GetArea()
{
    if (area_ < 0 )  // Если пока что не вычислена
    {
        CalcArea();  // Вычисляем
    }
    return area_;
}
```

6. Несмотря на то, что данная функция изменяет внутреннее состояние объекта, ее следует описать как `const`. Почему? Потому что эта функция не изменяет видимое состояние объекта. Здесь мы применяем определенное кэширование, однако это всего лишь особенность внутренней реализации, и логически объект остается константным, несмотря на неконстантность физическую.

   Вывод: член данных `area_` следует объявить как `mutable`. Если ваш компилятор пока что не поддерживает описание `mutable`, можно воспользоваться применением к `area_` приведения `const_cast`, обязательно отметив в комментарии, что как только `mutable` станет доступно, приведение `const_cast` надо будет удалить. Но функцию `GetArea()` в любом случае следует описать как `const`.

[1] Должен сказать, что далеко не все согласны с моим мнением. В [Lakos96] автор выступает против использования `const`, указывая на избыточность использования `const` со встроенными типами, что может мешать настройке шаблонов.

```
private:
    void InvalidateArea() { area_ = -1; }
```

7. Эта функция также должна быть const-функцией — единственно по причине согласованности. Принимая во внимание, что она должна вызываться только из неconst функций (поскольку ее предназначение — сделать недействительным кэшированное значение area_ при изменении состояния), семантически это неконстантная функция, но хороший стиль программирования предусматривает согласованность и ясность, так что мы вынуждены сделать константной и эту функцию.

```
void CalcArea()
{
    area_ = 0;
    vector<Point>::iterator i;
    for( i = points_.begin(); i != points_.end(); ++i )
        area_ += /* некоторое вычисление */;
}
```

8. Эта функция определенно должна быть const-функцией, поскольку она не изменяет видимое снаружи состояние объекта. Заметим также, что данная функция вызывается из другой константной функции-члена, а именно из GetArea().

9. Итератор не должен изменять состояние коллекции points_, так что нам следует использовать const_iterator.

```
    vector<Point> points_;
    double        area_;
};

Polygon operator+( Polygon& lhs, Polygon& rhs )
{
```

10. Конечно же, параметры должны передаваться как ссылки на константные объекты.

11. Возвращаемый по значению объект должен быть константным.

```
    Polygon ret = lhs;
    int last = rhs.GetNumPoints();
```

12. Поскольку значение last не должно изменяться, его тип следует сделать “const int”.

```
    for( int i = 0; i < last; ++i ) // Конкатенация
    {
        ret.AddPoint( rhs.GetPoint(i) );
    }
    return ret;
}
```

(Сделав rhs ссылкой на константный объект, мы получили еще одну причину сделать константной функцию GetPoint().)

```
void f( const Polygon& poly )
{
    const_cast<Polygon&>(poly).AddPoint( Point(0,0) );
}
```

Вот и ответ на вопрос на засыпку: использование результата приведения const_cast дает неопределенный результат, если ссылочный объект был объявлен как константный, как в случае с вызовом f(cpoly) ниже. Параметр на самом деле не является константным, так что не следует объявлять его таким образом — это попросту ложь.

```
void g( Polygon& const rPoly )
        { rPoly.AddPoint( Point(1,1) ); }
```

13. Этот `const` не только бесполезен, поскольку ссылки всегда являются константными (в том смысле, что они не могут поменять свое значение и начать указывать на другой объект), но и просто некорректен.

```
void h( Polygon* const rPoly )
        { rPoly->AddPoint( Point(2,2) ); }
```

14. Данный `const` тоже бесполезен, но уже по иной причине: поскольку указатель мы передаем по значению, такое объявление столь же бессмысленно, как и рассмотренное ранее "`const int`". Подумайте сами: в данном случае `const` просто обещает, что вы не станете изменять значение указателя. Это не самое полезное обещание, поскольку вызывающей функции нет до этого никакого дела.

(Если в поисках ответа на вопрос на засыпку вы решите, что данная функция не будет компилироваться, вы не правы. Она совершенно корректна с точки зрения C++. Вот если расположить `const` слева от `*`, то в таком случае тело функции действительно станет некорректным.)

```
int main()
{
    Polygon poly;
    const Polygon cpoly;
    f(poly);
```

Все в порядке.

```
f(cpoly);
```

Здесь мы получаем неопределенный результат, так как `f()` пытается выполнить приведение типа константного параметра и модифицировать его.

```
g(poly);
```

Все в порядке.

```
    h(&poly);
}
```

Здесь тоже все в норме.

`const` и `mutable` — ваши друзья

Мне часто приходится выслушивать жалобы на то, что кроме головной боли `const` не дает ничего. "Я уже замучился с этим `const`! Стоит поставить его в одном месте, как мне приходится использовать его везде! А ведь у других программистов (и даже в некоторых из используемых мною библиотек!) не встретишь и двух `const` на программу — и ничего, все отлично работает. Так зачем мне мучиться?" — такие речи далеко не редкость.

Представим себе примерно такую ситуацию: "Я уже замучился с этим предохранителем на моем кольте! Все время приходится то снимать оружие с предохранителя, то ставить на предохранитель... В конце концов, куча парней таскает свои кольты, не ставя их на предохранитель, и далеко не все из них отстрелили себе ногу!"

Небрежно обращающийся с оружием недолго протянет в этом мире. Как и строитель без каски, электрик без изоляции, программист без `const`... Нет никаких оправданий тому, кто пренебрегает элементарными средствами безопасности. (Кстати, интересный факт: обычно программисты, пренебрегающие `const`, не обращают внимания и на предупреждения компилятора. Очень показательно!)

Использование `const` там, где это возможно, делает код безопаснее и яснее, позволяя документировать интерфейсы и инварианты гораздо более эффективно, чем комментарии типа `/* я обещаю не изменять эту переменную */`. По сути обещание с использованием `const` гораздо более серьезно, поскольку за его выполнением тщатель-

но следит компилятор. Кроме того, описание const помогает компилятору создавать более быстрый и компактный код с высокой степенью оптимизации.

Не забывайте, что использование описания mutable является ключевой частью корректной работы с const. Если ваш класс содержит член, который может изменяться даже в константном объекте, сделайте его mutable. Тем самым вы облегчите написание константных функций-членов, а пользователи вашего класса смогут корректно создавать и использовать константные и неконстантные объекты вашего класса.

Да, интерфейсы не всех коммерческих библиотек корректны в плане использования const, но это не оправдывает написание вами некорректного с точки зрения const кода (кстати, работа с такими библиотеками — одна из причин применения в ваших программах приведения const_cast). Чужая лень — не оправдание собственной лени.

Далее приведен код с исправленными ошибками, связанными с const (все прочие недостатки остались нетронуты).

```
class Polygon
{
public:
    Polygon() : area_(-1) {}

    void AddPoint( Point pt )  { InvalidateArea();
                                 points_.push_back(pt); }
    const Point GetPoint( const int i ) const
                               { return points_[i]; }
    int   GetNumPoints() const { return points_.size(); }

    double GetArea() const
    {
        if (area_ < 0 )  // Если пока что не вычислена
        {
            CalcArea();  // Вычисляем
        }
        return area_;
    }
private:
    void InvalidateArea() const { area_ = -1; }

    void CalcArea() const
    {
        area_ = 0;
        vector<Point>::const_iterator i;
        for( i = points_.begin(); i != points_.end(); ++i )
            area_ += /* некоторое вычисление */;
    }
    vector<Point>  points_;
    mutable double area_;
};

const Polygon operator+( const Polygon& lhs,
                         const Polygon& rhs )
{
    Polygon ret = lhs;
    const int last = rhs.GetNumPoints();
    for( int i = 0; i < last; ++i ) // Конкатенация
    {
        ret.AddPoint( rhs.GetPoint(i) );
    }
    return ret;
}

void f( Polygon& poly )
{
    poly.AddPoint( Point(0,0) );
```

```
}

void g( Polygon& rPoly ) { rPoly.AddPoint( Point(1,1) ); }

void h( Polygon* rPoly ) { rPoly->AddPoint( Point(2,2) ); }

int main()
{
    Polygon poly;
    f(poly);
    g(poly);
    h(&poly);
}
```

Задача 10.4. Приведения *Сложность: 6*

Насколько хорошо вы знаете приведения типов в C++? Их использование может существенно повысить надежность вашего кода.

Приведения в новом стиле в стандарте C++ более мощны и безопасны, чем приведения в старом стиле (в стиле C). Насколько хорошо вы знакомы с новыми приведениями? Далее в задаче используются следующие классы и глобальные переменные.

```
class A { public: virtual ~A(); /* ... */ };
A::~A() {}

class B : private virtual A  { /* ... */ };

class C : public A           { /* ... */ };

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

A a1; B b1; C c1; D d1;

const A  a2;
const A& ra1 = a1;
const A& ra2 = a2;
char c;
```

В задаче требуется ответить на четыре вопроса.

1. Какие из следующих приведений в новом стиле *не* эквивалентны приведениям в стиле C?

   ```
   const_cast
   dynamic_cast
   reinterpret_cast
   static_cast
   ```

2. Для каждого из следующих приведений в стиле C укажите эквивалентное приведение в новом стиле. Какие из приведений некорректны, если их не переписать в новом стиле?

   ```
   void f()
   {
       A* pa; B* pb; C* pc;
       pa = (A*)&ra1;
       pa = (A*)&a2;
       pb = (B*)&c1;
       pc = (C*)&d1;
   }
   ```

3. Рассмотрите каждое из приведений в следующем фрагменте кода с точки зрения его стиля и корректности.

```
void g()
{
    unsigned char* puc = static_cast<unsigned char*>(&c);
    signed char*   psc = static_cast<signed char*>(&c);

    void* pv  = static_cast<void*>(&b1);
    B*    pb1 = static_cast<B*>(pv);

    B*    pb2 = static_cast<B*>(&b1);

    A*    pa1 = const_cast<A*>(&ra1);
    A*    pa2 = const_cast<A*>(&ra2);

    B*    pb3 = dynamic_cast<B*>(&c1);

    A*    pa3 = dynamic_cast<A*>(&b1);

    B*    pb4 = static_cast<B*>(&d1);

    D*    pd  = static_cast<D*>(&pb4);

    pa1 = dynamic_cast<A*>(pb2);
    pa1 = dynamic_cast<A*>(pb4);

    C* pc1 = dynamic_cast<C*>(pb4);
    C& rc1 = dynamic_cast<C&>(*pb2);
}
```

4. Почему обычно бесполезно использование приведения `const_cast` для преобразования не-`const` в `const`? Приведите пример, где такое приведение вполне корректно и обоснованно.

Решение

1. Какие из следующих приведений в новом стиле *не* эквивалентны приведениям в стиле C?

Только `dynamic_cast` не имеет эквивалентного приведения в стиле C. Все остальные приведения имеют эквиваленты в стиле C.

Рекомендация

Предпочтительно использование приведений в новом стиле.

2. Для каждого из следующих приведений в стиле C укажите эквивалентное приведение в новом стиле. Какие из приведений некорректны, если их не переписать в новом стиле?

```
void f()
{
    A* pa; B* pb; C* pc;
    pa = (A*)&ra1;
```

Вместо этого приведения воспользуйтесь `const_cast`: `pa = const_cast<A*>(&ra1);`.

```
pa = (A*)&a2;
```

Это выражение нельзя выразить с использованием приведений в новом стиле. Ближайший кандидат — `const_cast`, но поскольку `a2` является `const`-объектом, результат использования указателя не определен.

```
pb = (B*)&c1;
```

Вместо этого приведения воспользуйтесь `reinterpret_cast`:
`pb = reinterpret_cast<B*>(&c1);`.

```
    pc = (C*)&d1;
}
```

Это приведение в C некорректно. В C++ в данном случае никакого приведения не требуется: `pc = &d1;`.

3. Рассмотрите каждое из приведений в следующем фрагменте кода с точки зрения его стиля и корректности.

Сначала — общее замечание: все приведенные приведения `dynamic_cast` будут ошибочны, если классы не будут содержать виртуальных функций. К счастью, в нашем случае в `A` имеется виртуальная функция, так что все `dynamic_cast` в данном случае корректны с этой точки зрения.

```
void g()
{
    unsigned char* puc = static_cast<unsigned char*>(&c) ;
    signed char*   psc = static_cast<signed char*>(&c) ;
```

Ошибка: в обоих случаях следует использовать `reinterpret_cast`. На первый взгляд это может показаться удивительным, но причина этого в том, что `char`, `signed char` и `unsigned char` представляют собой три разные типа. Несмотря на наличие неявных преобразований между ними, эти типы не связаны друг с другом, а соответственно, не связаны и указатели на эти типы.

```
void* pv  = static_cast<void*> (&b1);
B*    pb1 = static_cast<B*>(pv);
```

Здесь все корректно, не считая того, что первое приведение не является необходимым в силу наличия неявного преобразования указателя на данные в тип `void*`.

```
B*    pb2 = static_cast<B*> (&b1);
```

Тут все в порядке, хотя данное приведение и излишне, поскольку аргумент имеет тип `B*`.

```
A*    pa1 = const_cast<A*>(&ra1);
```

Это приведение допустимо, но приведение типов `const` или `volatile` обычно говорит о плохом стиле. В большинстве случаев удаление константности указателя или ссылки связано с членами классов и решается с помощью ключевого слова `mutable` (см. также задачу 10.3, где говорится о корректности использования `const`).

Рекомендация

Избегайте приведения, отменяющего константность. Используйте вместо него ключевое слово `mutable`.

```
A*    pa2 = const_cast<A*>(&ra2) ;
```

Применение приведения в данном случае ошибочно, так как приводит к неопределенному поведению при записи посредством `pa2` в силу того, что `a2` является константным объектом. Для того чтобы понять суть проблемы, представьте, что компилятор в целях оптимизации разместил константный объект в памяти, доступной только для чтения. Опасность приведения с устранением константности таких объектов очевидна.

Рекомендация

Избегайте приведения, отменяющего константность.

```
B*    pb3 = dynamic_cast<B*>(&c1) ;
```

Потенциальная ошибка при попытке использования pb3 заключается в том, что поскольку c1 НЕ ЯВЛЯЕТСЯ B (класс C не является открытым производным от B классом; в действительности он вообще не является производным от B), значение pb3 окажется нулевым. Единственным корректным приведением в этом случае является reinterpret_cast, но его применение почти всегда связано с проблемами.

```
A*    pa3 = dynamic_cast<A*>(&b1) ;
```

Вероятная ошибка: поскольку b1 НЕ ЯВЛЯЕТСЯ A (в силу закрытого наследования), данное приведение некорректно, если только g() не является другом B.

```
B*    pb4 = static_cast<B*>(&d1);
```

Тут все в порядке, но данное приведение не является необходимым, поскольку преобразование указателя на производный класс в указатель на базовый класс может выполняться автоматически.

```
D*    pd  = static_cast<D*>(&pb4);
```

Это приведение корректно, и это может удивить тех, кто ожидает здесь приведения dynamic_cast. Причина этого в том, что понижающее приведение (из базового класса в производный) может быть статическим, если нам заранее известен целевой тип. Однако будьте осторожны: по сути вы говорите компьютеру, что вы знаете о том, на какой тип в действительности указывает данный указатель. Если вы ошибетесь, приведение не сможет информировать вас об ошибке (как это сделало бы приведение dynamic_cast, вернув нулевой указатель), и в лучшем случае вы получите немедленную ошибку времени выполнения.

Рекомендация

Избегайте понижающих приведений.

```
pa1 = dynamic_cast<A*>(pb2) ;
pa1 = dynamic_cast<A*>(pb4);
```

Эти два приведения выглядят очень похоже. Оба они используют dynamic_cast для преобразования B* в A*. Однако первое приведение некорректно, в то время как со вторым все в порядке.

Вот почему это так. Как упоминалось выше, вы не можете использовать dynamic_cast для преобразования указателя на то, что в действительности является объектом B, в указатель на A (а pb2 указывает на объект b1), поскольку B представляет собой закрытый, а не открытый производный от A класс. Второе же преобразование выполняется успешно, поскольку pb4 указывает на объект d1, а A представляет собой опосредованный базовый класс по отношению к D (через класс C); соответственно dynamic_cast может выполнить приведение по цепочке B*→D*→C*→A*.

```
C* pc1 = dynamic_cast<C*>(pb4);
```

Данное приведение также верно по только что описанной причине — dynamic_cast выполняет приведение по иерархии наследований.

```
    C& rc1 = dynamic_cast<C&>(*pb2) ;
}
```

И наконец, последняя, “исключительная” ошибка: так как `*pb2` в действительности не является `C`, `dynamic_cast` сгенерирует исключение `bad_cast` для сигнала о произошедшей ошибке. Почему? При попытке некорректного приведения указателей `dynamic_cast` возвращает нулевой указатель, но поскольку возврат нулевой ссылки невозможен, у `dynamic_cast` не остается другой возможности сообщить клиентскому коду о происшедшей ошибке, кроме как сгенерировать исключение.

4. Почему обычно бесполезно использование приведения `const_cast` для преобразования не-`const` в `const`?

Первые три вопроса не содержали примеры по использованию `const_cast` для добавления `const`, например, для преобразования указателя на неконстантный объект в указатель на константный. В конце концов, явное добавление `const` таким образом обычно избыточно, поскольку совершенно корректным является присваивание указателя на неконстантный объект указателю на константный. Таким образом, обычно приведение `const_cast` используется только с обратной целью.

И последняя часть вопроса: **приведите пример, где такое приведение вполне корректно и обоснованно.**

Имеется как минимум один случай, когда вы можете использовать данное приведение для преобразования неконстантной величины в константную, — для вызова определенной перегруженной функции или некоторой версии шаблона, например:

```
void f( T& );
void f( const T& );
template<class T> void g( T& t )
{
    f( t );                          // Вызов f( T& )
    f( const_cast<const T&>(t) );    // Вызов f( const T& )
}
```

Конечно, в случае выбора конкретной версии шаблона обычно проще указать ее явно, чем обеспечить корректное принятие решения компилятором на основе типов параметров. Например, проще явно написать `h<const T>(t)`, чем использовать запись `h(const_cast<const T&>(t))`.

Задача 10.5. `bool` *Сложность: 7*

Так ли нужен нам тип `bool`? Почему бы просто не эмулировать его имеющимися в языке средствами?

Помимо типа `wchar_t` (который в C представляет собой `typedef`), `bool` является единственным встроенным типом, добавленным в C++ со времен ARM [Ellis90].

Можно ли достичь того же эффекта без добавления встроенного типа? Если да, то покажите как. Если нет, то поясните, почему возможные реализации не ведут себя так же, как встроенный тип `bool`.

Решение

Ответ — нет, `bool` невозможно эмулировать полностью. Встроенный тип `bool` и зарезервированные ключевые слова `true` и `false` введены в C++ именно потому, что существующими средствами невозможно полностью их воспроизвести.

Данная задача призвана проиллюстрировать, о чем следует не забывать при проектировании собственных классов, перечислений и прочих инструментов.

Вторая часть вопроса задачи звучит так: **если нет, то поясните, почему возможные реализации не ведут себя так же, как встроенный тип `bool`**. Именно этим мы сейчас и займемся. Имеется четыре основных варианта реализации `bool`.

Вариант 1. typedef (оценка: 8.5/10)

Здесь используется инструкция типа “`typedef <нечто> bool;`”, например:

```
// Вариант 1.
//
typedef int bool;
const bool true  = 1;
const bool false = 0;
```

Это решение не худшее, но оно не допускает перегрузку с использованием типа `bool`, например:

```
// Файл f.h
void f( int );  // OK
void f( bool ); // OK, объявлена та же функция

// Файл f.cpp
void f( int )  {/* ... */}; // OK
void f( bool ) {/* ... */}; // Ошибка - это та же функция
```

Еще одна проблема заключается в том, что код наподобие приведенного ведет себя несколько неожиданно.

```
void f( bool b )
{
    assert( b != true && b != false );
}
```

Таким образом, первый вариант недостаточно хорош.

Вариант 2. #define (оценка: 0/10)

Здесь используется макроопределение типа “`#define bool <нечто>`”, например:

```
// Вариант 2.
//
#define bool  int
#define true  1
#define false 0
```

Это решение вообще никуда не годится. В нем не только присутствуют все недостатки варианта 1, но и все неприятности, связанные с применением директив `#define`. Представьте себе хотя бы мучения пользователя, у которого уже есть переменная по имени `false`...

Вариант 3. enum (оценка: 9/10)

Здесь используется перечисление типа “`enum bool;`”, например:

```
// Вариант 3.
//
enum bool { false, true };
```

Такое решение отчасти лучше, чем решение в первом варианте. Оно допускает перегрузку функций (основная проблема в первом варианте), однако не разрешает автоматического преобразования из условных выражений (что было возможно в первом варианте). Например, код

```
bool b;
b = (i == j);
```

не будет работоспособен, поскольку невозможно неявно преобразовать `int` в перечислимый тип.

Вариант 4. class (оценка: 9/10)

Давайте-ка вспомним, что C++ — язык объектно-ориентированный, и создадим класс для эмуляции bool.

```
class bool
{
public:
    bool();
    bool( int );            // Для преобразований
    bool& operator=( int ); // из условных выражений
    // operator int();      // Под вопросом
    // operator void*();    // Под вопросом
private:
    unsigned char b_;
};

const bool true ( 1 );
const bool false( 0 );
```

Операторы преобразования типов помечены как "*Под вопросом*" в силу следующих причин.

- Наличие преобразования bool в другие типы будет мешать разрешению перегрузки, как и в случае других классов, обладающих конструкторами преобразования типов, не объявленными как explicit, и/или операторами неявного преобразования типов. Особенно эта помеха проявляется при возможности автоматического преобразования в распространенные типы данных или из них. Дополнительную информацию о неявных преобразованиях вы можете найти в задачах 3.1 и 9.2.
- Отсутствие преобразования типа bool в тип наподобие int или void* приводит к тому, что объекты bool не могут естественным образом использоваться в условных выражениях, например:

  ```
  bool b;
  /* ... */
  if ( b )  // Ошибка при отсутствии автоматического
  {         // преобразования в тип наподобие int или void*
      /* ... */
  }
  ```

Итак, мы должны выбирать, какая именно возможность нам дороже, и соответственно включать или не включать в класс операторы преобразования типов. Однако никакой выбор не дает возможности полностью эмулировать встроенный тип bool. Резюмируем.

- typedef ... bool не позволяет использовать перегрузку.
- #define bool не позволяет использовать перегрузку и обладает всеми недостатками, присущими использованию макроопределений.
- enum bool разрешает перегрузку, но не обеспечивает автоматического преобразования в условных выражениях (как, например, в "b=(i==j);").
- class bool обеспечивает перегрузку, но не позволяет проверять значение объекта в условных выражениях (типа "if(b)"), если только класс не обеспечивает автоматическое приведение к типу наподобие int или void* (но в этом случае мы получаем все недостатки, присущие автоматическим преобразованиям типа).

И последнее — ни в одном случае (кроме, возможно, последнего варианта) мы не в состоянии указать, что условное выражение имеет тип bool. Итак, мы должны сделать вывод о том, что нам действительно необходим встроенный тип bool.

Задача 10.6. Пересылающие функции *Сложность: 3*

Каким образом следует создавать пересылающие функции? Оказывается, даже здесь есть свои тонкости.

Пересылающие функции (forwarding functions) — удобный инструмент для передачи работы другой функции или объекту, в особенности при эффективной передаче.

Рассмотрите следующую пересылающую функцию. Не хотите ли вы ее изменить? Если да, то как именно?

```
// Файл f.cpp
//
#include "f.h"
/* ... */
bool f( X x )
{
    return g( x );
}
```

Решение

Ключевым вопросом данной задачи, как сказано в ее условии, является эффективность. Имеется два улучшения, которые позволяют повысить эффективность данной функции.

1. *Передача параметра как константной ссылки (`const&`), а не по значению.*

"Неужели это не очевидно?" — можете спросить вы. В данном конкретном случае — нет. До 1997 года черновой вариант стандарта C++ гласил, что поскольку компилятор в состоянии определить, что параметр `x` не используется ни для каких иных целей, кроме передачи его в `g()`, то он в состоянии оптимизировать этот код, устранив излишнюю копию `x`. Например, если клиентский код выглядит как

```
X my_x;
f( my_x );
```

то использующий его компилятор может поступить одним из следующих способов.

- Создать копию `my_x` для использования в `f()` (это и есть параметр `x` в области видимости `f()`) и передать его функции `g()`.
- Передать `my_x` непосредственно в `g()` без создания копии вообще, так как известно, что копии могут использоваться исключительно для передачи в качестве параметра в функцию `g()`.

Последний способ весьма эффективен, не правда ли? Ведь это именно то, для чего и существуют оптимизирующие компиляторы?

Да и еще раз да — до Лондонской конференции в июле 1997 года. На этой конференции в черновой вариант были внесены поправки, ограничивающие количество ситуаций, когда компилятор может удалять "излишние" копии наподобие описанных. Внесение этих изменений было необходимо для устранения проблем, связанных с удалением вызовов конструкторов копирования, в особенности когда они обладали побочными действиями. Существуют случаи, когда код может обоснованно полагаться на количество реально созданных копий объекта.

На сегодня единственной ситуацией, когда компилятор может устранить конструктор копирования, является оптимизация возвращаемого значения и оптимизация временных объектов (детальную информацию об этом вы можете найти в руководствах по C++). Это означает, что в случае пересылающих функций типа `f()` компилятор

должен выполнить два копирования. Поскольку мы (как авторы `f()`) знаем, что в нашем случае такое копирование излишне, мы должны вернуться к нашему общему правилу и объявить параметр `x` как `const X&`.

Рекомендация

Предпочтительно использовать передачу объектов не по значению, а по ссылке, по возможности используя константные ссылки.

Примечание: если бы мы всегда следовали нашему общему правилу, не полагаясь на детальное знание того, что компилятор может сделать в некоторой конкретной ситуации, изменения в стандарте нас бы никак не затронули. Таким образом, данная ситуация может служить отличным примером того, что следует поступать наиболее простым способом, избегать неоправданных усложнений и не полагаться на особенности работы компилятора.

Рекомендация

Избегайте "темных закутков" языка программирования. Среди методов, эффективно решающих поставленную задачу, выбирайте простейший.

2. *Использовать встроенную функцию.* К этому следует подходить критически, и все функции по умолчанию не должны быть встроенными. Только после индивидуального рассмотрения и твердой уверенности в выигрыше вследствие этого действия, можно делать отдельные функции встроенными.

Рекомендация

Избегайте тонкой настройки производительности вообще и использования встроенности в частности до тех пор, пока необходимость этого не будет доказана с использованием профайлера.

Положительный эффект от использования встроенности состоит в том, что вы избегаете накладных расходов на лишние вызовы `f()`.

Отрицательный эффект состоит в том, что вы вынуждены раскрыть внутреннюю реализацию `f()` и делаете клиентский код зависящим от нее, так что при изменении этой функции следует перекомпилировать весь клиентский код. Кроме того, что ничуть не лучше, клиентскому коду требуется по меньшей мере прототип функции `g()`, что весьма неприятно, так как клиентский код не вызывает `g()` непосредственно и, вероятно, не нуждался в ее прототипе до объявления функции `f()` встроенной. Кроме того, если `g()` будет изменена и изменятся типы ее параметров, клиентский код будет зависеть от объявлений соответствующих классов.

Выбирать или нет использование встроенных функций, зависит от вас. И тот, и другой способы вполне корректны, и конкретный выбор зависит от конкретных функций `f()` и `g()`.

Задача 10.7. Поток управления — *Сложность: 6*

Насколько хорошо вы знаете порядок выполнения кода C++? Проверьте ваши знания, решив данную задачу.

Укажите по возможности большее число проблем в следующем (несколько надуманном) коде, сосредотачивая свое внимание на ошибках, связанных с потоком управления.

```
#include <cassert>
#include <iostream>
#include <typeinfo>
#include <string>
using namespace std;

// Следующие строки взяты из других заголовочных файлов
//
char * itoa( int value, char* workArea, int radix );
extern int fileIdCounter;

// Вспомогательный инструментарий для автоматизации
// проверки инварианта класса
template<class T>
    inline void AAssert( T& p )
{
    static int localFileId = ++fileIdCounter;
    if ( !p.Invariant() )
    {
        cerr << "Invariant failed: file " << localFileId
             << ", "   << typeid(p).name()
             << " at " << static_cast<void*>(&p) << endl;
        assert( false );
    }
}

template<class T>
class AInvariant
{
public:
    AInvariant( T& p ) : p_(p) { AAssert( p_ ); }
    ~AInvariant()              { AAssert( p_ ); }
private:
    T&p_;
};

#define  AINVARIANT_GUARD AInvariant<AIType> \
         invariantChecker( *this )

//--------------------------------------------------
template<class T>
class Array : private ArrayBase, public Container
{
    typedef Array AIType;
public:
    Array( size_t startingSize = 10 )
        : Container( startingSize ),
          ArrayBase( Container::GetType() ),
          used_(0),
          size_(startingSize),
          buffer_(new T[size_])
    {
        AINVARIANT_GUARD;
    }

    void Resize( size_t newSize )
    {
        AINVARIANT_GUARD;
        T* oldBuffer = buffer_;
        buffer_ = new T[newSize];
        copy( oldBuffer, oldBuffer + min(size_,newSize),
              buffer_ );
        delete [] oldBuffer;
        size_ = newSize;
    }
```

```
    string PrintSizes()
    {
        AINVARIANT_GUARD;
        char buf[30];
        return string("size = ") + itoa(size_,buf,10) +
            ", used = " + itoa(used_,buf,10);
    }

    bool Invariant()
    {
        if ( used_ > 0.9*size_ ) Resize( 2*size_ );
        return used_ <= size_;
    }

private:
    T*     buffer_;
    size_t used_, size_;
};

int f( int& x, int y = x ) { return x += y; }
int g( int& x )            { return x /= 2; }

int main( int, char*[] )
{
    int i = 42;
    cout << "f(" << i << ") = " << f(i) << ", "
         << "g(" << i << ") = " << g(i) << endl;
    Array<char> a(20);
    cout << a.PrintSizes() << endl;
}
```

Решение

Рассмотрим приведенный код строка за строкой.

```
#include <cassert>
#include <iostream>
#include <typeinfo>
#include <string>
using namespace std;

// Следующие строки взяты из других заголовочных файлов
//
char * itoa( int value, char* workArea, int radix );
extern int fileIdCounter;
```

Наличие глобальной переменной всегда должно беспокоить нас с той точки зрения, чтобы клиент не использовал ее до ее инициализации. Порядок инициализации глобальных переменных (включая статические члены классов) из различных единиц трансляции не определен.

Рекомендация

Избегайте использования глобальных или статических объектов. Если вы вынуждены использовать такие объекты, не забывайте о порядке их инициализации.

```
// Вспомогательный инструментарий для автоматизации
// проверки инварианта класса
template<class T>
    inline void AAssert( T& p )
{
    static int localFileId = ++fileIdCounter ;
```

Здесь мы сталкиваемся именно с такой, только что описанной ситуацией. Пусть, например, определение `fileIdCounter` выглядит следующим образом.

```
int fileIdCounter = InitFileId(); // Начальное значение 100
```

Если компилятор инициализирует `fileIdCounter` до инициализации любого из `AAssert<T>::localFileId`, то все в порядке, и `localFileId` получит ожидаемое значение. В противном случае устанавливаемое значение будет основываться на значении `fileIdCounter` до его инициализации, которое равно 0 для встроенных типов, и в результате значения `localFileId`, которые должны быть больше 100, окажутся меньше ожидаемого значения.

```
    if ( !p.Invariant() )
    {
        cerr << "Invariant failed: file " << localFileId
             << ", "   << typeid(p).name()
             << " at " << static_cast<void*>(&p) << endl;
        assert( false );
    }
}

template<class T>
class AInvariant
{
public:
    AInvariant( T& p ) : p_(p) { AAssert( p_ ); }
    ~AInvariant()              { AAssert( p_ ); }
private:
    T&p_;
};

#define  AINVARIANT_GUARD AInvariant<AIType> \
         invariantChecker( *this )
```

Здесь показана интересная идея, каким образом любой клиентский класс может автоматизировать проверку своего инварианта до и после вызова функции простым определением синонима `AIType` для самого себя и записи “`AINVARIANT_GUARD;`” в первой строке функций-членов.

Однако в приведенном ниже коде данная идея, к сожалению, не дает ожидаемого результата. Дело в том, что `AInvariant` скрывает вызов `assert()`, который будет автоматически удален при построении программы не в отладочном режиме. Этот код, похоже, был написан программистом, который не сильно беспокоился об этом.

```
//-------------------------------------------------
template<class T>
class Array : private ArrayBase, public Container
{
    typedef Array AIType;
public:
    Array( size_t startingSize = 10 )
        : Container( startingSize ),
          ArrayBase( Container::GetType() ),
```

Этот список инициализации конструктора содержит две потенциальные ошибки. Первая не обязательно является ошибкой, но вполне может сбить с толку.

1. Если `GetType()` является статической функцией-членом, или функцией-членом, который не использует `this` (т.е. не использует члены данных), и не опирается на какие-либо побочные действия конструктора (например, на какой-либо статический счетчик), то, несмотря на явный дурной стиль, этот конструктор будет работать корректно.

2. В противном случае (в основном, когда `GetType()` является обычной нестатической функцией-членом) у нас возникают проблемы. Невиртуальные базовые классы инициализируются в порядке их перечисления в описании класса слева направо, так что `ArrayBase` инициализируется до `Container`. Это означает, что мы пытаемся использовать член еще не инициализированного базового подобъекта `Container`.

Рекомендация

В списке инициализации конструктора всегда располагайте базовые классы в том же порядке, в котором они перечислены в определении класса.

```
used_(0),
size_(startingSize),
buffer_(new T[size_])
```

Это серьезная ошибка, поскольку на самом деле переменные инициализируются в том порядке, в котором они перечислены в определении класса.

```
buffer_(new T[size_])
used_(0),
size_(startingSize),
```

Такая запись делает ошибку очевидной. Вызов `new[]` возвращает буфер непредсказуемого размера — обычно нулевого или чрезвычайно большого, в зависимости от того, инициализирует ли компилятор память нулевыми значениями перед вызовом конструктора. В любом случае вряд ли мы получим буфер размером `startingSize` байт.

Рекомендация

В списке инициализации конструктора всегда перечисляйте члены данных в том же порядке, в котором они перечислены в определении класса.

```
{
   AINVARIANT_GUARD;
}
```

Тут возникает небольшой вопрос об эффективности: функция `Invariant()` вызывается дважды, один раз в конструкторе и второй — в деструкторе скрытого временного объекта. Впрочем, это мелочь, которой вряд ли стоит уделять особое внимание.

```
void Resize( size_t newSize )
{
    AINVARIANT_GUARD;
    T* oldBuffer = buffer_;
    buffer_ = new T[newSize];
    copy( oldBuffer, oldBuffer + min(size_,newSize),
          buffer_ );
    delete [] oldBuffer;
    size_ = newSize;
}
```

Здесь имеется проблема, связанная с потоком управления. Перед тем как читать дальше, еще раз посмотрите на эту функцию и постарайтесь обнаружить эту (довольно очевидную) проблему.

Ответ заключается в том, что данная функция небезопасна в смысле исключений. Если вызов `new[]` генерирует исключение `bad_alloc`, ничего плохого не происходит, и в данном случае не наблюдается никаких утечек. Но если исключение генерируется оператором копирующего присваивания `T` (в процессе операции `copy()`), то текущий объект оказывается в некорректном состоянии, и происходит утечка исходного буфера в силу потери всех указателей на него.

Цель данной функции — показать, как мало программистов заботятся о написании безопасного в смысле исключений кода.

Рекомендация

Всегда прилагайте максимум усилий для написания безопасного с точки зрения исключений кода. Всегда структурируйте код так, чтобы при наличии исключений происходило корректное освобождение ресурсов, а данные находились в согласованном состоянии.

```
string PrintSizes()
{
    AINVARIANT_GUARD;
    char buf[30];
    return string("size = ") + itoa(size_,buf,10)  +
        ", used = " + itoa(used_,buf,10) ;
}
```

Функция `itoa()` использует передаваемый ей буфер как рабочую область. В связи с этим в приведенном фрагменте возникает проблема, так как невозможно предсказать порядок вычисления выражения: порядок вычисления параметров функции не определен и зависит от конкретной реализации.

Вот что в действительности означает последняя инструкция (с учетом того, что `operator+()` левоассоциативен).

```
return
    operator+(
        operator+(
            operator+( string("size = "),
                       itoa(size_,buf,10) ),
            ", used = " ),
        itoa(used_,buf,10) );
```

Пусть, например, `size_` равно 10, а `used_` равно 5. Тогда, если первым во внешнем операторе + вычисляется первый параметр, то мы получим корректный вывод “`size = 10, used = 5`”, так как результат первого вызова `itoa()` сохраняется во временной строке до того, как второй вызов `itoa()` воспользуется тем же буфером. Если же первым вычисляется второй параметр внешнего оператора +, то вывод окажется некорректен (будет выведено “`size = 10, used = 10`”), поскольку второй вызов `itoa()` уничтожит результат первого вызова до того, как он будет использован.

Распространенная ошибка

Никогда не пишите код, зависящий от порядка вычисления аргументов функции.

```
bool Invariant()
{
    if ( used_ > 0.9*size_ ) Resize( 2*size_ ) ;
    return used_ <= size_;
}
```

Вызов `Resize()` создает две проблемы.

1. В нашем случае программа никогда не будет работать, поскольку если условие истинно, то будет вызвана функция `Resize()`, которая немедленно вызовет `Invariant()`, которая обнаружит, что проверяемое условие истинно и тут же вызовет `Resize()`, которая... — ну, вы поняли смысл происходящего.
2. Что если с целью повысить эффективность автор `AAssert()` примет решение убрать сообщение об ошибке и просто напишет “`assert(p->Invariant());`”? В этом случае клиентский код помещает в вызов `assert()` код с побочным

действием. Это означает, что поведение программы будет существенно отличаться при компиляции для отладки и при компиляции окончательной версии, когда отладка отключена. Даже если бы не было первой проблемы, такое решение привело бы к неприятностям, так как означало бы, что объекты `Array` изменяют размер в разное время, в зависимости от режима компиляции. Это превратило бы попытки воспроизвести возникающие у пользователей проблемы в пытки, так как отладочная версия и версия для пользователей отличались бы своей функциональностью и свойствами образа памяти во время выполнения.

Вывод: никогда не пишите код с побочным действием внутри вызова `assert()` (или подобном ему), а кроме того, всегда убеждайтесь, что у вас нигде нет бесконечной рекурсии.

```
private:
    T*     buffer_;
    size_t used_, size_;
};

int f( int& x, int y = x ) { return x += y; }
```

Второй параметр по умолчанию некорректен и не может использоваться в C++ ни в коем случае, так что эта строка не должна компилироваться компилятором, соответствующим стандарту. Далее для продолжения рассмотрения будем считать, что значение `y` по умолчанию равно 1.

```
int g( int& x )            { return x /= 2; }

int main( int, char*[] )
{
    int i = 42;
    cout << "f(" << i << ") = " << f(i) << ", "
         << "g(" << i << ") = " << g(i) << endl;
```

Здесь мы вновь сталкиваемся с проблемой порядка вычисления параметров. Поскольку порядок вызовов `f(i)` и `g(i)` неизвестен, выводимый результат может оказаться совершенно неверным. Одним из примеров может служить MSVC, выводящий строку “`f(22) = 22, g(21) = 21`”, говорящую о том, что компилятор вычисляет аргументы функций справа налево.

Но насколько ошибочен полученный результат? Компилятор прав, как будет прав и компилятор, который приведет к совершенно другому результату. Все дело в том, что в своем коде программист основывался на чем-то, что не определено в C++.

Распространенная ошибка

Никогда не пишите код, зависящий от порядка вычисления аргументов функции.

```
    Array<char> a(20);
    cout << a.PrintSizes() << endl;
}
```

Здесь, само собой, мы ожидаем вывода строки “`size = 20, used = 0`”, но из-за уже рассматривавшейся ошибки в функции `PrintSizes()` ряд компиляторов строят программы, выводящие строку “`size = 20, used = 20`”, некорректность которой очевидна.

Задача 10.8. Предварительные объявления *Сложность: 3*

Предварительные объявления предоставляют возможность удалить излишние зависимости времени компиляции. В этой задаче вы встретитесь с ловушками при применении предварительных объявлений. Сумеете ли вы обойти их?

1. Предварительные объявления — очень полезный инструмент. Однако в приведенном далее фрагменте они не работают так, как того ожидал программист. В чем тут дело?

```
// Файл f.h
//
class ostream;    // Ошибка
class string;     // Ошибка
string f( const ostream& );
```

2. Напишите корректные предварительные объявления `string` и `ostream` без включения каких-либо дополнительных файлов.

Решение

1. **Предварительные объявления — очень полезный инструмент. Однако в приведенном далее фрагменте они не работают так, как того ожидал программист. В чем тут дело?**

```
// Файл f.h
//
class ostream;    // Ошибка
class string;     // Ошибка
string f( const ostream& );
```

Увы, использовать предварительные объявления `ostream` и `string` нельзя, поскольку они не являются классами. Это синонимы шаблонов, определенные с помощью инструкций `typedef`.

(На самом деле вы могли использовать такие предварительные объявления в “достандартном” C++, но это было много лет назад. В стандартном языке это уже недопустимо.)

2. **Напишите корректные предварительные объявления `string` и `ostream` без включения каких-либо дополнительных файлов.**

Краткий ответ: это невозможно. Не существует стандартного и переносимого способа предварительного объявления `ostream` без включения дополнительного файла (то же относится и к `string`).

Причина, по которой мы не можем это сделать, заключается в том, что в соответствии со стандартом мы не имеем права писать собственные объявления в пространстве имен `std`, где и находятся `ostream` и `string`.

> *Добавление объявлений или определений в `namespace std` или в пространства имен внутри `namespace std` приводит к неопределенным результатам.*

Кроме прочего это правило призвано обеспечить возможность производителям реализовать стандартную библиотеку, у шаблонов которой большее количество параметров, чем это требуется стандартом (конечно, со значениями по умолчанию для обеспечения совместимости). Даже если бы нам было позволено использовать предварительные объявления шаблонов и классов стандартной библиотеки, мы бы не могли делать это переносимо, так как производители могут по-разному расширять эти объявления от реализации к реализации.

Лучшее, что вы можете сделать (хоть это и не удовлетворяет условию “без включения каких-либо дополнительных файлов”), — это добавить в программу следующие строки.

```
#include <iosfwd>
#include <string>
```

Включение этих файлов — это все, что вы можете сделать переносимо. К счастью, предварительные объявления `string` и `ostream` не так часто требуются на практике,

поскольку они невелики и широко используемы. То же самое можно сказать и о большинстве стандартных заголовочных файлов. Однако опасайтесь ловушек и не пытайтесь использовать предварительные объявления шаблонов или чего-либо другого, принадлежащего пространству имен `std`. Эта высокая привилегия принадлежит только разработчикам компилятора и стандартной библиотеки, и только им.

Рекомендация

Никогда не используйте директиву включения заголовочного файла `#include` там, где достаточно предварительного объявления. Кроме того, когда вы работаете с классами потоков, но вам не требуется их полное определение, предпочтительно использовать директиву `#include <iosfwd>`.

См. также задачу 4.1.

Задача 10.9. `typedef` ***Сложность: 3***

Зачем мы используем `typedef`? Кроме традиционных причин, в этой задаче мы рассмотрим технологии использования `typedef`, которые делают стандартную библиотеку C++ более простой и безопасной.

1. Почему мы используем `typedef`? Приведите как можно большее количество причин.
2. Почему использование `typedef` так широко распространено в коде, работающем со стандартными контейнерами?

Решение

typedef: управление именами типов

Вспомним, для чего вообще используется `typedef`. Эта инструкция позволяет нам дать другое, функционально эквивалентное имя некоторому типу, например, инструкция

```
typedef vector< vector<int> > IntMatrix;
```

позволяет в дальнейшем вместо громоздкой записи `vector< vector<int> >` пользоваться коротким именем типа `IntMatrix`.

1. Почему мы используем `typedef`? Приведите как можно большее количество причин.

Вот несколько основных преимуществ использования `typedef`.

Краткость

Короткие имена легче вводить.

Удобочитаемость

`typedef` может сделать код (в особенности с длинными именами шаблонных типов) гораздо более удобочитаемым. В качестве простого примера ответьте на вопрос, что означает следующий код?

```
int ( *t(int) )( int *);
```

Согласитесь, что с использованием `typedef`, пусть даже с таким малосодержательным именем, как `Func`, ситуация значительно проясняется.

```
typedef int (*Func)( int* );

Func t( int );
```

Теперь становится совершенно очевидно, что это — объявление функции с именем `t`, которая получает целочисленный аргумент и возвращает указатель на функцию, которая получает в качестве параметра указатель на `int` и возвращает целое число типа `int`.

Кроме того, `typedef` позволяет добавить в код семантическую значимость. Например, гораздо проще понять предназначение “`PhoneBook`”, чем “`map<string,string>`”, в котором гораздо меньше семантического содержания, да и применяться этот тип может почти для чего угодно.

Сообщения

`typedef` может служить для сообщений. Например, вместо того, чтобы использовать в программе огромное количество переменных типа `int`, им можно назначить значимые имена типов. Сравните фрагмент

```
int x;
int y;
y = x * 3;   // Может, так и надо - как знать?
```

с фрагментом, в котором типы имеют следующие имена.

```
typedef int Inches;
typedef int Dollars;

Inches x;
Dollars y;
y = x * 3;   // Непохоже, чтобы так и было задумано...
```

С каким из фрагментов вы предпочли бы иметь дело, будучи программистом, сопровождающим программу?

Переносимость

Если вы используете синонимы типов для специфичных для данной платформы или непереносимых по иной причине имен, вы существенно облегчите себе перенос на новую платформу. Гораздо проще написать код наподобие приведенного, чем использовать контекстную замену во всем коде всех имен, специфичных для данной системы (о причинах, по которым может оказаться неудобным использование для этой цели `#define`, можно узнать из задачи 8.4).

```
#if defined USING_COMPILER_A
    typedef __int32 Int32;
    typedef __int64 Int64;
#elif defined USING_COMPILER_B
    typedef int       Int32;
    typedef long long Int64;
#endif
```

2. **Почему использование `typedef` так широко распространено в коде, работающем со стандартными контейнерами?**

Гибкость

Изменение одного определения `typedef` существенно проще, чем выполнение изменения всех использований данного типа во всей программе. Рассмотрим, например, следующий код.

```
void f( vector<Customer>& custs )
{
    vector<Customer>::iterator i = custs.begin();
    ...
}
```

Но что если несколькими месяцами спустя мы обнаружим, что `vector` — контейнер, не совсем подходящий для наших целей? При хранении огромного количества объектов `Customer` тот факт, что используемая контейнером `vector` память представляет собой непрерывный блок,[2] является недостатком, и нам может потребоваться использовать дек вместо вектора. А может быть, мы обнаружим, что вставка и удаление объектов часто выполняются в средине списка, и нам лучше воспользоваться особенностями контейнера `list`.

В приведенном выше коде мы должны найти и заменить все вхождения имени типа `vector<Customer>`. Но насколько же проще оказалась бы наша задача, если бы исходный код выглядел иначе

```
typedef vector<Customer> Customers;
typedef Customers::iterator CustIter;

...

void f( Customers& custs )
{
    CustIter i = custs.begin();
    ...
}
```

и нам пришлось бы только заменить в инструкциях `typedef` один тип другим. Конечно, не всегда дела обстоят так просто, например, наш код может использовать тот факт, что итератор `Customers::iterator` — итератор произвольного доступа, а итератор `list<Customer>::iterator` таковым не является. Однако использование `typedef` спасает нас от необходимости внесения массовых изменений в исходный код программы.

Использование свойств

Идиома свойств представляет собой мощный способ связи информации с типом, и если вы настраиваете работу стандартных контейнеров или алгоритмов, то вы столкнетесь с необходимостью использовать свойства. (Об использовании свойств рассказывается в разделе “Обобщенное программирование и стандартная библиотека C++”; особое внимание обратите на задачи 1.2 и 1.3, в которых мы используем замену класса свойств `char_traits`.)

Резюме

Большинство причин использования `typedef` попадает в одну из рассмотренных категорий. В целом применение `typedef` делает код более простым в написании, чтении и изменении, путем введения пресловутого “дополнительного уровня косвенности” — косвенности в используемых именах типов.

Задача 10.10. Пространства имен. Часть 1 ***Сложность: 2***

Стандарт C++ включает поддержку пространств имен и управления видимостью имен с помощью объявлений и директив `using`. Эта задача представляет собой небольшое повторение материала, подготавливая вас к встрече со следующей задачей.

[2] См. задачу 1.14.

Что такое объявления и директивы using и как они используются? Приведите примеры. Что из них зависит от порядка использования?

Решение

Хотя в процессе стандартизации в C++ и были внесены большие изменения, только немногие из них приводят к неработоспособности существующего кода. Однако есть одно новшество, которое приводит к тому, что практически весь существующий код перестает компилироваться, — это добавление пространств имен и, в частности, тот факт, что вся стандартная библиотека C++ теперь находится в пространстве имен std.

Если вы программист на C++ и это новшество еще не коснулось вас, вероятно, это ненадолго, так как новейшие версии многих компиляторов уже соответствуют этому измененному стандарту.

Объявления using

Объявление using создает локальный синоним имени, объявленного в другом пространстве имен. С точки зрения перегрузки и разрешения имен такое объявление работает так же, как и любое другое.

```
// Пример 1а
//
namespace A
{
    int f( int );
    int i;
}

using A::f;

int f( int );

int main()
{
    f( 1 );     // Неоднозначность: A::f() или ::f() ?

    int i;
    using A::i; // Ошибка: двойное объявление (как если
                // бы вы написали "int i" еще раз
}
```

Объявления using вводят только те имена, чьи объявления уже видны, например.

```
// Пример 1б
//
namespace A
{
    class X {};
    int Y();
    int f( double );
}

using A::X;
using A::Y; // Только функция A::Y(), но не класс A::Y
using A::f; // Только A::f(double), но не A::f(int)

namespace A
{
    class Y {};
    int f( int );
```

```
}

int main()
{
    X x;    // Все в порядке, X - синоним A::X

    Y y;    // Ошибка - класс A::Y не видим, так как
            // объявление using предшествует объявлению
            // самого класса

    f( 1 ); // Здесь выполняется неявное преобразование и
            // вызывается A::f(double), а не A::f(int),
            // поскольку к моменту объявления using A::f
            // видимо только первое объявление - т.е.
            // объявление функции A::f(double)
}
```

Это свойство делает объявления `using` зависящими от порядка использования, в частности, при использовании пространства имен, распределенного между группой заголовочных файлов. Порядок включения заголовочных файлов, особенно то, какие заголовочные файлы включаются до объявлений `using`, а какие после, влияет на имена, вносимые этими объявлениями в область видимости. Само собой, под определение "распределенного между группой заголовочных файлов" попадает и пространство имен `std`, о чем мы поговорим подробнее в следующей задаче.

Директивы using

Директива `using` позволяет использовать все имена из другого пространства имен в области видимости этой директивы. В отличие от объявления `using` директива вносит в область видимости все имена, объявленные как до, *так и после* директивы. Само собой, что объявление имени должно находиться до его использования, как в приведенном примере.

```
// Пример 1в
//
namespace A
{
    class X {};
    int f( double );
}

using namespace A;

void f()
{
    X x;    // OK, X - синоним A::X
    Y y;    // Ошибка, класс Y еще не видим
    f( 1 ); // OK, вызов A::f(double)
}

namespace A
{
    class Y {};
    int f( int );
}

int main()
{
    X x;    // OK, X - синоним A::X
    Y y;    // OK, Y - синоним A::Y
    f( 1 ); // OK, вызов A::f(int)
}
```

Задача 10.11. Пространства имен. Часть 2 ***Сложность: 4***

Как наилучшим образом использовать возможности C++ по управлению пространствами имен и избежать при этом ошибок и ловушек? Как наиболее эффективно перенести существующий код на C++ для работы с компилятором и библиотекой, поддерживающими пространства имен?

Вы работаете над проектом в несколько миллионов строк кода с более чем тысячью заголовочных файлов и файлов с исходными текстами программ. Работающая над проектом команда переходит на последнюю версию компилятора, которая поддерживает пространства имен, а вся стандартная библиотека размещена в пространстве имен `std` (либо ваш компилятор и до этого поддерживал пространства имен, но вы использовали старые заголовочные файлы, такие как `<vector.h>`). К сожалению, побочное действие такого перехода — неработоспособность всего вашего кода. Как обычно, на детальный пересмотр всего кода и внесение необходимых объявлений `using` нет времени, несмотря на то, что это наиболее корректный путь.

Как наиболее эффективно решить эту проблему и безопасно перенести ваш код в новое (и более стандартное) окружение? Рассмотрите разные варианты и выберите наиболее быстрый путь, который не снизит безопасность и используемость вашего кода. Каким образом можно отложить работу по переносу так, чтобы это не привело в будущем к резкому увеличению количества этой работы?

Решение

Безопасный и эффективный переход к пространствам имен

Продемонстрированная ситуация достаточно типична для проектов, переносимых для работы с компиляторами, поддерживающими пространства имен. Основная проблема заключается в том, что стандартная библиотека теперь находится в пространстве имен `std`, и использование имен из стандартной библиотеки без квалификатора `std::` будет приводить к ошибке компиляции.

Так, например, код, выглядевший ранее как

```
// Пример 1
//
#include <iostream.h>  // "Достандартный" заголовок

int main()
{
    cout << "hello, world" << endl;
}
```

теперь для корректной работы должен либо указать, какие имена принадлежат пространству имен `std`

```
// Пример 2a
//
#include <iostream>

int main()
{
    std::cout << "hello, world" << std::endl;
}
```

либо использовать объявления using для внесения необходимых имен в область видимости

```
// Пример 2б
//
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    cout << "hello, world" << endl;
}
```

либо применить директиву using для внесения в область видимости всех имен из пространства имен std

```
// Пример 2в
//
#include <iostream>

int main()
{
    using namespace std;   // Это же можно сделать
                           // в области видимости файла
    cout << "hello, world" << endl;
}
```

либо использовать некоторую комбинацию из приведенных выше приемов.

Так какой же способ наиболее предпочтителен, если в дальнейшем мы планируем всегда использовать пространства имен? И какой способ обеспечит наиболее быстрый перенос старого кода, позволяя отложить серьезную работу по переносу на будущее, причем не увеличивая ожидающий нас объем работы?

Рекомендации для долгосрочного решения

Первое, что надо решить, — это каким должен быть корректный способ долгосрочного использования пространств имен. Знание ответа на этот вопрос поможет нам определить, к чему следует стремиться при разработке краткосрочной стратегии переноса.

Вкратце можно сказать, что корректное долгосрочное решение должно следовать как минимум следующим правилам.

Правило №1. Не используйте директивы using в заголовочных файлах

Причина использования данного правила заключается в том, что использование директив using приводит к необоснованному засорению пространства имен огромным количеством имен, подавляющее большинство которых обычно не нужно для работы программы. Наличие излишних имен существенно увеличивает риск возникновения конфликтов, причем не только в самом заголовочном файле, но и в каждом модуле, включающем данный файл. Директивы using в заголовочном файле представляются мне мародерствующей армией свихнувшихся варваров, разрушающих и уничтожающих все на своем пути; одно их присутствие вызывает “непреднамеренные конфликты”, даже если вы считаете их своими союзниками.

Правило №2. Не используйте объявления using в заголовочных файлах

(Уточним: здесь речь идет об объявлении using в смысле пространств имен; вы можете свободно использовать объявление using для внесения при необходимости в область видимости членов базового класса.)

Это правило может вас удивить, поскольку большинство авторов рекомендуют не использовать объявления `using` *в области видимости файла* в совместно используемых заголовочных файлах. Это хороший совет, поскольку в области видимости файла объявления `using` приводят, как и в предыдущем случае, к засорению (хотя и меньшему) пространства имен.

Однако, на мой взгляд, этот совет недостаточен. Вы вообще не должны использовать объявления `using` в заголовочных файлах, даже в области видимости пространства имен. Значение объявления `using` может изменяться в зависимости от того, какие другие заголовочные файлы включаются в программу перед данным. Этот тип непредсказуемости не имеет права на существование, так как может сделать программу совершенно неработоспособной. Рассмотрим одну из возможных ситуаций: если заголовочный файл содержит определение встроенной функции `f()` (пусть, например, `f()` — шаблон или часть шаблона) и некоторый код внутри `f()` зависит от порядка директив `#include`, то вся программа оказывается некорректной в силу нарушения правила одного определения (One Definition Rule, ODR), которое требует, чтобы определение любой сущности (в нашем случае `f()`) было идентично в каждом транслируемом модуле, использующем данную сущность. Из-за нарушения ODR мы получаем непредсказуемое поведение программы. Но даже если ODR окажется не нарушенным, непредсказуемость не приводит ни к чему хорошему, и я покажу это немного позже, в примере 3*в*.

Правило №3. В файлах реализации объявления и директивы using не должны располагаться перед директивами #include

Все объявления и директивы `using` должны идти *после* директив включения файлов `#include`. В противном случае имеется вероятность того, что использование `using` изменит семантику заголовочного файла из-за введения неожиданных имен.

Зато после директив включения файлов `#include` у вас полная свобода действий по использованию объявлений и директив `using`, поскольку ни к каким потенциальным побочным действиям на другие файлы реализации это не приведет. Объявления и директивы `using` существуют только для вашего удобства. Их использование вполне оправданно (если это не приводит к коллизиям имен, то почему бы не использовать *все* имена?) постольку, поскольку они никак не влияют ни на заголовочные файлы, ни на другие модули трансляции.

Правило №4. Используйте новые заголовочные файлы вместо старых

Это правило не так важно с практической точки зрения, и я упоминаю его для полноты картины. Для обеспечения обратной совместимости с C язык программирования C++ поддерживает все стандартные имена заголовочных файлов C (например, `stdio.h`), и при включении этих версий заголовочных файлов библиотечные функции, как и раньше, находятся в глобальной области видимости. Но в то же время указывается, что использование старых заголовочных файлов не приветствуется и что в будущем они могут быть удалены из стандартного C++. Таким образом, стандарт C++ требует от программистов использования новых заголовочных файлов, имена которых начинаются с “`c`” и не имеют расширения “`.h`”, например `cstdio`. При включении заголовочных файлов с использованием новых имен вы получаете те же библиотечные функции, но теперь они находятся в пространстве имен `std`.

Почему я сказал, что это правило не так уж важно? Во-первых, несмотря на то, что использование заголовочных файлов типа “`name.h`” официально не рекомендовано, вы можете смело поставить свои карманные деньги на то, что на самом деле они никогда не исчезнут. Говоря честно и строго между нами, даже если в какой-то прекрасный день они и будут удалены из стандарта, производители стандартной библиотеки все равно оставят старые заголовочные файлы в качестве расширения. Почему?

Потому что слишком много пользователей используют старые заголовочные файлы в своих программах. Во-вторых, у вас могут быть причины для использования заголовочных файлов типа “name.h” для соответствия другим стандартам, таким как Posix, которые требуют добавления новых заголовочных файлов к старым. Если вы используете такие стандарты, теоретически безопаснее работать со старыми именами заголовочных файлов, даже если производители стандартной библиотеки и предоставляют новые заголовочные файлы в стиле “cname”.

Долгосрочный подход: поясняющий пример

Для иллюстрации вышеупомянутых правил рассмотрим приведенный пример и два варианта переноса данного модуля для работы с пространствами имен.

```
// Пример 3а. Исходный код без пространств имен
//

// -------- Файл x.h --------
#include "y.h"   // Определяет Y
#include <deque.h>
#include <iosfwd.h>

ostream& operator<<( ostream&, const Y& );
Y operator+( const Y&, int );
int f( const deque<int>& );

// ------- Файл x.cpp -------
#include "x.h"
#include "z.h"   // Определяет Z
#include <ostream.h>

ostream& operator<<( ostream& o, const Y& y )
{
    // ... в реализации используется Z ...
    return o;
}

Y operator+( const Y& y, int i )
{
    // ... в реализации используется другой operator+() ...
    return result;
}

int f( const deque<int>& d )
{
    // ...
}
```

Как наилучшим образом перенести этот код для работы с компилятором, который соответствует требованиям стандарта относительно пространств имен, и стандартные имена библиотеки которого находятся в пространстве имен std? Перед тем как читать дальше, остановитесь на минутку и подумайте о разных возможных подходах к решению этой задачи. Какие из них хороши, а какие плохи, и почему именно?

У вас уже есть решение? Отлично. Имеется множество подходов, которые можно применить, но я опишу только два из них.

Хорошее долгосрочное решение

Хорошее долгосрочное решение состоит в явном использовании полностью квалифицированных имен в заголовочном файле и применении для удобства объявлений using в файле с исходным текстом (поскольку эти имена скорее всего широко используются во всем файле).

```
// Пример 3б. Хорошее долгосрочное решение
//

// -------- Файл x.h --------
#include "y.h"   // Определяет Y
#include <deque>
#include <iosfwd>

std::ostream& operator<<( std::ostream&, const Y& );
Y operator+( const Y&, int );
int f( const std::deque<int>& );

// ------- Файл x.cpp -------
#include "x.h"
#include "z.h"   // Определяет Z
#include <ostream>

using std::deque;      // "using" размещаются после
using std::ostream;    // заголовочных файлов
using std::operator+;
// Можно просто использовать "using namespace std"

ostream& operator<<( ostream& o, const Y& y )
{
    // ... в реализации используется Z ...
    return o;
}

Y operator+( const Y& y, int i )
{
    // ... в реализации используется другой operator+() ...
    return result;
}

int f( const deque<int>& d )
{
    // ...
}
```

Заметим, что объявления `using` в `x.cpp` следует размещать после всех директив `#include`, поскольку в противном случае они могут внести конфликты имен в другие заголовочные файлы. В частности, использование `operator+()` в `x.h` может привести к неоднозначности в зависимости от того, какие другие функции `operator+()` видимы (а это в свою очередь зависит от того, какие стандартные заголовочные файлы включены до или после каждого из объявлений `using`).

Мне иногда встречаются программисты, которые даже в `.cpp`-файлах используют только полностью квалифицированные имена. Я не рекомендую поступать таким образом, поскольку это означает огромное количество дополнительной работы, не дающей никаких реальных преимуществ.

Не самое лучшее долгосрочное решение

В противоположность описанному методу использовать одно часто рекомендуемое решение на самом деле просто опасно. Это “решение” предполагает внесение всего кода проекта в ваше собственное пространство имен (что не вызывает возражений, хотя и не настолько полезно, как можно подумать) и внесение объявлений или директив `using` в заголовочные файлы (что по сути широко распахивает дверь перед возможными проблемами). Причина популярности данного метода в том, что он требует внесения меньшего количества изменений, чем другие методы переноса.

```
// Пример 3б. Плохое долгосрочное решение
//            (почему никогда не следует использовать
//             объявления using в заголовочных файлах,
//             даже в пределах пространства имен)

// -------- Файл x.h --------
#include "y.h"   // Определяет MyProject::Y и добавляет
                 // объявления/директивы using в
                 // пространство имен MyProject
#include <deque>
#include <iosfwd>

namespace MyProject
{
    using std::deque;
    using std::ostream;
    // Возможно, "using namespace std;"

    ostream& operator<<( ostream&, const Y& );
    Y operator+( const Y&, int );
    int f( const deque<int>& );
}

// ------- Файл x.cpp -------
#include "x.h"
#include "z.h"   // Определяет MyProject::Z и добавляет
                 // объявления/директивы using в
                 // пространство имен MyProject

// Ошибка: потенциальная неоднозначность имен в объявлениях
// в файле z.h в зависимости от того, какие именно
// объявления using имеются в заголовочных файлах,
// включенных до z.h (в этом случае x.h и y.h могут вызвать
// потенциальные изменения смысла объявлений).

#include <ostream>

namespace MyProject
{
    using std::operator+;

    ostream& operator<<( ostream& o, const Y& y )
    {
        // ... в реализации используется Z ...
        return o;
    }

    Y operator+( const Y& y, int i )
    {
        // В реализации используется другой operator+()
        return result;
    }

    int f( const deque<int>& d )
    {
        // ...
    }
}
```

Обратите внимание на указанную ошибку. Причина ее, как указано, заключается в том, что значение объявления `using` в заголовочном файле может измениться (даже если объявление `using` находится в пространстве имен, а не в области видимости файла) в зависимости от того, какие другие клиентские модули включены до него (не говоря уже об изменениях при переходе от одной реализации стандартной библиотеки к другой).

Эффективное краткосрочное решение

Причина, по которой я потратил столько времени на описание долгосрочных решений, заключается в том, что знание того, чего мы хотим достичь в конечном итоге, поможет нам выбрать то простейшее краткосрочное решение, которое позволит нам достичь в будущем желаемого долгосрочного решения, а не будет ему противоречить. Таким образом, теперь мы достаточно подкованы теоретически для ответа на вопрос о том, каков наиболее эффективный путь переноса существующего кода в систему с использованием пространств имен.

Вообще говоря, переход на новую версию компилятора — обычная задача из списка выполняемых в процессе жизненного цикла продукта. Обновление компилятора, пожалуй, не самая неотложная из задач этого списка, и есть множество других вещей с более высоким приоритетом, которые должны быть сделаны. Соответственно, встает задача обеспечить работоспособность, отложив при этом излишнюю работу по переносу на будущее (при этом не увеличивая общий объем работы, которую придется выполнить позже).

Начнем с переноса заголовочных файлов.

Шаг №1. Добавление std:: в каждый заголовочный файл

"Действительно ли необходимо добавление квалификатора `std::` ко всем именам из пространства имен `std`? — можете спросить вы. — Может быть, проще будет добавить объявления или директивы `using` в заголовочный файл?" Конечно, добавление объявлений или директив `using` требует меньшего количества работы, но эту работу при переходе к долгосрочному решению придется переделывать. Кроме того, рассмотренные ранее правила 1 и 2 обоснованно отвергают такой подход, и единственным решением является добавление во всех нужных местах заголовочных файлов квалификатора `std::`.

Эту работу можно выполнить довольно быстро, просто заменив с помощью редактора во всех заголовочных файлах `string` на `std::string` (аналогично — `vector` на `std::vector` и т.д.). Будьте внимательны и случайно не замените имена из других пространств имен (возможна, например, ситуация, когда в некоторой библиотеке стороннего производителя имеется собственный нестандартный класс `string`; такие ситуации следует разрешать вручную).

Шаг №2. Создание и включение файла с директивой using namespace std; во все файлы реализации

На втором шаге мы создаем новый заголовочный файл (например, `myproject_last.h`), который содержит директиву `using namespace std;`, и включаем его в каждый файл реализации после всех остальных директив `#include`. Этим мы не нарушаем правило 1, поскольку это — специальный заголовочный файл. Это не обычный заголовочный файл, в котором объявляется нечто, что будет использовано позже, а всего лишь механизм, обеспечивающий внесение некоторого кода в файлы реализации в строго определенное место.

Этот способ имеет определенные преимущества перед помещением строки "`using namespace std;`" в каждый файл реализации. Во многих проектах уже имеется такой заголовочный файл, включаемый во все файлы реализации, так что все, что вам надо, — это дописать в него одну строку. Если же такого файла нет, то, создав его, позже вы сможете оценить все удобство работы с ним. В худшем случае внесение в каждый файл реализации строки "`using namespace std;`" требует того же количества работы, что и внесение строки "`#include "myproject_last.h"`".

И последнее замечание — о новых именах заголовочных файлов. К счастью, замена старых имен новыми не обязательна и может быть отложена на более поздний срок.

Вот результат применения описанной стратегии к примеру 3*а*.

```
// Пример 3г. Краткосрочное решение проблемы переноса
//

// -------- Файл x.h --------
#include "y.h"   // Определяет Y
#include <deque>
#include <iosfwd>

std::ostream& operator<<( std::ostream&, const Y& );
Y operator+( const Y&, int );
int f( const std::deque<int>& );

// ------- Файл x.cpp -------
#include "x.h"
#include "z.h"   // Определяет Z
#include <ostream>
#include "myproject_last.h" // После всех прочих #include

ostream& operator<<( ostream& o, const Y& y )
{
    // ... в реализации используется Z ...
    return o;
}

Y operator+( const Y& y, int i )
{
    // ... в реализации используется другой operator+() ...
    return result;
}

int f( const deque<int>& d )
{
    // ...
}

// -- Файл myproject_last.h --
//
using namespace std;
```

Это решение не ставит под угрозу долгосрочное решение в том смысле, что не требует отказа от уже внесенных в код изменений. В то же время оно существенно проще и требует внесения меньшего количества изменений в код, чем долгосрочное решение. По сути данное решение представляет минимальное количество работы, которую следует выполнить для переноса кода для работы с пространствами имен.

Заключение

Представим, что наступило счастливое время, когда на вас не давит срочная работа и вы можете взяться за долгосрочное решение проблемы переноса вашего кода (к которому было применено описанное выше краткосрочное решение). Для этого вы просто должны следовать следующей инструкции.

1. Закомментируйте директиву `using` в файле `myproject_last.h`.
2. Перекомпилируйте ваш проект, отмечая все ошибки компиляции. В каждом файле реализации добавьте необходимые объявления и/или директивы `using`, либо в области видимости файла (после всех директив `#include`), либо внутри каждой функции — на ваш вкус.

3. В каждом файле реализации можете заменить все заголовочные файлы C на новые (например, `stdio.h` на `cstdio`). Данный шаг не является обязательным, но легко выполняется с помощью какого-либо инструмента для поиска и замены текста.

Если вы последуете данным советам, то легко и быстро сможете перенести ваш код для работы с компилятором и библиотекой, использующими пространства имен.

Послесловие

Я надеюсь, что вам понравились не только задачи, но и их решения и использованные при этом технологии, большинство из которых вы можете тут же использовать в своих программах. Я надеюсь, что некоторые из прочитанных вами решений пригодятся вам в вашей повседневной работе прямо сегодня. Если это так, мои труды не пропали даром, и о большем я и не мечтаю.

Но работа продолжается, и на Web-узле *Guru of the Week* и в группе новостей *comp.lang.c++.moderated* и сегодня обсуждаются и публикуются новые задачи. В будущем они будут объединены в новую, заключительную книгу из серии *Exceptional C++ — Exceptional C++ Style.* Вот краткий перечень того, о чем вы сможете прочесть в ней.

- Новая информация на темы, освещенные в данной книге, включая обобщенное программирование и работу с шаблонами, проектирование классов, вопросы эффективности и оптимизации. Будут рассмотрены вопросы эффективного использования стандартной библиотеки, включая анализ того, какие ресурсы и какое количество памяти используют различные контейнеры на практике (а не только в теории) в настоящее время на различных платформах.
- Практическая информация о реальных задачах, такая как написание хорошо отлаживаемого кода, эффективные форматы данных и вопросы упаковки.
- Обсуждение вопросов управления памятью и их влияние на ваш код на разных платформах (даже при использовании компиляторов, полностью соответствующих стандарту).
- И что самое важное — изучение программ “реального мира”, разбор промышленного кода и опубликованных программ.

Еще раз спасибо всем читателям за ваш выбор. Я надеюсь, что материал книги будет полезен вам в вашей повседневной работе, и вы сможете писать более быстрые, ясные и безопасные программы на C++.

Список литературы

Alexandrescu00a Alexandrescu A. *Traits on Steroids*, C++ Report 12(6), June 2000.

Alexandrescu00b Alexandrescu A. *Mapping Between Types and Values*, C/C++ User Journal C++ Experts Forum, 18(10), October 2000. (`http://www.gotw.ca/publications/mxc++/aa_mappings.htm`).

Alexandrescu01 Alexandrescu A. *Modern C++ Design*, Addison-Wesley, 2001.

Barton94 Barton J., Nackman L. *Scientific and Engineering C++*, Addison-Wesley, 1994.

C++98 ISO/IEC 14882:1998(E), *Programming Languages — C++* (ISO and ANSI C++ standard).

Cargill92 Cargill T. *C++ Programming Style*, Addison-Wesley, 1994.

Cargill94 Cargill T. *Exception Handling: A False Sense of Security*, C++ Report, 9(6), Nov.-Dec. 1994.

Cline95 Cline M., Lomow G. *C++ FAQs*, Addison-Wesley, 1995.

Cline99 Cline M., Lomow G., Girou M. *C++ FAQs, Second Edition*, Addison Wesley Longman, 1999.

Coplien92 Coplien J. *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

Ellis90 Ellis M., Stroustrup B. *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

Gamma95 Gamma E., Helm R., Johnson R., Vissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

GotW Sutter H. *Guru of the Week*. (`http://www.gotw.ca/gotw`.)

Henney00 Henney K. *From Mechanism to Method: Substitutability*, C++ Report, 12(5), May 2000. (`http://www.gotw.ca/publications/mxc++/kh_substitutability.htm`.)

Keffer95 Keffer T. *Rogue Wave C++ Design, Implementation, and Style Guide*, Rogue Wave Software, 1995.

Knuth97 Knuth D. *The Art of Computer Programming, Vol. 1, Fundamental Algorithms, 3d Edition*, Addison-Wesley, 1997.
Кнут Д.Э. *Искусство программирования, т.1. Основные алгоритмы, 3-е изд.* — М.: Изд. дом “Вильямс”, 2000.

Koenig97 Koenig A., Moo B. *Ruminations on C++*, Addison Wesley Longman, 1997.

Koenig99 Koenig A. *Changing Containers Iteratively*, C++ Report, 11(2), February 1999.

Lakos96 Lakos J. *Large-Scale C++ Software Design*, Addison-Wesley, 1996.

Lippman97 Lippman S. *C++ Gems*, SIGS / Cambridge University Press, 1997.

Lippman98 Lippman S., Lajoie J. *C++ Primer, Third Edition*, Addison Wesley Longman, 1998.

Liskov88	Liskov B. *Data Abstraction and Hierarchy*, SIGPLAN Notices, 23(5), May 1988.
Martin00	Martin R. *C++ Gems II*. SIGS / Cambridge University Press, 2000.
Martin95	Martin R. *Designing Object-Oriented Applications Using the Booch Method*, Prentice-Hall, 1995.
Meyers01	Meyers S. *Effective STL*, Addison-Wesley, 2001.
Meyers96	Meyers S. *More Effective C++*, Addison-Wesley, 1996.
Meyers97	Meyers S. *Effective C++, 2d Edition*, Addison-Wesley, 1997.
Meyers98	Meyers S. *Effective C++, Second Edition*, Addison-Wesley Longman, 1998.
Meyers98a	Meyers S. *Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, 1999.
Meyers99	Meyers S. *Effective C++ CD*, Addison Wesley Longman, 1999. (`http://www.gotw.ca/publications/xc++/sm_effective.htm`).
Murray93	Murray R. *C++ Strategies and Tactics*, Addison-Wesley, 1993.
Myers97	Myers N. *The Empty Base C++ Optimization*, Dr. Dobb's Journal, Aug. 1997.
Niec00	Niec T. *Optimizing Substring Operations in String Classes*, C/C++ Users Journal, 18(10), October 2000.
Stroustrup00	Stroustrup B. *The C++ Programming Language, Special Edition*, Addison-Wesley, 2000.
Stroustrup94	Stroustrup B. *The Design and Evolution of C++*, Addison-Wesley, 1994.
Stroustrup97	Stroustrup B.. *The C++ Programming Language, 3d Edition*, Addison-Wesley Longman, 1997. Страуструп Б. *Язык программирования C++, 3-е изд.* — Спб.; М.; "Невский диалект", "Издательство БИНОМ", 1999.
StroustrupFAQ	Stroustrup B. *C++ Style and Technique FAQ*. (`http://www.gotw.ca/publications/mxc++/bs_constraints.htm`.)
Sutter01	Sutter H. *Virtuality*, C/C++ Users Journal, 19(9), September 2001.
Sutter98	Sutter H. *C++ State of the Union*, C++ Report, 10(1), Jan. 1998.
Sutter98a	Sutter H. *Uses and Abuses of Inheritance, Part 1*, C++ Report, 10(9), Oct. 1998.
Sutter99	Sutter H. *Uses and Abuses of Inheritance, Part 2,* C++ Report, 11(1), Jan. 1999.

Предметный указатель

A

auto_ptr, *159*; *265*
  и безопасность исключений, *271*
  и контейнеры, *270*
  и массивы, *274*
  и члены класса, *271*; *281*

B

bool, *363*

C

calloc, *259*
const, *353*; *357*
const_cast, *361*; *363*

D

delete, *259*; *261*
delete[], *107*; *261*
deque, *73*

E

explicit, *176*

G

GLSP, *23*

I

inline, *34*; *293*

L

LSP, *23*; *170*

M

malloc, *259*
mutable, *357*

N

new, *259*; *262*
  размещающий *119*
new[], *262*

R

realloc, *259*

S

static_cast, *52*

T

Try-блоки функций, *142*
typedef, *375*; *376*
typename, *62*

U

using, *378*
  директива, *379*; *381*
  объявление, *378*; *381*

V

vector, *73*; *263*; *278*; *298*

Б

Безопасность исключений, *31*; *338*; *346*
  потоков, *309*
Брандмауэр компиляции, *223*

В

Взаимоотношение
  быть частью, *243*; *245*
  использует, *226*
  работает как, *187*
  работать подобно, *23*
  реализован посредством, *121*; *187*
  содержит, *226*

является, *170*; *186*
является почти, *199*
Включение, *194*
Временные объекты, *176*
Выравнивание, *231*; *234*

Д

Деструктор, *107*; *118*
виртуальный, *182*
генерация исключений, *130*; *151*

З

Закрытое наследование, *194*
Замещение, *183*

И

Идиома
const auto_ptr, *273*
Fast Pimpl, *232*
handle/body, *223*; *228*
Pimpl, *223*; *267*
безопасности исключений, *110*
захвата ресурса при инициализации, *122*; *345*
свойства, *377*
скрытой реализации, *146*; *223*; *227*; *281*; *329*
создания временного объекта и обмена состояний, *133*; *164*; *337*
указателя на реализацию, *148*; *189*
Инициализация, *349*
глобальных переменных, *369*
Интеллектуальные указатели, *265*
члены класса, *282*
Исключения
в конструкторе, *142*
гарантии безопасности, *115*
и деструкторы, *130*; *151*
каноническая форма безопасности, *163*
спецификации *129*
Итератор *17*

К

Класс, *239*
виртуальный базовый, *343*
виртуальный деструктор, *182*; *213*
вложенный, *324*
друзья, *245*
интерфейс, *239*
свойств, *287*
ограничение, *55*
локальный, *325*
прокси, *323*
Конечный автомат, *320*
Конструктор
генерация исключений, *142*
копирования, *27*; *110*
по умолчанию, *106*
Контейнер
ассоциативный, *79*
прокси, *69*
Копирование при записи, *296*

М

Макрос, *330*; *333*
Массив, *74*; *278*
нулевой длины *275*
Многопоточность *309*
Множественное наследование, *201–11*
эмуляция, *205*

О

Обобщенный принцип подстановки Лисков, *23*
Объект, *340*
временный, *176*
время жизни, *341*
тождественность, *337*
функция, *45*; *326*
Оптимизация, *34*; *71*; *293–317*
и многопоточность, *309*
отложенное копирование, *296*

П

Память, *257*
куча, *258*
области памяти C++, *258*; *259*
стек *258*
Перегрузка, *183*; *340*
Поиск имен, *62*; *243*
Поиск К□нига, *238*; *242–43*
Порядок вычислений, *156*
Правило одного определения, *382*
Предварительное объявление, *373*
Предикат, *44*
с состояниями, *47*

Преобразование типов
неявное, *176*; *339*
Препроцессор, *330*; *333*
Приведение типов, *359*
Принцип интерфейса, *240*
подстановки Лисков, *23*; *226*
Пространство имен, *238*; *251*; *378*

С

Свойства *287*
Скрытая реализация, *223*
Сокрытие, *183*
имен, *250*
Специализация шаблона, *88*
Спецификация исключений, *282*
Срезка, *342*
Стек, *105*

У

Указатель, *169*
на функцию, *320*
Управление памятью, *232*

Ф

Функция
try-блоки, *142*
вложенная, *325*
встроенная, *367*
доступа, *295*
объект, *326*
пересылающая, *366*
указатель, *320*

Ш

Шаблон проектирования, *187*; *277*
Adapter, *276*
Decorator pattern, *214*
Factory, *332*
State pattern, *213*
Strategy pattern, *191*
Template Method, *188*

Научно-популярное издание

Герб Саттер

Решение сложных задач на C++

Серия *C++ In-Depth*, т. 4.

Литературный редактор *Е.П. Перестюк*
Верстка *О.В. Линник*
Художественный редактор *Е.П. Дынник*
Корректоры *Л.А. Гордиенко*, *Л.В. Коровкина*, *О.В. Мишутина*

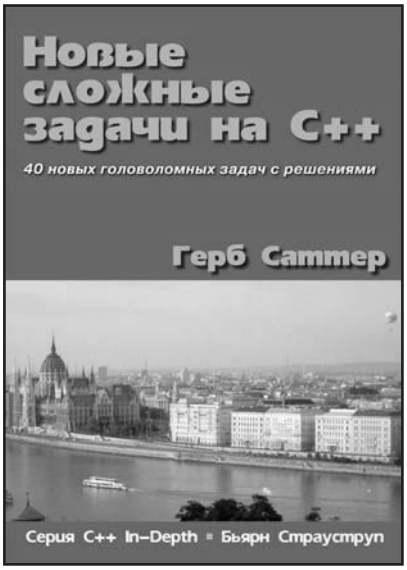
Издательский дом “Вильямс”
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 18.02.2008. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 32,3. Уч.-изд. л. 21,3.
Тираж 1000 экз. Заказ № 0000.

Отпечатано с диапозитивов в ФГУП “Печатный Двор”
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, С.-Петербург, Чкаловский пр., 15.

НОВЫЕ СЛОЖНЫЕ ЗАДАЧИ НА C++

Герб Саттер



www.williamspublishing.com

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.

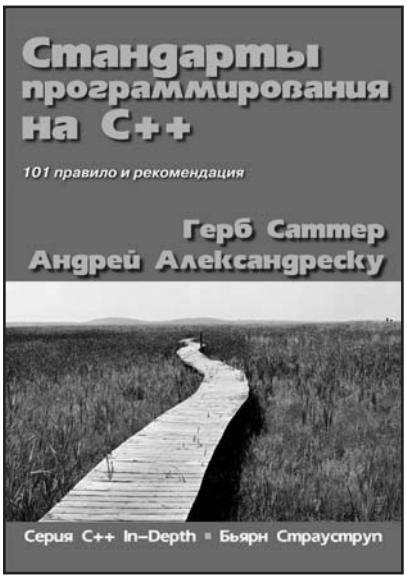
Книга рассчитана в первую очередь на профессиональных программистов с глубокими знаниями языка, однако она будет полезна любому, кто захочет углубить свои знания в данной области.

ISBN 978-5-8459-0823-0

в продаже

СТАНДАРТЫ ПРОГРАММИРОВАНИЕ НА C++

Герб Саттер
Андрей Александреску



www.williamspublishing.com

Эта книга поможет новичку стать профессионалом, так как в ней представлен сконцентрированный лучший опыт программистов на C++, обобщенный двумя экспертами мирового класса. Если вы — начинающий программист, эта книга для вас: вы сможете найти в ней простые и понятные рекомендации для ежедневного использования, подкрепленные примерами их конкретного применения на практике. Если вы — программист с опытом, эта книга для вас: в ней вы найдете как советы, до которых вы дошли самостоятельно, так и новые рекомендации, которые вы сможете по достоинству оценить и принять на вооружение. Если вы — программист-профессионал, эта книга для вас: вы сможете использовать ее как основу для разработки собственных стандартов кодирования для себя лично или для группы, которой вы руководите. Конечно, книга рассчитана в первую очередь на профессиональных программистов с глубокими знаниями языка, однако она будет полезна любому, кто захочет углубить свои знания в данной области.

ISBN 978-5-8459-0859-9 **в продаже**

СОВРЕМЕННОЕ ПРОЕКТИРОВАНИЕ НА C++

Андрей Александреску

www.williamspublishing.com

В книге изложена новая технология программирования, представляющая собой сплав обобщенного программирования, метапрограммирования шаблонов и объектно-ориентированного программирования на C++. Настраиваемые компоненты, созданные автором, высоко подняли уровень абстракции, наделив язык C++ чертами языка спецификации проектирования, сохранив всю его мощь и выразительность.
В книге изложены способы реализации основных шаблонов проектирования. Разработанные компоненты воплощены в библиотеке Loki, которую можно загрузить с Web-страницы автора. Книга предназначена для опытных программистов на C++.

ISBN 978-5-8459-0351-8 **в продаже**

C++ ДЛЯ ПРОФЕССИОНАЛОВ

Николас А. Солтер
Скотт Дж. Клепер

www.dialektika.com

В этом практическом руководстве с большим количеством примеров представлены все грани разработки C++-приложений, включая этапы проектирования, тестирования и отладки. Здесь описаны простые, но мощные методы, используемые C++-профессионалами, малознакомые, но весьма полезные средства и многократно применяемые шаблоны проектирования. В книге демонстрируются различные методики и хороший стиль программирования, а также предлагаются пути повышения качества кода и эффективности программирования в целом. Книга предназначена для программистов и разработчиков, которые хотят поднять свои навыки C++-программирования на профессиональный уровень.

ISBN 5-8459-1065-X

в продаже