Джефф Элджер

БИБЛИОТЕКА ПРОГРАММИСТА

C++

Содержание

БЛАГОДАРНОСТИ **9**
Извинения… или вроде того 9

ЧАСТЬ 1. ВВЕДЕНИЕ И КРАТКИЙ ОБЗОР **11**

ГЛАВА 1. ЗАЧЕМ НУЖНА ЕЩЕ ОДНА КНИГА О С++? **13**
Дао С++ 13
Три великие идеи С++ 15
Как читать эту книгу 16
Несколько слов о стиле программирования 17

ГЛАВА 2. СИНТАКСИС С++ **19**
Переменные и константы 19
const *19*
Стековые и динамические объекты *23*
Области действия и функции 25
Области действия *25*
Перегрузка *28*
Видимость *29*
Типы и операторы 33
Конструкторы *33*
Деструкторы *40*
Присваивание *41*
Перегрузка операторов *46*

ГЛАВА 3. ШАБЛОНЫ И БЕЗОПАСНОСТЬ ТИПОВ **55**
Что такое шаблоны и зачем они нужны? 55
Проблемы *55*
Обходные решения *56*
Шаблоны — усовершенствованные макросы *56*
Синтаксис шаблонов 57
Параметризованные типы *57*
Параметризованные функции *57*
Параметризованные функции классов *58*
Передача параметра *58*
Шаблоны с несколькими параметрами *59*
Долой вложенные параметризованные типы! *59*
Наследование *59*
Комбинации простых и параметризованных типов 59
Небезопасные типы в открытых базовых классах *60*
Небезопасные типы в закрытых базовых классах *60*
Небезопасные типы в переменных класса *60*

ГЛАВА 4. ИСКЛЮЧЕНИЯ **63**

ОБРАБОТКА ИСКЛЮЧЕНИЙ В СТАНДАРТЕ ANSI ... 63
Синтаксис инициирования исключений ... *63*
Синтаксис перехвата исключений ... *66*
Конструкторы и деструкторы ... *67*
НЕСТАНДАРТНАЯ ОБРАБОТКА ИСКЛЮЧЕНИЙ ... 69
УСЛОВНЫЕ ОБОЗНАЧЕНИЯ ... 69

ЧАСТЬ 2. КОСВЕННЫЕ ОБРАЩЕНИЯ ... 71

ГЛАВА 5. УМНЫЕ УКАЗАТЕЛИ ... 73

ГЛУПЫЕ УКАЗАТЕЛИ ... 73
УМНЫЕ УКАЗАТЕЛИ КАК ИДИОМА ... 75
Оператор -> ... *75*
Параметризованные умные указатели ... *75*
Иерархия умных указателей ... *76*
Арифметические операции с указателями ... *77*
Во что обходится умный указатель? ... *78*
ПРИМЕНЕНИЯ ... 78
Разыменование значения NULL ... *78*
Отладка и трассировка ... *80*
Кэширование ... *82*

ГЛАВА 6. ВЕДУЩИЕ УКАЗАТЕЛИ И ДЕСКРИПТОРЫ ... 85

СЕМАНТИКА ВЕДУЩИХ УКАЗАТЕЛЕЙ ... 85
Конструирование ... *86*
Уничтожение ... *87*
Копирование ... *87*
Присваивание ... *88*
Прототип шаблона ведущего указателя ... *89*
ДЕСКРИПТОРЫ В C++ ... 90
ЧТО ЖЕ ПОЛУЧАЕТСЯ? ... 90
Подсчет объектов ... *90*
Указатели только для чтения ... *92*
Указатели для чтения/записи ... *92*

ГЛАВА 7. ГРАНИ И ДРУГИЕ МУДРЫЕ УКАЗАТЕЛИ ... 93

ИНТЕРФЕЙСНЫЕ УКАЗАТЕЛИ ... 93
Дублирование интерфейса ... *93*
Маскировка указываемого объекта ... *94*
Изменение интерфейса ... *96*
ГРАНИ ... 96
Преобразование указываемого объекта в грань ... *97*
Кристаллы ... *98*
Вариации на тему граней ... *99*
Инкапсуляция указываемого объекта ... *102*
Проверка граней ... *103*
Обеспечение согласованности ... *103*
Грани и ведущие указатели ... *105*
ПЕРЕХОДНЫЕ ТИПЫ ... 106
Полиморфные указываемые объекты ... *106*
Выбор типа указываемого объекта во время конструирования ... *107*
Изменение указываемого объекта во время выполнения программы ... *107*
ПОСРЕДНИКИ ... 107
ФУНКТОРЫ ... 108

ГЛАВА 8. КОЛЛЕКЦИИ, КУРСОРЫ И ИТЕРАТОРЫ ... 111

МАССИВЫ И ОПЕРАТОР [] ... 111

Проверка границ и присваивание ... 111
Оператор [] с нецелыми аргументами ... 112
Имитация многомерных массивов ... 112
Множественные перегрузки оператора [] ... 113
Виртуальный оператор [] ... 113
Курсоры ... 114
Простой класс разреженного массива ... 114
Курсоры и разреженные массивы ... 115
Операторы преобразования и оператор -> ... 116
Что-то знакомое... ... 117
Итераторы ... 117
Активные итераторы ... 118
Пассивные итераторы ... 118
Что лучше? ... 119
Убогие, но распространенные варианты ... 119
Лучшие варианты ... 120
Итератор абстрактного массива ... 121
Операторы коллекций ... 123
Мудрые курсоры и надежность итераторов ... 124
Частные копии коллекций ... 126
Внутренние и внешние итераторы ... 127
Временная пометка ... 129
Пример ... 131

ГЛАВА 9. ТРАНЗАКЦИИ И ГЕНИАЛЬНЫЕ УКАЗАТЕЛИ ... 137

Тернистые пути дизайна ... 137
Транзакции ... 137
Отмена ... 138
Хватит? ... 138
Образы и указатели ... 138
Простой указатель образов ... 139
Стеки образов ... 140
Образы автоматических объектов ... 141
Образы указателей ... 144
Комбинации и вариации ... 145
Транзакции и отмена ... 145
Транзакции и блокировки ... 146
Класс ConstPtr ... 147
Класс LockPtr ... 149
Создание и уничтожение объектов ... 150
Упрощенное создание объектов ... 151
Отмена ... 152
Варианты ... 152
Вложенные блокировки ... 152
Взаимные блокировки и очереди ... 153
Многоуровневая отмена ... 154
Оптимизация объема ... 154
Несколько прощальных слов ... 155

ЧАСТЬ 3. СНОВА О ТИПАХ ... 157

ГЛАВА 10. МНОЖЕСТВЕННАЯ ПЕРЕДАЧА ... 159

Гомоморфные иерархии классов ... 159
Взаимозаменяемость производных классов ... 160
Нормальное наследование ... 160
Инкапсуляция производных классов ... 161
Множественная передача ... 162

Двойная передача ... *163*
Гетероморфная двойная передача ... *164*
Передача более высокого порядка ... *165*
Группировка передач и преобразования ... *166*
Это еще не все ... 167

ГЛАВА 11. ПРОИЗВОДЯЩИЕ ФУНКЦИИ И ОБЪЕКТЫ КЛАССОВ ... 169

Производящие функции ... 169
make-функции ... *170*
Символические классы и перегруженные make-функции ... *170*
Оптимизация с применением производящих функций ... *170*
Локализованное использование производящих функций ... *171*
Уничтожающие функции ... *172*
Снова о двойной передаче: промежуточные базовые классы ... *172*
Нет — конструкторам копий и оператору =! ... *173*
Объекты классов ... 173
Информация о классе ... *174*
Еще несколько слов об уничтожающих функциях ... *175*
Определение класса по объекту ... *176*
Представители ... 177

ГЛАВА 12. НЕВИДИМЫЕ УКАЗАТЕЛИ ... 179

Основные концепции ... 179
Инкапсуляция указателей и указываемых объектов ... *180*
Производящие функции ... *180*
Ссылки на указатели ... *181*
Неведущие указатели ... *181*
Ведущие указатели ... *183*
Снова о двойной передаче ... 184
Удвоенная двойная передача ... *185*
Самомодификация и переходимость ... *187*
Множественная двойная передача ... *189*
Применение невидимых указателей ... 189
Кэширование ... *189*
Распределенные объекты и посредники ... *189*
Нетривиальные распределенные архитектуры ... *189*

ЧАСТЬ 4. УПРАВЛЕНИЕ ПАМЯТЬЮ ... 191

ГЛАВА 13. ПЕРЕГРУЗКА ОПЕРАТОРОВ УПРАВЛЕНИЯ ПАМЯТЬЮ ... 193

Перегрузка операторов new и delete ... 193
Простой список свободной памяти ... *193*
Наследование операторов new и delete ... *196*
Аргументы оператора new ... *197*
Конструирование с разделением фаз ... *197*
Уничтожение с разделением фаз ... *198*
Кто управляет выделением памяти? ... 199
Глобальное управление ... *199*
Выделение и освобождение памяти в классах ... *200*
Управление памятью под руководством клиента ... *200*
Объекты классов и производящие функции ... *200*
Управление памятью с применением ведущих указателей ... *200*
Перспективы ... *204*

ГЛАВА 14. ОСНОВЫ УПРАВЛЕНИЯ ПАМЯТЬЮ ... 205

Строительные блоки ... 205
Поблочное освобождение памяти ... *205*

Скрытая информация ... *208*
Списки свободных блоков ... *208*
Подсчет ссылок ... 210
Базовый класс с подсчетом ссылок ... *210*
Укзатели с подсчетом ссылок ... *211*
Ведущие указатели с подсчетом ссылок ... *211*
Дескрипторы с подсчетом ссылок ... *212*
Трудности подсчета ссылок ... *213*
Подсчет ссылок и ведущие указатели ... *213*
Пространтсва памяти ... 214
Деление по классам ... *214*
Деление по размеру ... *215*
Деление по способу использования ... *215*
Деление по средствам доступа ... *215*
Пространства стека и кучи ... *216*

ГЛАВА 15. УПЛОТНЕНИЕ ПАМЯТИ ... 217

Поиск указателей ... 217
Мама, откуда берутся указатели? ... *217*
Поиск указателей ... *220*
Дескрипторы, повсюду дескрипторы ... 223
Общее описание архитектуры ... *223*
Ведущие указатели ... *223*
Вариации ... *227*
Оптимизация в особых ситуациях ... *229*
Алгоритм Бейкера ... 229
Пространства объектов ... *229*
Последовательное копирование ... *232*
Внешние объекты ... *233*
Алгоритм Бейкера: уход и кормление в C++ ... *234*
Уплотнение на месте ... 236
Базовый класс VoidPtr ... *236*
Пул ведущих указателей ... *237*
Итератор ведущих указателей ... *238*
Алгоритм уплотнения ... *238*
Оптимизация ... *239*
Последовательное уплотнение на месте ... *239*
Перспективы ... 239

ГЛАВА 16. СБОРКА МУСОРА ... 241

Доступность ... 241
Периметр ... *241*
Внутри периметра ... *242*
Анализ экземпляров ... *243*
Перебор графа объектов ... *244*
Сборка мусора по алгоритму Бейкера ... 245
Шаблон слабого дескриптора ... *245*
Шаблон сильного дескриптора ... *245*
Итераторы ведущих указателей ... *246*
Перебор указателей ... *248*
Оптимизация ... *251*
Внешние объекты ... *251*
Множественные пространства ... *251*
Сборка мусора и уплотнение на месте ... *251*
Нужно ли вызывать деструкторы? ... *252*
Только для профессиональных каскадеров ... 252
Концепции «матери всех объектов» ... *252*
Организация памяти ... *253*

Поиск периметра *254*
Перебор внутри периметра *254*
Сборка мусора *255*
Последовательная сборка мусора *255*
Итоговые перспективы 255

ПРИЛОЖЕНИЕ. JAVA ПРОТИВ C++ 257

Благодарности

Эта книга, как и любая другая, обязана своим существованием слишком многим, чтобы их можно было перечислить в одном списке. Книга — это нечто большее, чем просто страницы, покрытые забавными черными значками. Это смесь альтруизма, авторского ego и первобытного крика души. Кроме того, она сыграла заметную роль в жизни автора и его семьи. Я глубоко благодарен своей жене Синди и сыновьям Нику, Джей-Джею и Бобби за их терпение, поддержку и прощение, когда папа не мог уделять им достаточно времени для игр.

Если считать терпение добродетелью, то самые добродетельные люди, о которых мне известно, работают в издательстве AP Professional. Я в долгу перед всеми, кто с самого начала поддержал мою идею этой книги и продолжал нажимать на меня, чтобы работа не стояла на месте. Иногда мне кажется, что на мою долю выпала самая легкая часть — теперь я отдыхаю, а вы пытаетесь продать!

Я особенно благодарен Джону Трудо (John Trudeau) из компании Apple Computer, который впервые предложил мне изложить на бумаге свои разрозненные мысли и переживания в виде семинара для опытных программистов C++. Даже не знаю, что я должен выразить многим слушателям этих семинаров, которые пережили ранние варианты этого курса, прежде чем он начал принимать законченные формы, — то ли благодарность, то ли свои искренние извинения.

За эти годы многие люди повлияли на мое отношение к C++ и объектно-ориентированному программированию. В голову сразу же приходит несколько имен — Нил Голдстейн (Neal Goldstein), Ларри Розенштейн (Larry Rosenstein), Эрик Бердал (Eric Berdahl), Джон Брюгге (John Brugge), Дэйв Симмонс (Dave Simmons) и Дэйв Бьюэлл (Dave Buell). Никто из них не несет ответственности за то, с чем вы не согласитесь в этой книге, но именно они заронили в мою душу первые идеи.

Моя благодарность распространяется и на новых коллег из Microsoft Corporation, куда я был принят, когда книга была «почти готова» — то есть были готовы первые 90 процентов и оставалось сделать еще 90 процентов. Эта книга не была написана «под знаменем Microsoft», поэтому, пожалуйста, не обвиняйте их во всем, что в ней написано. Книга была начата и почти завершена до того, как я начал работать на Microsoft, и никто из работников Microsoft не помогал мне, не просматривал книгу и не одобрял ее.

Джеймс Коплин (James Coplien), мы никогда не встречались, но твоя книга «Advanced C++ Programming Styles and Idioms» оказала самое большое влияние на мое мировоззрение. Книга великолепно раскрывает тему нетривиального использования C++. Надеюсь, по твоим следам пойдут и другие авторы.

Наконец, хочу поблагодарить Бьярна Страуструпа (Bjarne Stroustrup) за то, что он изобрел такой странный язык. О простых, последовательных языках типа SmallTalk неинтересно не то что писать, но даже думать. Если бы в C++ не было всех этих тихих омутов и загадочных правил, пропала бы благодатная почва для авторов, консультантов и прочих личностей вроде вашего покорного слуги. Бьярн, я люблю твой язык… Честное слово, люблю — как Черчилль любил демократию. C++ — худший объектно-ориентированный язык… но остальные еще хуже.

Извинения… или вроде того

Заодно хочу воспользоваться случаем и извиниться перед всеми, кого я обидел в своей книге. Понятия не имею, кто вы такие, но на своем горьком опыте (по двум статьям, опубликованным в журнале IEEE *Computer*) я узнал, как много людей обижается на несерьезный подход к *серьезной* теме — такой как

C++. Если вы принадлежите к их числу, я сожалею, что задел ваши чувства. Пусть не так сильно, чтобы лишиться сна, но все же сожалею.

Я не претендую на авторство изложенных в книге идей. Если вы увидите в ней что-то, придуманное вами или кем-то другим, — смело заявляйте, что это ваших рук дело, спорить я не стану. Мастерство нетривиального использования C++ растет от свободного обмена идеями, а не от формального изучения, так что в действительности очень трудно однозначно определить, кто, что и когда сказал. Я написал эту книгу, чтобы как можно больше людей смогли быстро и безболезненно повысить свою квалификацию, поэтому вопросам авторства идей уделялось второстепенное внимание. Если вам это не нравится, примите мои искренние извинения и напишите свою собственную книгу.

С другой стороны, я взял на себя смелость использовать новые имена для старых концепций, вызывающих недоразумения, и нисколько об этом не жалею. Такова уж великая традиция сообщества C++, которое переименовало почти все объектно-ориентированные концепции: *субкласс* (производный класс), *суперкласс* (базовый класс), *метод* (функция класса) и т.д. Сторонники переименования не обошли вниманием даже такие традиционные концепции C, как поразрядный сдвиг (<< и >>). Вам не нравится, что для старых идей используются новые имена, — пусть даже в интересах ясности? Приятель, вы ошиблись с выбором языка.

Я сделал все возможное, чтобы все фрагменты кода в этой книге работали как положено, но без ошибок дело наверняка не обошлось. Действуйте так, словно ваша программу уже горит синим пламенем, — проверяйте, проверяйте и еще раз проверяйте мой код, прежде чем использовать его в своих программах. Помните: в этой книге я демонстрирую различные идиомы и концепции, а не создаю библиотеку классов. Все идиомы вполне работоспособны, но дальше вам придется действовать самостоятельно.

Джефф Элджер

Январь 1998 г.

Часть 1

Введение и краткий обзор

В этой части я отвечаю на главный вопрос: «Зачем было писать еще одну книгу о C++»? Далее в головокружительном темпе рассматриваются некоторые нетривиальные возможности языка. Все это делается исключительно для подготовки к следующим главам, поэтому материал можно читать или пропускать в зависимости от того, насколько уверенно вы владеете теми или иными тонкостями синтаксиса C++.

Зачем нужна еще одна книга о C++?

1

По последним данным, на рынке продается по крайней мере 2 768 942 книги о C++, не говоря уже о всевозможных курсах, обучающих программах, журналах и семинарах с коктейлями. И все же в этом изобилии наблюдается удручающее однообразие. Просматривать полку книг о C++ в книжном магазине ничуть не интереснее, чем литературу по бухгалтерии. В сущности, все книги пересказывают одно и то же и отличаются разве что по весу и количеству цветов в диаграммах и таблицах. По моим подсчетам, 2 768 940 из них предназначены для новичков, ориентированы на конкретный компилятор или представляют собой справочники по синтаксису C++. Для тех, кто уже знает язык и желает подняться на следующий уровень, существующая ситуация оборачивается сплошными разочарованиями и расходами. Чтобы узнать что-то новое, приходится дергать главу отсюда и раздел оттуда. Для знатока C++ такая трата времени непозволительна.

Эта книга — совсем другое дело. Прежде всего, она предполагает, что вы уже владеете C++. Вероятно, вы программировали на C++ в течение года-двух или более. Став настоящим асом, на вопрос о должности вы перестали скромно отвечать «Программист»; теперь ваш титул складывается из слов «Старший», «Специалист», «Ведущий», «Разработчик», «Проектировщик» (расставьте в нужном порядке). Вы уже знаете, что «перегрузка оператора» не имеет никакого отношения к телефонной компании, а «класс-коллекция» — вовсе не сборище филателистов. На вашей полке стоит книга Страуструпа «Annotated C++ Reference Manual», которую в профессиональных разговорах вы часто сокращенно именуете ARM и даже не считаете нужным расшифровывать.

Если вы узнали себя, добро пожаловать — эта книга для вас. Ее можно было бы еще назвать «C++: путь гуру». C++ в ней описывается совсем не так, как в книгах для начинающих. На этом уровне C++ — не столько язык, сколько целая субкультура со своими идиомами, приемами и стандартными архитектурными решениями, которые не следуют очевидным образом из формального описания языка. Об этом «языке внутри языка» редко упоминается с книгах и журналах. Одни программисты самостоятельно обнаруживают все эти возможности и с гордостью считают, что изобрели нечто потрясающее, пока не выяснится, что «нет ничего нового под солнцем». Другим везет, и они становятся учениками подлинных мастеров C++ — к сожалению, такие мастера встречаются слишком редко. В этой книге я попытался проложить третий путь истинного просветления — самостоятельное изучение. Кроме того, книга предназначена для тех, кто уже достиг заветной цели, но хочет пообщаться, поболтать в дружеской компании и пошевелить мозгами над очередной головоломкой.

Дао C++

C++ — язык, который изучается постепенно. Лишь после того, как будет сделан последний шаг, разрозненные приемы и фрагменты синтаксиса начинают складываться в общую картину. По-моему, изучение C++ чем-то напоминает подъем на лифте. Дзынь! Второй этаж. C++ — это усовершенствованный вариант C, с сильной типизацией (которую, впрочем, при желании можно обойти) и удобными комментариями //. Любой программист на C, если он не хочет подаваться в менеджеры, должен двигаться дальше… а Бьярн Страуструп (Господи, благослови его) придумал для этого отличную возможность.

Дзынь! Третий этаж. C++ — хороший, хотя и не потрясающий объектно-ориентированный язык программирования. Не Smalltalk, конечно, но чего ожидать от языка, работающего с такой головокружительной скоростью? C++ — это Cobol 90-х, политически выдержанный язык, которые гарантирует финансирование вашего проекта высшим руководством. А уж если C++ достаточно часто упоминается в плане, можно надеяться на удвоение бюджета. Это тоже хорошо, потому что никто толком не умеет оценивать проекты на C++ и управлять ими. А что касается инструментария — глаза разбегаются, не правда ли?

Дзынь! Последний этаж, все выходят. Но позвольте, где же «все»? Лифт почти пуст. C++ — это на самом деле не столько язык, сколько инструмент для создания ваших собственных языков. Его элегантность заключается отнюдь не в простоте (слова *C++* и *простота* режут слух своим явным противоречием), а в его потенциальных возможностях. За каждой уродливой проблемой прячется какая-нибудь умная идиома, изящный языковой финт, благодаря которому проблема тает прямо на глазах. Проблема решается так же элегантно, как это сделал бы *настоящий* язык типа Smalltalk или Lisp, но при этом ваш процессор не дымится от напряжения, а на Уолл-Стрит не растут акции производителей чипов памяти. C++ — вообще не язык. Это мировоззрение или наркотик, меняющий способ мышления.

Но вернемся к слову «элегантный». В программировании на C++ действует перефразированный принцип Дао: «Чтобы достичь истинной элегантности, нужно отказаться от стремления к элегантности». C++ во многом представляет собой C следующего поколения. Написанные на нем программы эффективно компилируются и быстро работают. Он обладает очень традиционной блочной структурой и сокращенной записью для многих распространенных операций (например, i++). В нем есть свои существительные, глаголы, прилагательные и свой жаргон:

```
cout << 17 << endl << flush;
```

Ревнители частоты языка часто нападают на C++. Они полагают, что высшее достижение современной цивилизации — язык, построенный исключительно из атомов и скобок. По мнению этих террористов от синтаксиса, если простую переменную с первого взгляда невозможно отличить от вызова функции или макроса — это вообще не язык, а шарлатанство для развлечения праздной толпы. К сожалению, теория расходится с практикой. В реальной жизни толпа платит лишь за то, чтобы видеть языки, в которых разные идеи *выглядят* по-разному. «Простые и последовательные» языки никогда не пользовались особым успехом за стенками академий, а языки с блочной структурой овладели массами. Стоит ли этому удивляться? Ведь компьютерные языки приходится изучать и запоминать, а для этого используется то же серое вещество, с помощью которого мы изучаем и запоминаем естественные языки. Попробуйте-ка назвать хотя бы один естественный язык без существительных, глаголов и скобок! Я бы не рискнул. Все наши познания в лингвистике говорят о том, что эти «плохие» особенности только ускоряют изучение компьютерного языка и делают его более понятным. `i++` во всех отношениях *действительно* понятнее, чем `i:=i+1`, а `x=17+29` читается лучше, нежели `(setq x(+17, 29))`. Речь идет не о строении компьютерного языка, а скорее о *нашем собственном* строении. Все уродства C++ — это в основном наши уродства. Когда вы научитесь понимать и любить его странности, когда перестанете беспокоиться о математической стройности, будет сделан ваш первый шаг к достижению элегантности в C++.

C++ наряду с Lisp, Smalltalk и другими динамическими языками (в отличие от C) обладает средствами для низкоуровневых манипуляций с компьютером. Вы можете создать свой собственный тип данных и подсунуть его компилятору так, чтобы он принял этот тип за встроенный. Вы можете управлять вызовами своих функций, обращениями к переменным классов, выделением и освобождением памяти, инициализацией и удалением объектов — и все это (в основном) происходит без потери эффективности или безопасности типов. Но в отличие от других языков, если эта сила будет применена неправильно, программа на C++ «грохнется». А если устоит программа, грохнутся ваши коллеги-программисты — если вы не придумаете, как пояснить свои намерения и использовать правильную идиому для особенно сложных моментов. Как известно, Дедал со своим сыном Икаром бежал и заточения на Крите с помощью крыльев, сделанных из перьев и воска. Дедал, главный архитектор и изобретатель, спокойно порхал где-то внизу. Его безрассудный сын поднялся слишком высоко к солнцу и упал в море. Хммм… Нет, пожалуй, аналогия получилась неудачная. Ведь именно Дедал построил Лабиринт — такой сложный, что в попытках выбраться из него люди либо умирали, либо попадали на обед к Минотавру. Может, попробуем более современную аналогию? Используя

низкоуровневые возможности C++, вы действуете, как суровый детектив с его сакраментальной фразой: «Доверься мне — я знаю, что делаю». Компилятор закатывает глаза и безмолвно подчиняется.

C++ интригует своими явными противоречиями. Его гибкость легко превращается в главный источник ошибок. За возможности его расширения не приходится расплачиваться скоростью или объемом кода. Он элегантен в одних руках и опасен в других, прост и сложен одновременно. После нескольких лет работы вы так и не можете решить, восхищаться им или проклинать. Да, настоящий знаток понимает все концепции, лежащие в основе языка и склоняющие чашу весов в его пользу. Эти концепции не видны с первого взгляда; чтобы понять их, необходимо в течение нескольких лет пытаться решать совершенно разные задачи. Некоторые архитектурные парадигмы лучше всего соответствуют конкретным языковым решениям. Их неправильное сочетание обернется хаосом, а правильное — элегантностью.

Три великие идеи C++

О нетривиальном использовании C++ написано так много, что я даже не знаю, с чего начать. Вам когда-нибудь приходилось видеть стереограммы? С первого взгляда они похожи на случайный узор, но после медленного и внимательного разглядывания на них проявляется слон, спираль или что-нибудь еще. Чтобы увидеть смысл в точках и пятнах, нужно рассматривать их в контексте объединяющей темы. Именно здесь кроется одно из самых больших разочарований при изучении архитектуры и идиом C++. Сначала кажется, что перед вами — огромная куча разрозненных приемов и ни одного правила того, как ими пользоваться. Эта книга научит вас «видеть слона». Существует множество классификаций нетривиальных аспектов C++, но я разделил их на несколько простых тем:

- Косвенные обращения.
- Гомоморфные иерархии классов.
- Пространства памяти.

В основе каждой темы лежит конкретный синтаксис и средства C++, и их совместное применение позволяет решать самые разные задачи. Существует много других приемов и принципов, которые тоже стоило бы включить в эту книгу, но эти три категории помогают организовать очень большое количество тем в логически стройную структуру.

В первой части приведен обзор многих важных аттракционов синтаксического цирка C++. Например, многие программисты C++ не имеют большого опыта работы с перегруженными операторами и лишь теоретически знают, как они применяются. Оказывается, большинство программистов никогда не использует шаблоны или обработку исключений и лишь немногие умеют пользоваться потоками ввода/вывода за рамками простейших обращений к объектам `cout` и `cin`. В части 1 стараюсь выровнять уровни подготовки читателей, заполнить пробелы в ваших знаниях C++ и подготовиться к игре. Часть 1 можно читать от корки до, бегло просматривать или пропускать целые разделы в зависимости от того, насколько хорошо вы знакомы с нюансами C++.

Термин *косвенное обращение (indirection)* относится к разным конкретным темам, однако везде используется одна и та же концепция: клиентский объект обращается с запросом к другому объекту, который, в свою очередь, поручает работу третьему объекту. Косвенность связана со средним объектом в цепочке. Иногда годится слышать, что это определение почти совпадает с определением *делегирования (delegation),* одного из краеугольных камней объектно-ориентированного программирования. Тем не менее, в C++ идиомы, используемые с этой концепцией, и ее языковая поддержка выходят далеко за рамки того, что считается делегированием в других языках. В этой книге часто используется термин *указатель (pointer);* вы встретите его в каждой главе. Указатели C++ способны на многое. Они могут определить, где в памяти, на диске или в сети находится объект, на который они ссылаются; когда он уничтожается; изменяется ли он или доступен только для чтения; и даже то, существует ли объект или просто представляет собой некую область в абстрактном пространстве памяти — и все это происходит без активного участия самого объекта, который может ничего не знать об этих низкоуровневых операциях. Возможности, что и говорить, впечатляющие, однако они основаны на нескольких очень простых идиомах.

О проектировании иерархии классов говорили все кому не лень — одни по делу, другие болтали об «имитации объектов реального мира». Большинство аргументов в равной степени относится к любому объектно-ориентированному языку, и я вовсе не намерен захламлять книгу по C++ своими личными

взглядами на объектно-ориентированный дизайн. Тем не менее, один конкретный тип наследования — *гомоморфное наследование (homotorphic derivation)* — оказывается исключительно полезным в сочетании со специфическими средствами C++. В гомоморфной иерархии все производные классы получают свой открытый интерфейс от некоторого базового класса-предка. Как правило, «мать всех базовых классов» девственно чиста — она не содержит ни одной переменной, а все ее функции являются чисто виртуальными. В C++ с этой концепцией ассоциируются многие полезные идиомы проектирования и программирования.

За концепцией *пространства памяти (memory space)* кроется нечто большее, чем обычное управление памятью. Перегружая в C++ операторы `new` и `delete`, вы определяете, где создаются объекты и как они уничтожаются. Кроме того, можно создавать абстрактные коллекции, в которых не всегда понятно, с чем вы имеете дело — с настоящим объектом или с абстракцией. На горизонте уже видны контуры новых распределенных объектно-ориентированных структур, разработанных такими фирмами, как Microsoft, Apple и Taligent. Правда, вам придется пересмотреть некоторые базовые представления о том, где находятся объекты и как они перемещаются в другое место — все эти темы я выделил в категорию пространств памяти. Пространства памяти позволяют определить тип объекта во время выполнения программы — возможность, которой до обидного не хватает в C++. Конечно, мы поговорим и об управлении памятью, но этим дело не ограничится.

Как читать эту книгу

Перед вами — не руководство с готовыми рецептами для конкретных ситуаций. Скорее это сборник творческих идей и головоломок. Если к концу книги вы почувствуете, что ваш арсенал приемов программирования на C++ расширился, значит, я достиг своей цели, а учить вас, когда и как пользоваться этими приемами, я не стану.

Материал каждой отдельной главы невозможно *в полной мере* понять без предварительного знакомства со всеми остальными главами. И все же я приложил максимум усилий, чтобы материал любой главы был полезен немедленно после знакомства с ней и чтобы главы логически следовали друг за другом, а наш воображаемый слон вырисовывался постепенно — бивни, уши, хобот и т. д. После прочтения книга может пригодиться в качестве справочника — что-то вроде личной и очень краткой энциклопедии приемов программирования и идиом C++.

За многие годы изучения и использования C++ я узнал, что даже у опытных программистов в познаниях встречаются пробелы; в оставшемся материале части я постараюсь выровнять уровень подготовки всех читателей. Это не вступительное описание языка, а скорее краткая сводка тем, которые будут использованы в последующих главах. В главе 2 мы стремительно пробежимся по некоторым особенностям языка. Глава 3 посвящена шаблонам — постепенно эта тема становится все более важной, поскольку шаблоны поддерживаются во все большем числе компиляторов. В главе 4 рассматривается обработка исключений на основе рекомендованного стандарта ANSI и приводится пара замечаний о нестандартных исключениях, встречающихся в реальном мире.

Часть 2 посвящена разным типам указателей — от глупых до гениальных. На этом фундаменте построена вся книга, и я уверен, что эти сведения будут полезны любому читателю.

В части 3 рассматриваются структура и реализация типов и иерархий классов в C++. Основное внимание уделено одному из частных случаев — гомоморфным иерархиям классов. Заодно мы поговорим об объектах классов, представителях и других любопытных темах. Большинству читателей стоит прочитать третью часть от начала до конца, но никто не запрещает вам просмотреть ее и отобрать темы по своему вкусу. И хотя вы будете полагать, что знаете об указателях все на свете, они совершенно неожиданно снова возникнут в контексте гомоморфных иерархий.

В части 4 нас поджидает самая ужасная тема C++ — управление памятью. Уровень изложения меняется от примитивного до нормального и сверхсложного, но основное внимание уделяется тем проблемам, которые могут возникнуть при программировании на C++, и их возможным решениям на базе разных языковых средств. Лично я считаю, что начальные главы этой части абсолютно необходимы для счастливой и полноценной жизни в C++, но если вас, допустим, совершенно не интересует процесс сборки мусора — оставьте последнюю пару глав и займитесь чем-нибудь более полезным для общества.

Несколько слов о стиле программирования

Вот эти несколько слов: стиль программирования меня не волнует. Я достаточно краток? Если хотя бы половина времени, израсходованного на правильную расстановку фигурных скобок, тратилась на обдумывание программы или еще лучше — на общение с пользователями, то вся отрасль работала бы намного эффективнее. Конечно, единство стиля — вещь хорошая, но я еще не видел книги или руководства по стилю, которые бы стоили даже часового собрания группы в начале проекта. К тому же ни одна книга или руководство по стилю не превратят код неаккуратного программиста в нечто осмысленное. В сущности, стиль часто используется как оправдание недостатка внимания к самой программе. Наконец, я еще не видел, чтобы в спорах о стиле один программист в чем-то убедил другого, поэтому любые дискуссии на эту тему считаю бесполезной тратой времени.

У меня есть свои собственные принципы и свой стиль, но в основном я собираюсь отказаться от своего пути и понемногу пользоваться всеми стилями, с которыми мне приходилось встречаться. Книга посвящена языковым идиомам, а не расположению фигурных скобок или регистру символов. Надеюсь, мое решение будет раздражать всех читателей в равной мере.

Я также весьма вольно обошелся с подставляемыми (inline) функциями классов, особенно с виртуальными. В каноническом варианте подставляемые функции должны выглядеть следующим образом:

```
class Foo {
public:
     void MemberFn();
};
inline void Foo::MemberFn()
{
...
}
```

В моей книге этот фрагмент будет выглядеть иначе:

```
class Foo {
public:
     void MemberFnO {...};
};
```

Я оформлял как подставляемые даже виртуальные функции классов, хотя одни компиляторы отвергают такой синтаксис, а другие обрабатывают его неправильно. Делалось это для экономии места. Если бы тексты всех подставляемых функций приводились отдельно, книга заметно выросла бы в размерах, а разрывы страниц чаще приходились на середину листинга. Так что не относитесь к подставляемым функциям слишком серьезно.

Садитесь в любимое кресло, заводите хорошую музыку, ставьте под руку чашку чая и попытайтесь получить удовольствие!

Синтаксис C++

2

За годы преподавания C++ я узнал, что подавляющее большинство программистов C++ (включая самых опытных) редко пользуется некоторыми возможностями языка. Конечно, дело это сугубо индивидуальное, но при всей сложности и глубине C++ небольшой обзор не повредит никому. В этой и двух следующих главах я постараюсь выровнять уровень подготовки читателей перед тем, как переходить к действительно интересным темам. Эта глава не заменит Annotated Reference Manual или другое справочное руководство — вы не найдете в ней полной спецификации языка. Я лишь рассмотрю некоторые языковые средства, которые часто понимаются неверно или не понимаются вовсе. Придержите шляпу и приготовьтесь к стремительному облету синтаксиса C++!

Переменные и константы

Болтовню о том, что такое переменные и для чего они нужны, пропускаем. Нашего внимания заслуживают две темы: константность и сравнение динамических объектов со стековыми.

const

Ключевое слово `const`, которое в разных контекстах принимает разные значения, одно из самых информативных в C++. Да, между этими значениями есть кое-что общее, но вам все равно придется запомнить все конкретные случаи.

Константные переменные

Если переменная объявлена с ключевым словом `const`, значит, она не должна меняться. После определения константной переменной вы уже не сможете изменить ее значение или передать ее в качестве аргумента функции, которая не гарантирует ее неизменности. Рассмотрим простой пример с константной целой переменной.

```
const int j = 17;        // Целая константа
j = 29;                  // Нельзя, значение не должно меняться
const int i;             // Нельзя, отсутствует начальное значение
```

Третья строка неверна, поскольку в ней компилятору предлагается определить случайную переменную, которую никогда не удастся изменить, — этакий странный генератор случайных целых констант. Вообще говоря, вы сообщаете компилятору, какой конструктор он должен использовать в конкретном случае. Если бы переменная `i` относилась к нетривиальному классу, то при объявлении константного экземпляра пришлось бы явно указать конструктор и его аргументы. `int` — вырожденный случай, поскольку на самом деле `const int j=17;` — то же, что и `int j(17)`.

Но вот компилятор узнал, что нечто должно быть константным. Он просыпается и начинает искать ошибки — не только фактические, но и потенциальные. Компилятор не разрешит использовать ваше константное нечто в любом неконстантном контексте, даже если шестилетний ребенок разберется в программе и докажет, что в ней нет ни одной ошибки.

```
const i = 17;
int& j = 1;          // Нельзя, потому что позднее j может измениться
```

Не важно, будете ли вы изменять величину, на которую ссылается j. Компилятор предполагает, что вам *захочется* это сделать, и на всякий случай устраняет искушение. Иначе говоря, константность — свойство переменной, а не данных, поэтому неконстантная переменная не может ссылаться на константную величину.

const и #define

Две следующие строки *не* эквивалентны:

```
const int i = 17;
#define i 17;
```

В первой строке определяется переменная, занимающая некоторую область памяти, а во второй — макрос. Обычно отличия несущественны, если не считать одного-двух лишних тактов, затраченных на каждое обращение к константной переменной. Однако если переменная является глобальной и принадлежит нетривиальному классу со своим конструктором, ситуация резко меняется. Дополнительные сведения приведены в разделе «Инициализация глобальных объектов» этой главы.

Константы в перечислениях

Перечисления (enum) не очень широко использовались в языке С по одной простой причине: символические имена констант имеют глобальную область действия и быстро захламляют пространство имен. В С++ эта проблема исчезла, поскольку область действия символических имен ограничивается классом или структурой.

```
class Foo {
public:
    enum Status { kOpen = 1, kClosed };
};
// Где-то в программе
Foo::Status s = Foo::kOpen;
```

Обратите внимание — область действия должна быть явно указана как в имени типа, так и в символическом имени. Следовательно, символические имена kOpen и kClosed можно использовать в программе и для других целей. Компилятор рассматривает символические имена перечислений как макросы, а не как константные переменные. Это обстоятельство может оказаться важным при инициализации глобальных переменных (см, далее в этой главе).

Указатель на константу

С указателями дело обстоит несколько сложнее, поскольку приходится учитывать два значения: адрес и содержимое памяти по этому адресу. В следующем примере p — это указатель на константу; находящийся в указателе адрес может измениться, но содержимое памяти по этому адресу — нет.

```
const int* p;
int i = 17;
p = &i;        // Можно
*p = 29;       // Нельзя
```

Сказанное также относится к структурам и объектам.

```
class foo {
public:
    int x;
};
const foo* f = new foo;
f->x = 17;    // Нельзя, присвоение членам класса не допускается
```

Константный указатель

С константными указателями все наоборот: адрес изменять нельзя, но зато можно изменять содержимое памяти по этому адресу.

```
int i = 17;
int j = 29;
int* const p;       // Нельзя! Должно быть задано начальное значение
int* const p1 = &i;     // Порядок
*p1 = 29;           // Можно; величина, на которую ссылается указатель,
                    // может изменяться
p1 = &j;            // Нельзя
```

Константный указатель на константу

Константный указатель на константу (попробуйте-ка трижды быстро произнести это вслух!) изменить вообще нельзя. Это неизменяемый адрес неизменяемой величины.

```
int i = 17;
int j = 29;
const int* const p;          // Нельзя. Должен быть задан начальный адрес
const int* const p1 = &i;    // Можно
*p1 = 29;                    // Нельзя
p1 = &j;                     // Нельзя
```

Константные аргументы функций

Константный аргумент функции должен подчиняться тем же правилам, что и любая другая константная переменная.

```
void f(const int* p)
{
    *p = 17;        // Нельзя
    int i = 29;
    p = &i;         // Можно, но зачем?
}
// Где-то в программе
int i = 17;
f(&i);    // Порядок, фактический аргумент не обязан быть константой
```

Обратите внимание — аргумент, указанный при вызове функции, не обязан быть константным. Этот вопрос целиком остается на усмотрение стороны-получателя. Передача по ссылке осуществляется по тем же правилам, что и передача по адресу.

```
void f(const int& p)
{
    p = 17;         // Нельзя
    int i = 29;
    p = i;          // Можно (на грани фола)
}
// Где-то глубоко в программе
int i = 17;
f(i);               // Порядок
```

Неконстантные аргументы функций

Если формальный аргумент функции объявлен неконстантным, то и фактический аргумент, используемый при вызове, тоже должен быть неконстантным.

```
void f(int*);
int i = 17;
const int* p = &i;
const int j = 29;
f(&i);          // Можно, потому что i – не константа
f(p);           // Нельзя
f(&j);          // Тоже нельзя, потому что j - константа
```

Это еще одно средство, с помощью которого компилятор соблюдает принцип «единожды константный всегда остается константным». Даже если функция `f` на самом деле не изменяет значения своего формального параметра, это ни на что не влияет.

Константные функции классов

В константных функциях классов переменная `this` интерпретируется как указатель на константу. Компилятор даст вам по рукам, если вы попробуете воспользоваться переменной `this` для изменения переменной класса или найти для нее иное, неконстантное применение. Смысл ключевого слова `const` зависит от его места в объявлении функции; для константных функций оно, словно бородавка, торчит после сигнатуры функции.

```
class foo {
private:
    int x;
public:
    void f() const;
    void g();
};
void h(int*);
void m(foo*);
void foo::f();
{
    x = 17;         // Нельзя: изменяется переменная класса
    this->g();      // Нельзя: g – некоторая функция
    h(&x);          // Нельзя: h может изменить x
    m(this);        // Нельзя: неконстантный аргумент в m()
}
```

Первая ошибка — попытка изменить переменную класса через `this`. В константных функциях класса `foo` переменная `this` фактически объявляется как `const foo* this;`. Вторая ошибка сложнее. Из приведенного фрагмента неизвестно, изменяет ли функция `g` какие-либо переменные класса `foo`, но это и не важно; одной возможности достаточно, чтобы ваш компилятор разразился негодующими воплями. Из константной функции класса нельзя вызывать неконстантные функции через `this`. Похожая ситуация возникает с третьей и четвертой ошибкой — компилятор попытается спасти вас от самого себя и не допустит потенциально опасные строки.

Один из верных признаков профессионала C++ — ключевые слова `const`, обильно разбросанные по функциям классов. Любая функция класса, которая гарантированно не изменяет `this`, должна без малейших размышлений объявляться константной. Впрочем, как видно из приведенного выше фрагмента, эта стратегия работает лишь в том случае, если все участники команды следуют вашему примеру и объявят константными *свои* функции, В противном случае возникают каскадные ошибки. Часто выясняется, что недавно купленная библиотека классов не использует константные функции и

нарушает ваш пуританский стиль кодирования. Мораль: константные функции классов нужно использовать либо с полным фанатизмом (желательно), либо не использовать вовсе.

Стековые и динамические объекты

Иногда мне кажется, что C++ лучше изучать без предварительного знакомства с C. В C++ часто используются те же термины, что и в C, но за ними кроются совершенно иной смысл и правила применения. Например, возьмем примитивный целый тип.

```
int x = 17;
```

В C++ это будет экземпляр встроенного «класса» `int`. В C это будет... просто `int`. Встроенные классы имеют свои конструкторы. У класса `int` есть конструктор с одним аргументом, который инициализирует объект передаваемым значением. Теоретически существует и деструктор, хотя он ничего не делает и ликвидируется всеми нормальными разработчиками компиляторов в процессе оптимизации. Важно осознать, что встроенные типы за очень редкими исключениями подчиняются тем же базовым правилам, что и ваши расширенные типы.

Вы должны понимать эту теоретическую особенность C++, чтобы правильно относиться к стековым и динамическим объектам и связанным с ними переменным.

Размещение в стеке

Чтобы выделить память для стековой переменной в области действия блока, достаточно просто объявить ее обычным образом.

```
{
    int i;
    foo f(constructor_args);
    // Перед выходом из блока вызываются деструкторы i и f
}
```

Стековые объекты существуют лишь в границах содержащего их блока. При выходе за его пределы автоматически вызывается деструктор. Разумеется, получение адреса стекового объекта — дело рискованное, если только вы абсолютно, стопроцентно не уверены, что этот указатель не будет использован после выхода за пределы области действия объекта. Все фрагменты наподобие приведенного ниже всегда считаются потенциально опасными:

```
{
    int i;
    foo f;
    SomeFunction(&f);
}
```

Без изучения функции `SomeFunction` невозможно сказать, безопасен ли этот фрагмент. `SomeFunction` может передать адрес дальше или сохранить его в какой-нибудь переменной, а по закону Мэрфи этот адрес наверняка будет использован уже после уничтожения объекта `f`. Даже если сверхтщательный анализ `SomeFunction` покажет, что адрес не сохраняется после вызова, через пару лет какой-нибудь новый программист модифицирует `SomeFunction`, продлит существование адреса на пару машинных команд и — БУМ!!! Лучше полностью исключить такую возможность и не передавать адреса стековых объектов.

Динамическое размещение

Чтобы выделить память для объекта в куче (heap), воспользуйтесь оператором **`new`**.

```
foo* f = new foo(constructor_args);
```

Вроде бы все просто. Оператор `new` выделяет память и вызывает соответствующий конструктор на основании переданных аргументов. Но когда этот объект уничтожается? Подробный ответ на этот вопрос займет примерно треть книги, но я не буду вдаваться в технические детали и отвечу так: «Когда

кто-нибудь вызовет оператор delete для его адреса». Сам по себе объект из памяти не удалится; вы должны явно сообщить своей программе, когда его следует уничтожить.

Указатели и ссылки

Попытки связать указатели с динамическими объектами часто приводят к недоразумениям. В сущности, они не имеют друг с другом ничего общего. Вы можете получить адрес стекового объекта и выполнить обратное преобразование, то есть разыменование (dereferencing) адреса динамического объекта. И на то, и на другое можно создать ссылку.

```
{
    foo f;
    foo* p = &f;
    f.MemberFn();       // Использует сам объект
    p->MemberFn();      // Использует его адрес
    p = new foo;
    foo& r = *p;        // Ссылка на объект
    r.MemberFn();       // То же, что и p->MemberFn()
}
```

Как видите, выбор оператора . или -> зависит от типа переменной и не имеет отношения к атрибутам самого объекта. Раз уж мы заговорили об этом, правильные названия этих операторов (. и ->) — *селекторы членов класса (member selectors).* Если вы назовете их «точкой» или «стрелкой» на семинаре с коктейлями, наступит гробовая тишина, все повернутся и презрительно посмотрят на вас, а в дальнем углу кто-нибудь выронит свой бокал.

Недостатки стековых объектов

Если использовать оператор delete для стекового объекта, то при большом везении ваша программа просто грохнется. А если вам (как и большинству из нас) не повезет, то программа начнет вести себя, как ревнивая любовница — она будет вытворять, всякие гадости в разных местах памяти, но не скажет, на что же она разозлилась. Дело в том, что в большинстве реализаций C++ оператор new записывает пару скрытых байтов перед возвращаемым адресом. В этих байтах указывается размер выделенного блока. По ним оператор delete определяет, сколько памяти за указанным адресом следует освободить. При выделении памяти под стековые объекты оператор new не вызывается, поэтому эти дополнительные данные отсутствуют. Если вызвать оператор delete для стекового объекта, он возьмет содержимое стека над вашей переменной и интерпретирует его как размер освобождаемого блока.

Итак, мы знаем по крайней мере две причины, по которым следует избегать стековых объектов — если у вас нет действительно веских доводов в их пользу:

1. Адрес стекового объекта может быть сохранен и использован после выхода за границы области действия объекта.

2. Адрес стекового объекта может быть передан оператору delete.

Следовательно, для стековых объектов действует хорошее правило: *Никогда не получайте их адреса или адреса их членов.*

Достоинства стековых объектов

С другой стороны, память в стеке выделяется с головокружительной быстротой — так же быстро, как компилятор выделяет память под другие автоматические переменные (скажем, целые). Оператор new (по крайней мере, его стандартная версия) тратит несколько тактов на то, чтобы решить, откуда взять блок памяти и где оставить данные для его последующего освобождения. Быстродействие — одна из веских причин в пользу выделения памяти из стека. Как вы вскоре убедитесь, существует немало способов ускорить работу оператора new, так что эта причина менее важна, чем может показаться с первого взгляда.

Автоматическое удаление — второе большое преимущество стековых объектов, поэтому программисты часто создают маленькие вспомогательные стековые классы, которые играют роль «обертки» для динамических объектов. В следующем забавном примере динамический класс Foo «упаковывается» в стековый класс PFoo. Конструктор выделяет память для Foo; деструктор освобождает ее. Если вы незнакомы с операторами преобразования, обратитесь к соответствующему разделу этой главы. В двух словах, функция operator Foo*() позволяет использовать класс PFoo везде, где должен использоваться Foo* — например, при вызове функции g().

```
class PFoo {
private:
    Foo* f;
public:
    PFoo() : f(new Foo) {}
    ~PFoo() { delete f; }
    operator Foo*() { return f; }
}
void g(Foo*);
{
    PFoo p;
    g(p);    // Вызывает функцию operator Foo*() для преобразования
    // Уничтожается p, а за ним – Foo
}
```

Обратите внимание, что этот класс не совсем безопасен, поскольку адрес, возвращаемый функцией operator Foo*(), становится недействительным после удаления вмещающего PFoo. Мы разберемся с этим чуть позже.

Мы еще не раз встретимся с подобными фокусами. Вся соль заключается в том, что стековые объекты могут пригодиться просто из-за того, что их не приходится удалять вручную. Вскоре я покажу вам, как организовать автоматическое удаление динамических объектов, но эта методика очень сложна и вряд ли пригодна для повседневного применения.

У стековых объектов есть еще одно преимущество — если ваш компилятор поддерживает ANSI-совместимую обработку исключений (exception). Когда во время раскрутки стека происходит исключение, деструкторы стековых объектов вызываются автоматически. Для динамических объектов этого не случается, и ваша куча может превратиться в настоящий хаос. Рискуя повториться, я скажу, что мы вернемся к этой теме позднее.

Области действия и функции

Одно из значительных преимуществ C++ над C — возможность ограничения области действия символических имен. Впрочем, это палка о двух концах, поскольку правила определения области действия иногда довольно запутанны. Кроме того, в C++ появилась перегрузка функций и — как ее расширение — перегрузка операторов. Предполагается, что вы уже знакомы с азами, поэтому в своем кратком обзоре я ограничусь лишь некоторыми нетривиальными особенностями функций и областей действия.

Области действия

Область действия создается следующими конструкциями:

- класс;
- структура;
- объединение;
- блок;
- глобальное пространство имен.

Символические имена, объявленные в области действия, относятся только к данной области. Они не ограничиваются перечислениями и простыми переменными. Структуры, классы и функции также могут определяться в конкретной области действия.

Классы

Класс в C++ — нечто большее, чем простая структура данных. Это аналог модуля из других языков программирования, средство упорядочения символьных имен.

```
class Foo {
public:
    static int y;           // Глобальная переменная
    static void GFn();      // Глобальная функция
    int x;                  // Переменная класса
    Foo();                  // Конструктор
    void Fn();              // Функция класса
    typedef int (*IntFn)();              // Тип
    enum Status { kOpen = 0, kClosed };  // Другой тип
    struct Bar {            // Вложенная структура
        int a;
        int b;
        static void BarFn();
    }
private:
    void Hn();
};
```

В этом фрагменте приведены некоторые вариации на тему классов. Переменная `y` — глобальная переменная, а `GFn()` — глобальная функция, хотя область действия их имен ограничивается классом `Foo`. Во всех функциях класса `Foo` к ним можно обращаться просто по имени, но за его пределами необходимо использовать оператор области действия `::`:

```
Foo::Foo()
{
    GFn();          // Мы уже находимся в области действия Foo
}
void f()
{
    Foo::GFn();     // Необходимо задать область действия
}
```

Аналогично, определение типа `IntFn`, перечисление `Status` и даже вложенную структуру `Bar` также можно использовать без указания области действия в функциях класса `Foo`, но в любом другом месте эту область необходимо задать. Для вложенных типов с открытой видимостью синтаксис указания области действия может принять несколько устрашающий вид, как видно из следующего примера для структуры `Bar`:

```
Foo::Bar b;
Foo::Bar::BarFn();
```

По этой причине вложенные структуры либо делаются тривиальными, либо доступ к ним ограничивается.

Члены класса `x`, `Foo` и `Fn()`, имеют смысл лишь в контексте конкретного экземпляра (instance) этого класса. Для обращения к ним используются операторы-селекторы членов класса, `.` и `->`. Широкие массы (и, как я выяснил на собственном горьком опыте, даже разработчики компиляторов C++) почти не знают о том, что с помощью селекторов можно вызывать статические функции класса и обращаться

к статическим переменным класса. Следующий фрагмент верен, хотя бедные читатели вашей программы придут в такое замешательство, что подобное можно проделывать только в последний день перед увольнением:

```
Foo f;
f.Gfn();      // То же, что и Foo::GFn();
```

Структуры

Структура в C++ — почти что полноценный класс. Со структурой можно делать все, что можно делать с классом. Например, структуры могут участвовать в наследовании; в них можно объявлять секции `public`, `private`, `protected` и даже виртуальные функции. Тем не менее, для структур действуют несколько иные правила: по умолчанию все члены считаются открытыми (`public`), чтобы готовые программы на C не приходилось переписывать заново под каноны C++.

Теория — вещь хорошая, но давайте вернемся на землю. Стоит ли демонстрировать свою «крутизну» и объявлять структуру с множественным наследованием и виртуальными функциями? На практике структуры используются вместо классов лишь при соблюдении следующих условий:

- Структура не содержит виртуальных функций.
- Структура не является производной от чего-либо, кроме разве что другой структуры.
- Структура не является базовой для чего-либо, кроме разве что другой структуры.

Нормальные программисты C++ обычно используют структуры лишь для маленьких удобных наборов данных с тривиальными функциями. В частности, структуры часто используются в ситуациях, когда объект C++ должен быть совместим на битовом уровне с внешней структурой данных (особенно со структурами C). При этом можно запросто объявлять конструкторы и невиртуальные функции (особенно тривиальные встроенные), поскольку для них не создается v-таблица, которая могла бы нарушить битовую совместимость.

Объединения

Объединения C++ почти не отличаются от объединений C. Они позволяют сэкономить несколько байт за счет наложения различных структур данных поверх друг друга. Объединения могут содержать невиртуальные функции, в том числе конструкторы и деструкторы, но при этом они должны подчиняться довольно жестким ограничениям:

- Члены объединения не могут иметь конструкторов (хотя само объединение — может).
- Объединение не может быть производным от чего-либо.
- Ничто не может быть производным от объединения.
- Деструкторы членов не вызываются, хотя деструктор самого объединения, если он есть, вызывается.

Поскольку объединения не участвуют в иерархии наследования, нет смысла объявлять в них виртуальные функции или защищенные члены. Члены объединений разрешается объявлять закрытыми (`private`) или открытыми (`public`). Объединения пригодятся лишь тогда, когда вам действительно нужно сэкономить память, когда вы не собираетесь делать объединение производным или базовым, а также включать в него виртуальные функции или конструкторы. Иначе говоря, пользы от них не так уж много.

Блоки

Все, что стоило бы сказать о блоках, уже известно вам из C или из предыдущего описания стековых объектов.

Глобальные пространства имен

Глобальные пространства имен C++ настолько сложны, что в моем представлении процесс компиляции глобальных конструкций напоминает магический ритуал с дымом благовоний и пением мантр. Я постараюсь изложить эти правила как можно проще. Область действия глобальных *типов*

ограничивается файлом, в котором они объявляются. Глобальные *переменные* и *функции* к тому же подчиняются правилам компоновки для нескольких исходных файлов. Рассмотрим следующую ситуацию:

```
// В файле Foo.cpp
typedef int Symbol;
// В файле Bar.cpp
typedef void (*Symbol)();
```

Никакого конфликта не возникнет, если только по мазохистским соображениям вы не включите один файл с расширением .cpp в другой директивой `#include`. Символическое имя `Symbol` известно компилятору лишь в тех исходных файлах, в которых оно встречается, поэтому в разных исходных файлах его можно использовать по-разному. Следующий фрагмент неверен, поскольку на этот раз символическое имя соответствует переменной, а не типу. Имя переменной должно быть уникальным для всех файлов, передаваемых компоновщику.

```
// В файле Foo.cpp
int Symbol;
// В файле Bar.cpp
void (*Symbol)();
```

Единственное исключение из этого правила относится к перегрузке функций, о которой будет рассказано в следующем разделе. Конечно, конфликты имен часто возникают в любом достаточно большом проекте, в котором несколько программистов работают над разными исходными файлами. Одно из возможных решений — использование статических членов; другое — объявление глобальных переменных и функций статическими. Если переменная или функция объявляется статической, она определена лишь в границах исходного файла.

```
// В файле Foo.cpp
static int Symbol;
// В файле Bar.cpp
static void (*Symbol)();
```

Увидев ключевое слово `static`, компилятор проследит за тем, чтобы компоновщик не перепутал две разные версии одного символического имени при условии что исходные файлы не компилируются вместе; будут сгенерированы две разные переменные.

К любому символическому имени, объявленному в глобальном пространстве имен, можно обратиться с помощью оператора `::` без указания области действия:

```
::Fn();            // Вызвать глобальную функцию с заданным именем
int x = ::i;       // Присвоить x значение глобальной переменной
::SomeType y;      // Использовать глобально объявленный тип
```

Явно заданная область действия всегда отменяет все символические имена, определенные локально — например, внутри блока или класса.

Перегрузка

В C++ существует несколько способов многократного использования имен функций. В частности, пространства имен функций формируются на основе классов. Одноименные функции в классах, не связанных друг с другом, выполняют совершенно разные задачи. Перегрузка функций развивает великую традицию разделения пространств имен функций и позволяет многократно использовать имена функций в границах одной области действия.

Аргументы

Две функции с одинаковыми именами считаются разными, если они отличаются по количеству, порядку или типу аргументов.

```
void Fn();
```

```
void Fn(int);
void Fn(long);      // Можно, если типы long и int отличаются размером
int Fn(int);        // Нельзя – отличается только тип возвращаемого значения
int Fn(char*);      // Можно, отличаются аргументы
void Fn(int, char*);
void Fn(char*, int);    // Можно, аргументы следуют в другом порядке
void Fn(char* s, int x, int y = 17); // Можно – три аргумента вместо двух
Fn("hello", 17);    // Ошибка – совпадают две сигнатуры
```

Пока аргументы отличаются, компилятор не жалуется на изменение возвращаемого типа. Инициализация по умолчанию (такая как y=17) может присутствовать при объявлении функции, хотя позднее она может стать причиной неоднозначности при вызове функции (как в последней строке примера).

Константные функции

Константная функция, аргументы которой совпадают с аргументами неконстантной функции, тем не менее считается другой функцией. Компилятор вызывает константную или неконстантную версию в зависимости от типа переменной, указывающей или ссылающейся на объект.

```
class Foo {
public:
     void Fn();
     void Fn() const;   // Другая функция!
};
Foo* f = new Foo;
f->Fn();            // Вызывается неконстантная версия
const Foo* f1 = f;
f1->Fn();           // Вызывается константная версия
```

Видимость

В C++ существует подробная (а по мнению некоторых, даже слишком подробная) система правил, по которым можно узнать, что вы видите прямо перед собой, а что вышло из вашего поля зрения. Базовые правила для открытых защищенных и закрытых символических имен в классах и структурах настолько просты, что я не стану их пересказывать. Ниже приведена краткая сводка наиболее каверзных вопросов, относящихся к понятию видимости (visibility) в C++.

Закрытое наследование

При закрытом наследовании от базового класса все его защищенные и открытые члены становятся закрытыми в производном классе; члены закрытого базового класса недоступны для пользователей производного класса. Доступ к ним возможен лишь из функций базового и производного класса, а также из друзей производного класса.

Кроме того, производный класс нельзя преобразовать к одному из его закрытых базовых классов или надеяться, что это сделает компилятор.

```
class Mixin {
private:
     int x;
protected:
     int y;
public:
     Mixin();
     Void a();
};
```

```
class Foo : private Mixin {...};
class Bar : public Foo {...};
```

Переменная `x` видна лишь в функциях класса `Mixin` — в конструкторе и `A()`. Переменная `y` видна лишь в функциях класса `Foo`, как и функция `Mixin::A()`. Все члены `Mixin` не видны в классах, производных от `Foo` (таких как `Bar` в этом фрагменте). Все друзья `Foo` видят `x` и `A()`, а друзья `Bar` — нет.

Переобъявление членов

Хотя описанная ситуация возникает довольно редко, допускается переобъявление виртуальных функций с целью изменения их атрибутов видимости по отношению к базовому классу.

```
class Foo {
protected:
    virtual void Fn();
};
class Bar : public Foo {
public:
    virtual void Fn();
};
```

В классе `Foo` функция `Fn()` была защищенной, но в новом варианте она объявлена открытой. Для переменных класса или невиртуальных функции это сделать нельзя. Переобъявление переменной или невиртуальной функции скрывает прототип из базового класса.

```
class Foo {
private:
    int x;
public:
    void Fn();
};
class Bar : public Foo {
private:
    int x;          // Вторая переменная с тем же именем
public:
    void Fn();      // Вторая функция
};
// В клиентской программе
Bar *b = new Bar;
b->Fn();        // Вызывает Bar::Fn()
Foo* f = b;     // Можно, потому что Foo – открытый базовый класс
f->Fn();        // Вызывает Foo::Fn()
```

Существуют две разные переменные с одним локальным именем `x`. В области действия `Foo` символическое имя `x` означает `Foo::x`. В области действия `Bar` символическое имя `x` означает `Bar::x`. Конечно, для открытой или защищенной переменной `x` это вызовет невероятную путаницу, но для закрытой переменной подобной двусмысленности не будет. Пример `Fn()` показывает, какой хаос возникает при скрытии открытой или защищенной функции класса. При попытке скрыть открытую или защищенную функцию хороший компилятор C++ выдает предупреждение.

Видимость перегруженных и виртуальных функций класса

Если в базовом классе функция объявлена невиртуальной, превращать ее в виртуальную в производном классе не рекомендуется. Она поведет себя не так, как виртуальная функция, и безнадежно запутает читателей вашей программы. Но на ситуацию можно взглянуть и под другим углом. Удивительно, но факт — ключевое слово `virtual` обязано присутствовать только в базовом классе. Если оно

пропущено в производном классе, компилятор должен интерпретировать версию функции в производном классе так, словно она и там была объявлена виртуальной. Я люблю называть подобную логику работы компилятора DWIMNIS: «Do what I mean, not what I say» («Делай то, что я подразумеваю, а не то, что я говорю»). Как правило, в C++ эта логика начисто отсутствует, поэтому ее редкие проявления (как в данном случае) смотрятся неестественно. В следующем примере для обоих указателей будет вызвана функция `Bar::Fn()`:

```
class Foo {
public:
    virtual void Fn();
};
class Bar {
public:
    void Fn();      // Все равно считается виртуальной
};
Bar* b = new Bar;
b->Fn();            // Вызывает Bar::Fn()
Foo* f = b;
f->Fn();            // Также вызывает Bar::Fn()
```

Подобные ситуации нежелательны по двум причинам. Во-первых, компилятор может неправильно интерпретировать их, и тогда в последней строке будет вызвана функция `Foo::Fn()`. Во-вторых, ваши коллеги ни за что не разберутся, почему в одном месте функция `Fn()` виртуальная, а в другом — нет. После бессонной ночи они могут устроить погром в конторе.

Если в производном классе создается функция с тем же именем, но с другой сигнатурой, она скрывает *все* сигнатуры базового класса для данной функции, но только в области действия производного класса. Понятно? Нет? Что ж, вы не одиноки.

```
class Foo {
public:
    virtual void Fn();
    virtual void Fn(char*);
};
class Bar {
public:
    virtual void Fn(int);     // Можно, но не желательно
};
```

Вероятно, многие новички-программисты допоздна засиживались на работе, пытаясь разобраться в происходящем. А происходит следующее:

- При попытке вызвать `Fn()` через `Bar*` доступной будет лишь одна сигнатура, `void Fn(int)`. Обе версии базового класса скрыты и недоступны через `Bar*`.
- При преобразовании `Bar*` в `Foo*` становятся доступными обе сигнатуры, объявленные в `Foo`, но не сигнатура `void Fn(int)`. Более того, это не переопределение, поскольку сигнатура `Bar::Fn()` отличается от версии базового класса. Другими словами, ключевое слово `virtual` никак не влияет на работу этого фрагмента.

Если вам когда-нибудь захочется сделать нечто похожее, встаньте с кресла, медленно прогуляйтесь вокруг дома, сделайте глубокий вдох, сядьте за компьютер и придумайте что-нибудь другое. Если уж перегружать, то перегружайте все сигнатуры функции. Никогда не перегружайте часть сигнатур и никогда не добавляйте новые сигнатуры в производный класс без переопределения всех сигнатур функции базового класса. Если это покажется слишком сложным, запомните хорошее правило: когда при чтении программы возникают вопросы, вероятно, ваше решение неудачное.

Друзья

Любой класс может объявить что-нибудь своим другом (friend). Друзья компилируются обычным образом, за исключением того, что все защищенные и закрытые члены дружественного класса видны так, словно друг является функцией этого класса. Друзьями можно объявлять функции — как глобальные, так и члены классов. Классы тоже могут объявляться друзьями других классов; в этом случае во всех функциях класса-друга «видны» все члены того класса, другом которого он является.

```
class Foo;
class BarBar {
public:
    int Fn(Foo*);
};
class Foo {
friend void GlobalFn();           // Дружественные глобальные функции
friend class Bar;                 // Дружественный класс
friend int BarBar::Fn(Foo*);      // Дружественная функция класса
friend class DoesNotExist;        // См. Ниже
private:
    int x;
    struct ListNode {
        ListNode* next;
        void* datum;
        ListNode() : next(NULL), datum(NULL) {}
    } head;
protected:
    int y;
public:
    void G();
};
void GlobalFn()
{
    Foo* f = new Foo;
    f->x = 17;          // Разрешается из-за дружеских отношений
}
class Bar {
private:
    Foo* f;
public:
    Bar() : f(new Foo) {}
    void WalkList();
};
void Bar::WalkList()
{
    Foo::ListNode* n = f->head.next;
    for (; n != NULL; n = n->next)
        cout << n->datum << endl;
}
int BarBar::Fn(Foo* f)
{
    return f->x;
}
```

Друзей принято объявлять сначала, перед членами класса и перед ключевыми словами public, protected и private. Это объясняется тем, что на друзей не действуют обычные атрибуты видимости; нечто либо является другом, либо не является. Весь фрагмент программы после определения класса Foo вполне допустим. Друзья имеют доступ ко всем членам Foo, включая закрытые. В этом примере есть одна действительно интересная строка — та, в которой другом объявляется несуществующий класс DoesNotExist. Как ни странно, она не вызовет ни предупреждения, ни ошибки компилятора. Объявления друзей игнорируются на момент компиляции Foo. Они используются лишь тогда, когда будет компилироваться друг. Даже когда друга не существует, компилятор остается в счастливом неведении.

Типы и операторы

Темы, рассматриваемые в этом разделе, на первый взгляд не кажутся близкими, однако все они вращаются вокруг общей концепции — абстрактных типов данных.

Конструкторы

Конструктор можно рассматривать двояко — как функцию, инициализирующую объект, или, с позиций математики, как отображение аргументов конструктора на домен класса. Я предпочитаю второй подход, поскольку он помогает разобраться с некоторыми языковыми средствами (например, операторами преобразования).

С конструкторами связаны очень сложные правила, но каждый программист C++ должен досконально знать их, иначе минимум три ночи в году ему придется проводить за отладкой.

Конструкторы без аргументов

Если в вашем классе имеется конструктор, который вызывается без аргументов, он используется по умолчанию в трех следующих случаях.

```
class Foo {
public:
     Foo();
};
class Bar : public Foo {     // 1. Базовый класс
public:
     Bar();
};
class BarBar {
private:
     Foo f;                  // 2. Переменная класса
};
Foo f;                       // 3. Созданный экземпляр Foo
Foo* f1 = new Foo;           // 3. То же, что и предыдущая строка
```

Если в списке инициализации членов (см. следующий раздел) конструктора Bar не указан какой-нибудь другой конструктор Foo, то при каждом создании экземпляра Bar будет вызываться конструктор Foo без аргументов. Аналогично, если f отсутствует в списке инициализации членов конструктора BarBar, будет использован конструктор Foo без аргументов. Наконец, при каждом создании экземпляра Foo без указания конструктора по умолчанию используется конструктор без аргументов.

Конструкторы с аргументами

Конструкторы, как и все остальные функции, можно перегружать. Вы можете объявить столько сигнатур конструкторов, сколько вам потребуется. Единственное настоящее отличие между сигнатурами конструкторов и обычных функций заключается в том, что конструкторы не имеют возвращаемого значения и не могут объявляться константными. Если вы объявите какие-либо

конструкторы с аргументами, но не объявите конструктора без аргументов, то компилятор не позволит конструировать объекты этого класса, даже в качестве базового для другого класса, с использованием конструктора без аргументов.

```
class Foo {
public:
    Foo(char*);
};

Foo f;   // Нельзя – нет конструктора без аргументов!
class Bar : public Foo {
public:
    Bar();
};
Bar::Bar()
{
    // Ошибка! Нет конструктора Foo без аргументов
}
```

Списки инициализации членов

Чтобы избавиться от этой проблемы, в C++ находится очередное применение символу **:** — для создания списков инициализации членов. Так называется список спецификаций конструкторов, разделенных занятыми и расположенных между сигнатурой конструктора и его телом.

```
class Foo {
public:
    Foo(char*);
};
class Bar : public Foo {
public:
    Bar(char*);
};
class BarBar {
private:
    Foo f;
    int x;
public:
    BarBar();
};
Bar::Bar(char* s) : Foo(s) {...}
BarBar::BarBar : f("Hello"), x(17) {...}
```

В конструкторе `Bar` список инициализации членов используется для инициализации базового класса `Foo`. Компилятор выбирает используемый конструктор на основании сигнатуры, определяемой по фактическим аргументам. При отсутствии списка инициализации членов сконструировать `Bar` было бы невозможно, поскольку компилятор не мог бы определить, какое значение должно передаваться конструктору базового класса `Foo`. В конструкторе `BarBar` список инициализации членов использовался для инициализации (то есть вызова конструкторов) переменных `f` и `x`. В следующем варианте конструктор работает не столь эффективно (если только компилятор не отличается сверхъестественным интеллектом):

```
BarBar::BarBar() : f("Hello")
{
    x = 17;
}
```

Во втором варианте переменная x сначала инициализируется значением 0 (стандартное требование C++) с использованием по умолчанию конструктора `int` без аргументов, а затем в теле конструктора ей присваивается значение 17. В первом варианте имеется всего одна инициализация и потому экономится один-два машинных такта. В данном примере это несущественно, поскольку переменная `x` — целая, но если бы она относилась к более сложному классу с конструктором без аргументов и перегруженным оператором присваивания, то разница была бы вполне ощутима.

Списки инициализации членов нужны там, где у базового класса или переменной нет конструктора без аргументов (точнее, есть один и более конструктор с аргументами, но нет ни одного определенного пользователем конструктора без аргументов). Списки инициализации членов не обязательны в тех ситуациях, когда все базовые классы и переменные класса либо не имеют конструкторов, либо имеют пользовательский конструктор без аргументов.

Порядок вызова конструкторов

Если класс не содержит собственных конструкторов, он инициализируется так, словно компилятор создал конструктор без аргументов за вас. Этот конструктор вызывает конструкторы без аргументов базовых классов и переменных класса. Четко определенный порядок вызова конструкторов не зависит от того, используются конструкторы стандартные или перегруженные, с аргументами или без:

1. Сначала вызываются конструкторы базовых классов в порядке их перечисления в списке наследования (еще один список, в котором после символа `:` перечисляются базовые классы, разделенные запятыми).

2. Затем вызываются конструкторы переменных класса в порядке их объявления в объявлении класса.

3. После того как будут сконструированы все базовые классы и переменные, выполняется тело вашего конструктора.

Описанный порядок применяется рекурсивно, то есть первым конструируется первый базовый класс первого базового класса... и т. д. Он не зависит от порядка, указанного в списке инициализации членов. Если бы дело обстояло иначе, для разных перегруженных конструкторов мог бы использоваться разный порядок конструирования. Тогда компилятору было бы трудно гарантировать, что деструкторы будут вызываться в порядке, обратном порядку вызова конструкторов.

Конструкторы копий

Конструктор копий (copy constructor) определяется специальной сигнатурой:

```
class Foo {
public:
    Foo(const Foo&);
};
Foo::Foo(const Foo& f)...
```

Конструктор копий предназначен для создания копий объектов. Эта задача может возникнуть в самых разных обстоятельствах.

```
void Fn(Foo f) {...}
void Gn(Foo& f) {...}
Foo f;
Foo f1(f);
Foo f2 = f;    // Конструирование, а не присваивание!
Fn(f);         // Вызывает конструктор копий для передачи по назначению
const Foo f3;
Gn(f3);        // Конструктор копий используется
               // для создания неконстантной копии
```

Давайте внимательно рассмотрим этот фрагмент. Строка `Foo f1(f);` создает новый экземпляр класса Foo, передавая другой экземпляр класса `Foo` в качестве аргумента. Это всегда можно сделать, если

класс `Foo` не содержит чисто виртуальных функций. Не важно, объявили ли вы свой собственный конструктор копий; если нет, компилятор построит его за вас. Не важно, есть ли в `Foo` другие пользовательские конструкторы; в отличие от конструкторов без аргументов, конструктор копий доступен *всегда.*

Строка `Foo f2 = f` выглядит как присваивание из-за присутствия оператора =, но на самом деле это альтернативный вариант вызова конструктора копий. Чтобы понять, чем присваивание отличается от инициализации, спросите себя: «Был ли объект сконструирован заранее или же его создание является частью команды?» Если объект уже существует, вы имеете дело с присваиванием. Если он создается на месте, как в пашем примере, используется конструктор копий.

При вызове функции `Fn()` происходит передача по значению копии `Foo`. Конструктор копий используется для создания временной копии, существующей лишь во время выполнения `Fn()`. После этого вызывается деструктор копии, который уничтожает ее.

Вызов функции `Gn()`, вероятно, ошибочен, и хороший компилятор прочитает вам суровую нотацию о стиле программирования на C++ — что-нибудь вроде:

«Создается временная неконстантная копия — поучись программировать, тупица!» По крайней мере, со *мной* компиляторы обычно поступают именно так. Проблема заключается в том, что аргумент передается по ссылке, однако фактический аргумент является константным, а формальный — нет. Все изменения аргумента внутри `Gn()` вносятся в копию, а не в оригинал.

В создаваемом компилятором конструкторе копий по умолчанию используется строго определенная последовательность вызова конструкторов копий базовых классов и переменных класса.

1. Конструкторы копий базовых классов вызываются в том порядке, в котором они объявлены в списке наследования.
2. Конструкторы копий переменных вызываются в том порядке, в котором они объявлены в объявлении класса.

Описанный порядок применяется рекурсивно, то есть первым копируется первый базовый класс первого базового класса... и т. д. Звучит знакомо, не правда ли? Тот же порядок, что и для любого другого конструктора.

С конструкторами копий, в отличие от всех остальных, компилятор ведет себя гордо и ревниво. Если вы перегрузите конструктор копий для некоторого класса, компилятор, фигурально выражаясь, умывает руки и отправляется домой. При отсутствии явного вызова конструкторов копий базовых классов и переменных класса в списке инициализации членов вашего собственного конструктора копий компилятор будет использовать *конструктор без аргументов* для инициализации базовых классов и переменных.

```
class Foo {...};
class Bar : public Foo {
private:
    Foo f;
public:
    Bar(const Bar&);
};
// Вероятно, ошибка
Bar::Bar(const Bar& b)
{
    // Стоп! Нет списка инициализации членов
    // Будут использованы конструкторы без аргументов
    // базового класса и переменной
}
// Вероятно, ошибки нет
Bar::Bar(const Bar& b) : Foo(b), f(b.f) {...}
```

Компилятор очень сильно обидится на первый конструктор копий — так сильно, что он спустит ваше произведение в мусоропровод и даже не сообщит об этом. Для инициализации базового класса и переменной будет использован конструктор Foo без аргументов. В 99 случаях из 100 это совсем не то, чего вы добивались; обычно требуется, чтобы базовые классы и переменные тоже копировались. Вероятно, второй вариант правилен. Базовый класс и переменная присутствуют в списке инициализации членов, поэтому будут вызваны их конструкторы копий (компилятор преобразует b к типу Foo в выражении Foo(b)).

В некоторых ситуациях вас интересует именно поведение компилятора по умолчанию. В качестве примера рассмотрим следующий базовый класс, который присваивает уникальный серийный номер каждому производному объекту.

```
class Serialized {
private:
     static int NextSerialNumber;
     int serialNumber;
public:
     Serialized(const Serialized&);
     Serialized();
     int SerialNumber();
};
// В Serialized.cpp
int Serialized::NextSerialNumber = 0;
Serialized::Serialized() : serialNumber(NextSerialNumber++)
{
}
Serialized::Serialized(const Serialized&) : serialNumber(NextSerialNumber++)
{
}
int Serialized::SerialNumber()
{
     return serialNumber;
}
```

Нас не интересует, какой конструктор — без аргументов или копий — выберет компилятор во время компиляции производного класса, поскольку мы перегрузили оба конструктора, и они делают одно и то же.

Закрытые и защищенные конструкторы

Конструкторы часто объявляются закрытыми и защищенными, чтобы пользователи не могли создавать экземпляры класса. Если конструктор объявлен закрытым; только обычные и статические функции класса могут создавать стековые экземпляры класса или использовать его в операторе new (по крайней мере, с данным конструктором). Если конструктор объявлен защищенным, пользователь может создавать экземпляры базового класса, поскольку конструктор базового класса может «вызываться» из конструктора производного класса. Пусть для этого потребуется некоторое воображение, но семантика именно такова. У этой логики есть один недостаток - она не совсем надежна. Если конструктор защищен, любая функция базового или производного класса (включая статические) может создать экземпляр базового класса.

```
class Foo {
protected:
     Foo();
};
```

```
class Bar : public Foo {
public:
    Foo* Fn();
};
Foo Bar::Fn()
{
    return new Foo;    // Работает вопреки всем вашим усилиям
}
```

Возможно, вы полагали, что `Foo` — абстрактный базовый класс и его экземпляры создать невозможно. Оказывается, ничего подобного! В системе защиты открывается зияющая дыра. Друзья классов `Foo` и Bar тоже могут создавать экземпляры `Foo`. Единственный «железный» способ, который стопроцентно гарантирует невозможность создания экземпляров класса — включение в него хотя бы одной чисто виртуальной функции.

Анонимные экземпляры

Анонимным экземпляром (anonymous instance) называется объект, который... впрочем, сейчас увидите.

```
struct Point {
    int X;
    int Y;
    Point(int x, int y) : X(x), Y(y) {}
};
double distance(Point p)
{
    return sqrt(double(p.X) * double(p.X) + double(p.Y) * double(p.Y));
}
double d = distance(Point(17, 29));
```

Аргумент функции `distance()` представляет собой анонимный экземпляр. Мы не создали переменной для его хранения. Анонимный экземпляр существует лишь во время вычисления выражения, в котором он встречается.

Анонимные экземпляры обычно связываются с простыми структурами вроде `Point`, но их можно использовать для любого класса.

Инициализация глобальных объектов

В спецификации языка порядок конструирования глобальных объектов выглядит довольно сложно. Если же учесть причуды коммерческих компиляторов C++, этот порядок становится и вовсе непредсказуемым. В соответствии со спецификацией должны вызываться конструкторы глобальных объектов, включая конструкторы статических переменных классов и структур, однако многие компиляторы этого не делают. Если вам повезло и ваш компилятор считает, что конструкторы важны для глобальных переменных, порядок конструирования глобальных объектов зависит от воображения разработчика компилятора. Ниже перечислены некоторые правила, которые теоретически должны соблюдаться:

1. Перед выполнением каких-либо операций все глобальные переменные инициализируются значением 0.
2. Объекты, находящиеся в глобальных структурах или массивах, конструируются в порядке их появления в структуре или массиве.
3. Каждый глобальный объект конструируется до его первого использования в программе. Компилятор сам решает, следует ли выполнить инициализацию до вызова функции `main()` или отложить ее до первого использования объекта.

4. Глобальные объекты, находящиеся в одном «модуле трансляции» (обычно файле с расширением .cpp), инициализируются в порядке их появления в этом модуле. В сочетании с правилом 3 это означает, что инициализация может выполняться по модулям, при первом использовании каждого модуля.

Вот и все. Внешне простая последовательность глобальных объявлений на самом деле полностью подчиняется всем капризам разработчика компилятора. Она может привести к нужному результату или сгореть синим пламенем.

```
// В файле file1.cpp
Foo foo;
Foo* f = &foo;
// В файле file2.cpp
extern Foo* f;
Foo f1(*f);             // Используется конструктор копий
```

Если бы все это находилось в одном исходном файле, ситуация была бы нормальной. Со строкой `Foo* f = &foo;` проблем не возникает, поскольку глобальные объекты одного исходного файла заведомо (хе-хе) инициализируются в порядке их определения. Другими словами, когда программа доберется до этой строки, объект `foo` уже будет сконструирован. Тем не менее, никто не гарантирует, что глобальные объекты в файле file1.cpp будут инициализированы раньше глобальных объектов в файле file2.cpp. Если file2.cpp будет обрабатываться первым, `f` оказывается равным 0 (`NULL` на большинстве компьютеров), и при попытке получить по нему объект ваша программа героически умрет.

Лучший выход — сделать так, чтобы программа не рассчитывала на конкретный порядок инициализации файлов .cpp. Для этого используется стандартный прием — в заголовочном файле .h определяется глобальный объект со статической переменной, содержащей количество инициализированных файлов .cpp. При переходе от 0 к 1 вызывается функция, которая инициализирует все глобальные объекты библиотечного файла .cpp. При переходе от 1 к 0 все объекты этого файла уничтожаются.

```
// В файле Library.h
class Library {
private:
     static int count;
     static void OpenLibrary();
     static void CloseLibrary();
public:
     Library();
     ~Library();
};
static Library LibraryDummy;
inline Library::Library()
{
     if (count++ == 0)
          OpenLibrary();
}
inline Library::~Library()
{
     if (--count == 0)
          CloseLibrary();
}
// В Library.cpp
int Library::count = 0;     // Делается перед выполнением вычислений
int aGlobal;
```

```
Foo* aGlobalFoo;
void Library::OpenLibrary()
{
     aGlobal = 17;
     aGlobalFoo = new Foo;
}
void Library::CloseLibrary()
{
     aGlobal = 0;
     delete aGlobalFoo;
     aGlobalFoo = NULL;
}
```

К этому нужно привыкнуть. А происходит следующее: файл .h компилируется со множеством других файлов .cpp, один из которых - Library.cpp. Порядок инициализации глобальных объектов, встречающихся в этих файлах, предсказать невозможно. Тем не менее, каждый из них будет иметь свою статическую копию `LibraryDummy`. При каждой инициализации файла .cpp, в который включен файл Library.h, конструктор `LibraryDummy` увеличивает счетчик. При выходе из `main()` или при вызове `exit()` файлы .cpp уничтожают глобальные объекты и уменьшают счетчик в деструкторе `LibraryDummy`. Конструктор и деструктор гарантируют, что `OpenLibrary()` и `CloseLibrary()` будут вызваны ровно один раз.

Этот прием приписывается многим разным программистам, но самый известный пример его использования встречается в библиотеке iostream. Там он инициализирует большие структуры данных, с которыми работает библиотека, ровно один раз и лишь тогда, когда это требуется.

Деструкторы

Деструкторы вызываются каждый раз, когда стековый объект выходит из области действия (включая анонимные экземпляры и временные объекты, создаваемые компилятором) или когда для динамического объекта вызывается оператор `delete`. К деструкторам относится ряд малоизвестных фактов.

Порядок вызова

Деструкторы гарантированно вызываются в порядке, обратном порядку вызова конструкторов. Это означает, что сначала вызывается тело конструктора объекта, затем деструкторы переменных класса в порядке, обратном порядку их перечисления в объявлении класса, и наконец деструкторы базовых классов, начиная с последнего в списке наследования и кончая первым базовым первого базового и т.д.

Уничтожение глобальных объектов

Если от разговоров об инициализации глобальных объектов у вас закружилась голова, могу вас обрадовать. Если разработчик вашего компилятора справился со своей работой, деструкторы глобальных объектов гарантированно вызываются в порядке, точно обратном порядку вызова конструкторов.

Глобальные объекты уничтожаются при выходе из области действия `main()` или при вызове `exit()`.

Невиртуальные деструкторы

C++ выбирает вызываемый деструктор по типу указателя на объект. Если указатель имеет тип `base*` (указатель на базовый класс), возникнут проблемы, если только деструктор класса не виртуален.

```
class Foo {
public:
     ~Foo();
};
```

```
class Bar : public Foo {
private:
     int* numbers;
public:
     Bar() : numbers(new int[17]) {...}
     ~Bar();
};
Bar* b = new Bar;
delete b;     // Вызывает Bar::~Bar()
Foo* f = new Bar;
delete f;     // Ой! Вызывается Foo::Foo()!
```

При удалении f массив, на который ссылается переменная numbers, превращается в некое подобие Летучего Голландца, обреченного на вечные скитания в памяти. Чтобы избежать беды, достаточно объявить оба деструктора виртуальными; в этом случае независимо от типа указателя (кроме, конечно, void*) уничтожение будет начинаться с Bar: :~Ваг().

Другая, более коварная проблема с невиртуальными деструкторами возникает при организации нестандартного управления памятью. Компилятор сообщает вашему перегруженному оператору размер уничтожаемого объекта — сюрприз! Для невиртуального деструктора этот размер может оказаться неверным. Представьте себе удивление вашей программы, когда ей сообщат, что объект имеет размер 20 байт, хотя на самом деле он равен 220 байтам! Разработчики компиляторов C++ любят похвастаться подобными проделками за кружкой пива после работы.

Мораль: деструкторы следует делать виртуальными. Исключение составляют ситуации, когда ваш класс или структура не имеет производных классов или у вас найдутся чрезвычайно веские причины поступить иначе.

Прямой вызов деструкторов

Деструктор можно вызвать и напрямую, не прибегая к оператору delete, поскольку это такая же функция, как и все остальные. Впрочем, до того, как мы займемся нестандартным управлением памятью, вряд ли это будет иметь какой-нибудь смысл.

```
class Foo {
public:
     ~Foo();
};
Foo* f = new Foo;
f->Foo::~Foo();
```

Позднее мы воспользуемся этой возможностью, а пока сохраните ее в своей коллекции C++.

Присваивание

Присваивание одного объекта другому в C++ — дело серьезное. Впрочем, в обилии запутанных правил есть и положительная сторона — благодаря им вы постоянно остаетесь начеку и уделяете больше внимания программе.

Синтаксис и семантика присваивания

Для присваивания одного объекта другому используется оператор =.

```
Foo f;
Foo f1;
f1 = f;
```

Присваивание выполняется в третьей строке. Если бы f и f1 были целыми или чем-нибудь столь же простым, смысл этой строки был бы предельно ясен: содержимое области памяти, на которую

ссылается f, копируется в область памяти, на которую ссылается f1. Только и всего. Но если Foo относится к нетривиальному классу, в C++ все заметно усложняется. В приведенном примере компилятор предоставляет оператор = по умолчанию, который вызывается для выполнения фактического копирования. Как и с конструкторами копий, вы можете спокойно сидеть и смотреть, как компилятор вкалывает за вас, или написать свой собственный оператор =. То, что делает версия по умолчанию, вам может и не понравиться, особенно в момент освобождения памяти деструктором класса.

```
class String {
private:
    char* s;
public:
    String(char*);
    ~String();
    void Dump(ostream& os);
};
String::String(char* str) : s(NULL)
{
    if (str == NULL) {      // NULL означает пустую строку
        s = new char[1];
        *s = ‘\0’;
    }
    else {
        s = new char[strlen(str) + 1];
        strcpy(s, str);
    }
}
String::~String()
{
    delete s;
}
void String::Dump(ostream& os)
{
    os << “\”” << s << “\””;
}
String* s1 = new String(“Hello”);
String* s2 = new String(“Goodbye”);
s2 = s1;
delete s1;          // Память освободилась, вроде все нормально...
s2->Dump();         // Облом! Ха-ха-ха!
delete s2;          // Помогите, убивают! Ха-ха-ха!
```

По умолчанию компилятор копирует содержимое s2->s поверх содержимого s1->s. При этом копируется значение указателя, а не символы, поэтому после присваивания возникают две большие проблемы. Два разных объекта ссылаются на одну область памяти, и никто не ссылается на копию Goodbye, созданную командой `String* s2 = new String( "Goodbye");`. Дальше — больше; при удалении s1 деструктор освобождает область памяти, на которую ссылается s1. Однако на эту память продолжает ссылаться указатель s2->s. Попытка вывести s2->s дает совершенно безумные результаты. «Комедия ошибок» достигает кульминации при попытке удалить s2, поскольку менеджер памяти попытается освободить ранее освобожденную область. Чего только не бывает в C++!

Разумеется, та же проблема возникает и при создании копий. Конструктор копий по умолчанию копирует указатель, а не данные, на которые он ссылается. По этой причине конструктор копий и оператор = обычно перегружаются одновременно.

Присваивание и инициализация

Мы уже обсуждали, чем инициализация отличается от присваивания, но эта тема настолько важна, что я повторю еще раз. Если объект слева от оператора = был сконструирован заранее, = означает присваивание. Если в этом выражении он конструируется впервые, речь идет о конструировании и конструкторах. В следующем примере первая строка с символом = выполняет инициализацию и вызывает конструктор копий. Вторая строка выполняет присваивание и вызывает оператор =.

```
Foo f;
Foo f1 = f;    // Инициализация; f1 еще не существует
f1 = f;        // Присваивание: объект f1 уже сконструирован
```

Присваивание по умолчанию

Оператор = по умолчанию, как и конструктор копий по умолчанию, ведет себя четко определенным образом. Как и конструктор копий, который рекурсивно вызывает другие конструкторы копий, оператор = по умолчанию не ограничивается простым копированием битов из одного объекта в другой. Последовательность его действий выглядит так:

1. Присваивание для базовых классов выполняется в порядке их перечисления в списке наследования. При этом используются перегруженные операторы = базовых классов или в случае их отсутствия — оператор = по умолчанию.
2. Присваивание переменных класса выполняется в порядке их перечисления в объявлении класса. При этом используются перегруженные операторы = базовых классов или в случае их отсутствия — оператор = по умолчанию.

Эти правила применяются рекурсивно. Как и в случае с конструкторами, сначала выполняется присваивание для первого базового класса первого базового класса и т. д.

Перегрузка оператора =

Перегрузка оператора = практически не отличается от перегрузки всех остальных операторов. Пока нас интересует сигнатура оператора =, которая выглядит так: `X& X::operator=(const X&)`.

```
class String {
private:
    char* s;
public:
    String(char*);
    ~String();
    String(const String&);    // Возможно, тоже решает проблему
    String& operator=(const String&);
    void Dump(ostream& os);
};
String::String(char* s) : s(NULL)
{
    if (str == NULL) {      // NULL означает пустую строку
        s = new char[1];
        *s = '\0';
    }
    else {
        s = new char[strlen(str) + 1];
        strcpy(s, str);
```

```
        }
    }
    String::~String()
    {
        delete s;
    }
    String::String(const String& s1) : s(NULL)
    {
        s = new char[strlen(s1.s) + 1];
        strcpy(s, s1.s);
    }
    String& String::operator=(const String& s1)
    {
        if (this == &s1) return *this;
        delete s;      // Уничтожить предыдущее значение
        s = new char[strlen(s1.s) + 1];
        strcpy(s, s1.s);
        return *this;
    }
    void String::Dump(ostream& os)
    {
        os << "\"" << s << "\"";
    }
```

Конструктор копий и оператор = вместо простого копирования адреса теперь создают копию новой строки. Деструктор стал безопасным, и миру ничего не угрожает.

Ниже показан обобщенный вид оператора =, который стоит занести в долговременную память (не компьютерную, а вашу собственную):

1. Убедитесь, что не выполняется присваивание вида `x=x;`. Если левая и правая части ссылаются на один объект, делать ничего не надо. Если не перехватить этот особый случай, то следующий шаг уничтожит значение до того, как оно будет скопировано.
2. Удалите предыдущие данные.
3. Скопируйте значение.
4. Возвратите указатель `*this`.

Оператор = возвращает `*this`, чтобы стало возможным вложенное присваивание вида `a=b=c`. В C++, как и в C, значением этого выражения является присваиваемая величина. Выражение интерпретируется справа налево, как `a=(b=c)`.

А теперь — плохие новости. Как и в случае с конструкторами копий, при перегрузке оператора = C++ умывает руки и отправляется домой. Если вы перегрузили оператор =, то на вас ложится ответственность за выполнение присваивания для переменных и базовых классов; по умолчанию базовые классы и переменные левостороннего объекта остаются без изменений.

Присваивание для переменных класса

Иногда переменные класса относятся к простейшим типам данных (например, `int`), и тогда присваивание выполняется с помощью оператора =, предоставленного компилятором. Иногда (например, для класса `String`) их приходится копировать вручную. В остальных случаях переменные относятся к какому-нибудь нетривиальному классу. Лучший выход из положения — присвоить что-нибудь таким переменным. При этом компилятор определяет, существует ли для переменной перегруженный оператор = или он должен использовать свой собственный вариант по умолчанию.

```
    class Foo {
```

```
public:
     Foo& operator=(const Foo&);
};
class Bar {
public:
     // Оператор = не перегружен
};
class FooBar {
private:
     Foo f;
     Bar b;
public:
     FooBar& operator=(const FooBar&);
};
FooBar& FooBar::operator=(const FooBar& fb)
{
     if (this == &fb) return *this;
     f = fb.f;      // Используется перегруженный оператор = класса Foo
     f = fb.b;      // Используется оператор = по умолчанию
     return *this;
}
```

Применяя эту методику, вы не заботитесь о том, существует ли для переменной перегруженный оператор =. Об этом должен думать компилятор.

Присваивание для базовых классов

Присваивание для базовых классов сопряжено с некоторыми синтаксическими ухищрениями. Если вы никогда их не видели, вероятно, на поиск правильной комбинации уйдет немало времени. Выглядит она так:

```
class Foo {...}
class Bar : public Foo {
public:
     Bar& operator=(const Bar&);
};
Bar& Bar::operator=(const Bar& b)
{
     if (this == &b) return *this;
     this->Foo::operator=(b);         // Чего-чего?
     return *this;
}
```

Другие варианты, которые могут придти в голову (например, `*((Foo)this)=b;`), не работают — поверьте мне на слово. Все они создают временные копии. Показанный вариант работает, поскольку компилятор знает, как преобразовать `Bar` в `Foo` в аргументе. Он работает независимо от того, перегружали вы `Foo::operator=` или нет. Даже если не перегружали, оператор все равно присутствует, и его можно вызвать по полному имени `Foo::operator=`.

Другие сигнатуры оператора =

Оператор = не ограничен одной сигнатурой. Его можно перегрузить так, чтобы в правой части присваивания мог стоять аргумент любого другого типа. Сигнатура `X& X::operator=(const X&)` выделяется на общем фоне тем, что компилятор предоставляет ее версию по умолчанию и использует эту сигнатуру в стандартном алгоритме рекурсивного присваивания.

```
class String {
// Как раньше
public:
    String& operator=(const String&);     // Нормальный вариант
    String& operator=(char*);             // Перегруженный вариант
    String& operator=(int);               // Вызывает atoi()
};
```

В показанном фрагменте создается несколько перегруженных вариантов оператора = для различных типов данных в правой части выражения. Второй вариант позволяет избежать конструирования временного объекта `String` из `char*` лишь для того, чтобы присвоить его объекту в левой части. Третий вариант выполняет преобразование другого рода. Тем не менее, лишь первый вариант перегружает (то есть заменяет) версию оператора по умолчанию.

Перегрузка операторов

Одна из приятных особенностей C++ — возможность расширения смысла операторов. Это упрощает чтение программы, поскольку вам уже не придется изобретать дурацкие имена функций вроде `Add` там, где знак + имеет совершенно очевидный смысл. Тем не менее, из личного опыта я знаю две проблемы, связанные с перегруженными операторами. Во-первых, их чрезмерное применение превращает программу в хаос. Во-вторых, большинство программистов никогда их не использует. Приведенный ниже список не претендует на полноту, однако он поможет подготовить поле для дальнейшего изложения материала.

Функциональная форма операторов

Операторы (например, +) используются в двух вариантах: как особая синтаксическая форма или как функция. В C++ функциональная форма всегда представляет собой ключевое слово `operator`, за которым следует символ оператора.

```
class Foo {...}
Foo x, y, z;
z = x + y;              // Инфиксная (нормальная) форма
z = operator+(x, y);    // Функциональная форма (внешняя функция)
z = x.operator+(y);     // Функциональная форма (функция класса)
```

С концептуальной точки зрения три последние строки эквивалентны, хотя на практике, вероятно, оператор будет определен либо в виде внешней функции, либо в виде функции класса, но не в обоих вариантах сразу. Для бинарных операторов знак оператора указывается между двух аргументов в инфиксной форме. В форме внешней функции оба аргумента передаются глобальной функции. В форме функции класса объект, которому принадлежит вызываемый оператор, указывается слева, а аргумент — справа от знака оператора. Унарные операторы (такие как ! и ~) тоже могут перегружаться. Форма внешней функции вызывается с одним аргументом, а форма функции класса вызывается без аргументов (операция выполняется с объектом, находящимся слева от оператора `.` или `->`).

Не разрешается перегружать встроенные операторы (например, оператор целочисленного сложения). Чтобы обеспечить выполнение этого условия, компилятор требует, чтобы хотя бы один аргумент каждого перегруженного оператора относился к пользовательскому типу (обычно к классу). Выбор ограничен операторами, уже определенными в C++. Во время долгих ночных отладок мне часто хотелось создать оператор с именем #$%^&, но C++ на этот счет неумолим.

Перегруженные операторы наследуют приоритеты и атрибуты группировки от встроенных операторов, поэтому вы не можете, например, изменить стандартный порядок группировки «слева направо» для оператора +. Не существует ограничений на тип значения, возвращаемого перегруженным оператором, и один оператор можно перегружать произвольное число раз при условии, что сигнатуры остаются уникальными.

Перегрузка операторов в форме внешних функций

Чтобы перегрузить оператор в форме внешней функции, необходимо определить глобальную функцию.

```
class String {
friend String& operator+(const String&, const String&);
private:
    char* s;
public:
    // Конструкторы и т.д.
}
String& operator+(const String& s1, const String& s2)
{
    char* s = new char[strlen(s1.s) + strlen(s2.s) + 1];
    strcat(s, s1.s, s2.s);
    String newStr(s);
    delete s;
    return newStr;
}
String s1 = "Hello";
String s2 = "Goodbye";
String s3 = s1 + s2;
```

Перегруженная функция выглядит так же, как и любая глобальная функция (если не считать странного имени). Именно для таких случаев и были придуманы друзья. Если бы мы не объявили функцию `operator+` другом, то она не имела бы доступа к переменной `s`, и мы оказались бы перед выбором: то ли разрешить всем на свете доступ к `char*`, то ли перейти к менее эффективной реализации, при которой строка копируется при каждом обращении к ней. С концептуальной точки зрения `operator+` является частью библиотеки `String`, поэтому нет ничего страшного в том, чтобы объявить эту функцию другом и вручить ей ключи к внутреннему устройству `String`.

Внешними функциями могут перегружаться любые операторы, кроме операторов преобразования, =, `[]`, `()` и `->` — все эти операторы должны перегружаться только функциями класса.

Перегрузка операторов в форме функций класса

Синтаксис напоминает обычную перегрузку функций класса, разве что количество аргументов уменьшается на 1 по сравнению с формой внешней функции.

```
class String {
private:
    char* s;
public:
    // Конструкторы и т.д.
    String& operator+(const String&) const;
};
String& String::operator+(const String& s1) const
{
    char* s2 = new char[strlen(s1.s) + strlen(s) + 1];
    strcat(s2, s1, s);
    String newStr(s2);
    delete s2;
    return newStr;
}
```

```
String s1 = "Hello";
String s2 = "Goodbye";
String s3 = s1 + s2;
```

Любой оператор может быть перегружен в форме функции класса. Если оператор может перегружаться как внешней функцией, так и функцией класса, какую из двух форм выбрать? Ответ: используйте перегрузку в форме функции класса, если только у вас не найдется веских причин для перегрузки внешней функцией. Из этих причин наиболее распространены следующие:

1. Первый аргумент относится к базовому типу (например, `int` или `double`).
2. Тип первого аргумента определен в коммерческой библиотеке, которую нежелательно модифицировать.

Компилятор ищет перегрузку в форме функций класса, просматривая левую часть бинарных операторов и единственный аргумент унарных. Если ваш тип указывается справа и вы хотите воспользоваться перегрузкой в форме функции класса, вам не повезло. Самый распространенный пример перегрузки в форме внешней функции — оператор << в библиотеке ostream.

```
ostream& operator<<(ostream& os, const String& s)
{
    os << str.s;       // Предполагается, что данная функция является другом
    return os;
}
```

Перегрузка должна осуществляться в форме внешней функции, поскольку ваш тип, `String`, находится справа — если, конечно, вы не хотите залезть в готовые заголовки iostream.h и включить в класс `ostream` перегрузку в форме функции класса для своего класса `String`. Наверное, все-таки не хотите.

Примечание: предыдущий пример может не работать в вашем компиляторе, если функции `strlen` и `strcat`, как это часто бывает, по недосмотру разработчиков получают `char*` вместо `const char*`. Вы можете решить, что игра не стоит свеч, и объявить функцию неконстантной, но это выглядит слишком жестоко. Лучше избавиться от константности посредством преобразования типов, если вы абсолютно уверены, что библиотечная функция не модифицирует свои аргументы, и готовы смириться с предупреждениями компилятора.

```
String& String::operator+(const String& s1) const
{
    char* s2 = new char[strlen((char*)s1.s) + strlen(s) + 1];
    strcat(s2, (char*)s1.s, s);
    String newStr(s2);
    delete s2;
    return newStr;
}
```

Видите, что происходит, если кто-то забывает о константности?

Операторы преобразования

Оператор преобразования — особый случай. Если конструктор представляет собой отображение аргументов на домен вашего класса, то оператор преобразования делает прямо противоположное: по экземпляру вашего класса он создает другой тип данных.

```
class String {
private:
    char* s;
public:
    operator long();   // Использует atol для преобразования к типу long
};
String::operator long()
```

```
{
    // Вероятно, здесь следует проверить, что строка
    // представляет собой число, принадлежащее к диапазону длинных целых
    return atoll(s);
}
String s(“1234”);
long x = s;   // Вызывается функция operator long()
```

Операторы преобразования должны быть функциями класса. Как видно из показанного фрагмента, операторы преобразования хороши тем, что компилятор обычно сам может определить, когда они должны вызываться. Если ему понадобится длинное целое, он ищет оператор `long()`. Если ему понадобится объект `Foo`, он ищет в классе `Foo` либо конструктор с аргументом `String`, либо `operator Foo()`. Возникает интересный вопрос: если оператор преобразования делает фактически то же, что и конструктор, почему бы не обойтись чем-нибудь одним? Преимущество конструкторов состоит в том, что они обеспечивают инкапсуляцию результирующего класса. Чтобы сконструировать объект другого класса, оператор преобразования должен очень много знать о нем. Вот почему для перехода от одного типа к другому обычно используются конструкторы. А если осуществляется переход к базовому типу вроде `int`? Вряд ли вы будете требовать, чтобы компилятор создавал для `int` новые конструкторы, которые знают о существовании ваших пользовательских типов. А если и будете, то не рискнете признаться в этом вслух. Только оператор преобразования может автоматически перейти к базовому типу. Даже если результирующий тип не является базовым, он может быть частью готовой библиотеки, которую нежелательно модифицировать. И снова оператор преобразования справляется с задачей.

Операторы преобразования можно объявлять для любых типов данных. Они вызываются без аргументов, а тип возвращаемого значения определяется по имени оператора. Операторы преобразования, как и все остальные операторы, бывают константными или неконстантными. Часто определяется как константная, так и неконстантная версии одного оператора. Как правило, константная версия работает более эффективно, поскольку неконстантная версия обычно выполняет копирование данных.

```
class String {
private:
    char* s;
public:
    operator const char*() const { return s; }
    operator char*():
};
String::operator char*()
{
    char* newStr = new char[strlen(s) + 1];
    strcpy(newStr, s);
    return newStr;
}
```

Клиентский код, использующий неконстантную версию, должен взять на себя ответственность за удаление дубликата.

Порядок поиска и неоднозначность

Если во время обработки программы компилятор C++ находит оператор, он выполняет описанные ниже действия в указанном порядке, чтобы решить, как его компилировать. Описание относится к бинарному оператору, но та же самая логика используется и для унарных операторов:

1. Если оба аргумента относятся к базовым типам, используется встроенный оператор.

2. Если слева указан пользовательский тип, компилятор ищет перепруженный оператор в форме функции данного класса, подходящей для всей сигнатуры подвыражения оператора. Если такой оператор будет найден, он используется при компиляции.
3. Если все остальные варианты испробованы, компилятор ищет перегрузку в форме внешней функции.

Неоднозначность может возникнуть лишь в том случае, если она присутствует в левостороннем классе или в глобальном пространстве, и никогда — из-за совпадения перегруженных операторов в форме функции класса и внешней функции. При наличии неоднозначности сообщение об ошибке выдается лишь после вашей попытки реально использовать оператор. Так компилятор внушает ложное чувство безопасности и ждет, пока вы утратите бдительность, чтобы огреть вас дубиной по голове.

Виртуальные операторы

Операторы классов можно объявлять виртуальными, как и все остальные функции классов. Компилятор динамически обрабатывает перегруженный левосторонний оператор, как и любую другую функцию класса. Такая возможность особенно полезна в ситуациях, когда вы пытаетесь создать семейство классов, но открываете внешнему миру лишь их общий базовый класс. С точки зрения синтаксиса все выглядит просто, но логика программы может стать довольно запутанной. Виртуальные операторы являются одной из важнейших тем части 3, поэтому сейчас мы не будем вдаваться в подробности.

Оператор ->

Оператор -> занимает особое место среди операторов. Для начала рассмотрим его базовый синтаксис.

```
class Pointer {
private:
    Foo* f;
public:
    Pointer(Foo* foo) : f(foo) {}
    Foo* operator->() const { return f; }
};
Pointer p(new Foo);
p->MemberOfFoo();
```

В приведенном фрагменте p используется для косвенного вызова функции класса Foo. Компилятор интерпретирует любой указатель на структуру или класс (*-переменная) как базовый тип >, а для всех базовых типов указателей существует встроенный оператор ->. Встретив ->, компилятор смотрит на левостороннее выражение; если оно представляет собой указатель па структуру или указатель на класс, для обращения к членам используется встроенный оператор ->. Если левостороннее выражение представляет собой пользовательский тип, этот тип должен перегрузить оператор ->. Перегруженный вариант должен возвращать либо указатель на структуру/класс, либо какой-нибудь другой пользовательский тип, который также перегружает оператор ->. Если возвращаемое значение относится к пользовательскому типу, компилятор заменяет левостороннее выражение возвращаемым значением оператора -> (в нашем примере Foo*) и продолжает свои попытки до тех пор, пока не доберется до встроенного указателя. Таким образом, следующее двухшаговое косвенное обращение также будет работать.

```
class Pointer2 {
private:
    Pointer p;
public:
    Pointer(Foo* foo) : p(foo) {}
    Pointer operator->() const { return p; }
};
Pointer2 p(new Foo);
p->MemberOfFoo();
```

Здесь оператор -> вызывается трижды:

1. Pointer2::operator-> возвращает Pointer.
2. Затем Pointer::operator-> возвращает Foo.
3. Компилятор интерпретирует Foo* как базовый тип и вызывает его функцию.

В отличие от всех остальных операторов, вы не можете контролировать возвращаемое значение или просто обратиться к нему после выхода из операторной функции. Оно используется исключительно самим компилятором. Концепция объекта, «переодетого» указателем, имеет фундаментальное значение для всей книги.

Оператор []

Оператор [] может быть перегружен, чтобы получать единственный аргумент произвольного типа и возвращать произвольный тип в качестве своего значения.

```
class String {
private:
     char* s;
public:
     String(char*);
     char operator[](int n) const;   // n-й символ
};
char String::operator[](int n)
{
     // Здесь должна выполняться проверка диапазона
     return s[n];
}
```

Поскольку оператор [] может вызываться лишь с одним аргументом, для имитации многомерных массивов часто применяют анонимные экземпляры.

```
struct Index3 {
     int X, Y, Z;
     Index3(int x, int y, int z) : X(x), Y(y), Z(z) {}
};
class Array3D {    // Трехмерный массив объектов String
private:
     // Настоящая структура данных
public:
     String& operator[](const Index3&);
};
String s = anArray[Index3(17, 29, 31)];
```

Хотя оператор [] вызывается лишь с одним аргументом, этот умеренно неуклюжий синтаксис позволяет создать произвольное количество псевдоаргументов.

Оператор ()

Наверное, вы и не подозревали, что этот оператор тоже можно перегрузить. Если у вас часто возникает непреодолимое желание превратить объект в функцию, возможно, ваша психика нестабильна и вам стоит серьезно подумать над сменой рода занятий. Тем не менее, C++ с пониманием отнесется к вашим проблемам. Левостороннее выражение оператора () представляет собой объект, который должен интерпретироваться как вызываемая функция. Аргументы оператора представляют собой формальные аргументы, передаваемые объекту-функции при вызове.

```
class Function {
public:
    int operator()(char*);
};
int Function::operator()(char* s)
{
    cout << "\"" << s << "\"";
}
Function fn;
int x = fn("Hello");    // Выводит в cout строку "Hello"
```

Оператор () может возвращать любой тип и принимать любые аргументы. При этом он подчиняется тем же правилам, что и любая другая функция. Оператор () перегружается только в форме функции класса.

Оператор new

Изобретая новый язык, приходится изобретать собственные правила. Одно из новых правил C++ гласит, что имя оператора не обязано состоять из одних служебных символов. Исключение составляют операторы new и delete. Оператор new вызывается всякий раз, когда компилятор считает, что настало время выделить из кучи память для нового объекта. Используемый здесь термин *объект* относится к любым динамическим данным, включая int, char* и т.д., а не только к экземплярам ваших классов. Оператор new по умолчанию имеет следующий интерфейс:

```
void* operator new(size_t bytes);
```

Единственный аргумент — количество выделяемых байт, возвращаемое значение — адрес выделенной области. Оператор new никогда не должен возвращать NULL; вместо этого при нехватке памяти инициируется исключение (см. главу 4). Реализация оператора new по умолчанию изменяется от простого вызова функции malloc или calloc до нестандартных средств управления памятью, прилагаемых к компилятору.

Оператор new может перегружаться как в форме внешней функции, так и в форме функции класса. При перегрузке в форме функции класса он наследуется, поэтому выделение памяти в производных классах будет осуществляться так же, как и в базовом классе, перегрузившем оператор new.

```
class Foo {
public:
    void* operator new(size_t bytes);
};
void* Foo::operator new(size_t bytes)
{
    if (bytes < MAXBYTES)
        // Нестандартное выделение памяти для блоков малого размера
    else return ::operator new(bytes);
}
```

Разумеется, всеобщая перегрузка глобального оператора new затруднит создание переносимого кода, поэтому обычно предпочтительным вариантом является перегрузка в форме функции класса. Она имеет дополнительные преимущества: из перегруженной функции можно вызвать оператор new по умолчанию, как это делается в предыдущем фрагменте. Оператор new также может перегружаться для других сигнатур. Хотя дополнительные сигнатуры не будут вызываться компилятором автоматически, они оказывают большую помощь при реализации нестандартного управления памятью, в которой можно вызвать конкретную версию оператора new.

```
class Pool {  // Нестандартный пул памяти
public:
    virtual void* Allocate(size_t bytes);
```

```
};
void* operator new(size_t bytes, Pool* p)
{
    return p->Allocate(bytes);
}
extern Pool* DefaultPool;
Foo* f = new(DefaultPool) Foo;
```

Дополнительные аргументы указываются между ключевым словом new и перед именем класса `Foo`; они передаются перегруженному оператору после размера блока, предоставленного компилятором.

Оператор `delete`

Оператор `delete` обычно перегружается вместе с оператором new для выполнения нестандартных операций управления памятью. Существуют два интерфейса, автоматически вызываемые компилятором при удалении объекта:

1. `void operator delete(void* address);`
2. `void operator delete(void* address, size_t bytes);`

Первая версия просто передает вам адрес освобождаемого блока; чтобы определить его размер, вам придется самостоятельно заглянуть в хрустальный шар. Вторая версия вроде бы передает размер освобождаемого блока, но... он может быть меньше истинного размера! Проблемы возникают в следующих ситуациях:

```
class Foo {
private:
    int x;
public:
    ~Foo();   // Невиртуальный деструктор
};
class Bar : public Foo {
private:
    int y;
public:
    ~Bar();
};
Bar* b = new Bar;
delete b;     // Правильный размер
Foo* f = new Bar;
delete f;     // Размер Foo, а не Bar
```

Компилятор определяет размер на основании вызываемого деструктора. Если невиртуальный деструктор вызывается для указателя на базовый класс, используется размер базового класса. Правильный размер будет передаваться при соблюдении любого из трех условий:

1. Деструктор является виртуальным.
2. Указатель ссылается на настоящий тип объекта.
3. Тип, на который ссылается указатель, имеет тот же размер, что и настоящий тип.

Последнее условие соблюдается лишь в том случае, если в производном классе не добавляется ни одной новой переменной, а базовый и производный классы одновременно либо содержат, либо не содержат виртуальных функций (и как следствие, указателей v-таблицы). Непонятно? Тогда запомните простое правило — объявляйте ваши деструкторы виртуальными.

Оператор `delete`, как и оператор `new`, можно перегружать как в форме функции класса, так и в форме внешней функции. Если оператор перегружается функцией класса, он наследуется. В отличие от

оператора `new`, для оператора `delete` нельзя создавать дополнительные сигнатуры. Два варианта, приведенные выше, — это все, что у вас есть.

Шаблоны и безопасность типов

3

Хотя стандарты шаблонов опубликованы уже давно, они все еще распространены недостаточно широко. Конечно, трудно использовать нечто, не поддерживаемое вашим компилятором, — наверное, это первая причина, по которой большинство программистов C++ не умеет работать с шаблонами. К счастью, сейчас все основные компиляторы уже вошли в двадцатый век, так что эта проблема уже отпала. Остается лишь понять, что такое шаблон, как обойти все синтаксические ловушки, но прежде всего — для чего он все-таки нужен. Эта глава не ограничивается обзором синтаксиса. В ней также рассматриваются основы безопасности типов в C++, причем особое внимание уделяется шаблонам.

Что такое шаблоны и зачем они нужны?

Интерфейс простого класса-коллекции (на примере связанного списка) выглядит так:

```
class ListNode {
private:
     ListNode* next;
     void* data;
public:
     ListNode(void* d, ListNode* n = NULL) : next(n), data(d) {}
     ~ListNode() { delete next; }
     void* Data() { return data; }
     ListNode* Next() { return next; }
};
```

Заметили что-нибудь особенное?

Проблемы

Прежде всего, в глаза бросаются все эти `void*`. И вы, и я прекрасно знаем, что на самом деле за ними кроется нечто совершенно иное. Где-то в клиентском коде придется сделать что-нибудь подобное:

```
for (ListNode* n = listHead; n != NULL; n = n->Next())
     f((Foo*)n->Data());
```

Иначе говоря, вам придется постоянно приводить `void*` к конкретному типу Но как убедиться в том, что полученный указатель действительно имеет тип `Foo*`? Здесь придется рассчитывать только на себя, потому что компилятор со словами «Надеюсь, ты знаешь, что делаешь» умывает руки. Допустим, вы уверены, что ваше использование класса надежно по отношению к типам. Но можно ли гарантировать, что другой программист не выкинет какую-нибудь глупость и не занесет в коллекцию объект другого типа? Если вы свято верите в это, я рекомендую держаться подальше от рискованных инвестиций и вложить деньги в правительственные бумаги, вряд ли вам повезет в этой жизни.

Вторая проблема заключается в том, что элементы списка не знают, на какой тип они указывают. Предположим, вам хочется, чтобы деструктор списка удалял не только сами узлы, но и данные, на которые они ссылаются. Нельзя передать оператору delete указатель void* и надеяться, что он сам выберет нужный деструктор.

Обходные решения

Одно из возможных решений — потребовать, чтобы все объекты вашей коллекции происходили от общего предка. В этом случае void* можно будет заменить указателем на базовый класс, создавая хотя бы видимость порядка. Если деструктор базового класса является виртуальным, по крайней мере мы сможем переписать деструктор ListNode так, чтобы при самоубийстве он уничтожал и содержимое списка. Но если этот базовый класс имеет производные классы, вы наверняка вернетесь к необходимости выполнения ненадежных операций приведения к этим производным типам.

Другое обходное решение — создать список, рассчитанный на конкретный тип. Скажем, для ведения списка объектов класса Foo создается класс-коллекция ListOfFoos. В этом случае вам не придется выполнять приведения типов, если Foo не имеет производных классов. Но стоит ли плодить классы-двойники, которые отличаются только типами, с которыми они работают? Конечно, вырезание и вставка в текстовых редакторах — замечательная вещь, а сценарии обработки текстов помогают быстро размножить код. Но если вам потребуется изменить представление всех этих списков, дело неизбежно кончится масштабной головной болью.

В прошлом подобные проблемы часто решались с помощью макросов #define:

```
#define ListNode(Type) \
class ListNode##Type { \
private: \
    ListNode##Type* next; \
    Type* data; \
public: \
    ListNode##Type(Type* d, ListNode* n = NULL) : next(n), data(d) {} \
    ~ListNode() { delete next; } \
    void* Data() { return data; } \
    ListNode* Next() { return next; } \
};
```

Если вы нечаянно забудете поставить знак \, компилятор разразится громкими негодующими воплями, но при должной осторожности эта методика работает. Символы ## означают конкатенацию. Конструкция становится еще уродливее, но с этим приходится мириться — вы должны обеспечить уникальность имен типов коллекций. Такая методика обладает многочисленными недостатками. Если функции класса не являются подставляемыми (inline), вам придется создавать для них дополнительные макросы и следить, чтобы они были реализованы в одном модуле компиляции. У некоторых компиляторов возникают проблемы со слишком длинными макросами. Директивы #define не могут быть вложенными, поэтому рекурсивные, безопасные по отношению к типам структуры данных отпадают. Хуже всего, что при обнаружении ошибки в макросе отладчик складывает руки и сообщает, что *где-то* в макросе допущена ошибка, но не указывает конкретного номера строки.

Шаблоны — усовершенствованные макросы

На сцену выходит механизм шаблонов — усовершенствованный макропроцессор для директив #define. Шаблоны представляют собой ничто иное, как макросы без всех перечисленных ограничений. Они могут быть вложенными. Вам не придется беспокоиться о дублировании их функций. Большинство отладчиков С++ при возникновении ошибки правильно указывает строку шаблона. Размер шаблона не вызовет никаких проблем. Наконец, вам не придется уродовать свою прекрасную программу закорючками вроде \ и ##.

Синтаксис шаблонов

Если вы собираетесь использовать шаблоны, привыкайте к тому, что в вашей речи будет часто звучать термин *параметризованный* (*parameterized*). Шаблоны используются для создания параметризованных типов (обычно классов) и параметризованных функций.

Параметризованные типы

Параметризованный тип внешне представляет собой обычное объявление класса, которому предшествует магическое заклинание `template <class Type>`, где `Type` — выбранное вами символическое имя (остальные элементы задаются жестко). Всюду, где символическое имя `Type` (или другое имя) встречается в объявлении класса оно интерпретируется как макрос, вместо которого при использовании класса подставляется конкретный тип. Класс `ListNode`, переписанный как параметризованный тип, выглядит следующим образом:

```
template <class Type>
class ListNode {
private:
    ListNode<Type>* next;
    Type* data;
public:
    ListNode(Type* d, ListNode<Type>* n = NULL) : next(n), data(d) {}
    ~ListNode() { delete next; }
    Type* Data() { return data; }
    ListNode<Type>* Next() { return next; }
};
ListNode<Foo> list = new ListNode<Foo> (new Foo);
Foo* f = list->Data();      // Возвращает правильный тип
```

В теле объявления класса формальный параметр шаблона резервирует место, на которое при использовании класса подставляется фактический параметр. При этом компилятор буквально генерирует правильный, безопасный по отношению к типам код.

Параметризованные функции

Параметризованные функции объявляются точно так же — перед их объявлениями указывается формула `template...`. Синтаксис шаблона должен повторяться как при объявлении, так и при определении функции. Помните, шаблоны на самом деле являются макросами, поэтому они должны находиться в файлах .h. Если определение будет находиться в файле .cpp, программа работать не будет (если только это не *единственный* файл .cpp, в котором вызывается данная функция).

```
// Объявление функции
template <class Type>
Type* fn(Type* t);
// Определение ее реализации
template <class Type>
Type* fn(Type* t) {
    // Тело функции, в котором имя Type
    // используется в качестве параметра макроса
}
Foo* f = fn<Foo>(new Foo);
```

Определение генерируется компилятором при необходимости, то есть при вызове функции. На этот раз параметризовано имя функции, а не имя класса.

Параметризованные функции классов

Параметризованные функции классов определяются так же (разве что вам понадобится больше угловых скобок < и >). Давайте модифицируем класс ListNode так, чтобы его функции не определялись при объявлении класса.

```
template <class Type>
class ListNode {
private:
    ListNode<Type*> next;
    Type* data;
public:
    ListNode(Type* d, ListNode<Type>* n = NULL);
    ~ListNode();
    Type* Data();
    ListNode<Type>* Next();
};
template <class Type>
ListNode<Type>::ListNode(Type* d, ListNode<Type>* n = NULL)
    : next(n), data(d)
{
}
template <class Type>
ListNode<Type>::~ListNode()
{
    delete next;
}
template <class Type>
Type* ListNode<Type>::Data()
{
    return data;
}
template <class Type>
ListNode<Type>* ListNode<Type>::Next()
{
    return next;
}
```

Помните: все это должно находиться в файле .h. Исключение составляют ситуации, когда функции класса вызываются только из файла .cpp, в котором они определяются. В этом случае определения функций класса должны предшествовать их первому использованию.

Передача параметра

Многочисленные символы < и > вызывают изрядную путаницу, поскольку C++ не всегда последователен. Вообще говоря, <Type> следует указывать везде, кроме трех мест в объявлениях классов или определениях их функций:

1. За ключевым словом class в самом начале.
2. При указании имени конструктора.
3. При указании имени деструктора.

Аргументы конструкторов и деструкторов должны быть параметризованными, как и все использования имени класса за исключением трех указанных случаев. При любом использовании параметризованного

типа или функции необходимо указывать параметр. Было бы намного проще, если бы C++ просто требовал присутствия параметра во всех случаях, но это же C++... Вдобавок можно сэкономить несколько символов в исходном тексте программы. В трех указанных ситуациях компилятор может сделать разумные предположения по поводу отсутствующих символов.

Шаблоны с несколькими параметрами

Тип может иметь более одного параметра, хотя такие ситуации встречаются довольно редко. Увидев такой класс, я обычно затеваю с автором долгую беседу о принципах построения программ. Тем не менее, иногда использование многоаргументных шаблонов бывает оправданным. Синтаксис выглядит аналогично, разве что вместо `<class Type>` используется список типа `<class Type1, class Type2>`. Наконец, параметр не обязан быть классом. Он может быть структурой или еще чем-нибудь, хотя именно классы прочно удерживают ведущие позиции на рынке параметров.

Долой вложенные параметризованные типы!

Увы, такая возможность существует, но пожалуйста, пользуйтесь ею с максимальной осторожностью. Вложенные шаблоны не только плохо читаются, но и генерируют огромное количество кода при расширении. Помните, что при использовании шаблона самого верхнего уровня будут расширены *все* шаблоны.

```
template <class Type>
class B {...};
template <class Type>
class A {
    B<A<Type>>* member;    // Жуть!
};
```

Посмотрите на этот омерзительный синтаксический мусор. При вложении параметризованных типов всегда происходит нечто подобное. Позднее мы поговорим о том, как переделать этот фрагмент; а пока избегайте вложенных параметризованных типов, как чумы.

Наследование

Параметризованные классы могут быть производными от других классов, параметризованных или нет. Кроме того, параметризованный класс может выступать в качестве базового; в этом случае производный класс также будет параметризованным с тем же форматом аргументов, что и в базовом классе. В производном классе могут добавляться новые параметры, но результат напоминает соревнования по поводу того, кто уместит в одной строке больше APL-кода. Другими словами, не делайте этого. Из всех перечисленных мутаций чаще всего встречается наследование параметризованного типа от непараметризованного. Мы рассмотрим это и другие сочетания простых и параметризованных типов в следующем разделе.

Комбинации простых и параметризованных типов

Предположим, у вас имеется параметризованный класс, реализация всех функций которого занимает 1000 строк. При каждом его использовании для нового типа компилятор радостно выплевывает очередные 1000 строк расширенного кода. Даже при нынешних ценах на память это слишком высокая цена за безопасность типа.

Допустим, вы продаете библиотеку классов и не хотите поставлять исходный текст, а только интерфейсы. Если библиотека содержит параметризованные функции, они должны находиться в открытом для всего мира файле .h. Обидно.

Допустим, кто-то передает вам замечательный, но небезопасный по отношению к типам класс или библиотеку классов. Может быть, он был написан на компиляторе, который не поддерживает шаблоны, или автор просто не верит в шаблоны. Вам хочется подправить код и сделать его безопасным с помощью шаблонов. Но хотите ли вы переделывать все подряд, включать весь код реализации в файлы .h и добавлять в объявления класса параметры и символы <>?

Во всех описанных ситуациях стоит использовать параметризованный тип в сочетании с простым, непараметризованным типом. Когда это будет сделано, в 99 случаях из 100 параметризованный тип «заворачивает» простой тип в симпатичную, мягкую и безопасную по отношению к типам оболочку. При этом простой класс не изменяется — просто параметризованный класс помещается между небезопасным классом и пользователем. Для таких ситуаций существует ряд стандартных приемов и многочисленные идиомы, основанные на этой идее.

Небезопасные типы в открытых базовых классах

Не делайте этого. Серьезно. Если вы попытаетесь ввести безопасность типов в производном классе с открытым наследованием, клиент получит полный доступ ко всем небезопасным средствам базового класса. Существуют невероятно изобретательные решения этой проблемы (особенно с применением методов, которые будут рассмотрены в следующих главах), но в любом случае у вас выйдет что-то вроде подвесного моста из бутылочных пробок: гениальная работа при плохом материале.

Небезопасные типы в закрытых базовых классах

Вот это уже больше похоже на истину. Самый простой способ обеспечить безопасность типов — сделать ненадежный класс закрытым базовым классом безопасного шаблона.

```
class UnsafeNode {      // ListNode из предыдущего примера
private:
    UnsafeNode* next;
    void* data;
public:
    UnsafeNode(void* d, UnsafeNode* n);
    virtual ~UnsafeNode();
    UnsafeNode* Next();
    void* Data();
};
template <class Type>
class SafeNode : private UnsafeNode {
public:
    SafeNode(Type* d, SafeNode* n) : UnsafeNode(d, n) {}
    virtual ~SafeNode() { delete (Type*)Data(); }
    SafeNode* Next() { return (SafeNode*)UnsafeNode::Next(); }
    Type* Data() { return (Type*)UnsafeNode::Data(); }
```

Мы добились чего хотели — базовый класс недоступен для клиентов производного шаблона. Приведенный пример демонстрирует еще один прием, связанный с пространствами имен C++. Нет необходимости создавать в производном классе новые имена для функций `Next()` и `Data()` только потому, что они отличаются типом возвращаемого значения; поскольку ни одна из этих функций не является виртуальной, производная версия скрывает базовую от клиентов. Некоторые компиляторы при попытке скрытия членов базовых классов укоризненно грозят пальцем, но для закрытого наследования это предупреждение абсолютно безобидно. После всех огорчений, доставляемых вам компилятором, бывает приятно отплатить ему той же монетой.

Один из недостатков закрытого наследования — необходимость дублирования всех функций базового класса, которые могут безопасно использоваться клиентами. Впрочем, это происходит не так уж часто и в любом случае не является слишком высокой ценой за дополнительную безопасность. Примером может послужить наша реализация функций `Next()` и `Data()`, за исключением того, что интерфейс идентичен интерфейсу закрытого базового класса.

Небезопасные типы в переменных класса

Следующим способом объединения двух классов является делегирование: вы создаете экземпляр небезопасного класса в качестве переменной параметризованного класса и поручаете ему всю

необходимую работу. Для обеспечения безопасности типов эта переменная класса делается невидимой для пользователя. Иногда эта задача не решается так просто; семантика оболочки нередко отличается от семантики переменной.

Рассмотрим знакомый пример со связанным списком UnsafeNode. Вместо закрытого наследования SafeNode от этого класса можно сделать UnsafeNode переменной класса SafeNode. Однако по имеющемуся SafeNode вам не удастся получить следующий SafeNode в списке! Попробуйте сами. Каждый UnsafeNode ссылается на другой UnsafeNode, а не на SafeNode. Возможное решение — использовать разную семантику для оболочки и содержимого.

```
// В SafeList.h
class UnsafeNode;       // Предварительное объявление
template <class Type>
class SafeList {        // Безопасная оболочка для UnsafeNode
private:
     UnsafeNode* head;
public:
     SafeList() : head(NULL) {}
     ~SafeList();
     UnsafeNode* Cursor();           // Для итераций
     Type* Next(UnsafeNode*&);       // Переход к следующему элементу
     void DeleteAt(UnsafeNode*&);    // Удаление элемента в позиции курсора
     void InsertFirst(Type*);        // Вставка в начало списка
     void InsertBefore(UnsafeNode*&);      // Вставка перед позицией курсора
     void InsertAfter(UnsafeNode*&); // Вставка после позиции курсора
};

// В SafeList.cpp
class UnsafeNode {      // ListNode из предыдущего примера
private:
     UnsafeNode* next;
     void* data;
public:
     UnsafeNode(void* d, UnsafeNode* n);
     virtual ~UnsafeNode();
     UnsafeNode* Next();
     void* Data();
};
```

Объект SafeList представляет весь список, а не отдельный элемент. Большинство операций (такие как InsertFirst) относятся к списку в целом, а не к отдельному элементу. Для операций, выполняемых с одним элементом, нам потребуется новая парадигма — курсор (маркер позиции). Чтобы перемещаться по списку, вы запрашиваете у него позицию курсора. Чтобы перейти к следующему элементу, вы передаете ссылку на указатель на курсор, которая обновляется объектом SafeList. Чтобы выполнить операцию с определенной позицией списка, вы передаете курсор, определяющий эту позицию. Обратите внимание: клиенту не нужно знать об UnsafeNode ничего, кроме самого факта его существования — предварительного объявления оказывается вполне достаточно. Концепция курсора будет подробно рассмотрена в следующих главах. А пока вы должны понять, что безопасная оболочка не сводится к нескольким параметрам и символам <>, разбросанным по программе, — мы переопределяем семантику структуры данных. Такая ситуация типична для ненадежных, рекурсивных структур данных и часто встречается в других контекстах.

Исключения

4

Если ваши программы всегда работают без малейших проблем, можете смело пропустить эту главу. А если нет — давайте поговорим о том, как обрабатывать исключения.

Базовый принцип, на котором основана обработка исключений, — восстановление состояния и выбор альтернативных действий в случае ошибки. Предположим, в вашей программе имеется некий блок и вы не уверены, что он доработает до конца. При выполнении блока может возникнуть нехватка памяти, или начнутся проблемы с коммуникациями, или нехороший клиентский объект передаст неверный параметр. Разве не хотелось бы написать программу в таком виде:

```
if (блок будет работать) {
    блок;
}
else {
    сделать что-то другое;
}
```

Иначе говоря, вы заглядываете в хрустальный шар. Если в нем виден фрагмент программы, который горит синим пламенем, вы изменяете будущее и обходите этот фрагмент. Не стоит с затаенным дыханием ожидать появления таких языков программирования в ближайшем будущем, но на втором месте стоит обработка исключений. С помощью исключений вы «допрашиваете» подозрительный блок. Если в нем обнаружится ошибка, компилятор поможет восстановить состояние перед выполнением блока и продолжить работу.

Обработка исключений в стандарте ANSI

Хорошая новость: для обработки исключений существует стандарт ANSI — или, как это всегда бывает в C++, предложенный стандарт. Интересно, почему у нас так много предложенных стандартов и ни одного реального? Скорее всего, дело в том, что наша экономика не может вместить всех безработных членов комитетов стандартизации. Лучше оставить им занимательное пожизненное хобби, пока мы будем выполнять свою работу. Впрочем, я отвлекся.

Плохая новость: стандартная обработка исключений все еще не поддерживается многими компиляторами C++. Хорошая новость: все больше и больше компиляторов выходит на передовые позиции. Плохая новость: осталось немало старого кода, предназначенного для старых компиляторов. Увы.

Давайте сначала поговорим о том, как все *должно* происходить, а уже потом займемся теми вариациями, которые встречаются в реальном мире.

Синтаксис инициирования исключений

Следующая функция шлепнет вас по рукам, если вызвать ее с неверным параметром. Вместо линейки она воспользуется секцией `throw`. В этой функции могут произойти две ошибки, представленные константами перечисления `Gotcha`.

```
enum Gotcha { kTooLow, kTooHigh };
void fn(int x) throw(Gotcha) {
```

```
        if (x < 0)
            throw kTooLow;     // Функция завершается здесь
        if (x > 1000)
            throw kTooHigh;    // Или здесь
        // Сделать что-то осмысленное
    }
```

В первой строке определяется тип исключения. Исключения могут иметь любой тип: целое, перечисление, структура и даже класс. Во второй строке объявляется интерфейс функции с новым придатком — *спецификацией исключений,* который определяет, какие исключения могут быть получены от функции вызывающей стороной. В данном примере инициируется исключение единственного типа Gotcha. В четвертой и шестой строке показано, как инициируются исключения, которые должны быть экземплярами одного из типов, указанного в спецификации исключений данной функции. Спецификации исключений должны подчиняться следующим правилам.

Объявления и определения

Спецификация исключений в объявлении функции должна точно совпадать со спецификацией в ее определении.

```
    void Fn() throw(int);   // Объявление
    // Где-то в файле .cpp
    void Fn() throw(int) {
        // Реализация
    }
```

Если определение будет отличаться от объявления, компилятор скрестит руки на груди и откажется компилировать определение.

Функции без спецификации исключений

Если функция не имеет спецификации исключений, она может инициировать любые исключения. Например, следующая функция может инициировать что угодно и когда угодно.

```
    void fn();    // Может инициировать исключения любого типа
```

Функции, не инициирующие исключений

Если список типов в спецификации пуст, функция не может инициировать никакие исключения. Разумеется, при хорошем стиле программирования эту форму следует использовать всюду, где вы хотите заверить вызывающую сторону в отсутствии инициируемых исключений.

```
    void fn() throw();      // Не инициирует исключений
```

Функции, инициирующие исключения нескольких типов

В скобках можно указать произвольное количество типов исключений, разделив их запятыми.

```
    void fn() throw(int, Exception_Struct, char*);
```

Передача исключений

Если за сигнатурой функции не указан ни один тип исключения, функция не генерирует новые исключения, но может передавать дальше исключения, полученные от вызываемых ею функций.

```
    void fn() throw;
```

Исключения и сигнатуры функций

Спецификация исключений не считается частью сигнатуры функции. Другими словами, нельзя иметь две функции с совпадающим интерфейсом за исключением (нечаянный каламбур!) спецификации исключений. Две следующие функции не могут сосуществовать в программе:

```
void f1(int) throw();
void f1(int) throw(Exception);      // Повторяющаяся сигнатура!
```

Спецификация исключений для виртуальных функций

В главе 2 мы говорили (точнее, я говорил, а вы слушали) об отличиях между перегрузкой (*overloading*) и переопределением (*overriding*). Если виртуальная функция в производном классе объявляется с новой сигнатурой, отсутствующей в базовом классе, эта функция скрывает все одноименные функции базового класса (если вы в чем-то не уверены, вернитесь к соответствующему разделу; это важно понимать). Аналогичный принцип действует и для спецификаций исключений.

```
class Foo {
public:
    virtual Fn() throw(int);
};
class Bar : public Foo {
public:
    virtual Fn() throw(char*);      // Осторожно!
};
```

Компилятор косо посмотрит на вас, но откомпилирует. В результате тот, кто имеет дело с `Foo*`, будет ожидать исключения типа `int`, не зная, что на самом деле он имеет дело с объектом `Bar`, инициирующим нечто совершенно иное.

Мораль ясна: не изменяйте спецификацию исключений виртуальной функции в производных классах. Только так вам удастся сохранить контракт между клиентами и базовым классом, согласно которому должны инициироваться только исключения определенного типа.

Непредусмотренные исключения

Если инициированное исключение отсутствует в спецификации исключений внешней функции, программа переформатирует ваш жесткий диск. Шутка. На самом деле она вызывает функцию с именем `unexpected()`. По умолчанию затем вызывается функция `terminate()`, о которой будет рассказано ниже, но вы можете сделать так, чтобы вызывалась ваша собственная функция. Соответствующие интерфейсы из заголовочного файла except.h выглядят так:

```
typedef void (*unexpected_function)();
unexpected_function set_unexpected(unexpected_function excpected_func);
```

В строке `typedef...` объявляется интерфейс к вашей функции. Функция `set_unexpected()` получает функцию этого типа и организует ее вызов вместо функции по умолчанию. Функция `set_unexpected()` возвращает текущий обработчик непредусмотренных исключений. Это позволяет временно установить свой обработчик таких исключений, а потом восстановить прежний. В следующем фрагменте показано, как используется этот прием.

```
unexpected_function my_handler(void) {
    // Обработать неожиданное исключение
}
{   // Готовимся сделать нечто страшное и устанавливаем свой обработчик
    unexpected_function old_handler = set_unexpected(my_handler);
    // Делаем страшное и возвращаем старый обработчик
    set_unexpected(old_handler);
}
```

Функция-обработчик не может нормально возвращать управление вызывающей программе, если в ней встречается оператор `return` или при выходе из области действия функции результаты будут неопределенными. Тем не менее, из функции можно запустить исключение и продолжить поиск перехватчика, подходящего для нового исключения.

Синтаксис перехвата исключений

Чтобы перехватить исключение, поставьте перед блоком ключевое слово try и поместите после него одну или несколько секций catch, которые называются обработчиками (handlers).

```
try {
    // Фрагмент, который может инициировать исключения
}
catch (Exception_Type t) {
    // Восстановление после исключения типа Exception_Type
}
catch (...) {
    // Восстановление после исключений всех остальных типов
}
```

Каждый обработчик, за исключением (опять нечаянный каламбур) обработчика с многоточием, соответствует одному конкретному типу ошибок. Если из фрагмента, называемого try-блоком, инициируется исключение, компилятор просматривает список обработчиков в порядке их перечисления и ищет обработчик, подходящий по типу запущенного исключения. Многоточие соответствует исключениям любого типа; если такой обработчик присутствует, он должен находиться последним в списке.

Внутри обработчика вы можете предпринимать любые действия для выхода из ситуации. Сведения об исключении можно получить из аргумента catch — кроме обработчика с многоточием, который понятия не имеет, что он должен перехватывать.

Выполнение программы после исключения

Если выполнение try-блока обходится без исключений, программа благополучно игнорирует все обработчики и продолжает работу с первого выражения за последним обработчиком. Если же исключение все же произошло, оно будет единственным из всего списка, и после его обработки выполнение программы продолжится за последним обработчиком списка. Существуют два исключения (последний нечаянный каламбур): обработчик может содержать вызов крамольного goto или запустить исключение. Если обработчик инициирует исключение, он может продолжить распространение того же исключения или создать новое.

```
catch(int exception) {
    // Сделать что-то, а затем
    throw(“Help!”);    // Инициируется исключение типа char*
}
```

Инициирование исключения из обработчика немедленно завершает выполнение вмещающей функции или блока.

Если исключение не перехвачено

Если для исключения не найдется ни одного обработчика, по умолчанию вызывается глобальная функция terminate(). Как вы думаете, что она делает? По умолчанию terminate() в конечном счете вызывает библиотечную функцию abort(), и дело кончается аварийным завершением всей программы. Вы можете вмешаться и установить собственную функцию завершения с помощью библиотечной функции set_terminate(). Соответствующий фрагмент файла except.h выглядит так:

```
typedef void (*terminate_function)();
termination_function set_terminate(terminate_function t_func);
```

В строке typedef... объявляется интерфейс к вашей функции завершения. Функция set_terminate() устанавливает функцию завершения, которую вместо функции abort() вызывает функция terminate(). Функция set_terminate() возвращает текущую функцию завершения, которую позднее можно восстановить повторным вызовом set_terminate().

Ваша функция завершения *обязана* завершить программу и не может инициировать другие исключения. Она может выполнить необходимые подготовительные действия, но никогда не возвращает управление вызывающей программе.

Вложенная обработка исключений

Да, вложение блоков `try/catch` разрешается, хотя пользоваться этой возможностью следует как можно реже, если только вы хотите сохранить дружеские отношения с персоналом сопровождения вашей программы.

```
{
    try {
        try {
            try {
                // Ненадежный фрагмент
            }
            catch(...) {
            }
        }
        catch(...) {
        }
    }
    catch(...) {
    }
}
```

Создавать подобную мешанину приходится довольно редко, но иногда возникает необходимость в разделении стековых объектов по разным областям действия.

Внешние исключения не перехватываются!

Вы можете перехватить любое исключение, инициированное посредством `throw`. Тем не менее, существуют и другие исключения, которые не удается перехватить переносимыми способами. Например, если пользователь применяет для завершения программы комбинацию клавиш с правым Ctrl, нет гарантии, что операционная система сгенерирует исключение, которое может быть перехвачено вашими обработчиками. Вообще говоря, обработка исключений относится только к исключениям, сгенерированным программой; все остальное непереносимо.

Конструкторы и деструкторы

Одно из принципиальных достоинств стандартной схемы обработки исключений — *раскрутка стека* (*unwinding the stack*). При запуске исключения автоматически вызываются деструкторы всех стековых объектов между `throw` и `catch`.

```
void fn() throw(int) {
    Foo aFoo;
    // Что-то не так!
    throw(bad_news);
}
```

Когда возникает исключение, до передачи стека соответствующему обработчику будет вызван деструктор `aFoo`. Тот же принцип действует и для `try`-блока вызывающей стороны.

```
{
    try {
        Bar b;
        fn();    // Вызывает исключение
    }
```

```
        catch(int exception) {
            // Перед тем, как мы попадем сюда, будет вызван деструктор b
        }
    }
```

Вообще говоря, гарантируется вызов деструкторов всех стековых объектов, сконструированных с начала выполнения try-блока. Это может пригодиться для закрытия открытых файлов, предотвращения утечки памяти или для других целей. Тем не менее, дело не обходится без некоторых нюансов.

Уничтожаемые объекты

Гарантируется вызов деструкторов всех стековых объектов, сконструированных с начала выполнения try-блока, но и только. Например, допустим, что к моменту возникновения исключения был сконструирован массив. Деструкторы вызываются лишь для тех объектов массива, которые были сконструированы до возникновения исключения.

Динамические объекты (то есть созданные посредством оператора new) — совсем другое дело. Вам придется самостоятельно следить за ними. Если в куче размещаются объекты, которые должны уничтожаться в результате исключения, обычно для них создается оболочка в виде вспомогательного стекового объекта.

```
class TempFoo {
private:
    Foo* f;
public:
    TempFoo(Foo* aFoo) : f(aFoo) {}
    ~TempFoo() { delete f; }
};
try {
    TempFoo tf(new Foo);
    // и т.д.
}
catch(...) {
    // Foo уничтожается деструктором tf
}
```

Исключения во время конструирования

Рассмотрим следующий процесс конструирования:

```
class Foo {...}
class Bar : public Foo {
private:
    A a;
    B b;
public:
    Bar();
};
Bar::Bar()
{
    X x;
    throw(bad_news);
    Y y;
}
```

Если во время конструирования объекта произойдет исключение, деструкторы будут вызваны для тех компонентов (базовых классов и переменных), конструкторы которых были выполнены к моменту возникновения исключения. Конструирование Bar к этому моменту еще не завершено. Тем не менее, конструкторы базовых классов (Foo) и переменных (a и b) уже отработали, поэтому их деструкторы будут вызваны до передачи исключения обработчику. По тем же причинам будет вызван деструктор локальной переменной x. Деструктор y не вызывается, поскольку переменная еще не сконструирована. Деструктор Bar тоже не вызывается, поскольку конструирование объекта не завершилось к моменту инициирования исключения.

Предположим, конструктор b инициирует исключение. В этом случае вызываются деструкторы Foo и a, но не деструкторы b, Bar и y.

Одни и те же принципы действуют как для стековых, так и для динамических объектов. Если исключение возникает при конструировании динамического объекта, вызываются точно те же деструкторы, что и для стековых объектов.

Порядок вызова деструкторов

Гарантируется, что порядок вызова деструкторов будет обратным порядку вызова конструкторов. Это относится как к локальным переменным, так и к переменным и базовым классам объектов.

Нестандартная обработка исключений

Многие библиотеки и некоторые компиляторы обрабатывают исключения нестандартным образом. Большинство имитирует парадигму try/catch с помощью макросов, но не организует правильной раскрутки стека за счет вызова деструкторов конструированных объектов. Некоторые реализации ориентированы на конкретные типы компьютеров и операционных систем.

К сожалению, многие компиляторы претендуют на стандартную обработку исключений, но не во всем следуют каноническим правилам. На своем горьком опыте я узнал, что обработку исключений желательно тестировать, если вы делаете нечто хоть сколько-нибудь нестандартное (даже если оно должно быть стандартным). Если вы окажетесь в подобной ситуации, окажите услугу всем нам: наймите разработчика такого компилятора якобы для серьезного проекта и заставьте его писать и отлаживать код обработки исключений для его собственного компилятора в течение ближайших пяти лет. А еще лучше, заставьте его перенести в свой компилятор код, идеально работающий на другом компиляторе.

Если по какой-либо причине вам придется заниматься нестандартной обработкой исключений, больше всего проблем вызовет освобождение памяти от динамических объектов, которые были созданы до возникновения исключения и внезапно стали недоступными переменным. Именно для этой цели используется 99% обработки исключений в реальной жизни, да еще изредка требуется закрыть открытые файлы. Вы можете либо создать хитроумные структуры данных, которые перемещают новые динамические объекты на верх стека и, следовательно, сохраняют их доступность, либо воспользоваться методикой сборки мусора, описанной в последней части книги. Оба подхода выглядят в равной степени отталкивающе, так что выбирайте сами.

Условные обозначения

В этой книге я не использую ключевые слова throw и catch, а вставляю общие комментарии в тех местах, где может возникнуть исключение. Такой подход упрощает работу с программой, если ваш компилятор не поддерживает стандартную обработку исключений. Если вы увидите что-нибудь вроде следующего фрагмента и располагаете стандартной обработкой исключений, мысленно превратите комментарий в блок throw:

```
f()
{
    if (pool->Allocate(size) == NULL)
        // исключение – нехватка памяти
}
```

Часть 2

Косвенные обращения

Когда указатель на самом деле не является указателем? Когда вы программируете на C++ и работаете с умными указателями. Одна из самых богатых (и самых недооцененных) концепций C++ — то, что объект может выполнять функции указателя. В этой части подробно рассматриваются различные способы, позволяющие это сделать, а заодно и примеры практического применения описанных идиом. Эта часть — не сборник рецептов, а скорее набор новых инструментов для вашей мастерской. Руки приложить все равно придется, но с хорошими инструментами работа пойдет гораздо легче.

Умные указатели

5

Забудьте все, что вам известно о С и примитивных операторах ->, и взгляните на проблему под новым углом. Рассмотрим следующий фрагмент:

```
class Foo {
public:
     void MemberOfFoo();
};
Foo* aFoo = new Foo;
aFoo->MemberOfFoo();
```

Представьте, что встроенный оператор -> применяется к встроенному классу указателя — адресу, хранящемуся в aFoo. C++ предоставляет такой оператор для любого объекта, имеющего тип указателя на структуру, указателя на класс или указателя на объединение. Встроенные операторы -> осуществляют доступ к членам, указанным справа (в данном примере MemberOfFoo()). Фактически вы ссылаетесь на члена объекта (Foo) с помощью другого объекта и его оператора ->. То, что другой объект является указателем — всего лишь частный случай; вместо него мог бы использоваться ваш собственный класс с написанным вами нестандартным оператором ->.

Именно так следует подходить к оператору -> в C++, поскольку он, как и все остальные операторы (кроме оператора .), может перегружаться. Синтаксис был в общих чертах обрисован в главе 2, однако его последствия для программирования огромны, и их обсуждение займет эту и несколько следующих глав.

Глупые указатели

В C++ предусмотрено немало встроенных типов данных — например, int, double и указатели. Большинство этих встроенных типов удается относительно легко «упаковать» в разработанные вами классы. Например, если простой тип int недостаточно хорош для вас, можно создать совместимый класс Integer, который выглядит примерно так:

```
class Integer {
private:
     int value;
public:
     Integer() : value(0) {}
     Integer(int v) : value(v) {}
     operator int() { return value; }
     Integer operator+(Integer i) { return Integer(value + i.value); }
     Integer operator+=(Integer i) { value += i.value; return *this; }
     // И т.д. для остальных арифметических операторов
};
int f(int);
f(Integer(17));    // Работает благодаря оператору int()
```

Конструкторы позволяют создавать экземпляры Integer из ничего, по существующему int или другому экземпляру Integer. Возможно, стоит создать дополнительные конструкторы для параметра char* (с преобразованием atoi()) и других числовых типов. Так как Integer иногда придется использовать вместо int (например, при вызове функции с аргументом int), мы предусмотрели оператор int() для выполнения автоматических преобразований. Скучная череда разных операторов воспроизводит всю семантику целочисленной арифметики. Ура! Отныне вместо int можно повсюду использовать Integer. Мы создали новый класс, который полностью заменяет примитивный int. Выпейте чашку кофе.

Уже вернулись? Предположим, в своем стремлении к абстрактным типам данных в C++ (политически правильный термин для классов-оболочек) вы решили, что *-указатели вас уже не устраивают, и вы хотите спрятать их в своих классах (не спрашивайте почему; я же сказал, что это была ваша идея!). Давайте проделаем мысленный эксперимент и посмотрим, что для этого нужно. Итак, первая попытка.

```
class PFoo {
private:
    Foo* foo;
public:
    PFoo() : foo(NULL) {}
    PFoo(Foo* f) : foo(f) {}
    operator Foo*() { return foo; }
    PFoo operator+(ptr_diff offset) { return PFoo(foo + offset); }
    PFoo operator+=(ptr_diff offset) { foo += offset; return *this; }
    ptr_diff operator-(PFoo pf) { return foo - pf.foo; }
    // И т.д. для всех остальных арифметических операций с указателями
};
```

Ура! У вас появился новый класс, совместимый с Foo*! С арифметическими операторами дело обстоит несколько сложнее. Например, вам наверняка пришлось докопаться в справочнике, чтобы узнать, что ptr_diff является переносимым способом описания разности между двумя адресами памяти. Все это выглядит занудно, поскольку класс ориентирован на Foo, но зато теперь вы можете всюду использовать PFoo вместо Foo*... Но так ли это? Подождите минутку и оставьте вторую чашку кофе. Наш вариант не работает.

```
PFoo pf(new Foo*);
pf->MemberOfFoo();          // Неверно
((Foo*)pf)->MemberOfFoo();  // Работает, но выглядит ОМЕРЗИТЕЛЬНО!
```

Оператор Foo*() позволит передавать PFoo в качестве аргумента функциям, которые должны получать Foo*. Также воспроизводятся все арифметические операции с указателями, но часть семантики все же теряется, поскольку оператор -> не работает в прежнем виде. Я называю такие указатели «глупыми», так как они очень наивны и ведут себя совсем не так, как должны себя вести настоящие указатели.

Итак, почему написать оболочку для указателей сложнее, чем для других базовых типов?

1. Компилятор создает отдельный тип указателя для каждого типа структуры, класса или объединения. Чтобы создать оболочки для всех указателей, вам придется думать, как сделать это для всех возможных типов указателей. Описанный выше класс PFoo работает лишь для класса Foo и производных от него.

2. Приходится учитывать и другие, взаимосвязанные типы данных (например, size_t и ptr_diff). Если мы хотим полностью «упаковать» указатели в наш класс, придется создавать эквиваленты и для этих типов.

3. Вся суть встроенных указателей — обращение к членам объекта с использованием оператора ->. Для воспроизведения этой семантики оператора преобразования оказывается недостаточно.

Умные указатели как идиома

Возникающие проблемы стоит разбирать последовательно. До арифметических операций с указателями мы доберемся позже, поэтому пока будем пользоваться ptr_diff.

Оператор ->

Теперь вы знаете, почему оператор -> был сделан перегружаемым. В полном соответствии с синтаксисом, описанным в главе 2, PFoo теперь обзаводится собственным оператором ->. Оператора преобразования хватает для вызова внешних функций. Приведенный ниже вызов функции f() работает, потому что у компилятора хватает ума поискать оператор преобразования, соответствующий сигнатуре функции, и в данном случае оператор Foo*() прекрасно подходит.

```
class PFoo {
private:
    Foo* foo;
public:
    PFoo() : foo(NULL) {}
    PFoo(Foo* f) : foo(f) {}
    operator Foo*() { return foo; }
    Foo* operator->() { return foo; }
};
void f(Foo*);
PFoo pf(new Foo);
f(pf);                  // Работает благодаря функции operator Foo*()
pf->MemberOfFoo();      // Работает благодаря функции operator->()
```

Причина, по которой работает pf->MemberOfFoo(), менее очевидна. В левой части оператора -> указан пользовательский тип, поэтому компилятор ищет перегруженную версию оператора ->. Он находит ее, вычисляет и заменяет pf возвращаемым значением, которое превращается в новое левостороннее выражение оператора ->. Этот процесс рекурсивно продолжается до тех пор, пока левостороннее выражение не преобразуется к базовому типу. Если таким базовым типом является указатель на структуру, указатель на класс или указатель на объединение, компилятор обращается к указанному члену. Если это что-то иное (например, int), компилятор злорадно хохочет и выдает сообщение об ошибке. В нем он оценивает ваш интеллект и перспективы будущей работы на основании того факта, что вы пытаетесь обратиться к члену чего-то, вообще не имеющего членов. В любом случае поиск заканчивается при достижении базового типа. Для самых любопытных сообщаю, что большинство компиляторов, которыми я пользовался, не отслеживает истинной рекурсии вида:

```
PFoo operator->() { return *this; }
```

Здесь оператор -> пользовательского типа возвращает экземпляр этого типа в качестве своего значения. Компиляторы C++ обычно предпочитают помучить вас в бесконечном цикле.

Итак, у нас появился класс-указатель, который можно использовать везде, где используются указатели Foo*: в качестве аргументов функций, слева от оператора -> или при определении дополнительной семантики арифметических операций с указателями — всюду, где Foo* участвует в сложении или вычитании.

Параметризованные умные указатели

Один из очевидных подходов к созданию универсальных умных указателей — использование шаблонов.

```
template <class Type>
class SP {
private:
    Type* pointer;
public:
```

```
        SP() : pointer(NULL) {}
        SP(Type* p) : pointer(p) {}
        operator Type*() { return pointer; }
        Type* operator->() { return pointer; }
    };
    void f(Foo*);
    Ptr<Foo> pf(new Foo);
    f(pf);                    // Работает благодаря функции operator Type*()
    pf->MemberOfFoo();        // Работает благодаря функции operator->()
```

Этот шаблон подойдет для любого класса, не только для класса Foo. Перед вами — одна из базовых форм умных указателей. Она используется достаточно широко и даже может преобразовать указатель на производный класс к указателю на базовый класс при условии, что вы пользуетесь хорошим компилятором.

Хороший компилятор C++ правильно обрабатывает такие ситуации, руководствуясь следующей логикой:

1. Существует ли конструктор P<Foo>, который получает P<Bar>? Нет. Продолжаем поиски.
2. Существует ли в P<Bar> операторная функция operator P<Foo>()? Нет. Ищем дальше.
3. Существует ли пользовательское преобразование от P<Bar> к типу, который подходит под сигнатуру какого-либо конструктора P<Foo>? Да! Операторная функция operator Bar*() превращает P<Bar> в Bar*, который может быть преобразован компилятором в Foo*. Фактически выражение вычисляется как Ptr<Foo>pf2(Foo*(pb.operator Bar*())), где преобразование Bar* в Foo* выполняется так же, как для любого другого встроенного указателя.

Как я уже говорил, все *должно* работать именно так, но учтите — некоторые компиляторы обрабатывают эту ситуацию неправильно. Даже в хороших компиляторах результат вложения подставляемой (inline) операторной функции operator Bar*() во встроенный P<Foo>(Foo*) может быть совсем не тем, на который вы рассчитывали; многие компиляторы создают вынесенные (а следовательно, менее эффективные) копии встроенных функций классов вместо того, чтобы генерировать вложенный код подставляемой функции. Мораль: такой шаблон *должен* делать то, что вы хотите, но у компилятора на этот счет может быть другое мнение.

Иерархия умных указателей

Вместо использования шаблонов можно поддерживать параллельные иерархии указателей и объектов, на которые они указывают. Делать это следует лишь в том случае, если ваш компилятор не поддерживает шаблоны или плохо написан.

```
    class PVoid {       // Заменяет void*
    protected:
        void* addr;
    public:
        PVoid() : addr(NULL) {}
        PVoid(void* a) : addr(a) {}
        operator void*() { return addr; }
    };
    class Foo : public PVoid {
    public:
        PFoo() : PVoid() {}
        PFoo(Foo* p) : PVoid(p) {}
        operator Foo*() { return (Foo*)addr; }
        Foo* operator->() { return (Foo*)addr; }
```

```
};
class Pbar : public PFoo {
public:
     PBar() : PFoo() {}
     PBar(Bar* p) : PFoo(p) {}
     operator Bar*() { return (Bar*)addr; }
     Bar* operator->() { return (Bar*)addr; }
};
pBar pb(new Bar);
pFoo pf(pb);        // Работает, потому что PBar является производным от PFoo
pf->MemberOfFoo(); // Работает благодаря PFoo::operator->
```

Этот вариант будет работать, если вас не огорчают многочисленные копирования/вставки текста и (в зависимости от компилятора) предупреждения о том, что `PBar::operator->()` скрывает `PFoo::operator->()`. Конечно, такое решение не настолько элегантно, как встроенные типы указателей шаблона `Ptr`.

Арифметические операции с указателями

Ниже показан пример арифметических операторов, обеспечивающих работу операций сложения/вычитания для умных указателей. Для полной, абсолютно совместимой реализации к ним следует добавить операторы ++ и `--`.

```
template <class Type>
class Ptr {
private:
     Type* pointer;
public:
     Ptr() : pointer(NULL) {}
     Ptr(Type* p) : pointer(p) {}
     operator Type*() { return pointer; }
     ptr_diff operator-(Ptr<Type> p) { return pointer - p.pointer; }
     ptr_diff operator-(void* v) { return ((void*)pointer) - v; }
     Ptr<Type> operator-(long index) { return Ptr<Type>(pointer - index); }
     Ptr<Type> operator-=(long index) { pointer -= index; return *this; }
     Ptr<Type> operator+(long index) { return Ptr<Type>(pointer + index); }
     Ptr<Type> operator+=(long index) { pointer += index; return *this; }
};
```

Важно понимать, чем `ptr_diff` отличается от целого индекса. При вычитании одного адреса из другого результатом является смещение, как правило, выраженное в байтах. В случае прибавления целого к указателю адрес изменяется на размер объекта, умноженный на целое. Помните: в C++, как и в C, указатель ссылается не на один объект, а на теоретический массив объектов. Индексы в описанных выше перегруженных операторах представляют собой индексы этого теоретического массива, а не количества байт.

После всего сказанного я бы не советовал пускаться на эти хлопоты для умных указателей — не из-за лени, а потому, что в этом случае вы обрекаете себя на решения, которые не всегда желательны. Если дать пользователю возможность складывать и вычитать указатели, вам неизбежно придется поддерживать идею, что указатель всегда индексирует воображаемый массив. Как выяснится в следующих главах, многие варианты применения умных указателей не должны или даже не могут правильно работать с парадигмой массива.

Во что обходится умный указатель?

Объект класса, не содержащего виртуальных функций, занимает столько места, сколько необходимо для хранения всех его переменных. В рассмотренных выше умных указателях используется всего одна переменная — *-указатель; то есть размер умного указателя в точности совпадает с размером встроенного указателя. Хороший компилятор C++ должен специальным образом обрабатывать тривиальные подставляемые функции, в том числе и находящиеся в шаблоне умного указателя.

```
template <class Type>
class Ptr {
private:
    Type* pointer;
public:
    Ptr() : pointer(NULL) {}
    Ptr(Type* p) : pointer(p) {}
    operator Type*() { return pointer; }
    Type* operator->() { return pointer; }
};
```

В частности, использование оператора -> из этого шаблона не должно требовать никаких дополнительных вычислений по сравнению со встроенными указателями. Как всегда, обращайте особое внимание на слова *хороший* и *должно*. В хорошей реализации описанные выше умные указатели не требуют никаких дополнительных расходов. По крайней мере, хуже пока не стало.

Применения

Умные указатели — существа на редкость полезные, и мы проведем немало времени, изучая их применение на практике. Для простых умных указателей, рассматриваемых в этой главе, находятся столь же простые, но мощные применения.

Разыменование значения NULL

Рассмотрим одну из вариаций на тему умных указателей:

```
template <class Type>
class SPN {
private:
    Type* pointer;
public:
    SPN() : pointer(NULL) {}
    SPN(Type* p) : pointer(p) {}
    operator Type*() { return pointer; }
    Type* operator->()
    {
        if (pointer == NULL) {
            cerr << "Dereferencing NULL!" << endl;
            pointer = new Type;
        }
        return pointer;
}
```

При попытке вызвать оператор -> для указателя `pointer`, равного `NULL`, в поток `stderr` выводится сообщение об ошибке, после чего создается фиктивный объект и умный указатель переводится на него, чтобы программа могла хромать дальше.

Существует столько разных решений, сколько найдется программистов, достаточно глупых для попыток разыменования значения NULL. Вот лишь несколько из них.

Использование #indef

Если вас раздражают дополнительные вычисления, связанные с этой логикой, проще всего окружить if-блок директивами #ifdef, чтобы код обработки ошибок генерировался только в отладочных версиях программы. При компиляции рабочей версии перегруженный оператор -> снова сравнивается по быстродействию со встроенным указателем.

Инициирование исключений

Выдача сообщений об ошибках может вызвать проблемы в некоторых графических программах. Вместо этого можно инициировать исключение:

```
template <class Type>
class Ptr {
private:
    Type* pointer;
public:
    enum ErrorType { DereferenceNil };

    Ptr() : pointer(NULL) {}
    Ptr(Type* p) : pointer(p) {}
    operator Type*() { return pointer; }
    Type* operator->() throw(ErrorType)
    {
        if (pointer == NULL) throw DereferenceNil;
        return pointer;
    }
};
```

(На практике ErrorType заменяется глобальным типом, используемым для различных видов ошибок; приведенный фрагмент лишь демонстрирует общий принцип.) Это решение может объединяться с другими. Например, программа может использовать фиктивный объект в отладочном варианте и инициировать исключение в рабочей версии.

Стукачи

Еще один вариант — хранить в статической переменной специальный объект, который я называю «стукачом» (screamer). Стукач ждет, пока кто-нибудь не попытается выполнить разыменование значения NULL.

```
template <class Type>
class AHHH {
private:
    Type* pointer;
    static type* screamer;
public:
    AHHH() : pointer(NULL) {}
    AHHH(Type* p) : pointer(p) {}
    Operator Type*() { return pointer; }
    Type* operator->()
    {
        if (p == NULL) return screamer;
        return pointer;
```

```
    }
};
```

«Ну и что такого?» — спросите вы. Предположим, `screamer` на самом деле не принадлежит к типу `Type*` а относится к производному классу, все функции которого (предположительно виртуальные) выводят сообщения об ошибках в поток `cerr` перед вызовом своих прототипов базового класса. Теперь вы не только удержите свою программу на плаву, но и сможете следить за попытками вызова функций фиктивного объекта.

Отладка и трассировка

Умные указатели также могут использоваться для наблюдения за объектами, на которые они указывают. Поскольку все обращения к объекту выполняются через операторную функцию `operator Type*()` или `operator->()`, у вас появляются две контрольные точки для наблюдения за происходящим во время работы программы. Возможности отладки безграничны, я приведу лишь один из примеров.

Установка точек прерывания

Самое простое применение упомянутых контрольных точек — сделать эти функции вынесенными (out-of-line) в отладочной версии и расставить точки прерывания в их реализации.

```
template <class Type>
class PTracer {
private:
    Type* pointer;
public:
    PTracer() : pointer(NULL) {}
    PTracer(Type* p) : pointer(p) {}
    operator Type*();
    Type* operator->();
};
template <class Type>
#ifdef DEBUGGING
inline
#endif
PTracer<Type>::operator Type*()
{
    return pointer;    // Здесь устанавливается точка прерывания
}
template <class Type>
#ifdef DEBUGGING
inline
#endif
Type* PTracer<Type>::operator->()
{
    return pointer;    // Здесь устанавливается точка прерывания
}
```

С непараметризованными версиями указателей это сделать несколько проще, поскольку не все среды разработки позволяют устанавливать точки прерывания в параметризованных функциях.

Трассировка

Оператор преобразования и оператор `->` могут выводить диагностическую информацию в поток `cout` или `cerr`, в зависимости от ситуации.

Ведение статистики класса

Также несложно организовать накопление статистики об использовании операторов `Type*` и `->` в статических переменных параметризованного класса.

```
template <class Type>
class SPCS {
private:
     Type* pointer;
     static int conversions;
     static int members;
public:
     SPCS() : pointer(NULL) {}
     SPCS(Type* p) : pointer(p) {}
     operator Type*() { conversions++; return pointer; }
     Type* operator->() { members++; return pointer; }
     int Conversions() { return conversions; }
     int Members() { return members; }
};
```

Глобальные переменные должны быть где-то определены. Обычно это делается в файле Foo.cpp:

```
// В файле Foo.cpp
int Ptr<Foo>::conversions = 0;
int Ptr<Foo>::members = 0;
```

Разумеется, вы можете воспользоваться директивами `#ifdef`, чтобы это относилось только к отладочной версии.

Ведение статистики объекта

Мы подошли к более сложной теме. Возможно, ее следует отложить до знакомства с ведущими указателями (master pointers), однако умные указатели также могут вести статистику по отдельным объектам, а не по классу в целом. Задача не сводится к тому, чтобы сделать только что показанные переменные нестатическими (то есть по одному экземпляру переменных на указатель), поскольку мы (пока) не можем обеспечить однозначное соответствие между указателями и объектами. Вместо этого статистику придется хранить в самих объектах. Ниже приведен полезный вспомогательный класс, который можно создать на основе множественного наследования как производный от класса указываемого объекта и от класса умного указателя, знающего о его свойствах. Объявляя указатель другом, вы предоставляете ему доступ к защищенным членам классов, производных от `Counter`.

```
class Counter {
protected:
     Counter() : conversions(0), members(0) {}
     Counter(const Counter&) : conversions(0), members(0) {}
     Counter& operator=(const Counter&) { return *this; }
public:
     int conversions;
     int members;
     int Conversions() { return conversions; }
     int Members() { return members; }
};
template <class Type>
class SPOP {
private:
     Type* pointer;
```

```
public:
    SPOS() : pointer(NULL) {}
    SPOP(Type* f) : pointer(f) {}
    operator Type*() { pointer->conversions++; return pointer; }
    Type* operator->() { pointer->members++; return pointer; }
};
```

На эту тему существует ряд вариаций, с некоторыми из них мы познакомимся при изучении ведущих указателей.

Кэширование

Иногда нельзя даже настаивать, чтобы объект физически находился в памяти все время, пока к нему нужен доступ. В следующем примере предполагается, что функция `ReadObject()` умеет использовать данные о местонахождении объекта на диске, чтобы создать экземпляр и занести его адрес в указатель `pointer`. Если при вызове операторов объект отсутствует в памяти, он автоматически считывается с диска.

```
typedef unsigned long DiskAddress;  // Заменить нужными данными
template <class Type>
class CP {
private:
    DiskAddress record_number;
    Type* pointer;
    void ReadObject();      // Считывает объект с диска
public:
    CP(DiskAddress da) : pointer(NULL), record_number(da) {}
    CP(Type* f) : pointer(f), record_number(f->RecordNumber()) {}
    operator Type*()
    {
        if (pointer == NULL) this->ReadObject();
        return pointer;
    }
    Type* operator->()
    {
        if (pointer == NULL) this->ReadObject();
        return pointer;
    }
};
```

Подробно говорить о кэшировании преждевременно, поскольку приходится учитывать множество проблем, к которым мы еще не готовы. Если вы хотите гарантировать, что читается лишь одна копия объекта независимо от того, сколько различных объектов на нее ссылается, или чтобы объект уничтожался сразу после завершения работы операторов, вам придется подождать следующих глав, посвященных ведущим указателям и управлению памятью. Тем не менее, в простых ситуациях с одним считыванием, в которых может существовать несколько копий объекта, такая методика достаточно хорошо работает и с простыми умными указателями.

Кэширующие указатели используют один распространенный прием — они экономят несколько бит за счет объединения дискового адреса и адреса памяти в одной переменной класса. При этом используются два упрощающих предположения:

1. Размер адреса памяти не более чем на один бит превышает размер дискового адреса.
2. Средства управления памятью, используемые оператором `new`, никогда не возвращают нечетный адрес.

Если оба предположения верны, для хранения обоих адресов можно использовать одну переменную класса. Если младший бит установлен (то есть если «адрес» четный), остальные 31 бит определяют дисковый адрес. Когда младший бит сброшен, все 32 бита определяют адрес памяти. Если вам потребуется не только считывание, но и запись, объекту лучше знать свой собственный дисковый адрес, поскольку адрес, хранящийся в указателе, при считывании портится.

За дымовой завесой кэширующих указателей прячется интересная концепция: умные указатели могут использоваться как общее средство для доступа к объектам *независимо от того, где находится объект и существует ли он вообще.* Углубляясь в джунгли C++, мы будем рассматривать эту концепцию под разными углами, пока она не превратится в один из принципов Дао, о которых я упоминал во вступительной главе.

Ведущие указатели и дескрипторы

6

После совсем непродолжительного знакомства с умными указателями мы успели наткнуться на целый ряд фундаментальных проблем. Многие из них связаны с тем фактом, что на один объект может ссылаться любое количество умных указателей. Как в этом случае узнать, когда можно удалить объект? Кто ведет статистику использования объекта и обращается к ней при необходимости (если вам вдруг понадобится такая возможность)? Кто создает объект? Что означает присваивание одного умного указателя другому? Заиграет ли наконец в этом сезоне наша любимая команда или она снова разобьет наши сердца? Ниже вы найдете ответы на эти и многие другие интригующие вопросы.

Семантика ведущих указателей

При работе с умными указателями имеется один важный частный случай — когда два умных указателя не должны одновременно ссылаться на один объект. Между указателем и объектом, на который он ссылается, существует однозначное соответствие (за исключением особого случая умных указателей, ссылающихся на `NULL`). Если в программном дизайне действует такое ограничение, говорят, что реализуется семантика *ведущих указателей* (master pointers).

Конечно, можно просто объявить через местную газету, что указатели должны использоваться таким и только таким образом. А можно защитить ваших пользователей от самих себя и подкрепить семантические правила языковыми средствами C++. Если вы мудро выберете второй вариант, придется учесть следующее:

1. Указываемые объекты должны создаваться указателями в конструкторах.
2. Деструктор указателя должен удалять указываемый объект.
3. Конструктор копий должен создавать точную копию указываемого объекта.
4. Оператор присваивания `operator=` должен удалять текущий указываемый объект, находящийся слева от него, и заменять его копией указываемого объекта справа.

Кроме того, было бы *разумно* сделать еще две вещи:

5. Ограничить доступ к конструкторам класса указываемого объекта.
6. Создавать указатели с помощью *производящих функций* (*factory functions*).

Обе рекомендации будут подробно рассмотрены в последующих разделах. Прототип ведущего указателя, который мы собираемся воплотить, выглядит так:

```
template <class Type>
class MP {
private:
    Type* t;
public:
```

```
    MP();                   // Создает указываемый объект
    MP(const MP<Type>&);    // Копирует указываемый объект
    ~MP();                  // Удаляет указываемый объект
    MP<Type>& operator=(const MP<Type>&); // Удаляет левосторонний объект,
                                          // копирует правосторонний
    Type* operator->() const;
};
```

Конструирование

Предположим, вы разрешили пользователям создавать собственные указываемые объекты и передавать их ведущим указателям во время конструирования.

```
template <class Type>
class MP {
private:
    Type* t;
public:
    MP() : t(NULL) {}
    MP(Type* pointer) : t(pointer) {}
    ~MP() { delete t; }
    // И т.д.
};
Foo* foo = new Foo;
MP<Foo> mp1(foo);
MP<Foo> mp2(foo);  // Облом!
```

Насколько проще была бы жизнь без таких пользователей! Когда `mp1` удаляется, пропадает и `foo`. В итоге `mp2` начинает указывать неизвестно куда. «Зачем кому-нибудь потребуется вытворять такое?» — спросите вы. Как говорилось в одном рекламном ролике: «Зачем спрашивать «Зачем»?» Если вы оставите такую дыру, можете не сомневаться: кто-нибудь когда-нибудь изобретет дьявольский план, использует ее и обвинит во всех смертных грехах вашу программу.

Пользователь прямо-таки кричит: «Держите меня, пока я не натворил бед». Для этого существует надежный способ: отобрать у него ключи от конструкторов класса указываемого объекта.

```
class Foo {
friend class MP<Foo>;
protected:
    Foo();   // Теперь доступ к конструктору имеет только MP<Foo>
public:
    // Оставшаяся часть интерфейса
};
template <class Type>
class MP {
private:
    Type* t;
public:
    MP() : t(new Type) {}
    // И т.д.
};
```

Ага, уже лучше. При создании указателя его конструктор также конструирует и указываемый объект. Объявляя указатель другом, мы можем сделать конструкторы `Foo` закрытыми или защищенными и

сохранить доступ к ним из конструкторов указателя. Теперь клиент никак не сможет добраться до Foo, кроме как через MP<Foo>. Мы еще неоднократно вернемся к вопросу о том, как, когда и где создавать указываемые объекты, а пока давайте немного отвлечемся.

Если конструкторы Foo вызываются с аргументами, существуют две альтернативы:

1. Вместо того чтобы пользоваться универсальным шаблоном ведущего указателя, создайте для Foo нестандартный класс ведущего указателя MPFoo. Для каждого конструктора Foo создайте конструктор MPFoo с точно такой же сигнатурой и передайте его аргументы конструктору Foo.
2. Воспользуйтесь безаргументным конструктором для создания объекта и предоставьте отдельную функцию инициализации, которая может вызываться клиентом после конструирования.

Второй вариант выглядит так:

```
class Foo {
friend class MP<Foo>;
protected:
    Foo();    // Единственный конструктор
public:
    Initialized(int, char*);
    // Оставшаяся часть интерфейса
};
MP<Foo> mpf;
mpf->Initialize(17, “Hello”);  // Завершить конструирование
```

Такое решение выглядит довольно неуклюже, но оно позволяет работать с универсальным шаблоном ведущего указателя. Существуют и другие причины для использования инициализирующих функций, о которых будет рассказано в следующих главах. Любой из этих вариантов вполне приемлем для решения наших текущих задач.

Уничтожение

Нет ничего проще: в деструкторе ведущего указателя удаляется и указываемый объект.

```
template <class Type>
class MP {
private:
    Type* t;
public:
    ~MP() { delete t; }
};
```

Копирование

Ой! Опять эти проклятые пользователи...

```
MP<Foo> mpf1;           // Создает Foo, на который ссылается указатель
MP<Foo> mpf2 = mpf1;    // Неудача!
```

Пусть знак равенства не вводит вас в заблуждение — здесь происходит конструирование, и эта строка эквивалентна строке MP<Foo> mpf2(mpfl);. Если не перегрузить конструктор копий и разрешить компилятору C++ внести свою лепту, мы получим два ведущих указателя, которые ссылаются на один и тот же объект Foo. По умолчанию конструктор копий, генерируемый компилятором, радостно копирует содержащийся в переменной адрес из старого указателя в новый. Проблема решается относительно просто.

```
template <class Type>
class MP {
```

```
    private:
        Type* t;
    public:
        MP();                                    // Нормальный
        MP(const MP<Type>& mp) : t(*(mp.t)) {}   // Конструктор копий
    };
```

Этот конструктор копий создает дубликат указываемого объекта, используя для этого конструктор копий указываемого объекта. Получается не очень эффективно, но работает. В некоторых ситуациях, с которыми мы столкнемся позже, лучше вообще запретить копирование. Проще всего для этого объявить конструктор копий закрытым и не назначать ему никаких действий.

```
    template <class Type>
    class MP {
    private:
        Type* t;
        MP(const MP<Type>&) : t(NULL) {}     // Никогда не будет вызываться
    public:
        MP();
    };
```

Тем самым мы предотвращаем непреднамеренное копирование в ситуациях вроде следующей:

```
    void f(MP<Foo>);
    MP<Foo> mpf;
    f(mpf);        // Создается временная копия
```

Для предотвращения копирования также можно воспользоваться дескрипторами, о которых будет рассказано ниже.

Присваивание

Ааааа! Эти зловредные пользователи когда-нибудь угомонятся?!

```
    MP<Foo> mpf1;
    MP<Foo> mpf2;
    mpf2 = mpf1;        // Нет, только не это...
```

В приведенном фрагменте возникают сразу две проблемы. Во-первых, указываемый объект, созданный конструктором mpf2, никогда не удаляется. Он превращается в Летучего Голландца, обреченного на вечные скитания в океане памяти. Во-вторых, оператор =, используемый компилятором по умолчанию, копирует адрес, находящийся в t, из одного указателя в другой, что приводит к появлению двух ведущих указателей, ссылающихся на один объект. В исправленном варианте перегруженный оператор = удаляет объект, на который ссылается левосторонний указатель, и заменяет его копией объекта, на который ссылается правосторонний указатель.

```
    template <class Type>
    class MP {
    private:
        Type* t;
    public:
        MP();          // Нормальный конструктор
        MP<Type>& operator=(const MP<Type>& mp)
        {
            if (&mp != this) {
                delete t;
                t = new Type(*(mp.t));
```

```
        }
        return *this;
    }
};
```

Разумеется, если вы вообще не хотите поддерживать присваивание, достаточно объявить оператор = закрытым.

Прототип шаблона ведущего указателя

Ниже приведен конечный результат наших усилий. Подставляемые функции переместились на привычное место после объявления класса. Параметр, класс указываемого объекта, должен удовлетворять следующим требованиям:

1. Он должен иметь безаргументный конструктор.
2. Он должен либо перегружать конструктор копий, либо предоставленный компилятором конструктор копий по умолчанию должен подходить для использования в шаблоне ведущего указателя.

Если хотя бы одно из этих требований не выполняется, придется внести изменения в класс указываемого объекта или ведущего указателя, или в оба класса сразу. Помните: в реализации конструктора копий и оператора присваивания ведущего указателя будет использоваться конструктор копий указываемого объекта (то есть параметра шаблона).

```
template <class Type>
class MP {
private:
    Type* t;
public:
    MP();                   // Создает указываемый объект
    MP(const MP<Type>&);    // Копирует указываемый объект
    ~MP();                  // Удаляет указываемый объект
    MP<Type>& operator=(const MP<Type>&);
    Type* operator->() const;
};
template <class Type>
inline MP<Type>::MP() : t(new Type)
{}
template <class Type>
inline MP<Type>::MP(const MP<Type>& mp) : t(new Type(*(mp.t)))
{}
template <class Type>
inline MP<Type>::~MP()
{
    delete t;
}
template <class Type>
inline MP<Type>& MP<Type>::operator=(const MP<Type>& mp)
{
    if (this != &mp) {
        delete t;
        t = new Type(*(mp.t));
    }
    return *this;
```

```
}
template <class Type>
inline Type* MP<Type>::operator->() const
{
    return t;
}
```

Дескрипторы в C++

Итак, после небольшого подогрева умные указатели превратились в ведущие. Теперь в нашем вареве появляется еще один ингредиент — дескрипторы (handles) C++. Не путайте этот термин с дескрипторами, используемыми в операционных системах Macintosh и Windows. Некоторое сходство существует, но идиома дескрипторов C++ имеет собственную уникальную семантику и набор правил.

Основная идея заключается в том, чтобы использовать умные указатели для ссылок на ведущие указатели. Эти дополнительные указатели и называются дескрипторами. Основа, на которой мы будем строить класс дескрипторов, в первом приближении выглядит так:

```
template <class Type>
class H {
private:
    MP<Type>& ptr;     // Ссылка на ведущий указатель
public:
    H() : ptr(*(new MP<Type>)) {}   // См. Ниже
    H(MP<Type>& mp) : ptr(mp) {}
    MP<Type>& operator->() const { return ptr; }
};
```

Безаргументный конструктор `H` создает новый ведущий указатель. Этот ведущий указатель, в свою очередь, создает указываемый объект. Существует второй конструктор, который получает ведущий указатель и инициализирует им переменную `ptr`. Конструктор копий и оператор = по умолчанию годятся, поскольку любому ведущему указателю может соответствовать несколько дескрипторов. Работа оператора `->` основана на рекурсивном алгоритме, используемом компилятором: оператор `->` дескриптора возвращает ведущий указатель; затем оператор `->` ведущего указателя возвращает указатель `Type*` который является одним из базовых типов компилятора.

Приведенное решение не назовешь изящным — вложенные шаблоны порождают путаницу, к тому же совершенно неясно, когда и как удалять ведущие указатели. Кроме того, следует ли разрешить пользователю напрямую создавать и уничтожать ведущие указатели или же заключить их внутри дескрипторов так, как мы заключили указываемые объекты внутри ведущих указателей? Неужели мы трудились над решением этих проблем для указываемых объектов лишь затем, чтобы столкнуться с ними снова для ведущих указателей? Терпение — в свое время мы найдем ответ на эти и многие другие вопросы.

Что же получается?

Мы начнем с простого примера ведущих указателей и усовершенствуем его до уровня, который удовлетворил бы и более требовательную аудиторию. На этой стадии еще трудно понять всю пользу дескрипторов, но в следующих главах они будут играть очень важную роль.

Подсчет объектов

Допустим, вы хотите следить за количеством созданных или находящихся в памяти объектов некоторого класса. Одно из возможных решений — хранить эти сведения в статических переменных самого класса.

```
class CountedStuff {
private:
```

```
        static int current;
    public:
        CountedStuff() { current++; }
        CountedStuff(const CuntedStuff&) { current++; }
        CountedStuff& operator=(const CountedStuff&)
            {}   // Не менять счетчик для присваивания
        ~CountedStuff() { current--; }
    };
```

С этим примером еще можно повозиться и улучшить его, но как бы вы ни старались, придется изменять код целевого класса — хотя бы для того, чтобы заставить его наследовать от нашего класса. Теперь предположим, что указываемый объект входит в коммерческую библиотеку. Обидно, да? Любые изменения нежелательны, а скорее всего, просто невозможны. Но вот на сцену выходит класс ведущего указателя.

```
    template <class Type>
    class CMP {
    private:
        static int current;
        Type* ptr;
    public:
        CMP() : ptr(new Type) { current++; }
        CMP(const CMP<Type>& cmp) : ptr(new Type(*(mp.t))) { current++; }
        CMP<Type>& operator=(const CMP<Type>& cmp)
        {
            if (this != &cmp) {
                delete ptr;
                ptr = new Type(*(cmp.ptr));
            }
            return *this;
        }
        ~CMP() { delete ptr; current--; }
        Type* operator->() const { return ptr; }
        Static int Current() { return current; }
    };
```

Теперь ведущий указатель выполняет все подсчеты за вас. Он не требует внесения изменений в класс указываемого объекта. Этот шаблон может использоваться для *любого* класса при условии, что вам удастся втиснуть ведущие указатели между клиентом и указываемым объектом. Даже если вы не возражаете против модификации исходного класса указываемого объекта, обеспечить такой уровень модульности без ведущих указателей было бы крайне сложно (например, если бы вы попытались действовать через базовый класс, то в результате получили бы одну статическую переменную `current` на все производные классы).

Этот пример тривиален, но даже он демонстрирует важный принцип программирования на C++, справедливость которого со временем становится очевидной: пользуйтесь умными указателями, даже если сначала кажется, что они не нужны. Если программа написана для умных указателей, все изменения вносятся легко и быстро. Если же вам придется переделывать готовую программу и заменять все операторы * умными указателями, приготовьтесь к ночным бдениям.

В главе 14 вариации на тему подсчета будут использованы для реализации простой, но в то же время мощной схемы управления памятью — сборки мусора с подсчетом ссылок.

Указатели только для чтения

Предположим, вы хотите сделать так, чтобы некоторый объект никогда не обновлялся (или, по крайней мере, не обновлялся обычными клиентами). Эта задача легко решается с помощью ведущих указателей — достаточно сделать операторную функцию operator->() константной функцией класса.

```
template <class Type>
class ROMP {
private:
    Type* t;
public:
    ROMP();                     // Создает указываемый объект
    ROMP(const ROMP<Type>&);   // Копирует указываемый объект
    ~ROMP();                    // Удаляет указываемый объект
    ROMP<Type>& operator=(const ROMP<Type>&);
    const Type* operator->() const;
};
```

Указываемый объект заперт так надежно, что до него не доберется даже ЦРУ. В принципе, то же самое можно было сделать с помощью более простых умных указателей, но ведущие указатели обеспечивают стопроцентную защиту, так как клиент никогда не получает прямого доступа к указываемому объекту.

Указатели для чтения/записи

Во многих ситуациях существует оптимальное представление объекта, которое действительно лишь для операций чтения. Если клиент хочет изменить объект, представление приходится изменять.

Это было бы легко сделать при наличии двух перегруженных версий оператора ->, одна из которых возвращает Foo*, а другая — const Foo*. К сожалению, разные возвращаемые типы не обеспечивают уникальности сигнатур, поэтому при попытке объявить два оператора -> компилятор от души посмеется. Программисту придется заранее вызвать функцию, которая осуществляет переход от одного представления к другому.

Одно из возможных применений этой схемы — распределенные объекты. Если копии объекта не обновляются локальными клиентами, они могут быть разбросаны по всей сети. Совсем другое дело — координация обновлений нескольких экземпляров. Можно установить правило, согласно которому допускается существование любого количества копий только для чтения, но лишь одна главная копия. Чтобы обновить объект, необходимо предварительно получить главную копию у ее текущего владельца. Конечно, приходится учитывать многие нюансы (в частности, процедуру смены владельца главной копии), однако правильное применение ведущих указателей позволяет реализовать эту концепцию на удивление просто и незаметно для клиентов.

Грани и другие мудрые указатели

Если переложить эту главу на музыку, она бы называлась «Вариации на тему умных указателей». В двух предыдущих главах я представил базовые концепции умного указателя — класса, заменяющего встроенные *-указатели, — и ведущего указателя, для которого существует однозначное соответствие с указываемым объектом. В этой главе мы продолжим эту тему и добавим в мелодию еще несколько гармоничных нот.

Интерфейсные указатели

Наверное, вы считали, что интерфейс класса полностью определяется объявлением класса, но в действительности любой класс может иметь несколько разных интерфейсов в зависимости от клиента.

- Класс и его друзья видят один интерфейс, включающий всех членов класса и всех защищенных и открытых членов его базовых классов.
- Производные классы видят только защищенных и открытых членов класса и его базовых классов.
- Все остальные клиенты видят только открытых членов класса и его базовых классов.
- Если указатель на объект преобразуется к указателю на его базовый класс, интерфейс ограничивается только открытыми членами базового класса.

Открытые, закрытые и защищенные члены; открытое и закрытое наследование; полиморфизм и дружба — все это лишь грубые синтаксические приближения более общей концепции дизайна: один объект может иметь много специализированных интерфейсов.

Дублирование интерфейса

Давайте посмотрим, можно ли обобщить эту концепцию с помощью еще более умных (назовем их «мудрыми») указателей (smarter pointers). Для начала нам придется на некоторое время покинуть своего старого друга, оператор ->. Одно из ограничений оператора -> заключается в следующем: чтобы использовать указатель, клиент также должен знать все об интерфейсе указываемого объекта.

```
class Foo {
// Интерфейсная часть, которую бы вам хотелось спрятать подальше
};
Ptr<Foo> pf(new Foo);
```

Хммм. Чтобы клиент мог пользоваться указателем, нам придется рассказать ему все что только можно об указываемом объекте Foo. Не хотелось бы. Ниже показан альтернативный вариант. Терпение — все не так страшно, как кажется на первый взгляд.

```
class Foo {
friend class Pfoo;
protected:
    Foo();
public:
```

```
    void DoSomething();
    void DoSomethingElse();
};
class PFoo {
private:
    Foo* foo;
public:
    PFoo() : foo(new Foo) {}
    PFoo(const PFoo& pf) : foo(new Foo(*(pf.foo))) {}
    ~PFoo() { delete Foo; }
    PFoo& operator=(const PFoo& pf)
    {
        if (this != &pf) {
            delete foo;
            foo = new Foo(*(pf.foo));
        }
        return *this;
    }
    void DoSomething() { foo->DoSomething(); }
    void DoSomethingElse() { foo->DoSomethingElse(); }
};
```

Произошло следующее: мы воспользовались удобными средствами копирования/вставки текста вашей среды программирования и продублировали в указателе интерфейс указываемого объекта. Чтобы не лениться и не взваливать всю тяжелую работу по делегированию на оператор **->**, мы решительно реализовали все функции класса так, что каждая из них перенаправляет вызов функции-прототипу указываемого объекта. Указатели, воспроизводящие интерфейс указываемого объекта, называются *интерфейсными указателями (interface pointers).*

Маскировка указываемого объекта

Поначалу кажется, что реально мы ничего не добились. Чтобы подставляемые функции работали, интерфейс класса Foo все равно должен находиться в файле .h перед объявлением класса PFoo. Тем не менее, смирившись с небольшими дополнительными вычислениями для наших указателей, мы получаем быстрый и ощутимый результат.

```
class Foo1;   // Все, что клиент видит и знает о Foo
class PFoo1 {
private:
    Foo1* foo;
public:
    PFoo1();
    PFoo1(const PFoo1& pf);
    ~PFoo();
    PFoo1& operator=(const PFoo1& pf);

    void DoSomething();
    void DoSomethingElse();
};
class Foo1 {
friend class PFoo1;
protected:
    Foo1();
```

```
public:
     void DoSomething();
     void DoSomethingElse();
};

PFoo1::PFoo1() : foo(new Foo1)
{}

PFoo1::PFoo(const PFoo1& pf) : foo(new Foo1(*(pf.foo)))
{}

PFoo1::~PFoo()
{
     delete foo;
}

PFoo1& PFoo1::operator=(const PFoo1& pf)
{
     if (this != &pf) {
          delete foo;
          foo = new Foo1(*(pf.foo));
     }
     return *this;
}

void PFoo1::DoSomething()
{
     foo->DoSomething();
}

void PFoo1::DoSomethingElse()
{
     foo->DoSomethingElse();
}

Foo1::Foo1()
{
}

void Foo1::DoSomething()
{
     cout << “Foo::DoSomething()” << endl;
}

void Foo1::DoSomethingElse()
{
     cout << “Foo::DoSomethingElse()” << endl;
}
```

Видите, что здесь происходит? Для клиента класс Foo перестает существовать. Для всех практических целей указатель *стал* самим объектом. С таким же успехом мы могли все переименовать, убрать P перед указателем и заменить имя Foo чем-нибудь более закрытым и загадочным. Единственное, что говорит о существовании второго класса, — предварительное объявление `class Foo;`.

Цена всего происходящего — вызов не подставляемых (noninline) функций в каждой функции класса указателя. Для некоторых немногочисленных приложений и классов даже эта малая цена может стать неприемлемой. В таких случаях существуют две альтернативы для повышения скорости: использование умных указателей на базе оператора -> и использование интерфейсных указателей с занесением объявления класса указываемого объекта в файл .h и отказом от всех преимуществ инкапсуляции. Как вы убедитесь в оставшейся части этой главы, второй вариант все же имеет некоторые достоинства.

Изменение интерфейса

Одно из преимуществ интерфейсных указателей — в том, что они фактически позволяют изменить интерфейс указываемого объекта без внесения изменений в его класс. Интерфейс, представленный интерфейсным указателем, находится полностью в вашем распоряжении; вы можете исключить из него некоторые функции указываемого объекта, изменить сигнатуры, добавить ваши собственные дополнительные функции и просто хорошо провести время, заново изобретая указываемый объект.

Грани

Многие библиотеки классов (особенно связанные с графическим интерфейсом пользователя) содержат большие классы из десятков, а то и сотен функций. Визуальная часть экрана в них называется по-разному — *видом, окном* или *панелью.* Ниже показан типичный набросок класса, который представляет это отображаемое нечто, как бы оно ни называлось.

```
class View {  // На практике будет производным от другого класса
protected:
    // Часть, предназначенная только для производных классов вида
public:
    // Функции конструирования и инициализации
    // Функции уничтожения и деактивизации
    // Общие функции объекта
    // Обработка событий
    // Функции отображения
    // Геометрия («где сработала мышь?»)
    // Управление иерархией видов
};
```

Каждый подраздел может насчитывать до нескольких десятков функций. Разбираться в этих функциях — все равно что блуждать в зеркальном лабиринте; куда бы вы ни повернулись, виднеются бесчисленные отражения одного и того же класса. Конечно, такой класс можно было бы организовать и более разумно — например, выделить каждое семейство интерфейсов в собственный базовый класс и затем объединить эти классы с помощью множественного наследования. Или построить комплекс из объектов, совместная работа которых основана на взаимном делегировании. Все эти варианты обладают своими недостатками.

- Пользователи должны помнить, как объединяются все фрагменты каждого составного класса. Вы уверены, что действительно хотите этого?
- Когда фрагменты объединяются в общий класс, от которого затем создаются производные классы, проблема возникает заново на другом уровне иерархии классов.
- Нелегко заставить один базовый класс правильно работать с другим, когда эти два класса не имеют ничего общего до их объединения в контексте некоторого составного класса.
- Проектирование больших комплексов взаимодействующих объектов — занятие не для слабонервных.

Можно ли разбить такой класс на составляющие, не прибегая к сложностям создания производных классов или делегирования? Конечно можно, если воспользоваться технологией мудрых указателей. Достаточно создать для одного объекта несколько указателей, каждый из которых отвечает за некоторый аспект деятельности этого объекта.

```
class ViewEvents {
private:
    View* view;
public:
    // Функции, относящиеся к обработке событий
};

class ViewDrawing {
private:
    View* view;
public:
    // Функции, относящиеся к графическому выводу
};
// И т.д.
```

Каждый из этих мудрых указателей воспроизводит интерфейс к некоторому подмножеству функций класса `View` и перенаправляет вызовы функциям-прототипам объекта вида. Сам объект вида может быть устроен как угодно: на основе одиночного и множественного наследования, делегирования в комплексе объектов или в форме одного гигантского конгломерата; клиентов это волновать не должно. Я называю такие интерфейсные указатели, ограничивающие клиента подмножеством полного интерфейса, *гранями (facets).*

Эта базовая идея укоренилась как минимум в одной коммерческой технологии — компонентной модели объекта (COM, Component Object Model) компании Microsoft, которая называет эти указатели *интерфейсами.* Один из мелких разработчиков, компания Quasar Knowledge Systems, предложила аналогичную идею для объектов SmallTalk и назвала такие указатели *комплексами (suites).* Как бы они ни назывались, этой идее суждено стать одной из важнейших идиом дизайна объектно-ориентированного программирования будущего, поскольку она обладает повышенной гибкостью и модульностью по сравнению с функциональным делением на основе наследования и делегирования.

При всей простоте концепции она будет правильно работать лишь в том случае, если позаботиться о многочисленных технических деталях. Давайте рассмотрим их одну за другой, не жалея времени.

Преобразование указываемого объекта в грань

Итак, вы хотите получить грань по имеющемуся указываемому объекту. Для этого существует много способов, однако наиболее соответствующий стилю C++ заключается в использовании операторов преобразования.

```
class View {
public:
    operator ViewEvents() { return new ViewEvents(this); }
    operator ViewDrawing() { return new ViewDrawing(this); }
};
```

Другой вариант — разрешить пользователю напрямую использовать конструкторы класса грани:

```
ViewEvents ve(aView);
```

Об их достоинствах и недостатках можно спорить долго, но лично я предпочитаю первый способ по причинам, о которых мы вскоре поговорим. Существует еще один способ, который тоже заслуживает упоминания — присвоить каждому типу грани уникальный идентификатор и затем создать единую, многоцелевую функцию генерации граней для всех классов. Такая функция получает идентификатор типа грани в качестве аргумента и возвращает грань, если она поддерживается объектом, и `NULL` - в

противном случае. Те из вас, кому приходилось пользоваться технологиями COM и OLE компании Microsoft, узнают знакомую функцию `QueryInterface`, поддерживаемую всеми объектами.

Кристаллы

Если у вас имеется одна грань и вы хотите получить другую грань того же объекта, наиболее прямолинейный подход также заключается во включении операторов преобразования в грань. Упрощенный подход выглядит так:

```
class ViewEvents {
private:
    View* view;
public:
    operator ViewDrawing() { return ViewDrawing(*view); }
    // И т.д. для других граней
};
```

В этом маленьком C++-изме работа поручается операторной функции `operator ViewDrawing()` целевого вида. При малом количестве граней такое решение вполне приемлемо. С ростом количества граней число операторов преобразования возрастает в квадратичной зависимости, поскольку каждая грань должна преобразовывать ко всем остальным. Следующая модификация возвращает задачу к порядку `n`, где `n` — количество граней. Продолжая свою откровенно слабую метафору, я называю объект, который собирает и выдает грани, *кристаллом (gemstone)*.

```
class View;
class ViewEvents;
class ViewDrawing;
class ViewGemstone {
private:
    View* view;
public:
    ViewGemstone(View* v) : view(v) {}
    bool operator!() { return view == NULL; }
    operator ViewEvents();
    operator ViewDrawing();
    // И т.д.
};

class ViewEvents {
friend class ViewGemstone;
private:
    View* view;
    ViewEvents(View* v) : view(v) {}
public:
    bool operator!() { return view == NULL; }
    operator ViewGemstone();
};

class ViewDrawing {
friend class ViewGemstone;
private:
    View* view;
    ViewDrawing(View* v) : view(v) {}
```

```
public:
     bool operator!() { return view == NULL; }
     operator ViewGemstone();
};
```

У нас есть один объект, кристалл, который умеет генерировать все грани; каждая грань, в свою очередь, знает, как найти кристалл. Кристалл является единственным объектом, который может создавать грани, так как последние имеют закрытые конструкторы и дружат с кристаллом. Концепция кристалла чрезвычайно гибка — он может быть самостоятельным объектом, абстрактным базовым классом объекта и даже одной из граней.

С первого взгляда кажется, что такое решение создает излишние неудобства для пользователя, которому приходится выполнять два последовательных преобразования типа. Наверное, кому-нибудь захочется сделать класс `ViewGemstone` базовым для всех остальных. Такой вариант возможен, но тогда исчезнут некоторые важные преимущества. Приведенная выше модель является абсолютно плоской; между гранями не существует отношений наследования. Благодаря этому возникает огромная степень свободы в реализации — для поддержания этих интерфейсов можно использовать наследование, делегирование и агрегирование (внедренные переменные класса). Все это с лихвой окупает одно лишнее преобразование типа.

Вариации на тему граней

Грани можно реализовать несколькими способами. В совокупности они образуют надмножество тех возможностей, которые в C++ поддерживаются с помощью наследования и переменных класса.

Грани — множества подфункций

Самая простая форма грани — та, которая предоставляет интерфейс к подмножеству функций указываемого объекта.

```
// В файле Pointee.h
class Pointee;
class Facet {
friend class PointeeGemstone;
private:
     Pointee* pointee;
     Facet(Pointee* p) : pointee(p) {}
public:
     void Fn1();
     int Fn2();
     void Fn17();
};

class PointeeGemstone {
private:
     Pointee* pointee;
public:
     PointeeGemstone(Pointee* p) : pointee(p) {}
     Operator Facet();
};

// В файле Pointee.cpp
class Pointee {
public:
     void Fn1();
```

```
        int Fn2();
        void Fn3();
        char Fn4();
        // И т.д.
        void Fn17();
    };
```

Здесь грань просто отбрасывает все функции, которые не входят в ее компетенцию. Клиент имеет дело с «объектом», который намного легче всего указываемого объекта, но за кулисами все равно прячется полный объект.

Грани — переменные класса

Грань может представлять собой интерфейсный указатель на переменную класса. Это позволяет многократно использовать грань в различных кристаллах или для организации интерфейса к отдельному экземпляру. Если указываемый объект имеет переменную класса `Bar`, грань может представлять собой простой интерфейсный указатель на `Bar`.

```
// В файле Pointee.h
class BarFacet {
private:
    Bar* bar;
public:
    BarFacet(Bar* b) : bar(b) {}
    // Интерфейсы к функциям класса Bar
};

class PointeeGemstone {
private:
    Pointee* p;
public:
    operator BarFacet();
    // И т.д.
};

// В файле Pointee.cpp
class Pointee {
friend class PointeeGemstone;
private:
    Bar bar;  // Внедренная переменная класса Pointee
public:
    // И т.д.
};

PointeeGemstone::operator BarFacet()
{
    return BarFacet(&p->Bar); // Грань переменной
}
```

Все прекрасно работает, если вам хватает относительно простых правил согласованности C++. Вероятно, в более общем случае стоит воспользоваться приемами, описанными далее, в разделе «Обеспечение согласованности». В частности, одна из проблем такого упрощенного подхода

заключается в том, что вы можете «перейти» от кристалла к грани BarFacet, но не сможете выполнить обратное преобразование по информации, доступной в грани.

Грани — базовые классы

Грани также могут использоваться для создания эквивалента встроенного преобразования типа от производного класса к базовому.

```
// В файле Pointee.h
class FooFacet {
private:
    Foo* foo;
public:
    FooFacet(Foo* f) : foo(f) {}
    // Интерфейсы к функциям класса foo
};

class PointeeGemstone {
private:
    Pointee* p;
public:
    operator FooFacet();
    // И т.д.
};

// В файле Pointee.cpp
class Pointee : public Foo {
friend class PointeeGemstone;
public:
    // И т.д.
};

PointeeGemstone::operator FooFacet()
{
    return FooFacet(p);    // Компилятор преобразует p к Foo*
}
```

Как и в случае с гранями-переменными, это может позволить вам многократно использовать одни и те же грани Foo для базовых классов, переменных или отдельных объектов, хотя для обеспечения более строгих правил согласованности, описанных ниже, потребуется более узкая специализация. Например, при таком подходе вы сможете выполнить преобразование от кристалла к грани FooFacet, но не сможете снова вернуться к кристаллу.

Грани — делегаты

Во всех трех описанных вариантах (грани в качестве подмножества интерфейсов, переменных класса и базовых классов) подразумевается объединение нескольких объектов в один. Возможно и другое решение — использовать сеть взаимодействующих объектов и иметь одну грань для каждого объекта в сети. Ситуация очень похожа на вариант с гранями-переменными, хотя адрес, на который указывает грань, не внедряется физически в указываемый объект как переменная класса.

```
// В файле Pointee.h
class BarFacet {
private:
    Bar* bar;
```

```
public:
     BarFacet(Bar* b) : bar(b) {}
     // Интерфейсы к функциям класса Bar
};

class PointeeGemstone {
private:
     Pointee* p;
public:
     operator BarFacet();
     // И т.д.
};

// В файле Pointee.cpp
class Pointee {
friend class PointeeGemstone;
private:
     Bar* bar;     // Уже не внедренная переменная класса
public:
     // И т.д.
};

PointeeGemstone::operator BarFacet()
{
     return BarFacet(&p->Bar);
}
```

Такое решение страдает недостаточной согласованностью, как и решение с гранями-переменными.

Комбинации и вариации

Если на вас накатит особенно творческое настроение, можно создать грани, в которых используются комбинации этих четырех подходов. Например, одна грань может включать подмножество интерфейсных функций указываемого объекта; одну интерфейсную функцию, делегирующую переменной класса; другую, делегирующую базовому классу; и третью, работающую с делегатом указываемого объекта. На Капитолийском холме такая ситуация была впервые представлена в Акте о Всестороннем Применении Идиом C++ от 1995 года.

Инкапсуляция указываемого объекта

Раз вся эта бурная деятельность сконцентрирована вокруг объекта `View`, то как убрать его подальше от глаз широкой публики? Собственно, это считалось одним из достоинств интерфейсных указателей, не правда ли? Еще раз посмотрите на кристалл вида. Так ли уж необходимо пользователю видеть указываемый объект при использовании этой стратегии? В действительности ему хватит возможности создавать кристаллы, которые, в свою очередь, будут использоваться для получения необходимых граней. Когда это будет сделано, грани и кристаллы можно поместить в файл .h и скрыть указываемый объект в файле .cpp.

Такой вариант особенно хорошо работает в сочетании с другими идиомами. С переходными типами указываемые объекты можно менять во время выполнения программы. Гомоморфные иерархии классов позволяют использовать один набор граней для организации интерфейса к деревьям семейств производных или перекомпонованных (recomposed) классов указываемых объектов. Ведущие указатели помогают соблюсти некоторые из правил согласованности, о которых будет рассказано позже.

Проверка граней

Давайте еще немного расширим эту идиому и предположим, что некоторая грань может поддерживаться или не поддерживаться объектом. О том, почему и как это может произойти, мы поговорим во время изучения переходных типов далее в этой главе, а пока ответим на вопрос: как клиент может определить, поддерживается ли некоторый объект некоторой гранью? Решение построено на другом C++-изме — перегрузке оператора !.

```
class ViewEvents {
private:
    View* view;
public:
    ViewEvents(View* v) : view(v) {}
    bool operator!() { return view == NULL; }
    // И т.д.
};

// В клиентском коде
ViewEvents ve(aViewGemstone);
if (!ve) {
    cerr << "Вид не обрабатывает событий!" << endl;
    throw interface_error;
}
```

Предполагается, что aViewGemstone относится к кристаллам вида, о которых говорилось выше, и содержит операторную функцию operator ViewEvents(). Ключевая строка ViewEvents ve(aViewGenistone) работает в два этапа: сначала вызывается операторная функция operator ViewEvents(), выполняющая преобразование, а затем ve конструируется с помощью конструктора копий ViewEvents. Если кристалл вида решает, что данный вид не обрабатывает событий, он может сконструировать экземпляр ViewEvents с view, равным NULL. Затем оператор ! проверяет возвращенную грань. Функции грани также могут проверять наличие объекта перед тем, как пытаться что-то делегировать ему.

```
Void ViewEvents::DoSomething()
{
    if (!*this)   // то есть if (view == NULL)
        // Инициировать исключение
    view->DoSomething();
}
```

Обеспечение согласованности

В одних отношениях C++ непроницаем, как гранитная скала, в других — дыряв, как решето. Одна из таких дыр — несогласованный подход к преобразованию типов. Вы можете получить адрес переменной класса, но по этому адресу нельзя безопасно перейти к объекту, которому принадлежит переменная. У компилятора хватает ума для автоматического приведения указателя на производный класс к указателю на базовый, но вернуться к указателю на производный класс он уже не сможет. Придется выполнять небезопасное явное преобразование, пользующееся дурной славой. Большинство программистов и проектировщиков лишь испускает глубокий вздох, обычно приберегаемый для супружеских ссор, и живет с этими несоответствиями. Однако благодаря идиоме граней открывается уникальная возможность «исправить» этот аспект C++. Существуют три свойства, которые в совокупности обеспечивают отсутствующую в C++ степень согласованности. В следующем описании выражение a=>b означает, что грани типа A содержат оператор преобразования, дающий грань типа B; a и b являются конкретными экземплярами типов A и B соответственно. Хотя эти правила относятся к специфике C++ и граней, они также являются частью стандартной дисциплины проектирования, используемой в технологии COM компании Microsoft:

1. *Симметричность.* Если a=>b, то b=>a. Последствия: грань — переменная класса должна уметь восстанавливать вмещающий объект, из которого она получена, а грань — базовый класс должна уметь безопасно «возвращаться» к правильному производному классу.
2. *Транзитивность.* Если a=>b, a=>c и b=>c2, то c и c2 должны быть эквивалентны; точнее, они должны быть гранями одного объекта, обеспечивающими идентичные наборы функциональных средств. При этом физически они могут быть разными гранями.
3. *Устойчивость граней.* Пока грань существует, она должна работать корректно. Иначе говоря, при удалении указываемого объекта все ссылающиеся на него грани после выполнения оператора ! начинают возвращать `false` и инициировать исключения при вызове функций. Это положение можно было бы сформулировать и более строго — указываемый объект не должен уничтожаться до тех пор, пока для него существует хотя бы одна грань. В частности, это правило используется в технологии COM.

Первые два правила выполнить несложно, хотя за это нередко приходится расплачиваться объемом и быстродействием. Посидите пару вечеров, набейте живот жирной пиццей, и вы наверняка сами придумаете, как обеспечить выполнение этих двух правил. Некоторые рекомендации приведены ниже. Устойчивость граней — совсем другое дело; мы констатируем проблему сейчас, но откладываем решение до следующих глав.

Симметричность

Самый простой способ выполнить это требование — включить в класс грани две переменные. В одной хранить адрес составного объекта, используемого гранью, а в другой — адрес кристалла или исходного указываемого объекта.

```
class BarFacet {
private:
    Bar* bar;
    PointeeGemstone* gemstone;
public:
    BarFacet(Bar* b, PointeeGemstone* agg) : bar(b), gemstone(agg) {}
    operator PointeeGemstone() { return *gemstone; }
    // Интерфейсы к функциям Bar
};
```

Наличие обратного указателя обеспечивает симметричность; по кристаллу можно .получить грань, а по грани — кристалл. Основная проблема состоит в том, что `BarFacet` не годится для повторного использования, если Bar является компонентом различных кристаллов; для каждого использования приходится создавать отдельную грань.

Транзитивность

На самом деле соблюдение этого принципа зависит от дисциплины проектирования. Транзитивность нарушается в ситуациях наподобие следующей:

```
class Foo {
private:
    Blob* blob;
    Bar* bar;
};

class Bar {
private:
    Blob* another_blob;
};
```

Если обратиться к классу Foo с запросом на получение Blob, он вернет вам то, на что ссылается его собственная переменная. Если запросить Bar, а затем у Bar запросить Blob, вы получите другой объект — тот, на который ссылается Bar.

Такая ситуация обычно встречается при множественном наследовании и многократном (хотя и косвенном) наследовании от одного базового класса. Если переработать приведенный выше фрагмент, ту же проблему можно легко воссоздать с применением наследования:

```
class Blob { ... };
class Foo : public Blob { ... };
class Bar : public Foo, public Blob { ... };
```

В этом фрагменте заложена неоднозначность. При преобразовании типа Bar* в Blob* непонятно, какой же Blob имеется в виду: непосредственно базовый для Bar или тот, от которого происходит Foo? Даже если вам удастся справиться с неоднозначностью, транзитивность все равно нарушается.

Как же избежать подобных проблем? Я мог бы пуститься в замысловатые объяснения, но все сводится к одному: следите за собой. Не наследуйте *открыто* от одного класса несколько раз в иерархическом дереве некоторого класса (с закрытым наследованием проблем обычно не возникает, поскольку нельзя выполнить преобразование к типу закрытого базового класса). Не внедряйте экземпляры одного класса в качестве переменных более чем одного компонента, а если уж вы это сделали — не предоставляйте интерфейсы более чем к одному из них.

Устойчивость граней

В эталонном C++ эта проблема возникает, например, при получении адреса переменной или базового класса и последующем удалении вмещающего объекта. Указатель на переменную или базовый класс превращается в мусор. Проблема обеспечения согласованности в такой неразберихе не решается за обедом в течение одного дня. Усиленная формулировка (указываемые объекты должны существовать до тех пор, пока для них существуют грани) откладывается до той части книги, которая посвящена управлению памятью. Слабая формулировка легко обеспечивается с помощью ведущих указателей. Если грань работает с объектом через ведущий указатель, то у ведущего указателя можно спросить: «А указываешь ли ты сейчас на существующий объект?» Если ответ будет отрицательным, оператор ! возвращает true, и функции начинают инициировать исключения. Правда, остается другая проблема — когда и как удалять ведущие указатели. Ее также лучше рассматривать при обсуждении различных стратегий управления памятью.

Грани и ведущие указатели

Концепции граней и кристаллов хорошо сочетаются с концепцией ведущих указателей. Существуют два основных подхода.

Ведущий указатель в середине

Традиционный ведущий указатель (вероятно, с использованием оператора ->) вставляется между указываемым объектом и гранью. В этом случае грани превращаются в разновидность дескрипторов, косвенно обращающихся к членам указываемого объекта через ведущий указатель. Ведущий указатель должен уметь хотя бы генерировать кристалл, который затем будет использоваться клиентами для генерации других граней. Если весь доступ к объекту осуществляется через грани, подумайте о том, чтобы сделать оператор -> ведущего указателя закрытым, а грани объявить друзьями.

Превращение кристалла в ведущий указатель

Кристаллу присваиваются функции ведущего указателя. Затем кристалл генерирует грани и поддерживает семантику ведущего указателя для конструирования и уничтожения. Грани получают доступ к указываемому объекту *через кристалл.* Чтобы вам было удобнее, можно предоставить закрытый оператор -> в кристалле и сделать грани друзьями кристалла. Грани фактически превращаются в дескрипторы и получают доступ к указываемому объекту косвенно, через кристалл.

Переходные типы

Основная идея переходного типа (malleable type) - класс, экземпляры которого как бы изменяют свой тип во время выполнения программы. Конечно, формально в C++ такого происходить не может — иначе вам пришлось бы всерьез и надолго подружиться с отладчиком. Тем не менее, чудеса современных указателей позволяют добиться почти того же эффекта.

Полиморфные указываемые объекты

В базовом варианте ведущего указателя присутствует переменная с типом `Pointee*`. Но кто сказал, что объект, на который она ссылается, должен быть настоящим `Pointee`, а не каким-нибудь классом, производным от `Pointee`?

```
// В файле Foo.h
class Foo {
protected:
    Foo();
public:
    // Члены Foo
};

class Pfoo {  // Ведущий указатель
private:
    Foo* foo;
public:
    PFoo();
    Foo* operator->() const { return foo; }
    // Остальные члены, характерные для ведущих указателей
};

// В файле Foo.cpp
class DerivedFromFoo : public Foo {
private:
    // Закрытые члены производного класса
public:
    DerivedFromFoo();  // Функция открытая, но спрятанная в файле .cpp
    // Переопределения функций класса Foo
};

PFoo::PFoo() : foo(new DrivedFromFoo)
{
}
```

Ловкость рук и никакого мошенничества! Ведущий указатель подсунул вместо `Foo` нечто совершенно иное, а клиенты ничего не замечают. Ага! Теперь вы понимаете, почему конструктор `Foo` объявлялся защищенным, а не закрытым! Класс `PFoo` уже можно не объявлять другом; доступ к конструкторам `Foo` нужен только конструктору `DerivedFromFoo`.

В части 3 мы поговорим о том, какое наследование нужно, чтобы эта схема работала (а именно, чтобы все классы, производные от `Foo`, имели тот же открытый интерфейс, что и сам `Foo`). А пока продолжим изучение указателей и всего, что с ними связано.

Выбор типа указываемого объекта во время конструирования

Если наш ведущий указатель может создать объект производного класса во время конструирования, почему бы не разрешить ему свободно выбрать нужный тип из нескольких производных классов?

```
// В файле Foo.cpp
class DeirvedFromFoo : public Foo { ... };
class AlsoDerivedFromFoo : public Foo { ... };
PFoo::PFoo(bool x) : foo(x ? new DerivedFromFoo : new AlsoDerivedFromFoo) {}
```

Вообще говоря, интерфейсный указатель может выбрать любой производный класс на основании сведений, доступных во время конструирования. Клиент об этом ничего не знает, поскольку все происходящее скрывается за интерфейсным указателем.

Изменение указываемого объекта во время выполнения программы

При желании интерфейсный указатель может сменить указываемый объект прямо во время выполнения программы.

```
class Foo;
class PFoo {
private:
     Foo* foo;
public:
     PFoo();
     void DoSomething(bool x);
     // Другие функции класса
};

void PFoo::DoSomething(bool x)
{
     if (x) {
          delete foo;
          foo = new DerivedFromFoo;
     }
     Foo->DoSomething();
}
```

Пример уже встречался в предыдущей главе: при попытке неконстантного обращения к указываемому объекту указатель выбирал другую форму этого объекта. Такой подход работает вполне нормально, если не делать глупостей (например, получать адреса членов указываемого объекта).

Посредники

Интерфейсные указатели также помогают скрыть тот факт, что указываемый объект находится где-то в киберпространстве, а не сидит в памяти по соседству. В распределенных объектных системах такое происходит сплошь и рядом. Первая цель проектировщика — держать клиентские объекты в блаженном неведении; они не знают и знать не хотят, где находится указываемый объект — на расстоянии плевка или где-то на краю земли. Объект, который заменяет другой, удаленный объект, называется *посредником (proxy)*. На эту тему тоже существует немало вариаций, но самая простая из них — локальное использование интерфейсного объекта или грани. Затем локальный посредник может воспользоваться дистанционными вызовами или другим механизмом отправки сообщений, подходящим для взаимодействия с оригиналом.

Эта концепция распространяется и на ситуации, в которых удаленный «объект» вообще не является объектом. Это может быть целое приложение, завернутое посредником в объектно-ориентированную оболочку, или, допустим, содержимое базы данных с библиотечными функциями «класса».

В самой идее внедрения «не объектно-ориентированного» кода в объекты C++ нет ничего нового или оригинального. Нас в первую очередь интересует уровень инкапсуляции. Что должен знать клиент о реальной ситуации? Умные указатели на основе операторов -> подходят плохо. Клиент должен знать интерфейс указываемого объекта; следовательно, он должен знать, существует ли указываемый объект, как устроен его интерфейс и т. д. Интерфейсные указатели, в том числе грани — более удачный вариант. Если ваша программа написана с применением интерфейсных указателей, вам будет намного проще вставить новый код, в котором некоторые из этих указателей реализуются в виде посредников. Проще, хотя и не совсем незаметно для клиента — пока. Помните базовую форму интерфейсного указателя с обязательным предварительным объявлением?

```
class Pointee;     // Предварительное объявление
class Interface {
private:
     Pointee* pointee;
public:
     // Функции класса
};
```

Проблема кроется в переменной `pointee`. Клиент должен знать, что указатель ссылается на *нечто,* даже если он понятия не имеет, на что именно. В части 3 мы попробуем устранить даже это ограничение, а пока будем считать его досадной мелочью.

В результате мы приходим к классическому компромиссу: понижение быстродействия интерфейсных указателей (с вынесенными (outline) функциями) за возможность кардинальных изменений реализации без модификации клиентского кода. В большинстве проектов и классов расходы с лихвой компенсируются ускорением цикла разработки.

Функторы

Напоследок мы познакомимся с одной диковинкой C++, которая называется *функтором (functor).* Функторы играют для функций ту же роль, что и интерфейсные указатели для объектов. Одна из проблем, вечно мучивших программистов на С — то, что все функции находятся в глобальном пространстве имен, то есть вызванная функция имеет доступ только к данным, хранящимся в ее аргументах, и глобальным переменным. Если передать адрес функции еще кому-то, то при вызове функции по адресу она не будет помнить, как выглядел окружающий мир во время получения ее адреса.

В таких языках, как Паскаль, эта проблема изящно решается получением *замыкания (closure)* на момент получения адреса функции.

```
procedure p(n: integer);
var
     procedure fn;
     begin
          do_something(n);
     end;
begin
     callback(@fn);
end;
```

В качестве аргумента процедура `callbackfn` получает адрес другой процедуры. В данном примере ей передается адрес `fn`. При вызове `fn` из `callbackfn` первая *имеет доступ к переменным, находившимся в стеке в момент получения адреса.* В нашем примере `fn` знает значение переменной `n` на момент вызова `callbackfn`.

Замыкания чрезвычайно полезны для обработки обратных вызовов (callback), поскольку функция обратного вызова кое-что знает о том, почему она была вызвана. В С вложенных функций не существует, а следовательно, замыкания невозможны — их место занимают функторы.

```
class Fn {
private:
    int number;
public:
    f(int n) : number(n) {}
    void operator() () { do_something(number); }
};

void callbackfn(Fn);

void p(int n)
{
    callbackfn(Fn(n));
}

void callbackfn(Fn fn)
{
    // Что-то делаем
    fn();    // Вызвать «функцию» fn с помощью функции operator()
}
```

Весь секрет кроется в двух выражениях. Функция `callbackfn(Fn(n))` передает функции анонимный экземпляр класса `Fn`. Аргумент его конструктора содержит информацию, включаемую в «псевдозамыкание», которое поддерживается переменными класса `Fn`. Выражение `fn();` может показаться обычным вызовом функции, но на самом деле в нем вызывается операторная функция `operator()` класса `Fn`. В свою очередь, эта функция вызывает глобальную функцию `do_something` с использованием данных замыкания. И кому после этого нужен Паскаль?

Операторная функция `operator()` может вызываться с произвольным набором аргументов. Чтобы добавить новые аргументы, укажите их во вторых скобках в объявлении класса. Также разрешается многократная перегрузка оператора `()` с разными сигнатурами. Ниже приведен тот же пример, в котором одна из версий операторной функции `operator()` вызывается с аргументом.

```
class Fn {
private:
    int number;
public:
    f(int n) : number(n) {}
    void operator() () { do_something(number); }
    void operator() (char* s)
    {
        do_something(number);
        cout << "Что-то делаю с " << s << endl;
    }
};

void callbackfn(Fn);

void p(int n)
{
    callbackfn(Fn(n));
}
```

```
void callbackfn(Fn fn)
{
    // Что-то делаем
    fn("callbackfn");
}
```

Эта маленькая идиома выглядит довольно изящно, однако того же эффекта можно добиться и без оператора ().

```
class Fn {
private:
    int number;
public:
    f(int n) : number(n) {}
    void do_something() () { ::do_something(number); }
    void do_something() (char* s)
    {
        do_something(number);
        cout << "Что-то делаю с " << s << endl;
    }
};

void callbackfn(Fn);

void p(int n)
{
    callbackfn(Fn(n));
}

void callbackfn(Fn fn)
{
    // Что-то делаем
    fn.do_something("callbackfn");
}
```

Как видите, с таким же успехом можно воспользоваться именем любой функции класса. Единственная причина для использования оператора () — в том, что он предельно ясно выражает ваши намерения. Если класс существует лишь для того, чтобы обслуживать обратные вызовы подобного рода, пользуйтесь оператором (); в противном случае пользуйтесь обычными функциями класса.

Коллекции, курсоры и итераторы

8

Проблема проектирования и реализации классов-коллекций (совокупностей объектов, находящихся под управлением другого объекта) стара, как само объектно-ориентированное программирование. Нет смысла повторять здесь все, что можно прочитать в других книгах о всевозможных коллекциях — изменяемых, сериализуемых, индексируемых и т. д. Если вам нужна информация о структурах данных и сопутствующих трансформациях, начните с изучения классов-коллекций SmallTalk и затем переходите к коммерческим библиотекам классов C++, рекламой которых забиты все журналы по программному обеспечению. Я же собираюсь сосредоточить ваше внимание на тех C++-измах, благодаря которым коллекции укладываются в мистическое учение программирования на C++ независимо от используемых структур данных и иерархий классов.

В начале этой главы рассматриваются индексируемые коллекции — то есть те, в которых некоторый объект используется в качестве индекса для получения другого объекта, скрытого глубоко в недрах коллекции. Мы воспользуемся оператором [], чтобы коллекция выглядела как абстрактный массив. Дело даже не в том, что программа от этого обычно становится более понятной — в этом подходе задействованы многие идиомы, используемые для коллекций. После знакомства с курсорами и их собратьями итераторами мы перейдем к изощренным коллекциям, модифицируемым в процессе перебора.

Массивы и оператор []

Оператор [] чрезвычайно гибок и универсален. К сожалению, большинство программистов C++ об этом и не подозревает.

Проверка границ и присваивание

Ниже приведен простой пример перегрузки оператора [] — массив, который при обращении к элементу проверяет, принадлежит ли индекс границам массива, и в любых ситуациях ведет себя более или менее разумно (или по крайней мере безопасно).

```
class ArrayOfFoo {
private:
     int entries;
     Foo** contents;          // Вектор Foo*
     static Foo* dummy;       // Для индексов, выходящих за границы массива
public:
     ArrayOfFoo() : entries(0), contents(NULL) {};
     ArrayOfFoo(int size) : entries(size), contents(new Foo*[size]) {};
     ~ArrayOfFoo() { delete contents; }
     Foo*& operator[](int index)
```

```
    {
        return (index < 0 || index >= entries) ? dummy : contents[index];
    }
};

// Где-то в файле .cpp
Foo* ArrayOfFoo::dummy = NULL;
```

Оператор `[]` возвращает `Foo*&`, ссылку на адрес `Foo`. Эта идиома часто встречается при работе с коллекциями, и одна из причин — в том, что возвращаемое значение может использоваться как в левой, так и в правой части выражения присваивания.

```
Foo* foo = array[17];
array[29] = foo;   // Работает – можно присваивать по конкретному индексу
```

Если бы оператор `[]` возвращал просто `Foo*`, то содержимое элемента массива копировалось бы, а копия возвращалась вызывающей стороне. Возвращая `Foo*&`, мы позволяем вызывающей стороне изменить содержимое элемента, а не только прочитать хранящееся в нем значение. Для индекса, выходящего за границы массива, возвращается адрес фиксированной переменной класса, значение которой на самом деле нас не очень интересует. По крайней мере, ваша программа сможет хромать дальше (возможно, при правильно расставленных `#ifdef` в отладочном режиме), а не извлекать из памяти номер телефона вашей тетушки или другую случайную информацию.

Если вы обеспокоены накладными расходами такого варианта по сравнению с обычными массивами C/C++, заключите все дополнительные вычисления и переменные между директивами `#ifdef`. В одном варианте компиляции ваш массив будет абсолютно безопасным, а в другом будет иметь те же размер и быстродействие, что и обычный массив.

Оператор [] с нецелыми аргументами

Оператор `[]` перегружается для аргументов любого типа, а не только для целых. Тогда оператор `[]` можно использовать для представления словаря — коллекции, в которой один ключевой объект однозначно идентифицирует другой. Ниже приведен набросок ассоциативного класса, в котором хранятся пары строковых объектов `String`, при этом первая строка каждой пары является индексом второй.

```
class Association {
// Пропускаем подробности реализации
public:
    const String& operator[](const String& key);
};

// В клиентской программе
String str = assotiation[another_string];
```

Такой вариант выглядит намного элегантнее и лучше выражает намерения разработчика, нежели интерфейс, построенный только на функциях класса:

```
String str = assotiation.LookUp(another_string);
```

Имитация многомерных массивов

В любом варианте перегрузки оператор `[]` вызывается с одним аргументом произвольного типа. Например, компилятор лишь посмеется над следующей попыткой создания многомерного массива, потому что в ней оператор `[]` имеет несколько аргументов:

```
class WontWork {
public:
    Foo& operator[](int x, int y);  // Ха-ха-ха
```

```
};
```

Компиляторы обожают подобные ситуации — вроде бы все выглядит вполне логично, но как-то выпало из спецификации языка. Это дает им возможность поразмяться и вывалить все туманные сообщения об ошибках, которые они приберегли на черный день. Но когда сообщения перестанут сыпаться, наступает ваш черед смеяться, поскольку существует простой обходной путь.

```
struct Index {
     int x;

     int y;
     Index(int ex, int why) : x(ex), y(why) {}
     bool operator==(const Index& i) { return x == i.x && y == i.y; }
};

class WorksFine {
public:
     Foo& operator[](Index i);
};

array[Index(17, 29)].MemberOfFoo();       // Работает
```

`Index` представляет собой структуру с тривиальным конструктором. Причина перегрузки оператора == станет ясна позже. Выражение `Index(17,29)` создает анонимный экземпляр, который упаковывает два измерения массива в один аргумент. Правда здорово? Получай, компилятор!

Множественные перегрузки оператора []

Оператор `[]` может иметь и несколько вариантов перегрузки для данного класса при условии, что сигнатуры остаются уникальными. Например, одна версия может получать аргумент типа `int`, а другая — аргумент `char*`, который преобразуется к `int` функцией `atoi()`. Скорее всего, ваша коллекция может индексироваться несколькими способами.

```
class StringArray {
public:
     const String& operator[](int index);
     int operator[](const String&);
};

String str = array[17];              // Первая форма
int index = array[String("Hello")];  // Вторая форма
```

Первый оператор `[]` реализует семантику массива: по целому индексу возвращается значение элемента с этим индексом. Второй оператор обеспечивает обратную возможность: по значению находится соответствующий индекс массива. В этой схеме используется пара допущений (например, уникальное целое, которое возвращается в качестве индекса несуществующего значения), но в целом идея вполне понятна.

Виртуальный оператор []

Оператор `[]`, как и любой другой оператор или функцию, можно объявить виртуальным и переопределить в производных классах. В некотором базовом классе определяется абстрактный интерфейс, а все подробности реализации предоставляются в производных классах. Такая схема хорошо сочетается с гомоморфными иерархиями классов, описанными в части 3.

Курсоры

В предыдущем разделе мы говорили о присваивании элементам массива. Для массива Foo* все прекрасно работало, но попытка присвоить что-нибудь «элементу» строковой ассоциации кончается неудачей.

```
association[String("Hello")] = String("Good looking");
```

Дело в том, что левая часть не является ни левосторонним выражением (lvalue), ни классом с перегруженным оператором =. В этом случае можно сконструировать аргумент с использованием интерфейса вставки в коллекцию на базе функций класса, поскольку это все-таки не *настоящий* массив, а нечто загримированное под него с помощью оператора []. Многие классы, перегружающие оператор [], с точки зрения семантики *являются* массивами, но используют хитроумные структуры данных для оптимизации. Давайте рассмотрим конкретный пример (разреженные массивы), а затем вернемся к более общим коллекциям (таким как ассоциации).

Простой класс разреженного массива

Разреженный массив относится к числу основных структур данных. Он представляет собой матрицу, у которой большинство ячеек в любой момент времени остается пустым. Возможно, вы принадлежите к числу счастливчиков с 256 гигабайтами памяти на компьютере, но большинству из нас просто не хватит места для хранения всех ячеек матрицы 1000x1000x1000. Да и не хочется выделять память под миллиард ячеек, если в любой момент из них используется не более 1000. Несомненно, в вашем мозгу всплывают различные структуры данных, знакомые по начальному курсу программирования в колледже: связанные списки, бинарные деревья, хеш-таблицы и все прочее, что упоминает Кнут. На самом деле не так уж важно, какая структура данных лучше подойдет для низкоуровневой реализации. Прежде всего необходимо понять, как же использовать эти низкоуровневые средства и одновременно создать для клиентских объектов впечатление, что они имеют дело с самым обычным массивом?

В следующей реализации «методом грубой силы» для хранения данных используются связанные списки. Структура Index уже встречалась нам выше.

```
class SparseArray {
private:
    struct Node {
        Index index;        // Индекс массива
        Foo* content;       // Содержимое массива по данному индексу
        Node* next;         // Следующий элемент списка
        Node(Index i, Foo* f, Node* n) : index(i), content(f), next(n) {};
    };
    Node* cells;  // Связанный список элементов
public:
    SparseArray() : cells(NULL) {}
    Foo* operator[](Index i);
};

inline Foo* SparseArray::operator[](Index i)
{
    SimpleSparseArray::Node* n = cells;
    while (n != NULL) {
        if (n->index == i)   // Использует перегруженный оператор ==
            return n->content;
        n = n->next;
    }
    return NULL;
}
```

```
Foo* foo = array[Index(17, 29)];     // Работает
```

С чтением массива проблем нет. Если индекс существует, возвращается содержимое массива по данному индексу. Если индекс в массиве отсутствует, значение `NULL` полностью соответствует идее предварительной инициализации массива значениями `NULL`. Минутку, но как добавить в массив новую ячейку или изменить уже существующую? Значение, возвращаемое операторной функцией `operator[]`, не является ни левосторонним выражением (lvalue), ни классом с перегруженным оператором = и по нему нельзя выполнить присваивание.

```
array[Index(31, 37)] = foo;    // Не работает
```

Ваш компилятор не спит ночами и ждет, когда же у него появится такая замечательная возможность забить поток `cerr` сообщениями об ошибках. Можно было бы создать интерфейс на базе функций, но тогда у клиента нарушится иллюзия того, что он имеет дело с нормальным, честным массивом. Существует ли способ использовать оператор `[]` в левой части операции присваивания для индексов, которых еще нет? Оказывается, существует, но для этой цели нам потребуется новая идиома — курсор.

Курсоры и разреженные массивы

Итак, вторая попытка. Наша основная цель — чтобы операторная функция `operator[]` возвращала нечто, обладающее следующими свойствами:

1. Оно должно преобразовываться к типу содержимого массива.
2. Оно может использоваться в левой части операции присваивания для изменения содержимого соответствующей ячейки.

Это «нечто» представляет собой особый класс, который называется *курсором (cursor)*. Ниже показан уже знакомый разреженный массив с курсором в операторной функции `operator[]`:

```
class ArrayCursor;
class SparseArray {
friend class ArrayCursor;
private:
    struct Node {
        Index index;
        Foo* content;
        Node* next;
        Node(Index i, Foo* c, Node* n) : index(i), content(c), next(n) {};
    };
    Node* cells;
public:
    SparseArray() : cells(NULL) {}
    ArrayCursor operator[](Index i);
};

class ArrayCursor {
friend class SparseArray;
private:
    SparseArray& array;       // Обратный указатель на массив-владелец
    Index index;              // Элемент, представленный курсором
    SparseArray::Node* node;  // Если существует индекс, отличный от NULL
    // Конструкторы объявлены закрытыми, поэтому пользоваться ими
    // может только SparseArray. Первый конструктор используется, когда
    // индекс еще не существует, а второй - когда индекс уже присутствует
    // в массиве.
```

```
        ArrayCursor(SparseArray& arr, Index i)
            : array(arr), index(i), node(NULL) {}
        ArrayCursor(SparseArray& arr, SparseArray::Node* n)
            : array(arr), node(n), index(n->index) {}
    public:
        // Следующий оператор = позволяет преобразовать присваивание курсору в
        // присваивание соответствующему элементу массива.
        ArrayCursor& operator=(Foo* foo);
    };

    ArrayCursor& ArrayCursor::operator=(Foo* foo) {
        if (node == NULL) {     // Индекс не существует
            node = new SparseArray::Node(index, foo, array.cells);
            array.cells = node;
        }
        else
            // Индекс уже существует, изменить значение элемента
            node->content = foo;
        return *this;
    }

    ArrayCursor SparseArray::operator[](Index i)
    {
        SparseArray::Node* n = cells;
        while (n != NULL)
            if (n->index = i)
                return ArrayCursor(*this, n);     // Существует
            else
                n = n->next;
        return ArrayCursor(*this, i);   // Еще не существует
    }
```

Ого! Что же происходит в этом хитроумном коде? Все волшебство заключено в двух операторных функциях, `SparseArray::operator[]()` и `ArrayCursor::operator=()`. `SparseArray::operator[]()` возвращает `ArrayCursor` независимо от того, существует индекс или нет (об этом `ArrayCursor` узнает по тому, какой конструктор был выбран). `ArrayCursor::operator=(Foo*)` делает одно из двух: если индекс уже существует, элемент изменяется, а *если не существует — он динамически добавляется в массив.* В этом проявляется вся суть курсорности (курсоризма?): перегруженный оператор = выполняет присваивание не для самого курсора, а для структуры данных, от которой происходит курсор. Теперь присваивание работает независимо от того, существует индекс или нет.

```
    array[Index(17, 29)] = new Foo;     // Добавляет индекс
    array[Index(17, 29)] = new Foo;     // Изменяет значение с заданным индексом
```

Неплохо для часовой работенки, не правда ли? Наш массив работает совсем как настоящий. Почти.

Операторы преобразования и оператор ->

Осталось добавить еще пару штрихов. Во-первых, оператор `[]` в правой части операции присваивания работает уже не так, как было написано, поскольку он возвращает `ArrayCursor`, а не `Foo*` или `Foo*&`. Но причин для беспокойства нет, потому что `Foo*()` в случае необходимости автоматически преобразует `ArrayCursor` к `Foo*`. Вторая проблема заключается в том, что оператор `[]` не может использоваться слева от оператора ->; на помощь приходит `operator->()`!

```
class ArrayCursor {
friend class SparseArray;
private:
     SparseArray& array;
     Index index;
     SparseArray::Node* node;
     ArrayCursor(SparseArray& arr, Index i)
          : array(arr), index(i), node(NULL) {}
     ArrayCursor(SparseArray& arr, SparseArray::Node* n)
          : array(arr), node(n), index(n->index) {}
public:
     ArrayCursor& operator=(Foo* foo);
     operator Foo*() { return node != NULL ? node->content : NULL; };
     Foo* operator->()
     {
          if (node = NULL)
               // Инициировать исключение
          else
               return node->contents;
     }
};

Foo* foo = array[Index(17, 29)];          // Работает
array[Index(17, 29)]->MemberOfFoo();      // Тоже работает
```

Ну вот, теперь можно расслабиться. Для клиентского объекта наш массив ничем не отличается от обычного. Это означает, что вы можете программировать для простой семантики массива и отдельно выбрать внутреннюю реализацию структур данных даже на поздней стадии проекта.

Что-то знакомое…

Взгляните еще раз на класс `ArrayCursor`. Он представляет собой объект, который косвенно ссылается на `Foo`, имеет операторную функцию `operator Foo*()` и перегруженный оператор **->**, позволяющий обращаться к членам `Foo` через курсор. Выглядит знакомо? Так и должно быть. Курсоры на самом деле представляют собой следующее поколение умных указателей. Все, что говорилось об умных указателях в трех последних главах, легко распространяется и на курсоры. И наоборот, изучение «курсорологии» помогает расширить некоторые концепции умных указателей. Перегружая оператор = для умного указателя, вы сумеете избежать многих неприятных проблем. Например, вспомните концепцию кэширующего указателя, который в последний момент считывал объект с диска в операторе ->. Подобная перегрузка оператора присваивания нередко очищает программу и избавляет код от ненужных технических деталей. Другой полезный прием — привязка умного указателя к некоторой структуре данных (подобно тому, как `ArrayCursor` привязывался к классу `SparseArray`). Такое гармоничное объединение идей проектирования является хорошим признаком — мы приближаемся к неуловимой высшей истине C++. Чем более передовыми идиомами вы пользуетесь, тем больше возникает сходства.

Итераторы

Итак, мы можем работать с любым отдельным элементом коллекции. Как насчет того, чтобы перебрать все элементы? Тупой перебор в цикле `for` не поможет:

```
for (int i = 0; i < ... чего?
```

При выбранной реализации разреженного массива измерения не имеют верхней границы. Впрочем, даже если бы она и была, хотелось бы вам перебирать 1 000 000 000 всевозможных индексов в поисках

какой-то тысячи используемых? Знаю, знаю, ваш RISC-компьютер прогоняет бесконечный цикл за семь секунд, но давайте мыслить реально. Если для коллекции существует оптимальный способ обращаться только к используемым элементам, мы должны предоставить его в распоряжение клиента. Но помните, клиент ничего не знает о внутреннем строении наших коллекций; собственно, именно для этого мы изобретали курсоры. Добро пожаловать в удивительный и безумный мир итераторов (iterators) — классов, предназначенных для перебора коллекций! Удивительный — поскольку итераторы просто решают многие проблемы проектирования. Безумный — поскольку два программиста C++ ни за что не придут к общему мнению о том, какие же идиомы должны использоваться в реализации итераторов.

Активные итераторы

Активным называется итератор, который сам перемещается к следующей позиции.

```
class Collection {
public:
    class Iterator {
    public:
        bool More();
        Foo* Next();
    };
    Collection::Iterator* Iterate();      // Создает итератор
};
Collection::Iterator* iter = collection->Iterator();
while (iter.More())
    f(iter.Next());
```

Как правило, итераторы относятся к конкретным коллекциям; по этой причине они часто объявляются в виде вложенных классов. Функция `More()` возвращает `true`, если в коллекции имеется следующий элемент в порядке перебора, и `false` — в противном случае. Функция `Next()` возвращает следующий элемент и перемещает итератор к следующей позиции.

Если вы готовы пойти на дополнительные расходы, связанные с виртуальными функциями, итератор также можно реализовать в виде универсального шаблона, работающего с любыми коллекциями.

```
template <class Type>
class Iterator {   // Подходит для любых коллекций и типов
public:
    virtual bool More() = 0;
    virtual Type* Next() = 0;
};
```

Каждая коллекция может реализовать итератор заново в производном классе, а клиенты по-прежнему понятия не имеют о том, как происходит перебор и даже какой класс находится по другую сторону забора. К сожалению, в некоторых коллекциях необходимо предоставить средства, не поддерживаемые коллекциями других типов (например, ограничение перебираемого диапазона), поэтому такой шаблон не настолько универсален, как кажется с первого взгляда.

Я называю такие итераторы *активными,* поскольку для выполнения всей основной работы вызываются их функции. Ситуация выглядит так, словно кто-то отломил кусок коллекции и вставил его в переносимый маленький объект. После конструирования такой объект сам знает, что ему делать дальше.

Пассивные итераторы

На другом фланге стоят итераторы, которые на самом деле не делают ничего существенного. В них хранится служебная информация, занесенная коллекцией, но перемещение и все остальные операции выполняются самой коллекцией. Нам понадобятся те же функции — просто из итератора они

перемещаются в коллекцию, а итератор применяется только для хранения служебной информации этих функций. Базовый итератор и цикл итерации выглядят так:

```
class Iterator;
class Collection {
public:
    Iterator* Iterate();    // Возвращает пассивный итератор
    bool More(Iterator*);
    Foo* Next(Iterator*);
};
Iterator* iter = collection->Iterate();
while (collection->More(iter))
    f(collection->Next(iter));
```

Такие итераторы называются *пассивными,* поскольку сами по себе они не выполняют никаких действий и предназначены лишь для хранения служебной информации.

Что лучше?

Выбор между активными и пассивными итераторами в основном зависит от стиля, но я предпочитаю активные итераторы по нескольким причинам:

- Законченный класс итератора проще использовать повторно, чем пару функций большого класса.
- Рано или поздно вам захочется предоставить несколько разных способов перебора содержимого коллекции. Один и тот же общий интерфейс класса `Iterator` подойдет для любого способа, а в форме функций класса для каждого типа перебора вам придется создавать пару новых функций.
- Пассивные итераторы не имеют открытого интерфейса, однако клиентские объекты видят их через адреса. Это выглядит довольно странно.
- При равенстве всех остальных показателей я обычно предпочитаю активные итераторы, поскольку они обеспечивают лучшую инкапсуляцию.

В коммерческих библиотеках классов можно встретить хорошие примеры обеих форм. В вопросе об активных и пассивных итераторах отражается общий спор об активных и пассивных объектах, поэтому в первую очередь следует учитывать собственные приемы проектирования.

Убогие, но распространенные варианты

Вряд ли вы встретите в коммерческих библиотеках классов итераторы именно в таком виде. У каждого находится свой подход к этой теме. Ниже перечислены некоторые варианты, которые часто встречаются в странствиях по C++, с краткими комментариями по поводу их достоинств и недостатков.

Мономорфные активные итераторы вне области действия

Даже жалко расходовать замечательный термин на такую простую концепцию. Итераторы называются мономорфными, поскольку в них не используются виртуальные функции, и находятся вне области действия, поскольку они не объявляются вложенными в коллекцию.

```
class Collection { ... };
class CollectionIterator {
private:
    Collection* coll;
public:
    CollectionIterator(Collection* coll);
    bool More();
    Foo* Next();
};
```

```
CollectionIterator iter(collection);      // Создать итератор
while (iter.More())
    f(iter.Next());
```

Просто удивительно, что всего несколько строк программы порождает столько проблем:

- При использовании класса, производного от `Collection`, каждый клиент должен знать, какие новые классы итераторов должны использоваться вместо старого `CollectionIterator`.
- Переменные класса итератора видны всем и каждому. Даже если они не составляют государственной тайны, весь клиентский код придется перекомпилировать каждый раз, когда вам захочется изменить реализацию итератора.
- Занесение итераторов в стек противоречит некоторым стратегиям многопоточности, рассматриваемым в следующей главе.
- Многократное использование такого кода — задача мерзкая.

Учитывая все это, будет намного, намного лучше попросить класс коллекции: «Пожалуйста, сэр, сделайте мне итератор» вместо того, чтобы самому создавать его в стеке. Невзирая на все проблемы, этот тип итераторов часто встречается в коммерческих библиотеках классов.

Пассивные итераторы типа `void*`

Самая распространенная вариация на тему пассивных итераторов — не возиться с предварительным объявлением класса итератора, а обмануть клиентов и внушить им, что на самом деле они имеют дело с типом `void*`. Все это часто маскируется каким-нибудь красивым именем с помощью `typedef`, но уродливый `void*` так легко не спрячешь.

```
typedef void* AprilInParis;
class Collection {
public:
    AprilInParis Iterate();   // Возвращает загримированный void*
    bool More(AprilInParis&);
    Foo* Next(AprilInParis&);
};
```

Конечно, во внутреннем представлении хранится что-то более разумное, чем `void*`, поэтому код реализации `Collection` должен постоянно преобразовывать `void*` к реальности. Не знаю как вас, но лично меня приводит в ужас одна мысль о том, что клиентский код будет возиться с `void*` до его преобразования. К тому же отладка такого кода дьявольски сложна, поскольку отладчик знает о том, с чем он имеет дело, ничуть не больше клиента. Красивое название итератора не скроет изначального уродства такого подхода.

Нетипизированные значения функции `Next()`

Многие классы итераторов пишутся в обобщенной форме для типа `void*` или какого-то абстрактного базового класса. Клиент должен сам приводить значение, возвращаемое функцией `Next()`, обратно к правильному типу — и горе ему, если он что-нибудь напутает. Шаблоны изобретались именно для этой цели, так что теперь подобный бред уже нельзя оправдать.

Лучшие варианты

Начиная с этого места, я буду говорить об активных итераторах, однако все сказанное в равной мере относится и к пассивным итераторам. Некоторые разновидности итераторов сильно зависят от характеристик коллекции, но другие обладают большей универсальностью. Перечисляю их, не придерживаясь какого-то определенного порядка.

Возврат в исходную точку

Некоторые классы итераторов содержат функцию, которая возвращает итератор к началу перебора. Я называю ее функцией возврата, Rewind(). Такая возможность поддерживается не всеми коллекциями — например, для потока данных из коммуникационного порта это невозможно.

Ограничение диапазона

Если совокупность объектов, представленных в виде коллекции, упорядочена, итератор должен обладать некоторыми средствами для ограничения перебора конкретным диапазоном объектов.

```
class Collection {
public:
     class Iterator {
     public:
          Iterator(Key* low, Key* high);
          // И т.д.
     };
     // И т.д.
};
```

В этом фрагменте `low` и `high` определяют минимальное и максимальное значение ключа соответственно. `More()` и `Next()` пропускают элементы, не входящие в заданный диапазон со включением границ. Другие вариации на эту тему — «все элементы больше X», «все элементы меньше X» и исключение границ (< вместо <=).

Откат

Итераторы также могут поддерживать функцию `Previous()` для отката на одну позицию, если такая возможность обеспечивается самой коллекцией. Эта функция часто используется вместе с функцией `Current()`, которая возвращает то, что `Next()` возвратила при последнем вызове.

Курсоры

Часто бывает очень полезно объединить концепции, описанные в этой главе, и возвращать из `Next()` не *-указатель, а курсор, который знает, откуда взялся элемент. Благодаря этому пользователь сможет выполнить удаление, замену или вставку до или после текущего объекта. Возможны два варианта реализации: возвращать курсор из функции `Next()` или включить «курсороподобные» операции в качестве функций класса самого итератора, работающих с последней полученной позицией. Первый вариант требуется взаимодействия итератора с курсором; во втором они объединяются в один класс.

Итератор абстрактного массива

Перейдем к простому примеру на основе нашего разреженного массива. Классы массива и курсора взяты из предыдущего обсуждения без изменений за исключением того, что класс массива теперь также возвращает итератор лишь для непустых ячеек. Универсальный шаблон итератора не используется, поскольку функция Next() возвращает как индекс, так и объект с этим индексом, а это требует нестандартного интерфейса к Next(). Классы курсора и разреженного массива остались в прежнем виде. Я не утверждаю, что это *хороший* разреженный массив — однако он обладает достаточно простым дизайном, который не будет нам мешать при обсуждении итераторов.

```
// SparseArray.h
class ArrayCursor;
class SparseArray {
friend class ArrayCursor;
private:
     struct Node {
          Index index;
          Foo* content;
```

```
            Node* next;
            Node(Index i, Foo* c, Node* n) : index(i), content(c), next(n) {}
        };
        Node* cells;
    public:
        class Iterator {
        private:
            Node* next;
        public:
            Iterator(Node* first) : next(first) {}
            bool More() { return next != NULL; ]
            Foo* Next(Index& index)
            {
                Foo* object = next->content;
                index = next->index;
                next = next->next;
                return object;
            }
        };
        Iterator* NonEmpty() { return new SparseArray::Iterator(cells); }
        SparseArray() : cells(NULL) {}
        ArrayCursor operator[](Index i);
    };

    class ArrayCursor {
    friend class SparseArray;
    private:
        SparseArray& array;
        Index index;
        SparseArray::Node* node;
        ArrayCursor(SparseArray& arr, Index i)
            : array(arr), index(i), node(NULL) {}
        ArrayCursor(SparseArray& arr, SparseArray::Node* n)
            : array(arr), node(n), index(n->index) {}
    public:
        ArrayCursor& operator=(Foo* foo);
        operator Foo*() { return node != NULL ? node->content : NULL; }
        Foo* operator->()
        {
            if (node == NULL)
                throw nil_error;
            else
                return node->current;
        }
    };
```

Пожалуй, я бы не рискнул показывать эту программу потенциальному работодателю как доказательство глубоких познаний в C++, но она проста, быстра и справляется со своей задачей. Ниже перечислены некоторые изменения, которые можно было бы внести в коммерческий вариант:

- Инкапсулируйте SparseArray::Iterator, превратив его в абстрактный базовый класс, а затем верните производный класс из скрытой реализации NonEmpty() (эта идея также хорошо подходит для классов массива и курсора, поэтому мы разовьем ее в части 3).
- Предоставьте дополнительные итераторы, которые включают как пустые, так и непустые ячейки.
- Гарантируйте определенный порядок перебора ячеек.
- Возвращайте из Next() курсор, а не указатель, чтобы клиенты могли изменять содержимое ячейки во время перебора. Если это будет сделано, индекс может храниться в курсоре, поэтому отпадает необходимость возвращать его в виде отдельного ссылочного аргумента Next().

Операторы коллекций

Многие коллекции индексируются одним или несколькими способами и хорошо соответствуют оператору [], однако в нашем обсуждении курсоров и итераторов нигде не выдвигалось требование непременно использовать оператор [] или индексировать коллекцию. Курсор лишь определяет некоторую внутреннюю позицию в коллекции; эта позиция не обязана быть чем-то понятным или представляющим интерес для пользователя. Если убрать из функции Next() аргумент Index&, описанный итератор можно будет с таким же успехом использовать не для массива, а для чего-то совершенно иного.

В большинстве коллекций имеются общие операции. Как и в случае с оператором [], операторы C++ обычно перегружаются для получения более понятного и удобочитаемого кода. Хотя не существует повсеместного стандарта операторов коллекций, приведенный ниже перечень поможет вам начать ваши собственные разработки. Во всех приведенных операторах сохраняется семантика соответствующих операций с числами.

```
template <class Element>
class Set {
public:
     Set();                      // Пустое множество
     Set(const Set<Element>&); // Дублировать множество
     Set(Element*);              // Множество с одним исходным элементом

     // Бинарные операции и операции отношения (множество, множество)
     // (также варианты |=, &=, -=, <, <=)
     Set<Element> operator|(const Set<Element>&) const;     // Объединение
                                                            // множеств
     Set<Element> operator&(const Set<Element>&) const;     // Пересечение
     Set<Element> operator-(const Set<Element>-) const;     // Разность
                                                            // множеств
     bool operator>(const Set<Element>&) const;  // Истинно, если this
          // является точным надмножеством аргумента
     bool operator>=(const Set<Element>&) const; // Истинно, если this
          // является надмножеством аргумента
     bool operator==(const Set<Element>&) const; // Истинно, если множества
               // имеют одинаковое содержимое

          // Бинарные операции и операции отношения (множество, элемент*)
          // (также варианты |=, -=)
          Set<Element> operator|(Element*);      // Добавить элемент в this
          Set<Element> operator-(Element*);      // this минус элемент
          bool operator>(const Element*) const;  // Истинно, если элемент
               // принадлежит множеству, но не является единственным
```

```
            bool operator>=(const Element*) const; // Истинно, если элемент
                // принадлежит множеству
            bool operator==(const Element*) const; // Истинно, если элемент
                // является единственным элементом множества
};
```

Существует еще один вариант перегрузки оператора, о котором я вынужден упомянуть. Я так и не привык к операторам << и >> в качестве операторов «поразрядных сдвигов» в поток и из него, но поскольку они прочно внедрились в культуру C++, эту идиому приходится использовать хотя бы как базу для дальнейших расширений. Это приводит нас к дополнительному применению << и >> в контексте коллекций и итераторов:

- Оператор << может использоваться в итераторах как синоним функции `Next()`.
- Оператор >> может использоваться как синоним более длинного оператора `Set& operator|=(Element*)` для «сдвига» новых элементов в коллекцию.

В обоих идиомах оператор должен перегружаться в форме внешней функции, поскольку в левой части оператора находится `Element*`, а не `Set`. Идиома >> выглядит наиболее естественно для коллекций, сохраняющих исходный порядок вставки (например, списков).

Мы вернемся к этой теме в части 3 при обсуждении гомоморфных иерархий классов.

Мудрые курсоры и надежность итераторов

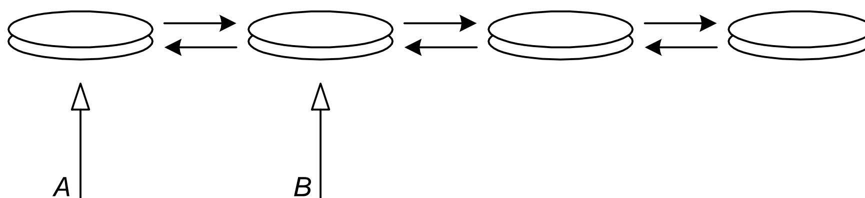
Курсор может использоваться для вставки объекта в некоторую позицию коллекции независимо от того, отражена ли данная позиция во внутренних структурах данных коллекции. Именно этот принцип бал заложен в основу перегрузки оператора = для курсоров. Его можно обобщить на другие операции с курсорами. Ниже перечислены типичные расширенные операции с курсорами, выраженные в виде функций класса курсора:

```
void InsertBefore(Foo* f);   // Вставить f перед курсором
void InsertAfter(Foo* f);    // Вставить f после курсора
void RemoveAt();             // Удалить объект в текущей позиции курсора
```

Сказанное относится к широкому диапазону коллекций, в которых существует четко определенная последовательность элементов. Перечисленные операции также могут обеспечиваться итераторами, которые не возвращают курсор как значение функции `Next()`, а скрывают текущую позицию в итераторе.

Эти операции усложняют соблюдение единой семантики перебора, а в худшем случае — порождают серьезные недостатки дизайна, которые могут угробить вашу программу. Впрочем, проблемы могут возникнуть и без расширенных операций, если изменения в коллекции могут происходить при наличии активных курсоров и итераторов. Представьте себе, что некий фрагмент программы удаляет объект, на который ссылается текущий активный курсор! Приходится особо заботиться, чтобы это не вызвало катастрофических последствий. Главная проблема — сделать курсор *надежным*, чтобы они могли пережить обновление своих базовых коллекций.

Для примера возьмем связанный список и его итератор.

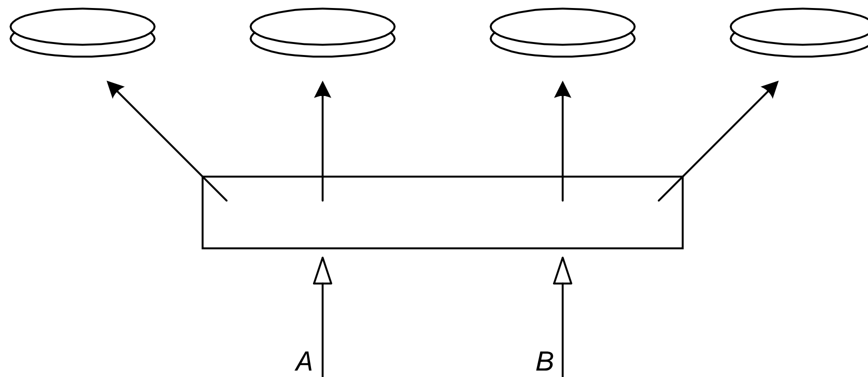


Курсоры A и B используются для отслеживания текущей позиции двух разных итераторов в одном списке. Все отлично работает, пока список остается без изменений. Но стоит клиенту одного из итераторов воспользоваться курсором для обновления списка, как немедленно возникают проблемы:

- Если клиент выполнит операцию «вставки после» с курсором B, оба итератора при возобновлении перебора увидят только что вставленный объект.
- Если клиент выполнит операцию «вставки до» с курсором B или «вставки после» с курсором A, итератор-владелец A увидит вставленный объект, а итератор-владелец B — нет.
- Если клиент удалит объект в позиции B, A никогда не увидит этого объекта, хотя тот находился на споем месте, когда итератор-владелец A начал перебор списка.
- Если клиент удалит объект в позиции A, B успеет увидеть этот объект перед тем, как произошло удаление.
- При выполнении вставки после любого курсора в некоторых алгоритмах вставки возникает бесконечная рекурсия, поскольку каждый только что вставленный объект может инициировать вставку другого объекта.
- Если A и B ссылаются на общий элемент списка, а один из них этот элемент удалит — «Здравствуй, дебаггер!»

Короче, при внесении изменений в коллекцию во время перебора семантика превращается в сплошной винегрет. А если при этом одновременно работают два и более итератора, результаты можно определять по броскам кубиков.

На самом деле списки — случай относительно простой. Подумайте, что произойдет, если коллекция хранится в виде массива (независимо от того, представляется ли она клиенту массивом или нет), а курсор располагает целочисленным индексом в этом массиве.



Чтобы обеспечить выполнение вставки «до/после» и удаления, приходится сдвигать элементы массива над позицией вставки или удалять одни элемент выше или ниже. Если на диаграмме удалить элемент в A и сдвинуть все, что находится выше, на одну позицию вниз, B пропустит позицию. Если вставить элемент в A, B увидит один и тот же объект дважды.

Семантика перебора должна быть намного более ясной и более предсказуемой. Для большинства приложений порядок элементов следует фиксировать на момент начала перебора, что на него не влияли последующие операции вставки и удаления. Как правило, идиомы итератора и курсора должны соблюдать два правила:

1. Итератор должен перебирать объекты коллекции на момент своего конструирования.
2. Курсор должен оставаться действительным от момента конструирования до момента уничтожения. Другими словами, программа не должна выполнять операции, после которых активный курсор может потерять работоспособность. Это не означает, что значение в позиции курсора должно оставаться одним и тем же — речь идет лишь о том, что курсор обязан сохранять работоспособность.

Эти правила вносят некоторый порядок в то, что грозило превратиться в абсолютный хаос. Первое из них можно назвать «принципом затенения», поскольку все изменения, вносимые в коллекцию после конструирования итератора, скрываются («затеняются») от него. Это одна из глобальных концепций дизайна, по которой запросто можно написать отдельную книгу, но, к счастью, у нас поставлена более приземленная цель — продемонстрировать те идиомы C++, в которых воплощаются практические решения.

Частные копии коллекций

Если итератор и его курсор не позволяют вносить изменения в коллекцию, существует простой выход: создать частную копию коллекции в конструкторе итератора. На псевдокоде это выглядит так:

```
class Iterator {
private:
    Collection collection;
    Cursor location;   // Текущая позиция в копии
public:
    Iterator(Collection& c)
        : collection(c), location(collection.First()) {}
    bool More();
    Foo* Next();
};
```

Конструктор итератора с помощью конструктора копий класса `Collection` создает вторую частную копию коллекции. Перед вами — один из редких случаев, когда действительно имеет значение тот факт, что переменные класса конструируются в порядке их перечисления; объект `collection` должен быть сконструирован раньше объекта location, в противном случае вам предстоят мучения с отладкой функции `First()`.

Коллекции объектов или коллекции указателей?

Эта схема обычно используется в ситуациях, когда коллекция состоит из указателей или ссылок на объекты, которые во всем остальном никак не связаны с коллекцией. В других коллекциях вместо указателей или ссылок содержатся собственно объекты.

```
template <class Type, int Size>
class Array {
private:
    int Size;     // Количество объектов Type
    Type elements[size];    // Объекты (внутренние)
// и т.д.
};
```

Здесь объекты буквально внедряются в коллекцию. Чтобы продублировать коллекцию, вам придется скопировать не только указатели, но и объекты — а это может обойтись слишком дорого. С другой стороны, может возникнуть необходимость в том, чтобы итератор возвращал указатель или ссылку на исходный объект исходной коллекции, а не на копию. В любом случае вариант с частными коллекциями отпадает.

Тот же принцип действует каждый раз, когда коллекция представляет собой набор ведущих указателей на ее содержимое. Да, она содержит указатели, а не объекты, однако коллекция имеет право удалять эти объекты, поэтому частная копия будет неустойчивой. Некоторые вопросы управления памятью, связанные с этой проблемой — конкретнее, сборка мусора — рассматриваются в части 4 этой книги.

Упрощение частной коллекции

Предположим, исходная коллекция представляет собой бинарное дерево или другую сложную структуру данных. Так ли необходимо воспроизводить в копии все дополнительные издержки древовидной структуры, если учесть, что вы не собираетесь пользоваться индексированным доступом? Существует общепринятое решение — создать в качестве частной копии упрощенный вариант коллекции. Это будет проще, если в классе коллекции имеется оператор преобразования, порождающий экземпляр упрощенной коллекции. Вместо конструктора копий коллекции итератор использует ее оператор преобразования:

```
class SimpleCollection;     // Упрощенный вариант
class ComplexCollection {
```

```
public:
    operator SimpleCollection*();
};
```

Существует и другой, похожий вариант — создать в классе SimpleCollection конструкторы для всех остальных типов коллекций. Однако с точки зрения дизайна такое решение неудачно — каждый раз, когда вы придумаете какую-нибудь новую экзотическую коллекцию, вам придется изменять класс SimpleCollection. Для таких случаев существуют операторы преобразования.

Если использовать этот вариант, итератор становится универсальным и подходящим для различных типов коллекций. Итератору не нужно ничего знать об исходной коллекции. Конструктору итератора передается адрес упрощенной коллекции вместо исходной, при этом интерфейс выглядит так:

```
class Iterator {
private:
    SimpleCollection* collection;
    Cursor location;        // Текущая позиция в копии
public:
    Iterator(SimpleCollection* c)
        : collection(c), location(collection->First()) {}
    bool More();
    bool Next();
};
```

Внутренние и внешние итераторы

Вернемся к итераторам, работающим с исходной коллекцией. Существуют два типа итераторов: относящиеся к внутренней реализации коллекции (например, для приведенного выше класса `SimpleCollection`) и открытые внешнему миру. Они называются *внутренними* (*internal iterator*) и *внешними* (*external iterator*) итераторами соответственно.

Внутренний итератор обычно представляет собой тупой, ненадежный итератор, который перебирает объекты коллекции в ее текущем состоянии. Если в коллекции происходит вставка или удаление, внутренние итераторы начинают выкидывать все те странные фортели, о которых говорилось в начале раздела. По этой причине их тщательно прячут от шаловливых рук клиента. Как правило, внутренние итераторы тесно связаны со структурами данных, использованными в реализации коллекции. Как и любые другие итераторы, они могут возвращать *-указатель или курсор в зависимости от ваших потребностей.

Внешние итераторы соблюдают принцип затенения. Затенения можно добиться многими способами, часть из которых рассматривается далее в этой главе и в главе 9. Как всегда, суть кроется не в конкретном алгоритме или структуре данных, а в том, как спрятать их от публики.

Временные внутренние итераторы

Если внешний итератор создает частную копию коллекции (см. предыдущий раздел) и при этом не существует оператора преобразования или конструктора, способного превратить исходную коллекцию в частную, в конструкторе внешнего итератора можно воспользоваться внутренним итератором. В следующем фрагменте два внутренних итератора объединяются в реализации одного внешнего:

```
class ExternalIterator {
private:
    SimpleCollection collection;
    SimpleIterator* my_iter;        // Возвращается коллекцией
public:
    ExternalIterator(ComplexCollection* c)
        {
            InternalIterator* iter = c->Iterator();
```

```
            while (c->More())
                collection += *(c->Next());
            delete iter;
            my_iter = collection->Iterator();
        }
    bool More() { return my_iter->More(); }
    bool Next() { return my_iter->Next(); }
};
```

ComplexCollection предоставляет внутренний итератор, который существует ровно столько, сколько необходимо для создания копии. SimpleCollection возвращает итератор, используемый для реализации функции More() и Next() внешнего итератора. Конечно, все могло бы выглядеть намного элегантнее, если бы у SimpleCollection был конструктор с аргументом ComplexCollection или у ComplexCollection — операторная функция преобразования operator SimpleCollection(). Но даже при их отсутствии класс итератора обеспечивает весь необходимый уровень инкапсуляции.

Устойчивые внутренние итераторы

Термин «устойчивый» (persistent) означает, что внутренний итератор существует до тех пор, пока существует внешний итератор (my_iter в предыдущем примере). Внутренний итератор может быть переменной класса внешнего итератора, как было показано, а при достаточной осторожности его можно создать как производный класс посредством закрытого наследования. Вариант с закрытым наследованием может выглядеть так:

```
// В файле .h
class Collection {
public:
    class ExternalIterator {
    public:
        virtual bool More() = 0;
        virtual Foo* Next() = 0;
    };
    ExternalIterator* Iterator();
};

// В файле .cpp
// Настоящий класс, возвращаемый клиентам
class RealExternalIterator
    : public ExternalIterator, private InternalIterator
(...);

Collection:ExternalIterator* Collection::Iterator()
{
    return new RealExternalIterator(this);
}
```

Обладающий локальной областью действия ExternalIterator обеспечивает абстрактный интерфейс, предоставляемый клиенту. Настоящий возвращаемый класс, RealExternalIterator, порожден от Collection::ExternalIterator посредством открытого наследования, а также (о чем клиент не подозревает) — от SimpleIterator посредством закрытого наследования. Как и в большинстве проблем дизайна C++, закрытое наследование проще реализуется, а делегирование переменной класса оказывается более универсальным. Например, вы можете на полпути заменить переменную, чтобы сцепить несколько внутренних итераторов в одном внешнем.

Фильтрующие итераторы

Одна из проблем, связанных с этой идиомой — реализация функции More(). Предполагается, что функция Next() внешнего итератора может пропустить объект, возвращаемый внутренней функцией Next(). Например, если внутренний итератор возвращает элемент, вставленный после конструирования внешнего итератора, внешний итератор может захотеть пропустить его. Функция More() внутреннего итератора заявляет, что в коллекции еще остались элементы, но при попытке извлечения они благополучно отвергаются функцией Next(). Один из вариантов решения — включить во внутренний итератор функцию «подсматривания» Peek(). Такая функция возвращает то же, что и Next(), но не перемещает курсор к следующей позиции. После такого добавления возникает стопроцентно надежный способ внедрить внутренний итератор во внешний:

```
class RealExternalIterator : public ExternalIterator {
private:
    InternalIterator* iter;
    bool Accept(Foo*);      // Фильтрующая функция
public:
    RealExternalIterator(Collection* c) : iter(c->Iterator()) {}
    virtual bool More()
    {
        while (iter.More()) {
            if (Accept(iter->Peek()))
                return true;
            (void)iter->Next();   // Отвергнуть и переместиться
        }
        return false;
    }
    virtual Foo* Next() { return iter->Next(); }
};
```

Каждый раз, когда клиент вызывает More() (в том числе и в начале цикла), внутренний итератор перемещается вперед до тех пор, пока не наткнется на элемент, который удовлетворяет фильтрующей функции Accept() внешнего итератора.

В особо зловредных коллекциях, чтобы реализовать функцию Peek(), вам придется сохранять копию последнего увиденного объекта, однако в большинстве случаев удается легко выполнить неразрушающее считывание текущей позиции.

Разумеется, возможности этой методики не ограничиваются затенением итераторов. Ее можно применять в любой ситуации, когда внешний итератор отвергает часть объектов, возвращаемых внутренним. Например, функция Accept() может накладывать ограничения для запроса к базе данных.

Временная пометка

Один из простейших фокусов для вставки — временная пометка вставок и итераторов. Если пометка итератора относится к более раннему моменту, чем пометка позиции, итератор пропускает объект в данной позиции. Временную пометку можно реализовать по-разному: буквально (как количество тактов таймера); в виде номера хранящегося где-то в статической переменной класса; или в переменной объекта коллекции.

Удаление обрабатывается аналогично. Если кто-то пытается удалить объект из коллекции, в действительности объект остается на своем месте до исчезновения всех старых итераторов. Текущие клиенты и новые итераторы игнорируют его, а итераторы, сконструированные до удаления, продолжают работать так, словно никто не сообщил им о печальной судьбе объекта. Фактически объект удаляется лишь тогда, когда он заведомо никому не нужен. Мы столкнулись с одним из вопросов сборки мусора, о котором пойдет речь в части 4. Удаленные объекты легко уничтожаются в

процессе сохранения коллекции на носителе информации или в любой момент при отсутствии активных итераторов и курсоров.

Иногда объект может находиться сразу как во вставленном, так и в удаленном состоянии; вставка произошла после того, как итератор был сконструирован, а удаление — до уничтожения итератора. В сущности, для каждого объекта можно вести целый журнал вставок и удалений, если для этих событий хватит жизненного срока вашего итератора. Для ведения такого журнала подходят многие различные структуры данных. Эта побочная тема достаточно интересна, хотя она и не имеет особого отношения к рассматриваемым идиомам C++. Чтобы обеспечить необходимый уровень инкапсуляции, мы внесем некоторые изменения в концепцию внутреннего итератора из предыдущего раздела.

Класс временных меток

Следующий класс инкапсулирует временные пометки. Его можно модифицировать, чтобы вместо последовательного счетчика использовались показания системных часов — для клиента это глубоко безразлично. Обратите внимание: конструктор копий и оператор = *не перегружаются*. При передаче `Timestamp` по значению создается временный объект `Timestamp` с тем же временем, что и в исходном объекте, а в случае присваивания одного `Timestamp` другому левосторонний объект приобретает ту же метку, что и правосторонний.

```
class Timestamp {
private:
    static int last_time;   // Используется для присваивания числа
    int stamp;
public:
    Timestamp() : stamp(++last_time) {}
    bool operator>(Timestamp ts) { return stamp > ts.stamp; }
    bool operator>=(Timestamp ts) { return stamp >= ts.stamp; }
    bool operator<(Timestamp ts) { return stamp < ts.stamp; }
    bool operator<=(Timestamp ts) { return stamp <= ts.stamp; }
    bool operator==(Timestamp ts) { return stamp == ts.stamp; }
    bool operator!=(Timestamp ts) { return stamp != ts.stamp; }
};
```

Внутренний итератор с временной пометкой

Внутренний итератор также необходимо модифицировать, чтобы он учитывал присутствие временных пометок. Соответствующие фрагменты интерфейса приведены ниже. Подробности реализации в значительной мере определяются структурами данных коллекции:

```
class InternalIterator {
public:
    bool More(Timestamp as_of);
    Foo* Next(Timestamp as_of);
    bool Peek(Timestamp as_of);
};
```

Внутренний итератор будет пропускать объекты, отсутствовавшие в коллекции в указанное время.

Способы внутренней реализации

Один из возможных путей реализации такого поведения — связать с каждой позицией коллекции вектор временных пометок, в котором самый старый элемент будет соответствовать исходной вставке, а последующие элементы — поочередно описывать последующие вставки и удаления. Если вектор содержит нечетное количество элементов, объект в настоящее время находится в коллекции, а первый элемент вектора относится к моменту последней вставки. Если число элементов четное, объект отсутствует в коллекции, а первый элемент описывает момент последнего удаления. В процессе уплотнения коллекции уплотняются и журналы удалений — в них остается лишь момент последней

вставки. Существуют и другие возможности реализации (например, через списки исключений), однако методика журналов удалений по крайней мере демонстрирует концепцию.

Внешний итератор с временной пометкой

Внешний итератор представляет собой тривиальную оболочку для только что описанного внутреннего итератора:

```
class RealExternalIterator : public ExternalIterator {
private:
    InternalIterator* iter;
    Timestamp my_time;      // Время конструирования ‘this’
    bool Accept(Foo*);      // Фильтрующая функция
public:
    RealExternalIterator(Collection* c) : iter(c->Iterator()) {}
    virtual bool More() {
        while (iter.More(my_time)) {
            if (Accept(iter->Peek(my_time)))
                return true;
            (void)iter->Next(my_time);
        }
        return false;
    }
    virtual Foo* Next() { return iter->Next(my_time); }
};
```

Безаргументный конструктор `Timestamp` использует для пометки текущее время. Во многих ситуациях функция `Accept()` всегда возвращает `true`, и ее можно убрать.

Пример

Давайте завершим эту затянувшуюся главу примером — шаблоном для набора объектов, тип которых определяется параметром шаблона. В этом наборе каждый объект присутствует ровно один раз; при попытке снова вставить уже существующий объект набор остается без изменений. Для реализации надежных итераторов будет использована методика временных пометок вместо частных копий, создаваемых при конструировании итератора. Заодно мы посмотрим, на какие ухищрения порой приходится идти, чтобы разумно использовать неразумные коммерческие классы.

Ненадежный коммерческий класс словаря

В реальном мире редко приходится начинать с пустого места. Обычно в вашем распоряжении уже имеются готовые классы коллекций. Даже если они работают не совсем так, как хочется, по крайней мере они избавят вас от нудного переписывания любимых глав из книг Кнута при реализации базовых структур данных. В том же реальном мире эти классы обычно не обращают особого внимания на проблему надежности итераторов, так что наш пример никак не назовешь абстрактным. Для пущего реализма мы будем считать, что коллекция представляет собой словарь, который индексирует значение типа `void*` и возвращает `void*` в качестве ключа. Имеется пассивный итератор `Slot`, реальная структура которого спрятана глубоко внутри класса словаря. Открытый интерфейс выглядит так:

```
class Dictionary {
public:
    Dictionary();      // Создает пустой словарь
    Dictionary(const Dictionary&);  // Конструктор копий
    ~Dictionary();
    Dictionary& operator=(const Dictionary&);

    void AddEntry(void* key, void* value);
```

```
        bool At(void* key, void*& value);
        void RemoveEntry(void* key);

        typedef void* Slot;     // Настоящее объявление надежно спрятано
        Slot First();           // Возвращает Slot для перебора
        // Эквивалент нашей функции “Peek”
        bool GetCurrent(Slot slot, void*& key, void*& value);
        // Эквивалент нашей функции “Next”
        bool GetNext(Slot& slot, void*& key, void*& value);
    };
```

Функция `AddEntry()` заносит в словарь новое значение с заданным ключом. Если ключ уже существует для другого значения, значение заменяется новым. Функция `At()` возвращает логический код, который показывает, присутствует ли ключ в словаре; если присутствует, функция возвращает `true` и `value` (значение, соответствующее данному ключу). Функция `RemoveEntry()` удаляет ключ и связанное с ним значение. Функция `First()` возвращает пассивный итератор, направленный таким образом, чтобы первый вызов `GetNext()` возвратил первый элемент словаря. Функция `GetCurrent()` возвращает пару ключ/значение, находящуюся в текущей позиции `Slot`. Функция `GetNext()` возвращает пару ключ/значение и перемещает итератор к следующему элементу. Единственное отличие между этими функциями заключается в перемещении логической позиции после получения объекта.

Мы воспользуемся словарем следующим образом: ключом будет объект, хранящийся в нашем наборе, а значением этого ключа — журнал вставок/удалений для данного объекта.

Класс журнала

Ниже приведен открытый интерфейс класса, в котором хранится история вставок/удалений для объекта. Этот класс использует рассмотренный выше класс Timestamp — он образует вторую составляющую пар ключ/значение, хранящихся в словаре. В реализации нет ничего интересного, поэтому я ее пропускаю.

```
    class History {
    public:
        History();    // Пустой журнал
        void Insert(Timestamp);   // Вставка в заданное время
        void Remove(Timestamp);   // Удаление в заданное время
        bool Exists(Timestamp);   // В заданное время
        bool Exists();            // В настоящий момент
    };
```

Функция `Exists(Timestamp)` возвращает `true`, если в заданное время последней операцией была вставка, и `false` — если последней операцией было удаление или до этого времени вообще не было операций. Безаргументная функция `Exists()` является синонимом для `Exists(Timestamp)` с очень большим значением времени — то есть до настоящего момента.

Абстрактный базовый класс

Класс набора делится на две части: абстрактный базовый класс, работающий с `void*`, и производный параметризованный класс, в который добавлена безопасность типов. Абстрактный базовый класс выглядит так:

```
    // В файле Set.h
    class SetBase : private Dictionary {
    friend class InternalIterator;
    protected:
        SetBase();    // Чтобы класс был абстрактным
```

```
public:
     class Iterator {   // Базовый класс внешнего итератора
     public:
          virtual bool More() = 0;
          virtual Type* Next() = 0;
     };
};
Iterator* SetBase::ProvideIterator()
{
     return new InternalIterator(this);
}
void SetBase::AddObject(void* entry)
{
     void* v;
     History* h;
     if (At(entry, v)) {     // Уже есть - проверить время
          h = (History*)v;
          if (!h->Exists())  // Необходимо выполнить вставку
               h->Insert(Timestamp());
     }
     else {    // Еще нет
          h = new History;
          h->Insert(Timestamp());
          AddEntry(entry, h);
     }
}
void SetBase::RemoveObject(void* entry)
{
     void* v;
     History* h;
     if (At(entry, v)) {     // Уже есть - проверить время
          h = (History*)v;
          if (h->Exists())   // Необходимо выполнить удаление
               h->Remove(Timestamp());
     }
}
bool SetBase::Exists(void* entry, Timestamp ts)
{
     void* v;
     return At(entry, v) && ((History*)v)->Exists(ts);
}
bool SetBase::Exists(void* entry)
{
     void* v;
     return At(entry, v) && ((History*)v)->Exists();
}
```

Существуют и другие возможности, которые можно было добавить, но и показанного вполне хватит для демонстрации рассматриваемой методики.

Внутренний итератор

Чтобы реализовать функцию ProvideIterator(), мы создаем как нетипизированный внутренний итератор, ограниченный файлом .cpp и производный от SetBase::Iterator, так и внешний — в виде параметризованной, безопасной по отношению к типам оболочки. Ниже приведен код внутреннего итератора, объявленного статически (то есть локального по отношению к файлу .cpp). Вся логика временных пометок спрятана в реализации этого класса.

```
// В файле set.cpp
class InternalIterator : public SetBase::Iterator {
private:
    Dictionary* dictionary;
    Dictionary::Slot* slot;   // Текущая позиция
    Timestamp my_time;        // Время рождения данного итератора
public:
    InternalIterator(Dictionary* d)
        : dictionary(d), slot(d->First()), my_time() {}
    virtual bool More();
    virtual void* Next();
};
bool InternalIterator::More()
{
    void* key;
    void* h;
    if (!dictionary->GetCurrent(slot, key, h))
        return false;      // позиция выходит за текущие границы

    do
        if (((History*)h)->Exists(my_time))
            return true;
    while (dictionary->GetNext(slot, key, h));
    return false;
}
void* InternalIterator::Next()
{
    void* key;
    void* key1;
    void* h;
    // Предполагается, что клиент сначала вызвал More(),
    // поэтому объект GetNext() не устарел
    dictionary->GetCurrent(slot, key, h);
    dictionary->GetNext(slot, key1, h);
    return key;
}
```

Параметризованная оболочка, безопасная по отношению к типам

Наконец, в приведенном ниже параметризованном классе все становится безопасным по отношению к типам и начинает радовать глаз. Внешний итератор представляет собой параметризованную оболочку для внутреннего итератора, предоставляемого SetBase.

```
template <class Type>
class Set : private SetBase {
```

```
    public:
        // Безаргументный конструктор – подходит
        // Конструктор копий по умолчанию – подходит
        // Оператор = по умолчанию – тоже подходит
        Set<Type>& operator+=(Type* object)
            { AddObject(object); return *this; }
        Set<Type>& operator-=(Type* object)
            { RemoveObject(object); return *this; }
        bool operator>=(Type* object)
            { return Exists(object); }
        class Iterator {
        private:
            SetBase::Iterator* iter;
        public:
            Iterator(Set&* i) : iter(s.ProvideIterator()) {}
            bool More() { return iter->More(); }
            Type* Next() { return (Type*)(iter->Next()); }
        };
        Iterator* ProvideIterator()
            { return new Set::Iterator(*this); }
    };
```

Существуют и другие возможности, которые было бы неплохо добавить в параметризованный класс, но я думаю, что вы поняли общий принцип. От подобных параметризованных классов редко создаются производные, поэтому вложенный класс итератора оформлен с использованием подставляемых функция. В сущности, этот итератор можно просто объявить в стеке:

```
    Set::Iterator iter(aSet);
```

Такой вариант работает вполне нормально, однако он не соответствует обычной идиоме возвращения итератора из коллекции. Заслуживает внимания и другой вариант: сделать `SetBase` переменной класса вместо закрытого базового класса. Это позволит вам упрятать `SetBase` в файл .cpp, чтобы клиент никогда не видел его. Для этого в шаблоне `Set` придется определить конструкторы и оператор =, но все остальные модификации шаблона будут простыми и незамысловатыми.

Транзакции и гениальные указатели

9

Мы подошли к последней главе, посвященной косвенным обращениям (которая, впрочем, далеко не исчерпывает этой темы). Давайте посмотрим, как развить идею умных указателей. Встречавшиеся до сих пор идиомы и примеры умных и мудрых указателей, курсоров и итераторов главным образом были локализованы в одном инкапсулированном объекте. Хватит мелочиться — пора поговорить о том, как будет выглядеть архитектура в целом, если применить эти идиомы в более широком масштабе. Указатели, рассматриваемые в этой главе, умны настолько, что становится просто страшно. Эпитеты «умный» и «мудрый» уже не описывают их в достаточной степени. «Потрясающие, ослепительные, невероятно одаренные» звучит слишком выспренне, поэтому я скромно назову их «гениальными».

Предупреждение: в этой главе проверяется не только ваше мастерство, но и воображение. Описанные ниже проблемы и идиомы напоминают узкую горную тропу, по которой можно проехать только на велосипеде. Надевайте шлем и почаще останавливайтесь, чтобы передохнуть.

Тернистые пути дизайна

Как правило, после описания идиом в этой книге сразу же рассматриваются их практические применения. Позвольте мне отклониться от этой традиции и обсудить некоторые проблемы дизайна, которые понадобятся нам позднее — по мере того, как будет разворачиваться действие этой главы.

Транзакции

В приложениях клиент/сервер и базах данных часто встречается ситуация, когда несколько удаленных пользователей обращаются с запросом на обновление одной базы или структуры данных. Квант обновления, или *транзакция* (*transaction*), с точки зрения клиента может состоять из нескольких элементарных изменений в базе или структуре данных. В таких ситуациях разработчики руководствуются несколькими правилами, позаимствованными из мира СУБД:

1. Транзакции должны быть атомарными. Либо вносятся все изменения, необходимые для совершения транзакции, либо никакие.
2. Во время обработки транзакции одного клиента данные должны выглядеть так, как они выглядели в начале незавершенных транзакций всех остальных клиентов. До момента своего завершения транзакция остается невидимой для других клиентов; все выглядит так, словно транзакция и не начиналась.
3. Если сразу несколько клиентов в своих транзакциях пытаются изменить один объект, допускается успешное завершение не более одной транзакции.

Последнее правило гарантирует, что каждый объект обновляется не более чем одним клиентом. Существуют две стратегии его реализации:

1. Перед выполнением обновления каждая транзакция блокирует все объекты, которые она собирается изменить. Если транзакции не удается установить необходимые блокировки, она вообще не начинается.
2. Запрос на блокировку выдается не вначале, а во время транзакции по мере выполнения обновлений. Если необходимый объект уже заблокирован другим клиентом, то транзакция либо ожидает его освобождения, либо завершается неудачей.

Вторая стратегия может привести к взаимной блокировке (deadlock): транзакция A блокирует объект X и ожидает возможности блокировки объекта Y, тогда как транзакция B блокирует объект Y и ожидает возможности блокировки объекта X.

В мире баз данных на эту тему написано огромное количество литературы, поэтому нет смысла ее здесь подробно рассматривать. Нас интересуют те идиомы и возможности C++, которые облегчают решение целого класса подобных задач.

Отмена

Многие графические приложения любезно разрешают пользователю отменить последнюю выполненную операцию. На первый взгляд в этом нет ничего сложного, но в действительности не все так просто. В объектно-ориентированном мире найдется не так уж много проблем дизайна, которые вызывают большую головную боль, чем проблемы отмены в сложных приложениях. Чтобы реализовать отмену, многие программисты «зашивают» в программу структуры данных, ориентированные на конкретную операцию, но такой подход чреват ошибками и неустойчив при сопровождении программы.

В некоторых языках (таких как Lisp) можно буквально делать «снимки» памяти в различные моменты времени. При наличии такого снимка вернуться к прежнему состоянию приложения очень просто. Увы. В C++ таких изящных возможностей не предусмотрено; зато *наши* программы не запрашивают 3 Гб памяти и успевают завершить работу еще до обеда. Так что это замечание стоит воспринимать не как критику C++, а как констатацию факта: проблема есть, и ее необходимо решать.

В дальнейшем обсуждении будут рассматриваться две вариации на эту тему. Одни приложения всегда предоставляют один уровень отмены, а в других пользователь может выбирать команду Undo снова и снова — программа каждый раз возвращается на одну операцию назад. Конечно, ограничения все же существуют, но обычно не очень жесткие. Это называется «многоуровневой отменой» — проблема, которой мы займемся после одноуровневой отмены. Вторая вариация относится к контекстной отмене. Если пользователь выполняет операцию отмены сначала в одном, а потом — в другом окне, то после возврата к первому окну и выполнения команды Undo обычно следует отменить последнюю операцию только для первого окна, а не для всего приложения. Иначе говоря, операция отмены учитывает контекст приложения на момент своего вызова.

Хватит?

Существуют и другие проблемы, относящиеся к тому же классу, но и сказанного вполне достаточно, чтобы подтолкнуть нас к дальнейшим исследованиям. Большинство программистов рассматривает эти проблемы как нечто изолированное в контексте конкретного приложения. Однако, как вы вскоре убедитесь, некоторые специфические (а кто-то скажет — извращенные) особенности синтаксиса C++ позволяют приблизиться к общим решениям. Помните: мы изучаем C++ и его идиомы, а не структуры данных или принципы проектирования программ. Многие архитектуры и варианты здесь не приводятся, однако это вовсе не говорит против них.

Образы и указатели

Разве можно начать новую главу и не ввести новую разновидность указателей? Состояние объекта можно восстановить двумя способами: сохранить его образ (image) до и после операции или же хранить информацию, достаточную для выполнения операций в обратном направлении. Вариант достаточно прост и универсален, а попытки возврата через выполнение операций в обратном направлении привязаны к конкретному объекту и приложению, поэтому я предпочитаю хранить несколько копий одного объекта. Ключевая концепция, обеспечивающая реализацию этой методики на C++ — указатель образов (image pointer). Такие указатели незаметно для клиента содержат несколько

копий указываемого объекта. Существует множество всевозможных комбинаций и вариаций, так что выбор зависит главным образом от вашего настроения.

Простой указатель образов

На нескольких ближайших страницах показано простейшее решение этой проблемы. А пока лишь скажу, что мы имеем дело с ведущим указателем, удаляющим объекты, на которые он ссылается.

```
template <class Type>
class ImagePtr {
private:
     Type* current;     // Текущий образ, предоставляемый компоненту
     Type* undo;        // Предыдущий образ
public:
     ImagePtr() : undo(NULL), current(new Type) {}
     ImagePtr(const ImagePtr<Type>& ip)
          : current(new Type(*(ip.current))), undo(NULL) {}
     ~ImagePtr() { delete current; delete undo; }
     ImagePtr<Type>& operator=(const ImagePtr<Type>& ip)
     {
          if (this != &ip) {
               delete current;
               current = new Type(*(ip.current));
          }
          return *this;
     }
     void Snapshot()
     {
          delete undo;       // На случай, если был старый образ
          undo = current;
          current = new Type(*undo);
     }
     void Commit()
     {
          delete undo;
          undo = NULL;
     }
     void Rollback()
     {
          if (undo != NULL) {
               delete current;
               current = undo;
               undo = NULL;
          }
     }
     Type* operator->() const { return current; }
};
```

Указатель всегда возвращает «текущий» образ как значение оператора `->`, однако за кулисами он прячет предыдущую версию указываемого объекта. Функция `Shapshot()` создает образ с помощью конструктора копий указываемого объекта. Если клиент передумает и захочет отказаться от изменений, он вызывает функцию `Rollback()`; если изменения его устраивают, он вызывает функцию `Commit()`.

Конструктор копий и оператор = поддерживают семантику ведущих указателей, но не создают снимков во время конструирования или сразу же после присваивания. Помните: нас интересует состояние *указываемого объекта*, а не *указателя*. Когда мы создаем новую копию объекта для поддержания семантики ведущего указателя, для этой копии еще не существует предыдущих образов, которые необходимо отслеживать.

Деструктор предполагает, что если клиент не вызвал функцию `Commit()`, что-то было сделано неверно, и уничтожает копию. Специалисты по базам данных — отъявленные пессимисты; если происходит что-то непредвиденное, они всегда предполагают самое худшее. Во многих приложениях при уничтожении указателя вместо `Rollback()` следовало бы просто вызвать `Commit()`, предполагая, что до настоящего момента все происходило вполне разумно.

Стеки образов

Для многоуровневой отмены вам может понадобиться стек предыдущих образов. Один из вариантов реализации — хранить стек в каждом указателе образов. В следующем фрагменте предполагается, что у вас имеется параметризованный класс `Stack` с функциями `Empty()`, `Push()`, `Pop()` и `DeleteAll()`. Функция `Pop()` возвращает верхний элемент стека или `NULL`, если стек пуст. Функция `DeleteAll()` опустошает стек и уничтожает все объекты по мере их извлечения. Каждый указатель хранит стек предыдущих образов. Если стек пуст, значит, образы не создавались. Если стек не пуст, в его нижней части находится исходный объект. Функция `Rollback()` находит и восстанавливает этот объект. Конструктор копий и оператор = работают так же, как и в описанном выше простейшем указателе образов.

```
template <class Type>
class ImageStackPtr {
private:
     Type* current;     // Текущий образ, предоставляемый клиенту
     Stack<Type> history;    // Предыдущие образы
public:
     ImageStackPtr() : current(new Type) {}
     ImageStackPtr(const ImageStackPtr<Type>& ip)
          : current(new Type(*(ip.current))) {}
     ~ImageStackPtr() { delete current; }
     ImageStackPtr<Type>& operator=(const ImageStackPtr<Type>& ip)
     {
          if (this != &ip) {
               history.DeleteAll();
               delete current;
               current = new Type(*(ip.current));
          }
          return *this;
     }
     void PushImage()
     {
          history.Push(current);
          current = new Type(*current);
     }
     void Commit() { history.DeleteAll(); }
     void PopImage()    // Вернуться на один уровень
     {
          if (!history.Empty()) {
               delete current;
               current = history.Pop();
```

```
            }
        }
        void Rollback()    // Вернуться к самому старому образу
        {
            Type* old = history.Pop();
            Type* oldere = NULL;
            if (old != NULL) {   // Хотя бы один раз
                while ((older = history.Pop()) != NULL) {
                    delete old;
                    old = older;
                }
                delete current;
                current = old;
            }
        }
        Type* operator->() const { return current; }
    };
```

Хранение отдельного стека в каждом указателе оправданно для транзакций, в которых участвует небольшое количество объектов. Но если одной транзакции приходится отслеживать множество обновляемых объектов, кучу мелких стеков лучше объединить в один большой. Мы сделаем это позднее, когда речь пойдет о транзакциях.

Образы автоматических объектов

Концепцию образа можно немного обобщить, чтобы она распространялась не только на объекты, созданные оператором `new` и обслуживаемые *-указателями, но и автоматические объекты. Автоматическими считаются стековые объекты, а также переменные и компоненты базовых классов вмещающего объекта независимо от того, выделялась память под вмещающий объект динамически или нет.

```
    template <class Type>
    class AutoImage {
    private:
        Type current;
        Type image;
        bool have_image;   // Истина, если образ существует
    public:
        AutoImage() : have_image(false) {}
        AutoImage(const AutoImage<Type>& ai)
            : current(ai.current), image(), have_image(false) {}
        AutoImage<Type>& operator=(const AutoImage<Type>& ip)
        {
            if (this != &ip) {
                current = ip.current;
                have_image = false;
            }
            return *this;
        }
        AutoImage<Type>& operator=(const Type& t)
        {
            current = t;
```

```
            return *this;
        }
        operator Type&() { return current; }
        void Snapshot()
        {
            image = current;
            have_image = true;
        }
        void Commit() { have_image = false; }
        void Rollback()
        {
            current = image;
            have_image = false;
        }
        bool HaveImage() { return have_image; }
    };
```

Этот шаблон работает со всеми классами, которые удовлетворяют двум условиям:

1. Тип, используемый в качестве параметра, имеет конструктор без аргументов. Он используется в конструкторе `AutoImage` для инициализации `current` и `image`.
2. Тип, используемый в качестве параметра, допускает присваивание с помощью оператора = по умолчанию, предоставленного компилятором, или перегруженного варианта для данного типа. Используется в функциях `Shapshot()` и `Rollback()`.

Все встроенные типы (такие как `int` и `double`) удовлетворяют этим условиям. Подходят и другие классы, имеющие конструктор без аргументов и рабочий оператор =. Чем дольше я имею дело с C++, тем чаще мне кажется, что нарушение этих требований — проявление злостного непрофессионализма, за которое следует наказывать парой лет каторжного программирования на BASIC. Заодно я бы издал закон о том, чтобы конструкторы копий *всегда* работали так, как им положено.

Конструктор копий `AutoImage` следует примеру `ImagePtr` и `ImageStackPtr` — он использует конструктор без аргументов для создания фиктивного объекта `image` и присваивает `have_image` значение `false`. Оператор = делает то же самое, однако в нем не удается найти удобный способ уничтожить объект переменной `image`. Мы выбираем меньшее из двух зол — объект остается без изменений и попросту игнорируется, поскольку переменная `have_image` равна `false`. Если вас это не устраивает и вы действительно хотите оставить объект `image` неинициализированным до тех пор, пока в нем не появится настоящий образ, и уничтожить его после присвоения `false` переменной `have_image`, имеются два возможных решения:

1. Изменить тип `image` с `Type` на `Type*` и выделять для него память оператором `new`. Это увеличит накладные расходы по сравнению с автоматическими объектами, однако вы сможете в полной мере контролировать процесс создания и уничтожения.
2. Воспользоваться идиомой «виртуальных конструкторов» из главы 13. Не вдаваясь в подробности, скажу, что это позволит вам объявить `image` чем-то приятным для глаза — например, `unsigned char image(sizeof Type)` — нежели вызывать конструктор и деструктор Type вручную. Компиляторы C++ недолюбливают подобные фокусы, поэтому, прежде чем пускаться на авантюры, внимательно прочитайте главу 13.

Если `AutoImage` будет использоваться только для структур или классов, добавьте оператор `->`:

```
    Type* operator->() { return &current; }
```

Обратите внимание: в отличие от предыдущих версий `->` этот оператор не может быть константной функцией, поскольку `current` находится внутри `*this` и мы не можем гарантировать, что `->` не будет использоваться для обращений к неконстантным функциям `current`.

Следующий класс демонстрирует возможное использование этого шаблона. Вмещающему объекту Foo незачем создавать свой образ, как в предыдущих указателях на объекты, поскольку все его переменные способны поддерживать свои образы по отдельности.

```
class Foo {
private:
     AutoImage<int> some_integer;
     AutoImage<Bar> bar;
public:
     void Rollback()
     {
          some_integer.Rollback();
          bar.Rollback();
     }
     void Commit()
     {
          some_integer.Commit();
          bar.Commit();
     }
     void Snapshot()
     {
          some_integer.Snapshot();
          bar.Snapshot();
     }
     int ProvideInt() const { return some_integer; }
     void ChanheInt(int new_value)
     {
          if (!some_integer.HaveImage())
               some_integer.Snapshot();
          int&(some_integer) = new_value;
     }
     const Bar& ProvideBar() const { return bar; }
     Bar& UpdateBar()
     {
          if (!bar.HaveImage())
               bar.Shapshot();
          return Bar&(bar);
     }
};
```

Предполагается, что Bar соответствует необходимым условиям. Последние четыре функции перед тем, как обновлять переменную, создают «моментальный снимок» объекта. Для int получение копии по определению является константным по отношению к копируемой переменной. При вызове функции, изменяющей значение переменной, настает время делать снимок. Для работы с другой переменной, bar, предоставляется как константная, так и неконстантная функция. Конечно, хотелось бы просто перегрузить функцию ProvideBar(), чтобы одна перегруженная версия возвращала const Bar&, а другая — неконстантный Bar&, но тогда их сигнатуры будут совпадать. Помните: две функции не могут иметь одинаковые имена и аргументы и отличаться только типом возвращаемого значения. Я никогда не понимал этого ограничения C++, которое запрещает создавать константную и неконстантную версию оператора ->:

```
const Type* operator->() const;      // Снимок не создается
Type* operator->() const;            // Создает снимок
```

Конечно, это намного упростило бы жизнь, но назвать эти загадочные ограничения бесполезными нельзя — они дают знатокам C++ хорошую тему для разговоров на семинарах с коктейлями.

Раз уж речь зашла об ограничениях C++, упомяну еще об одном. Взгляните на приведенный выше код класса Foo. Работа некоторых его функций сводится к вызову одной и той же функции для всех переменных класса и в более общем случае — базовых классов. Скажем, Foo::Commit() просто вызывает Commit() для всех переменных. Весь повторяющийся код приходится писать вручную; в языке сильно не хватает макросредств, которые бы позволяли сказать: «Вызвать функцию Commit() для каждой переменной класса». Компилятор знает, как составить список такого рода (и использует его в конструкторах), но вам ни за что не скажет.

Образы указателей

У шабона AutoImage есть одно довольно занятное применение — им можно воспользоваться для создания образов *-указателя. В некоторых ситуациях не хочется создавать лишние копии указываемого объекта только чтобы следить за тем, на что ссылался указатель в прошлой жизни. Собственно, дело обстоит так каждый раз, когда указатель не является ведущим. Указатель также помогает следить за объектами, которые были созданы или уничтожены в процессе транзакции.

```
AutoImage<Foo*> f;
```

Теперь вы можете восстановить состояние указателя f в начале транзакции, в том числе и NULL. Тем не менее, существует веский довод в пользу создания специализированного шаблона для *-указателей — необходимость перегрузки оператора ->, чтобы указатель образов можно было использовать в левой части выражений (что-нибудь типа ptr->MemberOfPointer();). Для *-указателей AutoImage похож на глупые указатели, с которыми мы расправились в начале главы 5. Следующий шаблон больше напоминает обычные умные (но не ведущие!) указатели.

```
template <class Type>
class PtrImage {
private:
    Type* current;
    Type* image;
    bool have_image;   // Истина, если образ существует
public:
    PtrImage() : current(NULL), image(NULL), have_image(false) {}
    PtrImage(const PtrImage<Type>& pi)
        : current(pi.current), image(NULL), have_image(false) {}
    PtrImage<Type>& operator=(const PtrImage<Type>& pi)
    {
        if (this != &pi)
            current = pi.current;
        return *this;
    }
    PtrImage<Type>& operator=(Type* t)
        { current = t; return *this; }
    operator Type*() { return current; }
    Type* operator->() const { return current; }
    bool operator!() { return current == NULL; }
    void Snapshot()
    {
        image = current;
        have_image = true;
    }
    void Commit() { image = NULL; have_image = false; }
```

```
        void Rollback()
        {
            if (have_image) {
                current = image;
                have_image = false;
            }
        bool HaveImage() { return have_image; }
    };
```

Если вам захочется еще немного автоматизировать этот шаблон, добавьте вызовы `Snapshot()` в обе операторные функции `operator=()`.

Указателей расплодилось слишком много, и терминология начинает запутываться. Термин «указатель образов» обозначает указатель, в котором содержится несколько образов объекта (и который почти всегда является ведущим), тогда как термин «образы указателей» относится к классу наподобие показанного выше, в котором хранится несколько предыдущих значений самого указателя.

Комбинации и вариации

В нашей коллекции скопилось уже немало «строительных блоков», а возможности их комбинирования безграничны. Выберите по одному варианту в каждой позиции, и вы получите некое специализированное решение для работы с образами:

- простые/ведущие указатели;
- образы автоматические/созданные оператором `new`;
- один образ/стеки образов;
- образы объектов/ образы указателей;
- спонтанное создание образа/ ручной вызов функции `Snapshot()`.

Приведенный список ни в коем случае не исчерпывает всех возможных вариантов. Скорее это представительный перечень концепций, сосредоточенных вокруг общей темы — управления образами. Все концепции ориентированы на C++ с его уникальным синтаксисом и семантикой (в частности, конструкторами и операторами). Пошевелите мозгами и подумайте, как эти концепции подходят к вашей конкретной задаче. Именно этим мы и займемся в оставшейся части этой главы.

Транзакции и отмена

Решение проблемы транзакций в значительной степени связано с проблемами отмены и многопоточных итераторов, поэтому сначала мы поговорим о транзакциях. Итак, мы хотим предотвратить обновление некоторого объекта более чем одной транзакцией. Похоже, в решении этой задачи нам помогут указатели образов. Давайте посмотрим свежим взглядом на обобщенный указатель образов, приведенный в начале главы:

```
    template <class Type>
    class ImagePtr {
    private:
        Type* current;     // Текущий образ, предоставляемый компоненту
        Type* undo;        // Предыдущий образ
    public:
        ImagePtr();
        ImagePtr(const ImagesPtr<Type>& ip);
        ~ImagePtr();
        ImagePtr<Type>& operator=(const ImagePtr<Type>& ip);
        void Snapshot();
        void Commit();
```

```
    void Rollback();
    Type* operator->() const;
};
```

Для мира транзакций придется внести ряд изменений:

- Объект в любой момент времени может быть заблокирован не более чем одной транзакцией.
- Объект не может быть изменен при снятой блокировке.
- Заблокировнный объект может быть измене лишь объектом, который принадлежит транзакции, установившей блокировку.

Следовательно, нам придется создать некоторое представление для транзакции, а заодно — поразмять мышцы C++ и построить соответствующую семантику. Транзакции будут представлены классом `Transaction`. Для блокировок мы воспользуемся специальным обновляющим указателем. Иначе говоря, обычные клиенты работают с умным указателем, не допускающим обновления, а клиенты транзакции-владельца получают доступ к другому указателю с возможностью обновления. Ниже приведена прямолинейная (хотя необязательно самая эффективная) реализация этой архитектуры. Позднее мы снимем эти упрощающие ограничения и расширим архитектуру:

1. Нас интересует только отмена изменений в существующих объектах, а не отмена создания и уничтожения объектов в процессе транзакции.
2. Вопрос о том, когда именно должна устанавливаться блокировка объекта выходит за рамки описанной упрощенной архитектуры.

Транзакции и блокировки

В действительности транзакция представляет собой нечто иное, чем коллекцию указателей образов, в которой имеется несколько функций для перебора. Одна из трудностей состоит в том, что одна транзакция может обновлять любое число объектов, относящихся к различным типам. Следовательно, класс транзакции должен быть расписан так, чтобы он мог работать с любыми типами — похоже, речь идет об абстрактном базовом классе.

```
// В файле Transaction.h
class Lock {
friend class Transaction;
protected:
    Transaction* transaction;       // Транзакция, которой принадлежит this
    Lock() : transaction(NULL) {}
    void RegisterLock(Transaction* t)
    {
        if (transaction != NULL) {
            // Конфликт - уже имеется другой владелец
            cerr << "Lock::RegisterLock - already locked" << endl;
        }
        else {
            t->AddLock(this);
            transaction = t;
        }
    }
    virtual ~Lock() {}
    virtual void Rollback() = 0;
    virtual void Commit() = 0;
};
class Transaction {
friend class Lock;      // Чтобы предоставить доступ к AddLock()
```

```
private:
    SafeSet<Lock>* locks;
    void AddLock(Lock*);    // Включить блокировку в транзакцию
public:
    ~Transaction();
    void Commit();          // Закрепить все образы
    void Rollback();        // Отменить все образы
    bool OwnsLock(Lock*);   // Истина, если блокировка
                            // принадлежит транзакции
};
```

Класс Transaction поддерживает коллекцию блокировок с помощью гипотетического шаблона Collection. Функция RegisterLock() включена в базовый класс Lock и потому может обратиться к закрытой функции AddLock() класса Transaction. Дружба не наследуется, поэтому объявление другом класса Lock не делает друзьями его производные классы. Реализации выглядят довольно просто.

```
void Transaction::AddLock(Lock* lock)
{
    *locks += lock;    // Использует перегрузку += для коллекции
}
void Transaction::Commit()
{
    SafeSetIterator<Lock>* iter = locks->Iterator();
    while (iter->More())
        iter->Next()->Commit();
    delete iter;
}
void Transaction::Rollback()
{
    SafeSetIterator<Lock>* iter = locks->Iterator();
    while (iter->More())
        iter->Next()->Rollback();
    delete iter;
}
bool Transaction::OwnsLock(Lock* lock)
{
    return *locks >= lock;
}
```

Предполагается, что шаблон Collection содержит функцию DeleteAll() для удаления всех объектов; что перегруженный оператор += (операторная функция operator+=(Type*)) включает элемент в коллекцию; что перегруженный оператор >= определяет принадлежность к коллекции, а функция Iterator() возвращает вложенный итератор. Это обобщенные условия; используйте те, которые действуют в вашем случае.

Класс ConstPtr

Классы, производные от Lock, должны ссылаться на нечто близкое к указателям, доступным только для чтения, с которыми на страницах книги вы уже познакомились.

```
template <class Type>
class LockPtr;     // Ссылка вперед на класс, производный от Lock
```

```
template <class Type>
class ConstPtr {
friend class LockPtr<Type>;
private:
    Type* old_image;          // Образ перед транзакцией
    LockPtr<Type>* lock;      // Текущая блокировка, если она есть
    ~ConstPtr() { delete old_image; }
    ConstPtr<Type>& operator=(const ConstPtr<Type>& cp)
        { return *this; }    // Присваивание не разрешается
public:
    ConstPtr() : old_image(NULL), lock(NULL) {}
    ConstPtr(const ConstPtr<Type>& cp)
        : old_image(new Type(*(cp.old_image))), lock(NULL) {}
    const Type* operator->() const { return old_image; }
    LockPtr<Type>& Lock(Transaction* t);
};
template <class Type>
LockPtr<Type>& ConstPtr<Type>::Lock(Transaction* t)
{
    if (lock != NULL && !t->OwnsLock(lock))
        // Конфликт – уже имеется другой владелец
    else {
        lock = new LockPtr<Type>(t, this);
        return *lock;
    }
}
```

Новый объект ConstPtr можно сконструировать на базе старого (хотя новый создается без блокировки), однако нам придется внести изменения для присваивания, которое разрешено только для LockPtr, но не для ConstPtr. Для этой цели мы определяем фиктивный оператор = и делаем его закрытым, чтобы до него никто не добрался. Поскольку указатель является ведущим, его удаление приводит и к удалению указываемого объекта (самое радикальное изменение, которое только можно представить). По этой причине конструктор также объявлен закрытым, чтобы никто не попытался удалить ConstPtr.

Конфликт в функции Lock можно обработать разными способами:

- Инициировать исключение.
- Изменить интерфейс и возвращать вместе с блокировкой флаг, показывающий, успешно ли прошла блокировка.
- Возвращать LockPtr<Type>* со значением NULL, свидетельствующем о неудачной блокировке.
- Возвращать конфликтную блокировку с перегруженным оператором !, с помощью которого можно проверить правильность блокировки.
- Предоставить отдельную функцию CanLock(Transaction*), которая возвращает логическое значение.

Выбор зависит от стиля. Вариант с исключением не так уж очевиден; неудача при блокировке представляет собой вполне разумный исход.

Класс LockPtr

Ага! Мы подошли к центральной идее всей концепции — указателям, которые разрешают обновление указываемого объекта. Предтранзакционный (предназначенный для отмены) образ хранится в ConstPtr, а текущий обновленный образ доступен только через LockPtr. Класс LockPtr содержит уже знакомые функции Rollback() и Commit(). В функции Snapshot() нет необходимости, поскольку LockPtr при необходимости создает образы в операторе ->.

```
template <class Type>
class LockPtr : public Lock {
friend class ConstPtr<Type>;
private:
     ConstPtr<Type>* master_ptr;
     Type* new_image;
     Transaction* transaction;
     LockPtr(Transaction* t, ConstPtr<Type>* cp);
     virtual ~LockPtr();
     virtual void Rollback();
     virtual void Commit();
public:
     Type* operator->() const { return new_image; }
};
template <class Type>
LockPtr<Type>::LockPtr(Transaction* t, ConstPtr<Type>* cp)
     : transaction(t), master_ptr(cp), new_image(new Type(*(cp->old_image)))
{
}
template <class Type>
LockPtr<Type>::~LockPtr()
{
     // В сущности происходит откат
     delete new_image;          // Отказаться от изменений
     master_ptr->lock = NULL;   // Оставить ConstPtr
}
template <class Type>
void LockPtr<Type>::Rollback()
{
     delete new_image;
     new_image = new Type(*(master_ptr->old_image));
}
template <class Type>
void LockPtr<Type>::Commit()
{
     delete master_ptr->old_image;
     master_ptr->old_image = new_image;    // Переместить в master_ptr
     new_image = new Type(*new_image);     // Нужна новая копия
}
```

Деструктор объявлен закрытым, чтобы никто не мог напрямую удалить LockPtr. Вместо этого транзакция-владелец должна сделать это через базовый класс Lock. Функции Rollback() и Commit() объявлены виртуальными, чтобы с их помощью можно было решать задачи, относящиеся к

конкретному типу (например, создание и уничтожение образов). Обе функции после завершения оставляют ConstPtr заблокированным.

Создание и уничтожение объектов

Пора заполнить кое-какие пробелы. Раз уж наши транзакции достаточно сложны, чтобы для них была оправдана вся эта возня, они наверняка будут создавать или уничтожать объекты. Операции создания и уничтожения также должны быть отменяемыми. Если объект создавался, операция отмены должна его уничтожать, а если уничтожался — возвращать его из мертвых. Для этого придется внести изменения как в класс ConstPtr, так и в класс LockPtr. Мы уже сделали первый шаг в этом направлении, объявив деструктор ConstPtr закрытым, чтобы им могли воспользоваться только ConstPtr или его друзья. Давайте разберемся с оставшимися проблемами.

Изменения в классе ConstPtr

Несомненно, создание указываемого объекта представляет собой изменение и потому должно осуществляться через LockPtr. Но для того чтобы получить LockPtr, мы должны сначала иметь ConstPtr и его функцию Lock(). Следовательно, только что описанный конструктор ConstPtr работать не будет — он создает уникальный объект перед вызовом Lock(). ConstPtr должен находиться в состоянии NULL до тех пор, пока LockPtr не выделит память под объект и не закрепит эти изменения. В ConstPtr необходимо внести следующие изменения:

- В конструкторе без аргументов присваивать переменной old_image значение NULL.
- Добавить оператор !, который проверяет, равен ли адрес значению NULL.
- Инициировать исключение в операторе ->, если адрес равен значению NULL.
- Либо запретить копирование, либо присвоить копии old_image значение NULL.

Проблема с обычным конструктором копий ConstPtr заключается в том, что он может создать новую копию указываемого объекта, но не позволит отменить ее создание. Ниже приводится новая версия конструктора ConstPtr. Определения функций, не изменившиеся по сравнению с показанной выше упрощенной версией не показаны.

```
private:
    ConstPtr(const ConstPtr&) : old_image(NULL), lock(NULL) {}
public:
    ConstPtr() : old_image(NULL), lock(NULL) {}
    bool operator!() { return old_image == NULL; }
    const Type* operator->() const
    {
        if (old_image == NULL)
            // Исключение
        return old_image;
    }
```

Изменения в классе LockPtr

Отныне LockPtr предстоит выполнять намного больше работы:

- Он должен при необходимости создавать указываемый объект по требованию. Для этого в него будет добавлена функция Make().
- Оператор -> должен инициировать исключение, если адрес равен NULL.

Ниже приведены определения только изменившихся функций.

```
// В объявлении LockPtr
public:
    void Make();       // Создать новый указываемый объект
    void Destroy();    // Уничтожить указываемый объект
```

```
// Изменившиеся определения
template <class Type>
LockPtr<Type>::LockPtr(Transaction* t, ConstPtr<Type>* cp)
     : transaction(t), master_ptr(cp),
       new_image(cp->old_image != NULL ? new Type(*(cp->old_image)) : NULL)
{
}
template <class Type>
void LockPtr<Type>::Commit()
{
     delete master_ptr->old_image;
     master_ptr->old_image = new_image;
     if (new_image != NULL)
          new_image = new Type(*new_image);
}
template <class Type>
Type* LockPtr<Type>::operator->() const
{
     if (new_image == NULL)
          // исключение
     return new_image;
}
template <class Type>
void LockPtr<Type>::Make()
{
     delete new_image;       // Если new_image не равен NULL
     new_image = new Type;   // Новый пустой объект
}
template <class Type>
void LockPtr<Type>::Destroy()
{
     delete new_image;
     new_image = NULL;
}
```

Функция `Make()` соблюдает семантику присваивания, что позволяет вызвать ее для существующего указываемого объекта. При этом объект, на который в данный момент ссылается `LockPtr`, уничтожается и заменяется новым пустым объектом.

Упрощенное создание объектов

Объекты теперь создаются в три этапа:

1. Создать `ConstPtr`, указывающий на `NULL`.
2. Запросить у него `Lock`.
3. Потребовать у `Lock` создать объект функцией `Make()`.

Конечно, это произведет впечатление на ваших коллег и лишний раз докажет вашу техническую квалификацию, но… они косо посмотрят на вас и вернутся к оператору `new`. Ведь он справляется с задачей за один этап, а нас окружают занятые, очень занятые люди. Существует несколько способов свести процесс создания к одному этапу, и самый простой из них — включить в `ConstPtr` другой конструктор.

```
ConstPtr<Type>::ConstPtr(Transaction* t) : old_image(NULL), lock(NULL)
{
    LockPtr<Type>& lp = Lock(t);
    lp.Make();
}
```

Последующий вызов `Lock()` возвращает уже созданный `LockPtr`.

Отмена

Все до смешного просто. Если вы внимательно следили за тем, как развивается тема транзакций, то для безопасной и универсальной реализации отмены вам потребуется совсем немного — включить необходимые фрагменты из вашей любимой библиотеки классов с графическим интерфейсом пользователя. Для создания и манипуляций со структурами данных транзакций идеально подходят классы *событий*. Объект-событие создается для выполнения некоторой команды или внесения изменений. Например, пользователь выбирает команду `Delete` в меню `Edit` или нажимает клавишу `Delete` на клавиатуре. Объект создается на базе класса, который прекрасно разбирается в удалениях. В дальнейшем этот объект не только вносит затребованные изменения, но и обеспечивает работу команды `Undo` меню `Edit`. Создайте транзакцию в его конструкторе, воспользуйтесь транзакцией для сохранения всех изменений, а затем — для поддержки отмены.

Варианты

Мы не будем пережовывать дополнительный код, относящийся к транзакциям. Если вы еще не уловили общий принцип, бессмысленно подливать масло в огонь, а если уловили, то без труда реализуете все, о чем говорится ниже. Так что я ограничусь лишь замечаниями, в которых описываются некоторые важные вариации на данную тему.

Вложенные блокировки

Чтобы вся методика имела хоть какой-то смысл, мы должны спрятать ключи от указываемых объектов и заставить всех работать с `ContrPtr` и `LockPtr`. Это относится не только к изменениям, вносимым в указываемые объекты непосредственно клиентами, но и к тем, которые один указываемый объект вносит в другой. Таким образом, в общем случае указываемые объекты должны обращаться друг к другу через `ConstPtr` и `LockPtr`, как и клиенты.

```
class A {
private:
    ConstPtr<B>& b;    // ConstPtr, как выше
    // И т.д.
};
```

Если экземпляр класса `A` захочет вызвать неконстантную функцию `b`, он должен сначала получить `LockPtr` для `b`. Таким образом, класс `A` должен знать, какую транзакцию он выполняет при попытке изменения `b`, и эта информация должна поступить от клиента. Для простоты назовем объекты, к которым клиенты обращаются напрямую, *первичными*, а объекты, косвенно обновляемые указываемым объектом, — *вторичными* по отношению к указываемому объекту (один и тот же объект может одновременно быть и первичным, и вторичным).

Если вторичный объект инкапсулируется в первичном (то есть вторичный объект невидим для окружающего мира), это правило можно обойти. Конструктор копий первичного объекта должен продублировать вторичный объект, а оператор `=` первичного объекта должен продублировать вторичный объект правостороннего выражения. При соблюдении этих условий логика создания образов в `ConstPtr` и `LockPtr` будет правильно работать с инкапсулированными объектами. Это происходит автоматически в тех случаях, когда вторичный объект внедряется в первичный (то есть когда в каждом экземпляре А объект `B` хранится не как указатель, а как обычная переменная класса):

```
class A {
private:
```

```
        B b;     // Внедренный объект
    };
```

Компилятор будет автоматически вызывать конструктор копий B при каждом копировании A, и оператор = класса B — при выполнении присваивания.

Предположим, класс A должен заблокировать b. Откуда берется Transaction*? Разумеется, извне A — то есть от клиента. Впрочем, эту задачу можно решить тремя способами:

- Передавать Transaction* в каждую функцию A, которой может потребоваться вызвать неконстантную функцию B.
- Один раз передать Transaction* в A во время блокировки и сохранить в переменной класса.
- Сделать указатель ConstPtr класса A еще более гениальным — научить его блокировать вторичные объекты по поручению A.

На последнем решении следует повесить табличку: «Только для профессиональных каскадеров. Не уверен — не пытайся». Тем не менее, в некоторых ситуациях лучшая реализация транзакций подразумевает использование коммерческих классов указываемых объектов. Этой стратегией можно воспользоваться даже для того, чтобы разрешить A хранить его обожаемый B* и ничего не знать о ConstPtr и LockPtr. Поумневшие ConstPtr и/или LockPtr класса A могут без его ведома менять адреса при создании образов и закреплении изменений. В дальнейшем мы еще неоднократно встретимся с концепцией объектов, которые описывают другие объекты и/или манипулируют ими. Ближе к концу мы снова вернемся к проблеме межобъектных ссылок.

Взаимные блокировки и очереди

Если в любой момент времени может существовать не более одной транзакции, все замечательно. Если же с вашими объектами могут возиться сразу несколько транзакций, придется сделать еще несколько шагов. Возможны разные причины — например, ваше приложение-сервер может обслуживать несколько пользователей, или вы напишете графическое приложение с контекстной отменой, в котором необходимо запоминать отменяемую команду для нескольких окон. Наконец, остается еще одно препятствие — необходимо продумать поведение вашей программы в ситуации, когда она пытается заблокировать уже заблокированный объект.

Консервативная блокировка

При *консервативной блокировке* (*conservative locking*) транзакция блокирует все объекты, которые ей могут понадобиться, до внесения каких-либо изменений. Если заблокировать все объекты не удается, транзакция либо ждет, пока они станут доступными, либо поднимает руки и сообщает пользователю, что при всем уважении к нему просьба отклоняется. Одна из возможных реализаций консервативной блокировки заключается в том, чтобы субклассировать Transaction и попытаться заблокировать все необходимое в конструкторе производного класса (при наличии стандартной обработки исключений) или в его отдельной инициализирующей функции (при отсутствии такой обработки).

Агрессивная блокировка

При *агрессивной блокировке* (*aggressive locking*) транзакция может начинаться в любой момент и блокировать объекты по мере необходимости. Я бы назвал ее «своевременной» блокировкой, поскольку блокируются только непосредственно обновляемые объекты и это происходит перед первым обновлением.

Очереди и взаимные блокировки

Вопрос о том, стоит ли создавать очереди запросов на блокировку, составляет отдельный аспект дизайна. Если очереди не поддерживаются и транзакция попытается заблокировать объект, ранее заблокированный другой транзакцией, она заканчивается неудачей. При консервативной блокировке транзакция вообще не начинается, а при аггрессивной она возвращается к прежнему состоянию, скрещивает руки на своей двоичной груди и сурово смотрит на пользователя. В некоторых приложениях это вполне нормально, но в действительности это решение из тех, о которых специалисты из службы поддержки могут лишь мечтать — ведь оно гарантирует им постоянную работу. Чтобы не

отказываться от второй транзакции, обычно стоит подождать снятия блокировки с объекта текущей транзакцией.

При отсутствии очередей вам не придется беспокоиться о ситуациях взаимной блокировки (когда A ожидает B, а B ожидает A). Если транзакция запросила блокировку и не смогла ее получить, она уничтожается. Если очереди поддерживаются, код блокировки должен определить, принадлежит ли этой транзакции какие-либо блокировки, которых дожидаются остальные, и если принадлежат, избавить одну из транзакций от бесконечного ожидания.

Происходящее оказывается в опасной близости от точки, в которой вам приходится либо разбивать свое приложение на несколько подзадач и поручать операционной системе их планирование, либо воспользоваться одной из коммерческих библиотек с поддержкой многопоточности. Так или иначе, в этой области мы не узнаем ничего принципиально нового, относящегося к C++, и поэтому не будем развивать эту тему. Об очередях, блокировках и многопоточности написано немало хороших книг, вы наверняка найдете их в разделе «Базы данных» своего книжного магазина.

Многоуровневая отмена

Семантика транзакций довольно легко распространяется и на многоуровневую отмену (если вспомнить концепции StackPtr, о которых говорилось выше). Существует два основных варианта реализации.

Класс LockPtr со стеками образов

Самый прямолинейный вариант реализации многоуровневой отмены для транзакций — включение стека старых образов в каждый LockPtr. Эта идея позаимствована из рассмотренного выше кода StackPtr. Тем не менее, она подходит лишь для консервативной блокировки. В случае с агрессивной блокировкой объект может быть впервые заблокирован уже после изменения остальных объектов. Это усложняет отмену нескольких изменений, поскольку стеки разных LockPtr не синхронизируются.

Стеки пар LockPtr/образ

К проблеме можно подойти и иначе — включить в Transaction стек, в котором хранятся пары старый образ/LockPtr. На каждом уровне стека хранятся лишь те LockPtr, которые существовали на момент закрепления. В общем случае это решение работает лучше, к тому же оно чуть более эффективно — вы используете один большой стек вместо множества маленьких.

Оптимизация объема

Другое решение — реорганизация структур данных для сокращения издержек, связанных с хранением незаблокированных ConstPtr (хотя и ценой некоторой потери скорости). ConstPtr лишь незначительно отличается от указателей только для чтения, которые рассматривались в главе 6 и не имели ничего общего с транзакциями: он имеет ссылку на ConstPtr и функцию Lock(). Мы избавимся от первого и внесем некоторые изменения во второе.

Представьте себе глобальную структуру данных (вероятно, хранящуюся в виде статического члена класса Transaction), в которой находится информация о том, какие Transaction блокируют какие ConstPtr. Для каждой пары в таблице содержится соответствующий LockPtr. Каждый раз, когда вызывается функция Lock() класса ConstPtr, она проверяет, присутствует ли this в таблице. Если присутствует, функция сравнивает транзакцию, переданную в качестве аргумента, с находящейся в таблице. Если ConstPtr не находит себя в таблице, он включает в нее новый триплет (ConstPtr, Transaction, LockPtr), а если находит с другой транзакцией — инициирует исключение.

Такая схема оказывается более экономной, чем описанная выше; она не тратит память на значения NULL для всех незаблокированных объектов. Разумеется, она сложенее и медленнее работает — структура данных еще только разогревает двигатель на старте, а косвенной обращение через переменную класса уже пересекает финишную черту.

Возможно, у вас возник вопрос — а почему функция Lock() должна оставаться с ConstPtr? Почему ее нельзя включить в другой класс или даже сделать глобальной функцией? Если мы избавимся от переменной LockPtr* и функции Lock(), ConstPtr превратится в самый обычный указатель только для чтения, который на вопрос о транзакциях лишь тупо смотрит на вопрошающего. Впрочем, так ли это? LockPtr по-прежнему приходится объявлять другом; следовательно, хотя бы тривиальных

изменений не избежать. Более того, Transaction не знает конкретный тип указываемого объекта и потому не может использовать шаблон LockPtr для создания объекта блокировки. Если вы еще не забыли, абстрактный базовый класс Lock создавался именно для этой цели. Увы. Нам хотелось бы оставить ConstPtr в счастливом неведении… но не суждено.

Несколько прощальных слов

Реализация полной схемы обработки транзакций занимает не так уж много места, но от этого она не становится проще. Изменения приходится вносить практически в любой класс, который ссылается на другие классы, поскольку большинство указателей должно соответствовать соглашениям ConstPtr/LockPtr. Впрочем, это не демонстрирует полную безнадежность подобных попыток на C++, а лишь подчеркивает важность соблюдения двух основных принципов:

1. Используйте умные указатели, даже если вы не уверены, что это нужно.
2. Фанатично следите за тем, чтобы константные обращения отделялись от неконстантных.

Если ваш код будет изначально устроен таким образом, подобная расширенная архитектура реализуется на порядок проще.

В этом кроется один из мистических принципов C++, о которых говорилось выше — вы должны понять не то, как язык справляется с конкретной проблемой, а то, как он обеспечивает построение полноценной, надежной программы. Некоторые инструменты C++, поддерживающие подобную архитектуру, не имеют аналогов в других языках (взять хотя бы конструкторы копий, операторы = и ->). Благодаря этим синтаксическим и семантическим странностям сложные библиотеки классов становятся более понятными и полезными. Только подумайте, насколько бы все усложнилось, если бы мы не могли перегрузить оператор -> в ConstPtr и LockPtr. При правильном выборе идиом C++ не подведет ни на скользкой дороге, ни в плохую погоду.

Наконец, помните о том, что эта глава предназначалась для тренировки творческого воображения — мы хотели узнать, куда нас заведет концепция умного указателя, если применить ее на свежем материале. Ответ: довольно далеко. После добавления некоторых идиом и принципов дизайна, рассмотренных в следующих главах, подобную архитектуру будет легче воплотить в реальной программе.

Часть 3

Снова о типах

О проектировании иерархий классов написано столько, что я даже не знаю, с чего начать. Большая часть подобной литературы не имеет отношения к специфике C++, и ее можно пропустить. Однако из того, что осталось, некоторые темы не получили должного внимания. Среди тем, имеющих особое значение для программирования на C++, — особая разновидность иерархии классов, так называемая гомоморфная иерархия. Существует и другая интересная концепция — для клиента указатель может выглядеть как настоящий, «честный» объект, а не как путь к нему. Эти две темы станут основными в этой части. Заодно мы, как всегда, исследуем ряд боковых тропинок, уводящих в сторону от главной дороги.

Множественная передача

10

Когда речь заходит об иерархии классов, сразу же хочется разразиться трескучей речью об объектно-ориентированном дизайне. Однако я справлюсь с искушением и ограничусь лишь той частью темы, которая развивает потенциал C++, а именно гомоморфными иерархиями. За длинным термином прячется простая концепция — иерархия классов с одинаковым открытым интерфейсом, унаследованным от общего базового класса. Суть проста, но возможности огромны.

Немедленно возникает первый вопрос: как выполнять передачу вызовов функций, когда единственное, что вам известно об аргументах, — все они происходят от некоторого общего предка? «Силовое» решение с конструкцией `switch/case`, нередко встречающееся в реальных программах, обычно удается заменить намного более элегантной, быстрой и простой в сопровождении архитектурой, известной под названием *множественной передачи* (*multiple dispatch*).

В этой и следующей главе мы временно отвлечемся от навязчивой темы — указателей. Поклонники указателей, не отчаивайтесь! В главе 12 гомоморфизм и умные указатели объединятся в новой разновидности умных указателей… настолько умных, что вы даже не будете подозревать об их присутствии.

Гомоморфные иерархии классов

Во главе гомоморфной иерархии классов всегда стоит абстрактный базовый класс, который определяет открытый интерфейс своих предков. Из чисто сентиментальных побуждений я назову этот класс-предок «дедушкой» (`Grandpa`). Как правило, `Grandpa` является *чисто абстрактным классом* — то есть он не содержит ни одной переменной, а все его функции являются виртуальными.

```
class Grandpa {
public:  // Закрытые и защищенные члены отсутствуют
     virtual void Fn1() = 0;
     virtual void Fn2(int) = 0;
};
```

Разумеется, классу `Grandpa` не нужны конструкторы. Наличие чисто виртуальных членов гарантирует, что экземпляры `Grandpa` непосредственно никогда создаваться не будут. Для чисто абстрактных базовых классов я иногда использую другой, неформальный термин — «класс-пенсионер». Вероятно, такие классы делали что-то полезное на ранних стадиях цикла разработки, но теперь они служат всего лишь для абстрактного объединения семейства.

Это еще не обеспечивает гомоморфизма. Все зависит от того, как от `Grandpa` порождаются новые классы. Гомоморфными по отношению к Grandpa называются производные классы, в которые не добавляется никаких открытых членов. Они могут иметь любые закрытые и защищенные члены, но только не новые открытые.

```
class Dad : public Grandpa {
private:
     // О чем папа никогда не рассказывал
```

```
protected:
    // Что было у папы, но не было у дедушки
public:
    virtual void Fn1();
    virtual void Fn2(int);
};
class AuntMartha : public Grandpa {
private:
    // Личная жизнь тетушки Марты
protected:
    // То, что она передала моим кузенам
public:
    virtual void Fn1();
    virtual void Fn2(int);
};
```

Иерархию можно продолжить и дальше (например, `class Me : public Dad`) при условии, что в открытый интерфейс не добавляется новых функций.

Мы рассмотрим множество примеров, демонстрирующих полезность гомоморфных иерархий, но сначала я приведу тройной аргумент в пользу этой концепции.

Взаимозаменяемость производных классов

Тот, кто пришел на объектно-ориентированную вечеринку раньше других, смог войти в историю. Некто Лисков заработал себе имя на следующей идее: если клиент имеет дело с базовым классом, его не должно интересовать, какой из производных классов на самом деле выполняет работу. Вы должны иметь возможность подставить экземпляр любого производного класса вместо экземпляра любого другого производного класса; клиенты базового класса просто пожимают плечами и продолжают работать так, словно ничего не произошло. Это называется «подстановочным критерием Лискова». Знатоки объектно-ориентированного программирования обычно сходятся на том, что это — Хорошая Мысль.

Строго говоря, данный критерий можно выполнить и без гомоморфизма. Если производный класс содержит дополнительные открытые функции, их можно просто не вызывать из клиента базового класса. Постойте-ка… а зачем добавлять открытые функции, если их не использовать? Если в одном производном классе были добавлены одни скрытые функции, а в другом — другие, со временем в вашей программе наверняка отыщется точка, в которой их нельзя свободно поменять местами.

Настоящая опасность заключается втом, чо без выполнения этого критерия клиентам придется думать о производных классах, а не только о базовом классе, который они знают и любят. Если бы в `Dad` присутствовали дополнительные открытые члены, клиента `Grandpa` со временем мог бы спросить свой объект: «Долой притворство — что ты представляешь собой в действительности?» В итоге было бы нарушено столько принципов модульного строения и инкапсуляции, что об этом можно было бы написать целую книгу.

Самый простой способ обеспечить взаимозаменяемость — воспользоваться гомоморфизмом. По крайней мере, для интерфейсов гомоморфизм обеспечивает взаимозаменяемость по определению, поскольку клиентам `Grandpa` не придется беспокоиться о существовании других функций, с которыми им положено работать.

Нормальное наследование

Класс `Grandpa` может содержать не чисто виртуальные функции и переменные и при всем этом обеспечивать полную взаимозаменяемость. Тем не менее, совпадение интерфейсов еще не означает взаимозаменяемости объектов. Приходится учитывать действие вторичных эффектов. Предположим, функция `Fn1()` класса `Grandpa` не является чисто виртуальной:

```
void Grandpa::Fn1()
```

```
{
    // Код, вызывающий вторичные эффекты
}
void Dad::Fn1()
{
    // Код, вызывающий другие вторичные эффекты
}
void AuntMartha::Fn1()
{
    Grandpa::Fn1();
    // Прочее
}
```

Клиент `Grandpa` может полагаться на вторичные эффекты этого класса. Знаю, знаю, инкапсуляция и все такое, на вторичные эффекты полагаться никогда не следует… но давайте спустимся на землю. Функции, которые мы вызываем, выполняют различные действия — скажем, рисуют на экране, создают объекты или записывают информацию в файле. Без этих вторичных эффектов толку от них будет немного. Если `Grandpa` обладает некоторыми встроенными вторичными эффектами, клиенты `Grandpa` могут с полным правом надеяться, что эти эффекты сохранятся во всех производных классах. Но вот `Dad` усомнился в авторитете `Grandpa` и в своем переопределении `Fn1()` не потрудился вызвать `Grandpa::Fn1()`. Вторичные эффекты `Grandpa::Fn1()` пропадают. Рано или поздно это начнет беспокоить клиента `Grandpa`, которые, возможно, ждал от `Dad` совсем иного. А вот `AuntMartha` в свом переопеределении вызывает `Grandpa::Fn1()` и потому сохраняет все вторичные эффекты `Grandpa::Fn1()`. Теперь `AuntMartha` может выполнять любые дополнительные действия в пределах разумного — клиентов `Grandpa` это совершенно не интересует.

Если переопределенная функция вызывает версию базового класса, говорят, что она *нормально наследуется* от этой функции. Не важно, где находится этот вызов — в начале, в конце или середине переопределенной функции. Важно лишь то, что в какой-то момент он все же происходит. Если все переопределенные функции производного класса наследуются нормально, говорят, что весь класс наследуется нормально. Если все производные классы гомоморфного базового класса наследуются нормально и ни один из них не обладает особо вопиющими вторичными эффектами, их можно подставлять вместо друг друга.

Самый простой способ обеспечить взаимозаменяемость — сделать все функции `Grandpa` чисто виртуальными. Это вырожденный случай нормального наследования; если функция базового класса является чисто виртуальной, то все ее вторичные эффекты (которых на самом деле нет) сохраняются по определению.

Инкапсуляция производных классов

Мы все еще не рассмотрели всех причин размещения чисто абстрактного базового класса во главе иерархии. Взаимозаменяемость и нормальное наследование можно обеспечить как с переменными и невиртуальными функциями в `Grandpa`, так и с виртуальными функциями, которые нормально наследуются производными классами. Зачем настаивать, чтобы `Grandpa` был чисто виртуальным базовым классом? Ответ состоит всего из одного слова: *инкапсуляция*. Если клиент имеет дело только с чисто абстрактным базовым классом, содержащим только открытые функции, он получает абсолютный минимум информации, необходимой для использования класса. Все остальное (в том числе и сами производные классы) может быть спрятано от чужих глаз в файле .cpp.

```
// В файле .h
class Grandpa { ... };
// В файле(-ах) .cpp
class Dad : public Grandpa { ... };
class AuntMartha : public Grandpa { ... };
```

Инкапсуляция производных классов — одно из редких проявлений истинного просветления программиста; верный признак того, что автор программы хорошо разбирается в том, что он делает.

Чтобы усилить эффект, закрытые классы можно объявить статистическими и тем самым ограничить их пространтсво имен исходным файлом, в котором они находятся.

С инкапсулированными производными классами связаны определенные проблемы. Например, как создать экземпляры таких классов, как Dad, которые не видны клиенту из файла .h? Эти проблемы легко решаются с помощью идиом, описанных в двух следующих главах. А пока мы продолжим публиковать производные классы в файле .h, зная, что существует возможность их полной инкапсуляции.

Множественная передача

Самый распространенный пример гомоморфной иерархии — набор классов, соответствующих различным видам чисел: целым, комплексным, вещественным и т.д. Класс-предок такой иерархии может называться Number и иметь интерфейс следующего вида:

```
class Number {
public:
    virtual Number operator+(const Number&) = 0;
    virtual Number operator-(const Number&) = 0;
    // И т.д.
};
class Integer : public Number {
private:
    int i;
public:
    Integer(int x) : i(x) {}
    virtual Number operator+(const Number&);
    // И т.д.
};
```

На бумаге все выглядит проще, чем в действительности. Как реализовать Integer::operator+(Number&), если нам не известно, что в скобках находится вовсе не Number, а некоторый производный класс? Для каждой пары типов, участвующих в сложении, существует свой алгоритм. Суммирование Complex + Integer отличается от Integer + Real, которое, в свою очередь, отличается от Integer + ArbitraryPrecisionNumber. Как программе разобраться, какой из алгоритмов следует использовать? Что-что? Кто сказал: «Запросить у аргумента оператора + его настоящий тип»? Немедленно встаньте в угол.

```
class Number {
protected:
    int type;     // Хранит информацию о настоящем типе
    int TypeOf() { return type; }
    // И т.д.
};
// Где-то в программе
    switch (type) {
    case kInteger: ...
    case kComplex: ...
    }
```

Именно этого знания типов мы постараемся избежать. Кроме того, все прямые реализации подобных схем не отличаются особой элегантностью. Вы когда-нибудь видели код, генерируемый компилятором для конструкции switch/case? Ни красоты, ни эффективности. Вместо этого мы объединим знания компилятора о типах с чудесами современной технологии — v-таблицами.

Двойная передача

В обобщенном виде задачу можно представить в виде матрицы, строки которой соответствуют типам левого операнда, а столбцы — всевозможным типам правого операнда. В каждой ячейке матрицы находится конкретный алгоритм для обработки сочетания типов. Чаще всего такая ситуация возникает для гомоморфных иерархий вроде нашей, но вообще типы левого операнда не обязаны совпадать с типами правого операнда.

Конечно, возможны силовые решения — например, запрятать в каждом экземпляре сведения о его типе. Однако более элегантное решение (и обычно более эффективное) решение носит название *двойной передачи (double dispatch)*.

```
class Number {
protected:
    // Диспетчерские функции для оператора +
    virtual Number& operator+(const Integer&) = 0;
    virtual Number& operator+(const Complet&) = 0;
    // И т.д. для всех производных типов
public:
    virtual Number& operator+(const Number&) = 0;
    virtual Number& operator-(const Number&) = 0;
    // И т.д.
};
class Integer : public Number {
private:
    int I;
protected:
    virtual Number& operator+(const Integer&);
    virtual Number& operator+(const Complex&);
public
    Integer(int x) : i(x) {}
    virtual Number& operator+(const Number&);
    // И т.д.
};
Number& Integer::operator+(const Number& n)
{
    return n + *this;  // Поменять местами левый и правый операнд
}
Number& Integer::operator+(const Integer& n)
{
    // Ниже приведен псевдокод
    if (i + n.i слишком велико для int) {
        return ЦелоеСПовышеннойТочностью
    }
    else return Integer(i + n.i);
}
```

С этим фрагментом связана одна нетривиальная проблема, к которой мы вернемся позже, а пока сосредоточьте все внимание на концепции. Она похожа на стереограмму — чтобы скрытая картинка проявилась, вам придется расслабить глаза и некоторое время рассматривать код. Когда клиент пытается сложить два `Integer`, компилятор передает вызов `Integer::operator+()`, поскольку `operator+(Number&)` является виртуальным — компилятор правильно находит реализацию производного класса. К моменту выполнения `Integer::operator+(Number&)` настоящий тип левого

операнда уже известен, однако правый операнд все еще остается загадкой. Но в этот момент наступает второй этап двойной передачи: `return n + *this`. Левый и правый операнды меняются местами, а компилятор приступает к поискам v-таблицы `n`. Однако на этот раз он ищет переопределение `Number::operator+(Integer&)`, так как он знает, что `*this` в действительности имеет тип `Integer`. Это приводит к вызову `Integer::operator+(Integer&)`, поскольку типы обоих операндов известны и можно наконец произвести вычисления. Если вы так и не поняли, что же происходит, прогуляйтесь на свежем воздухе и попробуйте снова, пока не поймете. Возможно, вам поможет следующая формулировка: вместо кодирования типа в целой переменной мы определили настоящий тип `Number` с помощью v-таблицы.

Такое решение не только элегантно. Вероятно, оно еще и более эффективно, чем те, которые приходили вам в голову. Скажем, приходилось ли вам видеть код, генерируемый компилятором для конструкции `switch/case`? Он некрасив и вдобавок куда менее эффективен, чем последовательное индексирование двух v-таблиц.

Несмотря на всю элегантность, двойная передача довольно дорого обходится по объему кода и сложности:

- Если у вас имеется m производных классов и n операторов, то *каждый* производный класс должен содержать m*(n+1) виртуальных функций, да еще столько же чисто виртуальных заглушек в классе-предке. Итого мы получаем (m+1)*m*(n+1) диспетчерских функций. Для всех иерархий, кроме самых тривиальных, это довольно много.
- Если оператор не является коммутируемым (то есть ему нельзя передать повторный вызов с аргументами, переставленными в обратном порядке), это число удваивается, поскольку вам придется реализовать отдельные функции для двух вариантов порядка аргументов. Например, `y/x` — совсем не то же, что `x/y`; вам понадобится оператор `/` и специальная функция `DivideInto` для переставленных аргументов.
- Клиенты базового класса видят все устрашающие защищенные функции, хотя это им совершенно не нужно.

Тем не менее, в простых ситуациях двойная передача оказывается вполне разумным решением — ведь проблема, как ни крути, достаточно сложна. Специфика ситуации неизбежно требует множества мелких фрагментов кода. Двойная передача всего лишь заменяет большие, уродливые, немодульные конструкции `switch/case` более быстрой и модульной виртуальной диспетчеризацией.

Как правило, количество функций удается сократить, но при этом приходится в той или иной степени идти на компромисс с нашим строгим правилом — никогда не спрашивать у объекта, каков его настоящий тип. Некоторые из этих приемов рассматриваются ниже. Видимость производных классов для клиентов `Number` тоже удается ликвидировать минимальной ценой; об этом будет рассказано в главе 12. Как и во многих проблемах дизайна в C++, в которых задействованы матрицы операций, вам придется на уровне здравого смысла решить, стоит ли повышать модульность за счет быстродействия или объема кода.

Гетероморфная двойная передача

Двойная передача обычно возникает в ситуациях, когда оба аргумента происходят от общего предка, но это не обязательно. Левый и правый операнды могут принадлежать к разным классам, не имеющим общего предка.

Один из моих любимых примеров относится к обработке событий в графических средах. Существует множество возможных событий: операции и мышью, события от клавиатуры, события операционной системы и даже такая экзотика, как распознавание голоса или световое перо. С другой стороны, в пользовательский интерфейс обычно входят разнообразные *виды*, *панели* или *окна* (терминология зависит от операционной системы и используемого языка) — внешние окна с заголовками и кнопками закрытия, поля для редактирования текста и области, в которых можно рисовать красивые картинки. Для каждой комбинации конкретного события с конкретным типом вида может потребоваться уникальная реализация. Возникает та же проблема, что и с иерархией чисел, хотя на этот раз события и виды не имеют общего базового класса. Тем не менее, методика двойной передачи все равно работает.

```
class Event {      // Чисто виртуальный базовый класс для событий
public:
```

```
    virtual void Process(View* v) = 0;
};
class MouseClick : public Event {
public:
    virtual void Process(View* v) { v->Process(*this); }
};
class View {  // Чисто виртуальный базовый класс для видов
public:
    virtual void Process(MouseClick& ev) = 0;
    virtual void Process(Keystroke& ev) = 0;
    // И т.д.
};
```

Хотя на первый взгляд кажется, что проблема отличается от иерархии `Number`, присмотритесь повнимательнее. Реализация функции `Process()` класса `Event` всего лишь «разворачивает» операцию и перенаправляет вызов. Поскольку функция `Event::Process()` является виртуальной, когда дело доходит до класса `View`, точный тип `Event` уже известен, и компилятор вызывает правильную перегруженную версию `View::Process()`.

Каждый раз, когда вам захочется забить код типа в переменную класса, чтобы узнать, с каким производным классом вы имеете дело, подумайте, нельзя ли переложить хлопоты на компилятор с помощью двойной передачи (или одного из описанных ниже вариантов).

Передача более высокого порядка

До сих пор мы рассматривали только бинарные функции, однако та же методика распространяется на функции с производным количеством аргументов неизвестного типа. Если функция имеет n аргументов, передавать придется n раз. Предположим, у вас имеется функция, которая получает три аргумента Number и возвращает Number&. Можно устроить, чтобы первый аргумент оказался в левой части оператора -> (или .), а дальше начинаются комбинаторные игры.

```
class Number {
protected:
    // 'this' – неявный второй аргумент
    virtual Number& fn1(Integer& n1, Number& n3) = 0;
    virtual Number& fn1(Complex& n1, Number& n3) = 0;
    // И т.д. для всех типов в первой позиции

    // 'this' – неявный третий аргумент
    virtual Number& fn2(Integer& n1, Integer& n2) = 0;
    virtual Number& fn2(Integer& n1, Complex& n2) = 0;
    virtual Number& fn2(Complex& n1, Integer& n2) = 0;
    virtual Number& fn2(Complex& n1, Complex& n2) = 0;
    // И т.д. для всех сочетаний
public:
    // 'this' – неявный первый аргумент
    virtual Number& fn(Number& n2, Number& n3) = 0;
};
class Integer : public Number {
protected:
    // 'this' – неявный второй аргумент
    virtual Number& fn1(Integer& n1, Number& n3)
        { return n3.fn2(n1, *this); }
    virtual Number& fn1(Complex& n1, Number& n3)
```

```
            { reutrn n3.fn2(n1, *this); }
        // И т.д. для всех типов в первой позиции

        // 'this' – неявный третий аргумент
        virtual Number& fn2(Integer& n1, Integer& n2)
        {
            // Настоящая реализация – известны все три аргумента
        }
        // И т.д. для всех сочетаний
    public:
        // 'this' – заданный первый аргумент
        virtual Number& fn(Number& n2, Number& n3)
            { return n2.fn1(*this, n3); }
    };
```

Такой вариант, как и двойная передача, обычно работает быстрее других предлагаемых архитектур. Все делается за три элементарных переходя через v-таблицы, а это намного быстрее, чем хеширование и просмотр таблиц. Однако он некрасив и быстро становится неуправляемым. Возьмите решение с двойной передачей и вообразите, что его сложность начинает возрастать по экспоненте; вы получите некоторое представление о том, с чем связано поддержание такой структуры.

Описанная методика применима и к гетероморфным иерархиям, хотя в результате у вас получится «программа-кузнечик»: она совершает головокружительные прыжки из стороны в сторону. Если вы проделаете нечто подобное в нетривиальной ситуации, в день сдачи программы прихватите пончиков или коробку конфет и будьте необычайно любезны со своими коллегами. Даже если это оптимальное решение, его наверняка сочтут… как бы это сказать… в лучшем случае, спорным.

Если функция имеет больше двух аргументов, число сочетаний растет быстрее, чем грибы после дождя. Организуя множественную передачу для нескольких аргументов, серьезно подумайте над описанными ниже приемами группировки, уменьшающими количество сочетаний.

Группировка передач и преобразования

В реальной жизни редко встречаются уникальные реализации для всех сочетаний левого и правого операндов. Например, в любой операции с участием комплексного и какого-то другог числа результат будет комплексным. Преобразование некомплексного аргумента в комплексный сокращает количество диспетчерских функций. Процесс сокращения матрицы передач я описываю общим термином *группировка (clustering)*. На самом деле для большинства задач не существует элегантного, универсального и притом головокружительно быстрого способа группировки. К тому же эти способы практически никак не связаны с синтаксисом или идиомами C++. Они либо требуют знания типов (тема, к которой мы вернемся в следующей главе), либо основаны на логике `if/then/else` или `switch/case`, которой мы пытаемся всячески избегать в этой части.

Существут два основных подхода:

1. Использовать иерархию классов для обслуживания нескольких сочетаний различных типов аргументов одной реализацией.
2. Сформировать *иерархию преобразований* и преобразовать один или оба аргумента к более универсальному типу, после чего выполнить передачу.

Их нетрудно спутать, но на самом деле они отличаются.

Группировка в базовых классах

Первый подход обычно связан с созданием специфической иерархии классов, которая отображает структуру групп. При этом диспетчерские функции поддерживаются только на высоких уровнях иерархии классов. При поиске сигнатур компилятор автоматически «преобразует» производные классы к промежуточным базовым классам. Такой вариант хорошо подходит лишь для не очень глубоких

иерархий, поскольку при совпадении сигнатуры в двух базовых классах компилятор начнет кричать «Караул, неоднозначность!».

```
class foo { ... };
class bar : public foo { ... };
class banana : public bar { ... };
void fn(bar&);
void fn(foo&);
fn(*(new banana));      // Неоднозначность! Ха-ха-ха!
```

Компиляторы обожают подобные шутки, поскольку они могут ждать и не сообщать об ошибке до тех пор, пока им не встретится заветное сочетание типов. Если бы существовала перегрузка `fn()` для аргумента `banana&`, никаких проблем не возникло бы — компилятор всегда предпочитает точное совпадение преобразованию. Но тогда пропадает весь смысл группировки посредством автоматического преобразования к базовому классу.

Отделение типов от иерархии классов

Второй подход сопровождается мучительными логическими построениями. Перед выполнением передачи аргументы преобразуются от специализированных типов к более универсальным. Например, при любой операции, в которой учавствует комплексное число (`Complex`), второй аргумент заранее преобразуется к типу `Complex`. Тем самым из матрицы фактически исключается целая строка и столбец. Если ни один из аргументов не является комплексным, мы ищем среди них вещественный (`Real`); если он будет найден, второй аргумент также преобразуется в `Real`. Если не будут найдены ни `Complex`, ни `Real`, ищем `Rational` и т.д. Подобная иерархия преобразований — от `Integer` (или чего угодно) к `Rational`, затем `Real` и `Complex` — не совпадает с иерархией классов, поскольку было бы глупо порождать легкий `Integer` от тяжеловесного `Complex`. Кстати, весьма интересный вопрос: почему иерархии типов (в данном случае числовых) часто плохо укладываются в иерархии классов, основанных на общих свойствах?

Это еще не все

Конечно, только что описанная методика передачи имеет некоторые недостатки. Во-первых, производные классы не удается нормально инкапсулировать, поскольку их все приходится перечислять в интерфейсах диспетчерских функций базового класса. В сущности, чтобы не вводить код типа, видимый широкой публике, мы выставили на всеобщее обозрение все производные типы — не очень хороший компромисс. Во-вторых, что произойдет с оператором вроде +=? Если в программе встречается `Integer+=Complex`, результат будет иметь тип `Complex`, а мы пока не располагаем средствами для преобразования типа «на месте».

В главах 11 и 12 обе проблемы будут решены для гомоморфных иерархий классов, хотя приведенные методики обобщаются и для других ситуаций (скажем, для упомянутой выше проблемы с классами `Event/View`). А пока лишь скажу, что описанные в этой главе приемы приносят непосредственную пользу в тех ситуациях, когда на первое место выходит быстродействие, а не инкапсуляция. Два перехода по v-таблицам почти всегда работают быстрее традиционных подходов.

Наконец, не удивляло ли вас то, как наши функции возвращали `Number&`?

```
Number& Integer::operator+(const Integer& n)
{
    // Ниже приведен псевдокод
    if (i + n.i слишком велико для int) {
        return ЦелоеСПовышенной точностью
    }
    else return Integer(i + n.i);
}
```

Возвращение ссылок на переменную величину — дело опасное. Многие компиляторы пускают в расход возвращаемую величину до того, как вам удастся ее использовать. Выделение памяти под

возвращаемый объект оператором new (вместо стека) решает проблему, поскольку величина заведомо остается жить после завершения функции. Но тогда возникают проблемы с управлением памятью — когда и как удалять возвращаемую величину? Чтобы решить ее, нам понадобится материал глав 11 и 12, а также одна из методик управления памятью (скажем, подсчет ссылок), рассматриваемых в последней части книги.

Итак, приведенной в этой главе информации хватит для решения простых проблем (например, связанных с событиями и видами), но она лишь закладывает основу для построения более общих решений.

Производящие функции и объекты классов

11

В предыдущей главе мы стролкнулись с гомоморфными иерархиями классов и с головой погрузились во множественную передачу. В этой главе мы продолжим исследовать царство иерархий классов, рассматривая объекты классов и сопутствующие темы.

Примеры из предыдущей главы обладали одним недостатком — все производные классы были видны клиентам. Но если производные классы погребены в файлах .cpp на глубине нескольких метров, как клиенту создавать экземпляры этих спрятанных классов? Этой теме посвящено начало главы. *Производящей функцией (factory function)* называется функция, которая инкапсулирует применение оператора new для создания экземпляров класса. Знатоки C++ обычно сходятся на том, что производящие функции — Хорошая Вещь, и вскоре вы поймете почему. Вы оказались на семинаре с коктейлями и хотите найти хорошую тему для разговора? Поднимите стакан и небрежно упомяните о том, как производящие функции выручили вас в трудную минуту.

Во многих программах нам хотелось бы в процессе их выполнения сделать нечто, не поддерживаемое динамической моделью C++ (например, запросить у объекта его класс). Для этой цели существует предложенный стандарт RTTI (Run Time Type Information, Динамическая информация о типе), но по причинам, о которых будет сказано ниже, его вряд ли можно считать универсальным или хотя бы полезным средством. Вместо этого мы рассмотрим нестандартные решения, основанные на концепции объектов классов, в том числе особый случай — представителей классов.

Производящие функции

Предположим, вы согласились, что гомоморфизм — это хорошо, и тут же сотворили свою собственную гомоморфную иерархию классов.

```
// В файле Grandpa.h
class Grandpa { ... };

// Скрыто в файле Grandpa.cpp
class Dad : public Grandpa { ... };
class AuntieEm : public Grandpa { ... };

// Где-то в нашей программе
#include “Grandpa.h”
Grandpa* g = new ...    // Стоп! Как создать «папу»?
```

Допустим, с позиций биологии все понятно, но мы говорим о C++, не правда ли? Проблема заключается в том, что мы надежно изолировали «папу» (`Dad`) от внешнего мира — по крайней мере для любого кода, расположенного за пределами файла `Grandpa.cpp`. Замечательный интерфейс

Grandpa позволяет нам как угодно манипулировать любым экземпляром производного класса, включая Dad, но при это не существует способа создать этот экземпляр!

make-функции

На сцену выходят производящие функции. По общепринятому соглашению их простейшая форма называется makeFoo(), где Foo — имя генерируемого класса.

```
class Grandpa {
public:
     static Grandpa* makeDad();      // Создает экземпляры Dad
     static Grandpa* makeAuntieEm();
};
// В Grandpa.cpp
Grandpa* Grandpa::makeDad()
{
     return new Dad;
}
Grandpa* Grandpa::makeAuntieEm()
{
     return new AuntieEm;
}
```

О существовании конкретных производных классов по-прежнему известно всем, однако настоящие интерфейсы Dad и AuntieEm надежно спрятаны от любопытных глаз.

Символические классы и перегруженные make-функции

Все эти функции makeFoo расходятся с перегружаемой природой C++, не правда ли? Следующий фрагмент помогает выбрать нужную версию make.

```
class Variation Dad {};     // Пустое объявление класса
class VariationAuntieEm {};
class Grandpa {
public:
     static Grandpa* make(VariationDad);
     static Grandpa* make(VariationAuntieEm);
};
// В вашей программе
Grandpa* g = Grandpa::make(VariationDad());
```

Вызов VariationDad() создает *анонимный экземпляр (anonytous instance)* класса VariationDad — то есть экземпляр, не связанный ни с какой переменной в исходной области действия. Он живет лишь столько, сколько необходимо для вызова, а затем исчезает. Экземпляр VariationDad нужен лишь для одного — он сообщает компилятору, какая перегруженная версия make должна вызываться. Класс, который не имеет членов и используется подобным образом, называется *символическим классом (symbol class)*. Он играет ту же роль, что и символический тип данных в языках типа Lisp: строка, которая заранее компилируется в более эффективную форму.

Поскольку производные классы все равно инкапсулированы в файле .cpp, мы можем воспользоваться этим обстоятельством и заменить имя исходного Dad другим локальным по отношению к файлу .cpp, а затем просто воспользоваться Dad вместо VariationDad в файле .h.

Оптимизация с применением производящих функций

Задержимся на минутку. Допустим, ни один из производных классов Grandpa не интересует публику настолько, чтобы различать их при выборе производящей функции. Даже в этом случае

конструирование экземпляров с помощью одной или нескольких производящих функций вполне оправдано. В частности, при этом скрывается логика выбора производного класса, экземпляр которого создается на основании информации, предоставленной на момент создания. Иначе говоря, производящие функции могут послужить средством оптимизации ваших программ.

```
class Number {
public:
    static Number* make();    // Конструирует число по умолчанию
    static Number* make(int);
    static Number* make(double);
    static Number* make(string);    // Например, “-1.23”
};
// В файле .cpp
class Integer : public Number { ... };
class Real : public Number { ... };
class ArbitraryPrecisionNumber : public Number { ... };

Number* Number::make()
{
    return new Integer(0);
}
Number* Number::make(int x)
{
    return new Integer(x);
}
Number* Number::make(double x)
{
    return new Real(x);
}
Number* Number::make(string s)
{
    // Псевдокод
    if (малое целое)
        return new Integer(atoi(s));
    if (большое целое)
        return new ArbitraryPrecisionNumber(s);
    if (вещественное)
        // И т.д.
}
```

Сделать то же с помощью конструкторов не удастся. К тому моменту, когда компилятор начинает искать конструктор с подходящей сигнатурой, вы уже сообщили ему, экземпляр какого класса создается. Более того, такая методика отделяет структуру производных классов от аргументов производящей функции.

Локализованное использование производящих функций

Одно из самых замечательных применений производящих функций — возможность изолирования кода, изменяемого при переносе программы на другой компьютер или среду. Открытый интерфейс, выраженный средствами базового класса, остается прежним, а в файле .cpp прячется специализированный производный класс с кодом, ориентированным на данную платформу.

```
class Window {
public:
```

```
        static Window* make();
        // Далее следует гомоморфный интерфейс
    };
    // В window.cpp для ОС Windows
    class MS_Window : public Window { ... };
    Window* Window::make()
    {
        return MS_Window();
    }

    // или в window.cpp для Mac OS
    class Mac_Window : public Window { ... };
    Window* Window::make()
    {
        return Mac_Window*();
    }
```

Чтобы переключиться с одной платформы на другую, достаточно перекомпилировать и перекомпоновать файл .cpp. Все клиенты класса `Window` ничего не будут знать о произошедшей локализации (предполагается, что вы собираетесь создать действительно универсальное, гомоморфное представление окна в графическом интерфейсе — задача, перед которой дрогнет даже самый отчаянный проектировщик).

Уничтожающие функции

Уничтожающие функции (junkyard functions) уничтожают экземпляры классов. Идея заключается в том, чтобы сделать деструктор закрытым или защищенным, а затем предоставить функцию, в которой инкапсулируется вызов оператора `delete`.

```
class Grandpa {
protected:
    virtual ~Grandpa();
public:
    static Grandpa make();
    static void destroy(Grandpa*);
};
// В файле grandpa.cpp
void Grandpa::destroy(Grandpa* g)
{
    delete g;
}
```

Возможно, сейчас это не имеет особого смысла, но в нестандартных схемах управления памятью уничтожающие функции способны принести очень большую пользу. А пока скажите начальнику и коллегам, что уничтожающие функции нужны вам ради симметрии. Они косо посмотрят на вас и отойдут подальше, так что на какое-то время вам будет спокойнее работать.

Снова о двойной передаче: промежуточные базовые классы

Приняв на вооружение производящие функции, мы легко повысим инкапсуляцию двойной передачи.

```
// В файле grandpa.h

class Grandpa {
public:
    // Производящие функции и гомоморфный интерфейс
```

```
};
// В файле grandpa.cpp
class RealGrandpa : public Grandpa {
// Промежуточный гомоморфный базовый класс
protected:
    // Функции двойной передачи
};
class Dad : public RealGrandpa { ... };
class AuntieEm : public RealGrandpa { ... };
```

Наличие производящих функций означает, что производные классы можно скрыть. Добавляя промежуточный базовый класс RealGrandpa, мы полностью прячем все жуткие подробности двойной передачи в файле .cpp. Никаких защищенных функций в файле .h!

Нет — конструкторам копий и оператору =!

Предполагается, что Grandpa — чисто гомоморфный базовый класс, содержащий хотя бы одну чисто виртуальную функцию. Это предотвращает непосредственное создание экземпляров Grandpa клиентом. Если вы используете производящую функцию для класса с возможностью создания экземпляров, конструкторы следует сделать защищенными, чтобы экземпляры могли создаваться только производящей функцией.

Раз уж мы заговорили на эту тему, после непродолжительных размышлений становится ясно, что клиент гомоморфного базового класса не должен использовать конструктор копий или оператор =. Если кто-нибудь захочет подублировать экземпляр, создайте специальную версию make-функции для копирования this.

```
class Grandpa {
public:
    virtual Grandpa* makeClone() = 0;
};
```

Эта функция не объявляется статической, поскольку в каждом производном классе она должна решать специализированную задачу. С присвоением дело обстоит сложнее. Если переопределить оператор = для левого операнда, непонятно, что же тогда делать с правым операндом, тип которого неизвестен. Первое практическое решение — полностью запретить присваивание в таких ситуациях и сделать оператор = закрытым. Второе — использовать вариацию на тему двойной передачи: сделать оператор = виртуальным и в каждом производном классе вызывать виртуальную функцию AssignTo(), перегружаемую для кажого производного класса. Смотрится уродливо, но работает.

Объекты классов

Объектом класса называется объект, предназначенный для создания экземпляров представляемого им типа. В нашей терминологии объект класса представляет собой объект, основные функции которого являются производящими. Позднее мы возложим на объекты классов и другие обязанности (например, описание структуры экземпляра), а пока рассмотрим практический пример.

```
class GrandpaClass {    // Объект класса для Grandpa
public:
    Grandpa* make();    // Создает экземпляры Grandpa
};
class Grandpa { ... };
```

Все сказанное о производящих функциях относится и к объектам классов, включая спрятанные инкапсулированные производные классы, оптимизацию и прозрачность локализации. «Хорошо, — скажете вы, — но зачем это нужно?» Во-первых, мы избавляемся от некрасивых статических функций и переходим к более чистому, объектно-ориентированному варианту, при котором все происходит

посредством отправки сообщений объекту. Во-вторых, мы получаем удобную возможность следить за другими характеристиками экземпляров и классов.

Информация о классе

В объектах класса удобно хранить сведения о самом классе. Для этого лучше всего создать гомоморфный базовый класс для объектов классов.

```
class Class {
protected:
    Collection<Class> base_classes;
    Collection<Class> derived_classes;
    String class_name;
    Class() {};   // Класс становится абстрактным базовым
public:
    // Функции доступа для получения имени и т.д.
};
```

Состав хранимой информации в огромной степени зависит от потребностей приложения и от того, насколько увлеченно вы занимались SmallTalk на этой неделе.

Имя класса и создание экземпляров по имени

Базовая информация, которую может сообщить объект класса, — имя класса в виде некоторой символьной строки. Кроме того, можно хранить словарь всех объектов классов, индексируемый по имени класса. Если добавить к этому универсальный интерфейс к производящей функции, вам удастся реализовать возможность, которая не поддерживается в C++ напрямую — *создание экземпляров по имени (instantiate by name)*.

```
// Где-то в клиентской программе
Class* c = gClasses.Find("Grandpa");
???* g = (Grandpa*)c->make(???);
```

Как видите, практическая польза такого подхода ограничивается некоторыми проблемами. Если мы уже знаем, что создается экземпляр `Grandpa`, то создание экземпляра по имени выглядит неразумно — нам будет трудно определить, к какому классу выполняется преобразование. Вдобавок данная схема не позволяет предоставить отдельные сигнатуры для производящих функций производных классов. Тем не менее, в некоторых ситуациях такая методика оказывается чрезвычайно полезной. Предположим, вы сохранили объект в виде потока байтов и теперь загружаете его. Если первые `n` байт содержит имя класса в виде символьной строки, вы сможете найти нужный объект класса для создания экземпляра. Как правило, реализация заканчивается созданием во всех классах `Class` функции `make(istream&)` или ее эквивалента. Программа приобретает следующий вид:

```
// В коде чтения потока
cin << className;
Class* c = gClasses.Find(className);
BasClass* obj = c->make(cin);
```

Иерархия классов

Сведения об иерархии классов можно хранить по разному, но в конечном счете все сводится к структурам данных с экземплярами `Class`. Выше был представлен один из вариантов: вести в каждом `Class` две коллекции, по одной для базовых и производных классов. Конечно, это следует понимать условно — речь идет об иерархии не объектов `Class`, а представленных ими классов. Необходимость различать понятия `Class` и «класс» наверняка вызовет у вас головную боль, но у поклонников SmallTalk и Lisp это считается хорошим развлечением и признаком мастерства. Другой способ — ввести одну глобальную структуру данных с парами (базовый, производный), индексируемую в обоих направлениях. В некоторых ситуациях вместо пар используются триплеты (базовый, производный, порядок), чтобы базовые классы перечислялись в порядке их объявления в соответствующем классе.

Описания переменных класса

В некоторых ситуациях объекты `Class` стоит сделать достаточно умными, чтобы они могли получать экземпляры представляемого класса и описывать их структуру. Все начинается с переменных класса. В наиболее прямолинейном варианте `Class` возвращает итератор, который в свою очередь возвращает пару смещение/`Class`, описывающую смещение переменной внутри экземпляра и ее `Class`. Имеет смысл создать перечисления для трех подмножеств переменных класса:

1. Все переменные, включая встроенные типы (такие как `int`). Для представления примитивных типов вам придется создать фиктивные классы, производные от `Class`.
2. Только переменные невстроенных типов, для которых обычно и так существует свой `Class`.
3. Только указатели и ссылки на другие объекты.

Последний вариант играет особенно важную роль в некоторых нетривиальных алгоритмах сборки мусора.

Описания функций класса

В еще более редких ситуациях объект `Class` должен описывать набор функций представленного им класса. При этом вы фактически начинаете играть роль компилятора C++, так что не перегибайте палку. И снова оптимальным представлением оказывается итератор. Для каждой функции можно возвращать любую информацию, от простейшей (ее имени) до более сложной (адрес, имена, типы и порядок аргументов и тип возвращаемого значения). Некоторые проблемы просто выходят за рамки C++; если вам захочется проделать нечто подобное, стоит серьезно подумать об использовании настоящего динамического языка.

Коллекции экземпляров

Класс `Class` предоставляет еще одну интересную возможность — ведение коллекции всех экземпляров класса. Это может быть либо отдельная коллекция для каждого `Class`, либо одна глобальная структура данных с парами (`Class`, экземпляр). Если выбран второй вариант и коллекция индексируется в обоих направлениях, она оказывается чрезвычайно полезной при отладке («Покажи мне все экземпляры класса x»), а также может применяться для ответов на вопросы вроде «Каков класс данного экземпляра?» без физического хранения адреса объекта `Class` в каждом экземпляре. Такое решение работает только для экземпляров верхнего уровня, создаваемых производящими функциями; вложенные объекты (переменные или базовые классы) в этот реестр не попадут.

Статистика

Показатели сыплются как из рога изобилия — количество экземпляров, в данный момент находящихся в памяти; общее количество экземпляров, созданных с момента запуска программы; статистические профили с описанием, когда и как создавались экземпляры… Возможности ограничены только тем, сколько времени вы сможете им посвятить. Если вы работаете на повременной оплате, не ограничивайте себя — ведь при желании обоснование можно придумать для любого показателя. А если нет, подумайте, окупятся ли потраченные усилия.

Еще несколько слов об уничтожающих функциях

После долгого разговора о том, какие замечательные штуки можно проделывать с объектами классов, вернемся к уничтожающим функциям. Многие концепции, представленные в предыдущем разделе (такие как скрытые коллекции экземпляров и статистика), реализуются лишь в том случае, если вам удастся отследить время создания и уничтожения экземпляра.

Конечно, для ведения статистики можно воспользоваться статистическими переменными, производящими функциями и т.д., принадлежащими целевому классу, однако методика, связанная с объектами классов, обеспечивает намного лучшую модульность. Возьмите существующий класс. Добавьте класс объекта `Class`. Влейте одну-две производящие функции, перемешайте с уничтожающей функцией. Поставьте на огонь статистики и доведите до кипения. Ура! Все административные средства были добавлены без модификации исходного класса. Об ихменениях придется сообщать клиентам, но если в начале работы никаких клиентов еще не было, а управляемый

класс должен оставаться неизменным или его исходные тексты недоступны, нам удалось довольно многого добиться, не создавая никаких побочных эффектов для критически важного кода.

Определение класса по объекту

Для существующего экземпляра довольно часто требуется определить его класс. Вроде бы ничего сложного, но в действительности это очень глубокая тема. Помните, что объекты могут создаваться в стеке или в куче, внедряться в другие объекты в виде переменных или базовых классов, а также создаваться производящими функциями. Ниже описано несколько основных решений.

Внедрение указателя на объект класса

Самое очевидное решение — внедрять указатель на Class в любой объект, вложенный или нет.

```
class Object {     // Предок всех реальных классов
protected:
     static ObjectClass s_my_class;
     Class* my_class;   // == &s_my_class;
public:
     Object() : my_class(&s_my_class) {}
     Class* My_Class()  { return my_class; }
};
class Foo : public Object
protected:
     static FooClass s_my_class;
public:
     Foo() { my_class = &s_my_class; }
};
```

Все классы порождаются от общего предка Object, в котором определяется протокол для получения объекта Class. Вы имеете полное право использовать одни и те же имена членов на разных уровнях иерархии классов (как это сделано с s_my_class в нашем примере). Компилятор выбирает имя, находящееся в непосредственной области действия. Более того, конструкторы выполняются в порядке «базовый класс/переменные класса/ производные классы», поэтому последний конструктор оставит my_class правильное значение. Эта схема позволяет всегда получить объект Class независимо от того, сколько выполнялось преобразований типа от производных к базовым классам.

Издержки составляют четыре байта, необходимые для хранения указателя. Виртуальные функции не требуются, поэтому нам не придется добавлять v-таблицу в класс, обходившийся без нее. На избыточное конструирование my_class будут потрачены дополнительные такты процессора, но для большинства приложений это несущественно. Пожалуй, основные издержки сводятся к дополнительному коду, находящемуся в конструкторах.

В более «чистом» варианте указатель на Class задается производящими функциями объекта Class:

```
class Object {
friend class Class;
private:
     Class* my_class;
public:
     Class* My_Class() { reutrn my_class; }
};
class Class {
protected:
     void SetClass(Object& obj) { obj.my_class = this; }
};
class Foo : public Object { ... };
```

```
class FooClass : public Class {
public:
     Foo* make()
     {
          Foo* f = new Foo;
          this->SetClass(f);
          return f;
     }
};
```

Выглядит получше, поскольку производные от `Object` классы и не подозревают об этих фокусах… но так ли это? Недостаток этого подхода — в том, что он не работает для экземпляров `Foo`, объявленных в стеке или вложенных в другие классы в виде структур данных. Перед вами одна из ситуаций, в которых приходится принимать трудное решение: то ли ограничить класс только динамическими экземплярами, то ли искать более сложное решение и без того сложной проблемы.

Существует еще один вариант — управлять выделением памяти и хранить адеса объекта класса прямо над самим объектом в памяти вместо того, чтобы делать его переменной класса предка. Для этого нам понадобятся приемы управления памятью, описанные в части 4.

Внешние структуры данных

Как упоминалось выше, вы также можете создать глобальную коллекцию с парами экземпляр/`Class`. Все не так скверно, как выглядит на первый взгляд, особенно если информация `Class` нужна только в процессе отладки и будет исключена в рабочем режиме. Если соблюдать осторожность в реализации, решение также распространяется и на такие вложенные объекты, как стековые переменные или экземпляры, хотя для этого вам понадобится соответствующая поддержка со стороны конструктора и деструктора основного класса.

Нестандартные пространства памяти

Другое решение, рассматриваемое в главах 15 и 16 — физическое разделение объектов по различным пространствам памяти в соответствии с их классом. Оно отличается повышенной сложностью и попросту не работает для вложенных объектов, но зато обладает впечатляющим быстродействием и малым расходом памяти.

Представители

В книге «Advanced C++ Programming Styles and Idioms» Джеймс Коплин (James Coplien) выдвинул особую разновидность объекта класса, который он назвал *представителем (exemplar)*. Представитель — это объект класса, который является экземпляром представляемого им класса. Понятно? У меня от таких высказываний голова идет кругом.

```
class Exemplar {};      // Символический класс
class Foo {
private:
     Foo();    // И все остальные конструкторы, кроме одного
public:
     Foo(Exemplar);     // Конструктор представителя
     Foo* make();       // Производящие функции представителя
};
extern Foo* foo;        // Сам представитель
```

Подробности несущественны. Важно то, что один экземпляр (на который ссылается переменная `foo`) выполняет функции объекта класса для класса `Foo`. В реализациях производящих функций необходимо убедиться, что `this` совпадает с представителем, поскольку производящая функция не должна вызываться для всех остальных экземпляров класса.

С представителями можно делать все то же самое, что и с объектами класса. При этом необходимо учитывать следующее:

- Для представителей не нужно создавать отдельный класс `Class`.
- С представителями не приходится объявлять друзей (обычно объект класса должен быть другом того класса, который он представляет).
- Настоящие объекты классов заметно упрощают моделирование иерархий классов.
- Настоящие объекты классов упрощают создание гомоморфного интерфейса для описанных выше сведений о классах.
- С объектами классов отличия между тем, что относится к объектам класса/представителям, и тем, что относится к реальным экземплярам, становятся более отчетливыми.

В общем и целом, объекты классов оказываются более гибкими при незначительно усложнении реализации по сравнению с экземплярами.

Невидимые указатели

12

Нетривиальное использование C++ напоминает один известный эпизод из фильма «Сумеречная зона». Героиня попадает в автомобильную аварию. Она безуспешно ждет, пока кто-нибудь проедет по дороге, и в конце концов решает отправиться за помощью. Но куда бы она ни шла, как бы внимательно ни следила за направлением, она всегда возвращалась к обломкам машины. Так и с указателями: куда бы вы ни шли, вы все равно вернетесь к обломкам. Хм… пожалуй, мне следовало подыскать более оптимистичное сравнение.

В этой главе мы снова возвращаемся к теме указателей, на этот раз — в свете гомоморфных иерархий классов. Рассматриваемые здесь указатели я называю *невидимыми (invisible pointers)*, поскольку в большинстве случаев можно устроить так, чтобы клиент абсолютно ничего не знал о присутствии указателя между ним и целевым объектом. Джеймс Коплин (James Coplien) рассматривает частный случай невидимых указателей и называет его «парадигма конверт/письмо»; мы же поговорим о более общем случае.

Основные концепции

Если гомоморфизм хорошо подходит для других классов, значит, он подойдет и для указателей. Концепция проста: указатель и указываемый объект порождаются от одного и того же чисто абстрактного базового класса.

```
class Foo {
public:
     virtual void do_something() = 0;
     virtual void do_something_else() = 0;
};
class PFoo : public Foo {
private:
     Foo* foo;
public:
     virtual void do_something() { foo->do_something(); }
     virtual void do_something_else() { foo->do_something_else(); }
};
class Bar : public Foo {
// Все для производного класса
};
```

Вместо перегрузки оператора `->` в `PFoo` используется делегирование. Приведенный фрагмент лишь слегка затрагивает данную тему. На практике приходится учитывать множество деталей, начиная с того, как скрыть указатели и указываемые объекты от клиентов.

Инкапсуляция указателей и указываемых объектов

Одно из величайших преимуществ гомоморфных указателей заключается в том, что указатель вместе с указываемым объектом можно инкапсулировать в файле .cpp. Взгляните на только что приведенный фрагмент. Указатель ничего не добавляет к открытому интерфейсу, представленному в классе Foo, поэтому клиентам не нужно видеть PFoo или производные классы, на которые он ссылается. В сущности, при достаточной аккуратности можно убедить клиентов, что они работают непосредственно с указываемым объектом, хотя на самом деле в качестве центрального звена цепочки присутствует указатель. Отсюда и термин — *невидимый указатель*.

```
// В файле foo.h
class Foo {
public:
     static Foo* make();     // Производящая функция
     virtual void do_something() = 0;
     virtual void do_something_else() = 0;
};
// В файле foo.cpp
class PFoo : public Foo {
private:
     Foo* foo;
public:
     PFoo(Foo* f) : foo(f) {}
     virtual void do_something() { foo->do_something(); }
     virtual void do_something_else() { foo->do_something_else(); }
};
class Bar : public Foo {
// Все для производного класса
};
Foo* Foo::make()
{
     return new PFoo(new Foo);
}
```

Вставить PFoo в существующую программу совсем несложно — при условии, что вы приняли все меры предосторожности, спроектировали его с расчетом на гомоморфную иерархию классов и инкапсулировали производные классы вроде Bar. Ведь вы это сделали, не правда ли? Перед нами очередной мистический принцип — вы делаете что-то не для того, чтобы извлечь непосредственную пользу, а для сохранения мировой гармонии. В один прекрасный день вам потребуется вставить умный указатель, и в гармоничном мире это не вызовет никаких проблем.

Производящие функции

Конечно, производящие функции пригодятся вам каждый раз, когда вы инкапсулируете производные классы. В приведенном выше фрагменте мы изменили производящую функцию так, чтбы она создавала два объекта — указатель и указываемое значение.

Обратите внимание: указываемый объект не создается в конструкторе указателя. Для этого существует веская причина. Вероятно, нам захочется использовать класс указателя PFoo для всех производных классов Foo. Это означает, что некто за пределами класса указателя (производящая функция) решает, что именно следует создать и спрятать в указателе.

В предыдущих главах, посвященных умным указателям, основное внимание уделялось шаблонам и обобщенным классам указателей, соответствующим классам указываемых объектов. С невидимыми указателями шаблоны уже не имеют никакого реального значения.

Все, что говорилось о производящих функциях и объектах классов в предыдущей главе, в равной степени относится и к невидимым указателям. Оптимизируйте и локализуйте, сколько душе угодно. Как правило, сам класс указателя в этом не участвует.

Ссылки на указатели

Производящая функция не обязана возвращать Foo*. С таким же успехом подойдет и Foo&.

```
class Foo {
public:
    static Foo& make();     // Производящая функция
    virtual void do_something() = 0;
    virtual void do_something_else() = 0;
};
// В файле foo.cpp
class PFoo : public Foo {
private:
    Foo* foo;
public:
    PFoo(Foo* f) : foo(f) {}
    virtual void do_something() { foo->do_something(); }
    virtual void do_something_else() { foo->do_something_else(); }
};
class Bar : public Foo {
// Все для производного класса
};
Foo& Foo::make()
{
    return *(new PFoo(new Foo));
}
```

Единственная проблема заключается в том, что копирование с помощью конструктора копий, как вы вскоре убедитесь, строго воспрещается. И все же люди, вооруженные оператором &, неизменно пытаются копировать объект. С оператором * соблазн намного слабее. Во всем остальном выбор — дело вкуса.

Неведущие указатели

Парадигма нивидимых указателей реализуется как для ведущих, так и неведущих указателей. От того, какое решение будет принято на стадии дизайна, зависят и способы решения некоторых проблем. Ниже приведены некоторые подробности реализации неведущих указателей.

Копирование

Клиент не может копировать указатель нормальными средствами, поскольку он не знает его настоящего класса. Зато хорошо подходят средства, описанные в предыдущей главе (в особенности копирование объектов с помощью специального виртуального варианта make-функции). Для неведущих указателей достаточно просто скопировать адрес указываемого объекта.

```
class Foo {
private:
    Foo(const Foo&) {}
public:
    virtual Foo* makeClone();       // Копирование
};
// В файле foo.cpp
```

```
class PFoo : public Foo {
private:
    Foo* foo;
public:
    PFoo(Foo* f) : foo(f) {}
    virtual Foo* makeClone() { return new PFoo(foo); }
};
Foo* Foo::makeClone()
{
    return NULL;        // Несущественно для всего, кроме указателей
}
```

Реализация функции makeClone() необходима только для класса указателя. По этой причине, чтобы каждому производному классу не пришлось ее переопределять, в класс-предок включается заглушка.

Присваивание

Разумеется, если не принять специальных мер, оператор = тоже не будет работать, поскольку клиент не знает фактический тип указателя. Так как мы имеем дело с неведущими указателями, операция присваивания стоит согласовать с копированием — то есть присваивание должно затрагивать только указатель, но не указываемый объект. Как и в случае копирования, это нетрудно реализовать — достаточно создать виртуальный оператор = для указателя и заглушку для указываемого объекта.

```
class Foo {
public:
    virtual Foo& operator=(const Foo&);
};
// В файле foo.cpp
class PFoo : public Foo {
private:
    Foo* foo;
public:
    virtual Foo& operator=(const Foo& f)
    {
        foo = f.foo;
        return *this;
    }
};
Foo& Foo::operator=(const Foo&)
{
    return *this;
}
```

Сборка мусора: взгляд в будущее

Поскольку производные классы инкапсулированы, применение неведущих указателей подводит нас к серьезной проблеме дизайна: как узнать, когда нужно удалять указываемый объект? В главах, посвященных управлению памятью, мы серьезно займемся этой проблемой, а пока я лишь в общих чертах опишу две базовые стратегии:

1. В указываемый объект включается счетчик, который показывает, сколько указателей ссылается на него в данный момент. Когда состояние счетчика изменяется с 1 на 0, объект должен удалять себя.

2. Реализуется схема сборки мусора, которая позволяет найти все указатели и указываемые объекты. Мы помечаем все указываемые объекты, на которые ссылается хотя бы один указатель, а затем удаляем все непомеченные указываемые объекты.

Подсчет ссылок используется не так часто, но наша задача прямо-таки создана для него. Сборка мусора — тема, над которой нам предстоит основательно поразмыслить в главе 16.

Ведущие указатели

Невидимые указаьели чаще всего являются ведущими (то есть между указателем и указываемым объектом существует однозначное соответствие). С момента рассмотрения ведущих указателей прошло немало времени, поэтому я кратко напомню, что нам предстоит сделать для поддержания семантики ведущих указателей в нашей ситуации:

1. Когда указатель уничтожается, должен уничтожаться и указываемый объект.
2. Когда указатель копируется, должен копироваться и указываемый объект.
3. Когда для указателя выполняется присваивание, он изменяется так, чтобы ссылаться на копию указываемого объекта из правой части выражения.

Недостаточно присвоить указываемый объект из правой части указываемому объекту из левой части, поскольку они могут принадлежать к разным классам.

Уничтожение

Деструктор класса-предка должен быть виртуальным, а деструктор указателя должен уничтожать указываемый объект.

```
class Foo {
public:
    virtual ~Foo() {}
};
// В файле foo.cpp
class PFoo : public Foo {
private:
    Foo* foo;
public:
    virtual ~PFoo() { delete foo; }
};
```

Копирование

Копирование невидимых ведущих указателей продолжается с того, на чем мы остановились при копировании неведущих указателей. Каждый производный класс должен переопределить функцию `makeClone()`.

```
class Foo {
protected:
    Foo(const Foo&) {}
public:
    virtual Foo* makeClone() = 0;   // Для копирования
};
// В файле foo.cpp
class PFoo : public Foo {
private:
    Foo* foo;
public:
    PFoo(Foo* f) : foo(f) {}
```

```
    virtual Foo* makeClone() { return new PFoo(foo->makeClone()); }
};
class Bar : public Foo {
protected:
    Bar(Bar&);    // Конструктор копий
public:
    virtual Foo* makeClone();
};
Foo* Bar::makeClone()
{
    return new Bar(*this);
}
```

Наконец мы добрались и до применения настоящего конструктора копий. Указатель создает копию — не только свою, но и указываемого объекта. В свою очередь, указываемый объект перекладывает всю тяжелую работу на свой собственный конструктор копий. Обратите внимание: чтобы это стало возможным, мы сделали конструктор копий Foo защищенным.

Присваивание

Присваивание для невидимых ведущих указателей похоже на присваивание для любых других типов ведущих указателей. И снова мы продолжаем с того места, на котором остановились при присваивании неведущих указателей.

```
class Foo {
public:
    virtual Foo& operator=(const Foo&);
};
// В файле foo.cpp
class PFoo : public Foo {
private:
    Foo* foo;
public:
    virtual Foo& operator=(const Foo& f)
    {
        if (this == &f) return *this;
        delete foo;
        foo = f.foo->makeClone();
        return *this;
    }
};
Foo& Foo::operator=(const Foo&)
{
    return *this;
}
```

Снова о двойной передаче

Невидимые указатели позволяют элегантно решить многие проблемы. Одна из таких проблем — улучшенная инкапсуляция двойной передачи, и в том числе решение неприятной проблемы, связанной с оператором +=. Мы объединим двойную передачу с концепцией переходных типов, о которых говорилось давным-давно, в главе 7.

Удвоенная двойная передача

Итак, давайте попробуем реализовать двойную передачу для невидимых указателей. Этот раздел представляет собой элементарное распространение приемов, которыми мы пользовались без связи с указателями.

Первая попытка

Сейчас мы сделаем первый заход на арифметические операции с невидимыми указателями. Он работает, но обладает некоторыми ограничениями, на которые следует обратить внимание и должным образом исправить. Чтобы избежать проблем, связанных с возвращением ссылок на временные значения (см. окончание главы 11), я перехожу на использование оператора new. Проблемы сборки мусора будут рассматриваться позже.

```
// В файле number.h
class NBase;  // Клиентам об этом ничего знать не нужно
class Number {
protected:
     Number(const Number&) {}
     Number() {}
public:
     virtual NBase& operator+(const NBase&) = 0;
     virtual Number& operator+(const Number&) = 0;
     // И т.д.
};
// В файле number.cpp
class Integer;
class Real;
class PNumber : public Number {
private:
     NBase* number;
protected:
     virtual NBase& operator+(const NBase& n) const
          { return *number + n; }    // #2
public:
     PNumber(NBase* n) : number(n) {}
     virtual Number& operator+(const Number& n) const
          { return *(new PNumber(&(n + *number))); }   // #1
};

class NBase : public Number {
// Промежуточный базовый класс
// Традиционная двойная передача в NBase
public:
     virtual NBase& operator+(const Integer&) const = 0;
     virtual NBase& operator+(const Real&) const = 0;
     // И т.д.
     virtual NBase& operator+(const NBase&) const = 0;
     virtual Number& operator+(const Number& n) const
          { return Integer(0); }     // Заглушка не вызывается
};
class Integer : public NBase {
```

```
    private:
        int value;
    protected:
        virtual NBase& operator+(const Integer& i) const
            { return *(new Integer(value + i.value)); }  // #4
    public:
        Integer(int i) : value(i) {}
        virtual NBase& operator+(const NBase& n) const
            { return n + *this; }      // #3
    };
    class Real : public NBase { ... };
```

Как и в исходном варианте двойной передачи, постарайтесь не сосредотачивать взгляд и медленно отодвигайте страницу от носа, пока ну ловите суть происходящего. Ниже подробно расписано, что происходит, когда клиент пытается сложить два `Number` (а на самом деле — два `PNumber`, но клиент об этом не знает). Предположим, складываются два `Integer`:

1. Вызывается операторная функция `PNumber::operator+(const Number&)` левого указателя. Выражение переворачивается, и вызывается аналогичная функция правого указателя, при этом аргументом является левый указываемый объект. Однако перед тем, как это случается, функция создает `PNumber` для результата.
2. Вызывается операторная функция `PNumber::operator+(const NBase&)` левого указателя. Вызов делегируется оператору + указываемого объекта.
3. Вызывается операторная функция `Integer::operator+(const NBase&)` правого указываемого объекта. Выражение снова переворачивается.
4. Вызывается операторная функция `Integer::opeator+(const Integer&)` левого указываемого объекта, где наконец и выполняется реальная операция вычисления суммы.

В итоге происходит четыре передачи — две для указателей и две для указываемых объектов. Отсюда и название — *удвоенная двойная передача*. Мы обходимся без преобразований типов, но зато о существовании `NBase` приходится объявлять на первых страницах газет.

Сокращение до трех передач

Если мы разрешим программе «знать», что изначально слева и справа стоят `PNumber`, и выполним соответствующее приведение типов, количество передач можно сократить до трех: оставить одну передачу для операторной функции `PNumber::operator+(const Number&)` плюс две обычные двойные передачи. Первый `PNumber` приходит к выводу, что справа также стоит `PNumber`, выполняет понижающее преобразование от `Number` к `PNumber`, а затем напрямую обращается к указываемому объекту. При этом удается обойтись без `PNumber::operator+(const NBase&)`. Есть и дополнительное преимущество — при должной осторожности можно удалить из файла .h все ссылки на NBase.

Проблема заключается в том, что какой-нибудь идиот может вопреки всем предупреждениям породить от `Number` свой класс, выходящий за пределы вашей тщательно построенной иерархии. Это будет означать, что не все `Number` будут обязательно «запакованы» в `PNumber`. Только что показанная методика предотвращает создание производных от `Number` классов за пределами файла .cpp и даже правильно работает с производными классами без оболочек (`Number` без `PNumber`) при условии, что они правильно реализуют схему удвоенной двойной передачи.

Как долго результат остается действительным?

В показанной выше реализации клиент должен помнить о необходимости избавляться от `Number`, вызывая `delete &aResult`. Это серьезное ограничение среди прочего усложняет вложенные вычисления, поскольку для всех промежуточных результатов приходится создавать указатель для их последующего удаления. В комитет ANSI поступило предложение (так и не принятое), в соответствии с которым компилятор должен гарантировать, что временная величина в стеке остается действительной

до полного вычисления самого большого вмещающего выражения. Если ваш компилятор следует этому правилу, то строку

```
{ return *(new Integer(&(value + i.value)); }
```

можно записать в виде

```
{ return Integer(value + i.value); }
```

Аналогично создается и `PNumber`. Возвращаемое значение будет оставаться действительным внутри вычисляемого выражения. Любая ссылка, которая может существовать за пределами вмещающего выражения, должна быть получена вызовом функции `makeClone()`. Эта функция создает `PNumber` в куче или присваивает другой `Number` виртуальным оператором = для невидимых ведущих указателей, о которых говорилось выше. Чтобы ликвидировать эти раздражающие мелкие утечки памяти, можно воспользоваться приемами уплотнения и сборки мусора, рассмотренными в части 4.

Самомодификация и переходимость

Невидимый ведущий указатель, как и любой умный указатель, может интерпретироваться как переходный тип. Если просто заменить указываемый объект каким-нибудь производным классом, вы фактически изменяете тип всей видимой клиенту комбинации. На этом основано решение проблемы оператора +=, которая требует самомодификации левого операнда, а также возможного оперативного изменения типа «на ходу». Если правый операнд `Complex` складывается с левым операндом `Integer`, тип левого операнда приходится менять.

```
// В файле number.h
class NBase;  // Клиентам об этом ничего знать не нужно
class Number {
protected:
     Number(const Number&) {}
     Number() {}
public:
     virtual NBase& AddTo(const NBase&) = 0;
     virtual Number& operator+(const Number&) = 0;
     // И т.д.
};
// В файле number.cpp
class Integer;
class Real;
class PNumber : public Number {
private:
     NBase* number;
protected:
     virtual NBase& AddTo(const NBase& n) const
          { return number->AddTo(n); }     // #2
public:
     PNumber(NBase* n) : number(n) {}
     virtual Number& operator+(const Number& n) const
     {
          number = &(n.AddTo(*number));    // #1 - замена
          return *this;
     }
};

class NBase : public Number {
// Промежуточный базовый класс
```

```
// Традиционная двойная передача в NBase
public:
    virtual NBase& operator+=(const Integer&) const = 0;
    virtual NBase& operator+=(const Real&) const = 0;
    // И т.д.
    virtual NBase& AddTo(const NBase&) const = 0;
    virtual Number& operator+(const Number& n) const
        { return Integer(0); }     // Заглушка не вызывается
};
class Integer : public NBase {
private:
    int value;
protected:
    virtual NBase& operator+=(const Integer& i) const
    {
        if (value + i.value достаточно мало) {
            value += i.value;
            return *this;
        }
        else {
            ArbitraryPrecisionInteger api(value);
            api += i.value;
            delete this;
            return api;
        }
public:
    Integer(int i) : value(i) {}
    virtual NBase& AddTo(const NBase& n) const
        { return n + *this; }      // #3
};
class Real : public NBase { ... };
```

Все как и раньше, разве что операторы + превратились в +=, а двойная передача теперь проходит через `+=(левый, правый)` и `AddTo(правый, левый)`, чтобы мы могли различать два порядка аргументов. Это важно, поскольку в конечном счете мы хотим заменить указываемый объект левого операнда новым. Это происходит в двух местах:

1. Операторная функция `PNumber::operator+=(const Number&)` автоматически заменяет число полученным новым значением.
2. Операторная функция `Integer::operator+=(const Integer&)` возвращает управление, если ей не приходится изменять тип; в противном случае после удаления своего объекта она возвращает новый объект другого типа.

По вполне понятным причинам я назову вторую из этих функций *заменяющей*. Заменяющие функции обладают одной экзотической (если не выразиться сильнее) особенностью: нельзя рассчитывать, что адрес объекта перед вызовом остается действительным и после вызова. Разумеется, пользоваться этим обстоятельством можно лишь в том случае, если эту логику удастся запрятать в самую глубокую и темную дыру, чтобы никто в нее не сунулся, но если это удается сделать, хлопотные алгоритмы невероятно упрощаются.

Показанный пример надежно работает, пока `PNumber` действует как ведущий указатель и пока можно гарантировать, что ни один объект, производный от `NBase`, не будет существовать без ссылающегося на него `PNumber`. В нашем случае, когда все прячется в файле .cpp, дело обстоит именно так.

Для такой простой проблемы программа получилась довольно большой. Я не утверждаю, что ваши хлопоты оправдаются во всех проектах. Мое решение в основном предназначено для ситуаций, в которых вы тратите много времени на разработку иерархии классов многократного использования и можете позволить себе потратить время на повышение модульности. Я привел его, поскольку оно соответствует основной идее книги — выжать из C++ все возможное и невозможное и щедро разбросать головоломки, представляющие интерес даже для самых выдающихся экспертов.

Множественная двойная передача

Множественная передача и все ее разновидности тоже имеют свои аналоги в мире невидимых указателей, но я бы не рискнул рекомендовать их для использования в реальных проектах.

Применение невидимых указателей

В оставшихся главах речь пойдет об управлении памятью. Сейчас я забегу вперед и перечислю некоторые стратегии управления памятью, которые упрощаются за счет применения невидимых указателей.

Кэширование

Кэширование уже упоминалось в контексте обычных умных указателей, однако для невидимых указателей оно приобретает дополнительное значение. Невидимый указатель может содержать адрес указываемого объекта на диске и в последнюю секунду перед тем, как передавать полномочия объекту, считывать объект. Все это происходит незаметно для клиента, поэтому со схемой можно экспериментировать, обходясь без изменений больших объемов кода.

Распределенные объекты и посредники

Раз уж мы заговорили об этом, стоит ли ограничиваться диском, если доступны и другие компьютеры? Благодаря невидимым указателям клиент не заботится о том, находится ли объект, к которому он обращается, на том же компьютере или же он затерян в глобальной сети где-то в горах Тибета. Когда невидимый указатель используется для делегирования удаленному объекту, он называется *посредником* (*proxy*) для этого объекта.

Нетривиальные распределенные архитектуры

В некоторых распределенных архитектурах посредник должен содержать локальный кэшированный образ удаленного объекта. Это приводит к снижению сетевого трафика для редко изменяемых объектов. В частности, в данной стратегии нередко используется схема с *главным маркером (master token)*: чтобы обновить объект, вы должны сначала получить его копию у процесса и компьютера, которым она принадлежит в настоящий момент. Все это может происходить незаметно для клиента, с использование невидимых указателей и различением константных и неконстантных функций.

Невидимые указатели — это замечательная лаборатория, в которой можно поэкспериментировать с различными стратегиями и выяснить, какая из них работает лучше. Методики, в которых реализуются переходные типы, также позволяют оперативно заменять одно представление объекта другим.

Часть 4

Управление памятью

Об управлении памятью и идиомах C++, упрощающих этот процесс, написано на удивление мало — особенно если учесть, сколько сил тратится на такое управление в реальном программировании. Здесь мы займемся этой таинственной темой, начнем с самого простого и перейдем к невероятно сложному. При этом мы будем под разными углами крутить, складывать и расчленять синтаксис C++, чтобы спрятать управление памятью или извлечь из него пользу. Приемы управления памятью вполне могли бы стать темой для отдельной книги. Зажигайте благовония и запевайте мантры!

Перегрузка операторов управления памятью

13

Давайте отдохнем от указателей и поговорим об управлении памятью. Говорят, типичный программист на C++ (если он вообще существует) тратит 50 процентов своего рабочего времени на управление памятью. Когда удалять объект? Как гарантировать, что старый адрес объекта нигде не останется после его уничтожения? Как добиться приличного быстродействия от популярных классов со стандартной схемой управления памятью, которую компилятор использует на все случаи жизни? В отличие от таких языков, как SmallTalk и Lisp, стандартные средства C++ не окажут вам особой помощи в этом вопросе. К счастью, в C++ имеется несколько прекрасных «лазеек»; тот, кто сумеет найти эти обходные пути, при необходимости всегда сможет добиться хорошего быстродействия, а по возможностям управления памятью — осмелюсь ли я произнести это вслух? — его программа не уступит SmallTalk.

Управление памятью — одна из самых мистических тем в компьютерных технологиях; то, над чем бьются выдающиеся умы в великих университетах. Я не претендую на глубокое освещение темы в целом. В оставшейся части книги мы посмотрим, как синтаксис и идиомы C++ помогают построить основу для подключения тех алгоритмов и структур данных, которые вы захотите реализовать. Тем не менее, даже краткие примеры из этой и следующей главы могут пригодиться на практике, если ваша задача не отличается особой сложностью.

Глава начинается с самого важного — перегрузки операторов `new` и `delete`. Затем мы рассмотрим несколько упрощенных, но очень полезных приемов управления памятью в C++. В последующих главах описываются нетривиальные методы, основанные на идеях этой главы.

Перегрузка операторов new и delete

Многие удивляются тому, что операторы `new` и `delete` можно перегружать, как и все остальные операторы. Понять, как это делается, проще всего на примере.

Простой список свободной памяти

Рассмотрим простой пример. Оператор `delete` включает освобождаемые блоки в список свободной памяти. Оператор `new` сначала пытается выделить блок из списка и обращается к глобальному оператору `new` лишь в том случае, если список свободной памяти пуст.

```
class Foo {
private:
	struct FreeNode {
		FreeNode* next;
	};
	static FreeNode* fdFreeList;
```

```
public:
    void* operator new(size_t bytes)
    {
        if (fgFreeList == NULL)
            return ::operator new(bytes);
        FreeNode* node = fgFreeList;
        FgFreeList = fgFreeList->next;
        return node;
    }
    void operator delete(void* space)
    {
        ((FreeNode*)space->next = fgFreeList;
        fgFreeList = (FreeNode*)space;
    }
};
```

Как вы вскоре убедитесь, приведенный фрагмент неполон, однако он демонстрирует общие принципы перегрузки операторов `new` и `delete` для конкретного класса. Оператор new получает один аргумент, объем выделяемого блока, и возвращает адрес выделенного блока. Аргументом оператора `delete` является адрес освобождаемой области. Не пытайтесь объявлять их виртуальными; компилятор лишь посмеется над вами. При вызове оператора `new` компилятор точно знает, с каким классом он имеет дело, поэтому v-таблица ему не нужна. При вызове оператора `delete` деструктор определяет, какому классу этот оператор должен принадлежать. Если вы хотите гарантировать, что будет вызываться оператор `delete` производного класса, то виртуальным нужно сделать деструктор, а не оператор `delete`. Перегрузки будут унаследованы производными класса `Foo`, поэтому это повлияет и на процесс выделения/освобождения памяти в них. На практике нередко создается абстрактный базовый класс, который не делает почти ничего (как и в приведенном примере) и используется только для создания классов с данной схемой управления памятью.

```
class Bar : public Baseclass, public Foo { ... };
```

Здесь `Bar` наследует все базовые характеристики типа `Baseclass`, а нестандартное управление памятью — от `Foo`.

Ограничения минимального размера

Перед тем как хвататься за компилятор, необходимо привести в порядок показанный фрагмент. Во-первых, предполагается, что экземпляр `Foo` содержит по крайней мере не меньше байт, чем `Foo::FreeNode*`. Для классов вроде нашего, не имеющего переменных и виртуальных функций, этого гарантировать нельзя. Он будет иметь определенный размер (во многих компиляторах — два байта), чтобы объекты обладали уникальным адресом, но по количеству байт он может быть меньше указателя на `FreeNode`. Мы должны гарантировать, что размер `Foo` не меньше размера указателя — для этого нужно включить в него v-таблицу или хотя бы переменные, дополняющие его до размера указателя.

Производные классы с добавленными переменными

Другая проблема заключается в том, что приведенный фрагмент не работает с производными классами, в которых добавляются новые переменные. Рассмотрим следующую иерархию классов.

```
class Bar : public Foo {
private:
    int x;
};
```

Каждый экземпляр `Bar` по крайней мере на пару байт больше, чем экземпляр `Foo`. Если удалить экземпляр `Foo`, а затем попытаться немедленно выделить память для экземпляра `Bar`… караул! Выделенный блок оказывается на ту же на пару байт короче. Возможное силовое решение — сделать оператор `new` достаточно умным для того, чтобы он перехватывал только попытки выделения правильного количества байт. Позднее будут рассмотрены и более изящные решения.

```
class Foo {
public:
    void* operator new(size_t bytes)
    {
        if (bytes != sizeof(Foo) || fgFreeList == NULL)
            return ::operator new(bytes);
        FreeNode* node = fgFreeList;
        FgFreeList = fgFreeList->next;
        Return node;
    }
};
```

Мы избавились лишь от проблем, связанных с выделением памяти. Процесс освобождения необходимо изменить в соответствии с этой стратегией. Альтернативная форма оператора `delete` имеет второй аргумент — количество освобождаемых байт. На первый взгляд кажется, что из затруднений появился изящный выход:

```
class Foo {
public:
    void* operator new(size_t bytes);     // См. выше
    void operator delete(void* space, size_t bytes)
    {
        if (bytes != sizeof(Foo))
            ::operator delete(space);
        ((FreeNode*)space)->next = fgFreeList;
        fgFreeList = (FreeNode*)space;
    }
};
```

Теперь в список будут заноситься только настоящие `Foo` и производные классы, совпадающие по размеру. Неплохо, но есть одна проблема. Как компилятор поведет себя в следующем фрагменте?

```
Foo* foo = new Bar;
delete foo;   // Какой размер будет использован компилятором?
```

`Bar` больше `Foo`, поэтому `Foo::operator new` перепоручает работу глобальному оператору `new`. Но когда подходит время освобождать память, компилятор все путает. Размер, передаваемый `Foo::operator delete`, основан на догадке компилятора относительно настоящего типа, а эта догадка может оказаться неверной. В данном случае мы сказали компилятору, что это `Foo`, а не `Bar`; компилятор ухмыляется и продолжает играть по нашим правилам. Чтобы справиться с затруднениями, необходимо знать точную последовательность уничтожения, возникающую в операторах вида `delete foo;`. Сначала вызываются деструкторы, начиная с производного класса, и далее вверх по цепочке. Затем оператор `delete` вызывается кодом, окружающим деструктор производного класса. Это означает, что проблема возникает только для невиртуальных деструкторов. Если деструктор является виртуальным, аргумент размера в операторе `delete` всегда будет правильным — 2438-й довод в пользу применения виртуальных деструкторов, если только у вас не находится действительно веских причин против них.

Рабочий класс списка свободной памяти

Учитывая все сказанное, следующий фрагмент всегда будет правильно работать на компиляторах, использующих v-таблицы.

```
class Foo {
private:
    struct FreeNode {
        FreeNode* next;
    };
```

```
        static FreeNode* fdFreeList;
    public:
        virtual ~Foo() {}
        void* operator new(size_t bytes)
        {
            if (bytes != sizeof(Foo) || fgFreeList == NULL)
                return ::operator new(bytes);
            FreeNode* node = fgFreeList;
            FgFreeList = fgFreeList->next;
            return node;
        }
        void operator delete(void* space, size_t bytes)
        {
            if (bytes != sizeof(Foo))
                return ::operator delete(space);
            ((FreeNode*)space)->next = fgFreeList;
            fgFreeList = (FreeNode*)space;
        }
    };
```

Указатель v-таблицы гарантирует, что каждый `Foo` по крайней мере не меньше указателя на следующий элемент списка (`FreeNode*`), а виртуальный деструктор обеспечивает правильность размера, передаваемого оператору `delete`.

Повторяю: рассмотренная схема управления памятью не предназначена для практического применения (встретив производный класс, она собирает вещи и отправляется домой). Она лишь демонстрирует некоторые базовые принципы перегрузки операторов `new` и `delete`.

Наследование операторов new и delete

Если перегрузить операторы `new` и `delete` для некоторого класса, перегруженные версии будут унаследованы производными классами. Ничто не помешает вам снова перегрузить `new` и/или `delete` в одном из этих производных классов.

```
    class Bar : public Foo {
    public:
        virtual ~Bar();    // Foo::~Foo тоже должен быть виртуальным
        void* operator new(size_t bytes);
        void operator delete(void* space, size_t bytes);
    };
```

С виртуальным деструктором все работает. Если деструктор не виртуальный, в следующем фрагменте будет вызван правильный оператор `new` и оператор `delete` базового класса:

```
    Foo* foo = new Bar;
    delete foo;
```

Хотя этот фрагмент работает, подобное переопределение перегруженных операторов обычно считается дурным тоном. Во всяком случае в кругу знатоков C++ о таких вещах не говорят. Когда производный класс начинает вмешиваться в управление памятью базового класса, во всей программе начинают возникать непредвиденные эффекты. Если вам захочется использовать несколько стратегий управления памятью в одной иерархии классов, лучше сразу включить нужную стратегию в конкретный производный класс средствами множественного наследования, чем унаследовать ее и потом заявить в производном классе: «Ха-ха, я пошутил».

Аргументы оператора new

Оператор new можно перегрузить так, чтобы помимо размера он вызывался и с другими дополнительными аргументами. Перегрузка лишает вас стандартной сигнатуры void* operator new(size_t), и, если вам этого не хочется, ее придется включить в программу вручную.

```
#define kPoolSize 4096
struct Pool {
    unsigned char* next;   // Следующий свободный байт
    unsigned char space[kPoolSize];
    Pool() : next(&space[0]) {}
};
class Foo {
public:
    void* operator new(size_t bytes)
        { return ::operator new(bytes); }
    void* operator new(size_t bytes, Pool* pool)
    {
        void* space = pool->next;
        pool->next += bytes;
        return space;
    }
};
void f()
{
    Pool localPool;
    Foo* foo1 = new Foo;   // Использует оператор new по умолчанию
    Foo* foo2 = new(&localPool) Foo;     // Использует перегрузку
}
```

Здесь клиент, а не класс указывает, где следует разместить объект. Показан лишь фрагмент полной стратегии. Например, как оператор delete узнает, откуда была взята память — из глобального пула, используемого оператором new по умолчанию, или настандартного пула, который используется перегруженным оператором new? Впрочем, основная идея проста: предоставить клиенту класса некоторую степень контроля над размещением экземпляров в памяти. Это означает, что способ выделения памяти может выбираться для конкретных объектов и не обязан совпадать для всех экземпляров класса.

Оператор new можно перегружать с любыми новыми сигнатурами при условии, что все они различаются, а первым аргументом каждой перегруженной версии является size_t — количество нужных байт. Перегрузки могут быть как глобальными, так и принадлежать конкретным классам. Когда компилятор встречает аргументы между new и именем класса, он подставляет размер в начало списка аргументов и ищет подходящую сигнатуру.

Конструирование с разделением фаз

Эта идиома предложена Джеймсом Коплином (James Coplien), который назвал ее «виртуальным конструктором». Что делает следующий перегруженный оператор new?

```
class Foo {
public:
    void* operator new(size_t, void* p) { return p; }
};
```

На первый взгляд — ничего; пустая трата времени. Но так ли это? Что произойдет в следующем фрагменте?

```
union U {
    Foo foo;
    Bar bar;
    Banana banana;
};
U whatIsThis;
```

Компилятор C++ не может определить, какой конструктор следует вызывать для `whatIsThis` — `Foo::Foo()`, `Bar::Bar()` или `Banana::Banana()`. Разумеется, больше одного конструктора вызывать нельзя, поскольку все члены занимают одно и то же место в памяти, но без инструкций от вас не может выбрать нужный конструктор. Как и во многих других ситуациях, компилятор поднимет руки; он сообщает об ошибке и отказывается принимать объединение, члены которого имеют конструкторы. Если вы хотите, чтобы одна область памяти могла инициализироваться несколькими различными способами, придется подумать, как обмануть компилятор. Описанный выше «пустой» конструктор подойдет лучше всего.

```
unsigned char space[4096];
Foo* whatIsThis = new(&space[0]) Foo;
```

Фактически происходит то, что в C++ происходить не должно — вызов конструктора. При этом память на выделяется и не освобождается, поскольку оператор `new` ничего не делает. Тем не менее, компилятор C++ сочтет, что это новый объект, и *все равно вызовет конструктор*. Если позднее вы передумаете и захотите использовать ту же область памяти для другого объекта, то сможете снова вызвать хитроумный оператор `new` и инициализировать ее заново.

При создании объекта оператором new компилятор всегда использует двухшаговый процесс:

1. Выделение памяти под объект.
2. Вызов конструктора объекта.

Этот код запрятан в выполняемом коде, генерируемом компилятором, и в обычных ситуациях второй шаг не выполняется без первого. Идиома виртуального конструктора позволяет надеть повязку на глаза компилятору и обойти это ограничение.

Оперативное изменение типа объекта

Если позднее `Foo` вам надоест и вы захотите использовать ту же область для `Banana`, то при наличии у Banana того же перегруженного оператора new вы сможете быстро сменить тип объекта.

```
Banana* b = new(&space[0]) Banana;
```

Пуф! Был `Foo`, стал `Banana`. Это и называется идиомой *виртуального конструктора*. Такое решение полностью соответствует спецификации языка.

Ограничения

Применяя эту идиому, необходимо помнить о двух обстоятельствах:

1. Область, передаваемая оператору `new`, должна быть достаточна для конструирования класса.
2. Об изменении должны знать все клиенты, хранящие адрес объекта!

Будет весьма неприятно, если вы сменили тип объекта с `Foo` на `Banana` только для того, чтобы какой-нибудь клиентский объект тут же вызвал одну из функций `Foo`.

Уничтожение с разделением фаз

Объект, переданный в качестве аргумента оператору `delete`, обычно уничтожается компилятором в два этапа:

1. Вызов деструктора.
2. Вызов оператора `delete` для освобождения памяти.

Довольно часто мы качаем головой и говорим: «Хорошо бы вызвать деструктор, но не трогать память». Допустим, вы разместили объект в пуле, а теперь не хотите, чтобы часть локально созданного пула

вернулась в главное хранилище памяти. По аналогии с тем, как мы разделили двухшаговый процесс конструирования, можно разделить и двухшаговый процесс уничтожения, напрямую вызывая деструкторы. Однако в отличие от тех выкрутасов, которыми сопровождалось разделение процесса конструирования, с уничтожением дело обстоит очень просто — достаточно вызвать деструктор так, словно это обычная функция класса.

```
void f()
{
    Pool localPool;
    Foo* foo1 = new Foo;    // Использует оператор new по умолчанию
    Foo* foo2 = new(&localPool) Foo;      // Использует перегрузку
    delete foo1;       // Для оператора new по умолчанию
    foo2->~Foo();      // Прямой вызов деструктора
}
```

`localPool` — большой блок памяти, локальный по отношению к функции. Поскольку он создается в стеке, при завершении `f()` он выталкивается из стека. Выделение происходит молниеносно, поскольку локальные объекты заполняют пул снизу вверх. Освобождение происходит еще быстрее, поскольку уничтожается сразу весь пул. Единственная проблема заключается в том, что компилятор не будет автоматически вызывать деструкторы объектов, созданных внутри `localPool`. Вам придется сделать это самостоятельно, используя только что описанную методику.

Кто управляет выделением памяти?

Довольно разговоров о конкретных механизмах; поговорим об архитектуре. Существуют три основные стратегии для определения того, где объект будет находиться в памяти и как занимаемая им память в конечном счете возвратится в систему:

1. Глобальное управление.
2. Управление в классах.
3. Управление под руководством клиента.

Впрочем, это не окончательная классификация: клиентский код может определять, а может и не определять место размещения объектов. Эти решения могут приниматься, а могут и не приниматься объектами классов. Наконец, вся тяжелая работа может выполняться ведущими указателями, а может и не выполняться.

Глобальное управление

По умолчанию объекты создаются глобальным оператором `new` и уничтожаются глобальным оператором `delete`. Перегрузка этих операторов позволяет вам реализовать нестандартную схему управления памятью, но это считается дурным тоном.

- Очень трудно объединить раздельно написанные библиотеки, каждая из которых перегружает заданные по умолчанию операторы `new` и `delete`.
- Ваши перегрузки влияют не только на ваш код, но и на код, написанный другими (включая библиотеки, для которых нет исходных текстов).
- Все перегрузки, принадлежащие конкретному классу, перегружают ваши глобальные версии. На языке С++ это звучит так, словно вы заказываете чай одновременно с молоком и лимоном. Если вам захочется проделать нечто подобное у себя дома или в офисе — пожалуйста, но я не советую упоминать об этом на семинарах по С++.
- Пользователи могут изменить вашу предположительно глобальную стратегию, перегружая операторы `new` и `delete` в конкретных классах. Перегрузка стандартных глобальных операторов дает меньше, чем хотелось бы.

Выделение и освобождение памяти в классах

Перегрузка операторов new и delete как функций класса несколько повышает ваш контроль над происходящим. Изменения относятся только к данному классу и его производным классам, так что побочные эффекты обычно оказываются минимальными. Такой вариант работает лучше всего при выделении нестандартной схемы управления памятью в отдельный класс и его последующем подключении средствами множественного наследования. Для некоторых схем управления памятью такая возможность исключается, но это уже трудности архитектора — показать, почему ее *не следует* реализовывать на базе подключаемых классов.

Если управление памятью реализовано в классе и вы можете создать от него производный класс, деструктор следует сделать виртуальным, чтобы тот же класс мог и исвобождать память. Производные классы не должны перегружать перегруженные версии.

Управление памятью под руководством клиента

Как демонстрируют приведенные выше фрагменты, клиентский код может выбирать, где объект должен находиться в памяти. Обычно это делается с помощью перегруженного оператора new, имеющего дополнительные аргументы помимо size_t. В управлении памятью открываются новые перспективы — управление на уровне отдельных объектов, а не класса в целом. К сожалению, эта стратегия перекладывает на клиента хлопоты, связанные с освобождением памяти. Реализация получается сложной, а модульность — низкой. Например, стоит изменить аргументы нестандартного оператора new, и вам придется вносить изменения во всех местах клиентского кода, где он используется. Пока все перекомпилируется заново, можно погулять на свежем воздухе. Впрочем, несмотря на все проблемы, эта стратегия легко реализуема, очень эффективна и хорошо работает в простых ситуациях.

Объекты классов и производящие функции

Расположение объекта в памяти также может выбираться объектом класса или производящей функцией (или функциями). Возможны разные варианты, от простейших (например, предоставление одной стратегии для всего класса) до выбора стратегии на основании аргументов, переданных производящей функции. При использовании нескольких стратегий вы неизменно придете к стратегии невидимых указателей, рассмотренной в следующем разделе. Более того, эти две концепции прекрасно работают вместе.

Управление памятью с применением ведущих указателей

Похоже, я соврал; в этой главе нам все же придется вернуться к умным указателям. Управление памятью под руководством клиента можно усовершенствовать, инкапсулируя различные стратегии в умных ведущих указателях. Расширение архитектуры с локальными пулами демонстрирует основную идею, которая может быть приспособлена практически для любой схемы с управлением на уровне объектов.

Специализированные ведущие указатели

Простейшая стратегия заключается в создании специализированного класса ведущего указателя или шаблона, который знает о локальном пуле и использует глобальную перегрузку оператора new.

```
struct Pool { ... };    // Как и раньше
void* operator new(Pool* p);  // Выделение из пула
template <class Type>
class PoolMP {
private:
     Type* pointee;
     PoolMP(const PoolMP<Type>&) {}  // Копирование не разрешено...
     PoolMP<Type>& operator=(const PoolMP<Type>&)
          { return *this; }          // ...и присваивание тоже
public:
```

```
        PoolMP(Pool* p) : pointee(new(p) Type) {}
        ~PoolMP() { pointee->~Type(); }
        Type* operator->() const { return pointee; }
    };
```

При желании клиент может использовать PoolMP для выделения и освобождения памяти в локальном пуле. Деструктор ведущего указателя вызывает деструктор указываемого объекта, но не освобождает память. Поскольку ведущий указатель не следит за исходным пулом, копирование и присваивание поддерживать не удастся, так как ведущий указатель понятия не имеет, в каком пуле создавать новые копии. Если не считать этих недостатков, перед нами фактически простейший указатель, не отягощенный никакими издержками.

На это можно возразить, что копирование и присваивание все же следует поддерживать, но с использование операторов new и delete по умолчанию. В этом случае конструктор копий и оператор = работают так же, как и для обычного ведущего указателя.

Обратные указатели на пул

Чтобы поддерживать копирование и присваивание в пуле, можно запоминать адрес пула.

```
    template <class Type>
    class PoolMP {
    private:
        Type* pointee;
        Pool* pool;
    public:
        PoolMP(Pool* p) : pointee(new(p) Type), pool(p) {}
        ~PoolMP() { pointee->Type::~Type(); }
        PoolMP(const PoolMP<Type>& pmp) : pointee(new(pool) Type(*pointee)) {}
        PoolMP<Type>& operator=(const PoolMP<Type>& pmp)
        {
            if (this == &pmp) return *this;
            delete pointee;
            pointee = new(pool) Type(*pointee);
            return *this;
        }
        Type* operator->() const { return pointee; }
    };
```

Это обойдется вам в четыре лишних байта памяти, но не потребует лишних тактов процессора по сравнению с использованием обычных ведущих указателей.

Сосуществование с обычными ведущими указателями

Предложенное решение отнюдь не идеально. Интерфейс PoolMP открывает многое из того, о чем следовало бы знать только классам. Более того, если вам захочется совместно работать с объектами из пула и объектами, размещенными другим способом (например, с помощью стандартного механизма), начинаются настоящие трудности. Ценой добавления v-таблицы мы сможем значительно лучше инкапсулировать отличия в стратегиях управления памятью.

```
    template <class Type>
    class MP {
    protected:
        MP(const MP<Type>&) {}  // Копирование не разрешено
        MP<Type>& operator=(const MP<Type>&)
            { return *this; }    // Присваивание - тоже
        MP() {}  // Используется только производными классами
```

```
public:
    virtual ~MP() {}   // Освобождение выполняется производными классами
    virtual Type* operator->() const = 0;
};

template <class Type>
class DefaultMP : public MP<Type> {
private:
    Type* pointee;
public:
    DefaultMP() : pointee(new Type) {}
    DefaultMP(const DefaultMP<Type>& dmp)
        : pointee(new Type(*dmp.pointee)) {}
    virtual ~DefaultMP() { delete pointee; }
    DefaultMP<Type>& operator=(const DefaultMP<Type>& dmp)
    {
        if (this == &dmp) return *this;
        delete pointee;
        pointee = new Type(*dmp.pointee);
        return *this;
    }
    virtual Type* operator->() const { return pointee; }
};

template <class Type>
class LocalPoolMP : public MP<Type> {
private:
    Type* pointee;
    Pool* pool;
public:
    LocalPoolMP(Pool* p)
        : pointee(new(p) Type), pool(p) []
    LocalPoolMP(const LocalPoolMP<Type>& lpmp)
        : pointee(new(lpmp.pool) Type(*lpmp.pointee)), pool(lpmp.pool) {}
    virtual ~LocalPoolMP() { pointee->Type::~Type(); }
    LocalPoolMP<Type>& operator=(const LocalPoolMP<Type>& lpmp)
    {
        if (this == &lpmp) return *this;
        pointee->Type::~Type();
        pointee = new(pool) Type(*lpmp.pointee);
        return *this;
    }
    virtual Type* operator->() const { return pointee; }
};
```

Теперь `DefaultMP` и `LocalPoolMP` можно использовать совместно — достаточно сообщить клиенту, что они принадлежат к типу `MP<Type>&`. Копирование и присваивание поддерживается для тех классов, которые взаимодействуют с производными классами, но запрещено для тех, которые знают только о базовом классе. В приведенном коде есть одна тонкость: операторная функция `LocalPoolMP::operator=` всегда использует `new(pool)` вместо `new(lpmp.pool)`. Это повышает

безопасность в тех ситуациях, когда два ведущих указателя поступают из разных областей действия и разных пулов.

Невидимые указатели

Раз уж мы «заплатили вступительный взнос» и создали иерархию классов ведущих указателей, почему бы не пойти дальше и не сделать эти указатели невидимыми? Вместо применения шаблона нам придется реализовать отдельный класс указателя для каждого класса указываемого объекта, но это не слишком большая цена за получаемую гибкость.

```
// В файле foo.h
class Foo {
public:
     static Foo* make();        // Использует выделение по умолчанию
     static Foo* make(Pool*);  // Использует пул
     virtual ~Foo() {}
     // Далее следуют чисто виртуальные функции
};
// В файле foo.cpp
class PoolFoo : public Foo {
private:
     Foo* foo;
     Pool* pool;
public:
     PoolFoo(Foo* f, Pool* p) : foo(f), pool(p) {}
     virtual ~PoolFoo() { foo->~Foo(); }
     // Переопределения функций класса, делегирующие к foo
};
class PFoo : public Foo {
// Обычный невидимый указатель
};
class ConcreteFoo : public Foo { ... };
Foo* Foo::make()
{
     return new PFoo(new ConcreteFoo);
}
Foo* Foo::make(Pool* p)
{
     return new PoolFoo(new(p) ConcreteFoo, p);
}
```

Такой вариант намного «чище» для клиента. Единственное место, в котором клиентский код должен знать что-то о пулах, — создание объекта функцией `make(Pool*)`. Остальные пользователи полученного невидимого указателя понятия не имеют, находится их рабочий объект в пуле или нет.

Стековые оболочки

Чтобы добиться максимальной инкапсуляции, следует внести в описанную архитектуру следующие изменения:

- Сделать `Pool` чисто абстрактным базовым классом с инкапсулированными производными классами, производящими функциями и т.д.
- Предоставить функцию `static Foo::makePool()`. Функция `make(Pool*)` будет работать и для других разновидностей `Pool`, но `makePool()` позволяет `Foo` выбрать производящую функцию `Pool`, оптимальную для хранения `Foo` (например, с передачей размера экземпляра).

- Переработать старый шаблон `MP` из предыдущих глав (с операторной функцией `operator Type*()`), чтобы при выходе из пула и указателей за пределы области действия все необходимое автоматически уничтожалось.

Ниже показан примерный вид полученного интерфейса, с фрагментом клиентского кода и без виртуального оператора =.

```
// В файле foo.h
// Подключить объявление чисто абстрактного базового класса
#include “pool.h”
class Foo {
private:
    Foo(const Foo&) {}
    Foo& operator=(const Foo&) { return *this; }
public:
    static Pool* makePool();  // Создать пул, оптимизированный для Foo
    static Foo* make();       // Не использует пул
    static Foo* make(Pool*);  // Использует пул
    // И т.д.
};
// Клиентский код
void g(Foo*);
void f()
{
    MP<Pool> pool(Foo::makePool());
    MP<Foo> foo(Foo::make(pool));
    foo->MemberOfFoo();     // Использует операторную функцию operator->()
    g(foo);            // Использует операторную функцию operator Type*()
    // Выход из области действия – удаляется сначала foo, затем pool
}
```

Перспективы

Глава заканчивается хорошо — умной, эффективной инкапсуляцией очень сложной проблемы дизайна. Единственным уязвимым местом является вызов функции `g()`, которая должна пообещать не сохранять долговременный указатель на свой аргумент. Впрочем, подобный анализ необходимо проводить для любой архитектуры, в которой используются временные пулы; в нашем случае ключом является инкапсуляция.

На время забудьте о пулах, временных или иных, и вы увидите разнообразные стратегии применения ведущих указателей для поддержки и инкапсуляции управления памятью в C++.

Основы управления памятью

14

В этой главе описаны некоторые простые стратегии управления памятью. Если вы пропустили предыдущую главу, вернитесь и прочитайте ее. Весь материал, изложенный в этой и следующих главах, требует досконального понимания базового материала, приведенного выше.

Первая группа стратегий имеет одну общую черту: клиентский код сам решает, когда следует удалить объекты и вернуть память в систему. Иногда это осуществляется косвенно, но в любом случае память возвращается лишь после выполнения клиентом определенных действий.

Стратегии второй группы построены на концепции подсчета ссылок. Это первый пример автоматической сборки мусора — темы, которая будет обобщена в последних главах книги. Подсчет ссылок способен принести огромную пользу, но, как мы вскоре увидим, он также обладает рядом довольно жестких ограничений.

Наконец, мы рассмотрим общую концепцию, на основе которой строятся более изощренные методики: пространство памяти. На самом деле это всего лишь новый подход к низкоуровневым методикам, при котором они предстают в свете архитектуры, а не оптимизации.

Строительные блоки

В числе основных принципов нестандартного управления памятью в C++ должен быть следующий: «Придумайте откровенную глупость; вполне возможно, из этого что-нибудь получится». Даже если идея не сработает сама по себе, она может пригодиться в качестве отправной точки.

Поблочное освобождение памяти

Если выделение и освобождение памяти плохо влияет на быстродействие программы, иногда самое простое решение проблемы заключается в выполнении операций с блоками. Память выделяется снизу блока к его верху, а возвращается в систему сразу целым блоком (а не отдельными объектами).

Фиктивное удаление

Задача многих программ — побыстрее отработать свое и уйти. Это особенно справедливо в среде Unix, где сценарии оболочки объединяют множество крошечных, недолговечных программ. Нередко выделение памяти для новых объектов оказывается самым серьезным фактором, снижающим быстродействие таких программ. Простая стратегия оптимизации заключается в том, что вы выделяете память под объекты снизу вверх большого блока *и не удаляете их*.

```
struct Pool {
    static Pool* gCurrentPool;      // Пул для выделения памяти
    enum { block_size = 8096 };     // Выберите свой любимый размер
    unsigned char* space;           // Следующая выделяемая область
    size_t remaining;               // Количество оставшихся байт в блоке
```

```
    Pool() : space((unsigned char*)calloc(block_size, '\0')),
        remaining(block_size) {}
    void* Allocate(size_t bytes)
    {
        if (bytes > block_size)
            return ::operator new(bytes);     // Слишком большой запрос
        if (gCurrentPool == NULL || bytes ? remaining)
            gCurrentPool = new Pool;
        void* memory = space;
        space += bytes;
        remaining -= bytes;
        return memory;
    }
};
class Foo {
public:
    void* operator new(size_t bytes)
    {
        if (Pool::fCurrentPool == NULL)
            Pool::gCurrentPool = new Pool;
        return Pool::gCurrentPool->Allocate(bytes);
    }
    void operator delete(void*) {}
};
```

Быстрее некуда! Выделение занимает лишь несколько машинных тактов, а освобождение происходит мгновенно. Конечно, этот код не завоюет приза на олимпиаде по C++, но я видел, как он всего за несколько часов работы спасал проекты с серьезными проблемами быстродействия. Как минимум, он поможет определить, на что лучше направить усилия по оптимизации, поскольку выделение и освобождение памяти исключается из рассмотрения. Последовательно применяя его к разным классам, вы сможете установить, с какими классами связаны основные затруднения.

Обратите внимание, что для выделения блоков вместо операторной функции `::operator new` используется функция `calloc()`. Большинство компиляторов C++ выделяет большой блок с помощью функции `calloc()` (или функции операционной системы), а затем управляет объектами в полученном блоке. Если использовать `::operator new` для выделения блоков, скорее всего, дело кончится двойными затратами и двойной фрагментацией, поскольку эти блоки будут существовать в стандартных блоках менеджера памяти. При нашей стратегии лучше обойти `::operator new`.

Описанная стратегия также хорошо работает в программах с более продолжительным сроком жизни, которые изначально создают множество объектов некоторого класса, но удаляют их относительно редко (если вообще удаляют). Если операторы `new` и `delete` перегружаются на уровне классов, то оптимизацию можно ограничить классами, обладающими указанным свойством.

Сборка мусора на уровне поколений

Многие алгоритмы выглядят примерно так:

```
void Eval(Structure s)
{
    // Создать локальные объекты
    Eval(s.SomePart());     // Произвести вычисления для подструктуры
    // Удалить локальные объекты
}
```

Обход деревьев, вычисление рекурсивных выражений в языках типа Prolog — эти и многие другие рекурсивные алгоритмы имеют показанную структуру. Размещение локальных объектов в стеке может ускорить выделение и освобождение памяти, но этот вариант часто непрактичен. Альтернативный вариант — создать пул, локальный для `Eval()`, и уничтожить его целиком при выходе из `Eval()`. В области действия `Eval()` все временные объекты размещаются в этом пуле.

```
void* operator new(size_t size, Pool* p)
{
    return p->Allocate(size);
}
template <class Type>
class PoolP {      // Указатель, использующий пул
private:
    Type* pointee;
public:
    PoolP(Pool* p) : pointee(new(p) Type) {}
    ~PoolP { pointee->Type::~Type(); }
    // Все остальное для ведущих указателей
};
void Eval(Structure s)
{
    Pool p;  // Объявляется в стеке!
    PoolP<Foo> foo(&p);     // Использует пул
    Eval(s.SomePart());     // Использует свой собственный пул
    f(p, s); // Вместо f(s); f будет использовать тот же пул
    // Деструктор p уничтожает все сразу
}
```

`Pool` может представлять собой любую разновидность пула памяти. Скорее всего, это «тупой» пул, который просто выделяет память снизу вверх и не беспокоится о возвращении памяти. Умный указатель `PoolP` вызывает деструктор указываемого объекта, но не освобождает его память. На эту тему существует множество вариантов:

- Обойтись без `PoolP`. Либо пусть пул сам вызывает деструкторы содержащихся в нем объектов из своего собственного деструктора (для этого все они должны иметь общий базовый класс и виртуальный деструктор), либо вообще не вызывайте деструкторы. (О господи! Неужели я сказал *это*? Но довольно часто такой вариант *работает*; главное — не хвастайтесь этим подвигом на семинарах по C++.)
- Назначить «текущий пул» в глобальной переменной или статической переменной класса, а затем перегрузить оператор `new` для использования текущего пула, каким бы он ни был. При этом вам не придется передавать текущий пул всем подфункциям вроде упоминавшейся выше `f()`.
- Предоставить средства для перемещения или копирования объектов из локального пула в то место, где они смогут жить вне области действия `Eval()`. Эта тема выходит за рамки данной главы, а возможно, и книги, но для нее можно приспособить методику дескрипторов из следующей главы.

Последний вариант используется в стратегиях управления памятью настолько сложных, что голова начинает болеть *заранее*, еще до подробного знакомства с темой. К счастью, все трудности обусловлены необходимостью выбора — стоит или не стоит перемещать объекты из-за возможных обращений к ним со стороны чего-то, пережившего данный блок. Этот вопрос не из области C++, он скорее относится к алгоритмам и структурам данных.

Скрытая информация

Многие менеджеры памяти выделяют несколько дополнительных байт в операторе new, чтобы при вызове оператора delete был использован правильный размер независимо от того, является деструктор виртуальным или нет.

```
void* Foo::operator new(size_t bytes)
{
    size_t real_bytes = bytes + sizeof(size_t);
    unsigned char* space = (unsigned char*)::operator new(real_bytes);
    ((size_t)space) = real_bytes;
    return space + sizeof(size_t);
}
```

Теперь информацию о размере можно использовать в операторе delete или в любом другом месте, где вам захочется узнать настоящий размер. Тем не менее, при этой стратегии необходимо учесть ряд обстоятельств.

Лишние затраты

В зависимости от компилятора и операционной системы эта методика уже может использоваться незаметно для вас. Если вы самостоятельно добавите информацию о размере, она может продублировать уже хранящиеся сведения. Скорее всего, это произойдет при делегировании ::operator new на уровне объектов (см. выше). Такая методика приносит наибольшую пользу в сочетании с блочными схемами выделения памяти, она сокращает затраты ::operator new или calloc для блока, а не для отдельного объекта.

Оптимизация размера кванта

Большое преимущество этой методики заключается в том, что вы можете выделить больше места, чем было запрошено. Как это? Может ли принести пользу выделение лишней неиспользуемой памяти? На самом деле может, если подойти к этому разумно. Один из возможных вариантов — всегда выделять память с приращением в n байт, где минимальное значение n равно 4. При это повышается вероятность того, что после удаления 17-байтового объекта занимаемое им место удастся использовать, скажем, для 18- или 19-байтового объекта. Более изощренный подход состоит в выделении памяти по степеням 2, или, если вы относитесь к числу истинных гурманов управления памятью, — по числам Фибоначчи. Такие системы называются *системами напарников (buddy systems)*, поскольку для каждого выделенного блока вы сможете найти его напарника (то есть другую половинку большого блока, из которого он был выделен) исключительно по размеру и начальному адресу. Появляется возможность эффективного воссоединения соседних освобожденных блоков. Если вы интересуетесь подобными вещами, в главе 16 рассматриваются основы системы напарников для степеней 2 в контексте автоматической сборки мусора.

Другая информация

Кроме размера блока может сохраняться и другая информация, например:

- адрес объекта класса для данного объекта;
- флаги блока (например, «флаг зомби», о котором будет рассказано ниже);
- статистическая информация — например, время создания объекта.

Списки свободных блоков

Упрощенный список свободных блоков, приведенный в предыдущей главе, может использоваться только для объектов постоянного размера. Например, он не будет нормально работать с производными классами, в которых добавляются новые переменные, поскольку они будут иметь другой размер. Сам список тоже ограничивался одним размером; для передачи оператору delete правильного размера требовались виртуальные деструкторы. Впрочем, создать более универсальные списки уже не так уж трудно.

Одно из несложных усовершенствований заключается в том, чтобы хранить в каждом узле списка не только адрес следующего указателя в списке, но и размер блока. Это позволит хранить в одном списке блоки разных размеров. При выделении памяти мы просматриваем список, пока не находим блок, размеры которого достаточны для наших целей. Для этого придется хранить скрытую информацию о размере блока даже после его выделения, чтобы при уничтожении объекта восстанавливать весь блок. Стратегия, прямо скажем, тупая и подходит разве что для очень маленьких списков, поскольку время поиска возрастает линейно с размером списка. Впрочем, она послужит отправной точкой для нашего обсуждения.

Более эффективное представление — коллекция списков свободной памяти, в которой каждый список представляет блоки определенного размера. Ниже показана урощенная, но полезная реализация.

```
class MemManager {
private:
    struct FreeList {  // Список с блоками определенного размера
        FreeList* next;       // Следующий FreeList
        void* top_of_list;    // Верх текущего списка
        size_t chunk_size;    // Размер каждого свободного блока
        FreeList(FreeList* successor, size_t size) : next(successor),
            top_of_list(NULL), chunk_size(size) {}
    };
    FreeList* all_lists;    // Список всех FreeList
public:
    MemManager() : all_lists(NULL) {}
    void* Allocate(size_t bytes);
    void Deallocate(void* space, size_t bytes);
};
void* MemManager::Allocate(size_t bytes)
{
    for (FreeList* fl = all_lists;
        fl != NULL && fl->chunk_size != bytes;
        fl = fl->next)
    {
        if (fl->top_of_list != NULL)
        {
            void* space = fl->top_of_list;
            fl->top_of_list = *((void**)(fl->top_of_list));
            return space;
        }
        return ::operator new(bytes);    // Пустой список
    }
    return ::operator new(bytes);   // Такого списка нет
}
void MemManager::Deallocate(void* space, size_t bytes)
{
    FreeList* fl = NULL;
    for (fl = all_lists; fl != NULL; fl = fl->next)
        if (fl->chunk_size == bytes) break;
    if (fl == NULL)    // Списка для такого размера нет
    {
        fl = new FreeList(all_lists, bytes);
        all_lists = fl;
```

```
        }
        *((void**)space) = fl->top_of_list;
        fl->top_of_list = space;
    }
```

Функции `Allocate()` и `Deallocate()` вызываются из перегруженных операторов `new` и `delete` соответственно. Такой подход предельно упрощен, но работает он неплохо. Вы можете воспользоваться им для любого сочетания классов, и он будет работать с производными классами, в которых добавились новые переменные. Он также может использоваться в схеме управления памятью на базе ведущих указателей. Существуют многочисленные усовершенствования, которые можно внести в показанную основу:

- Ограничить размеры блоков числами, кратными некоторому числу байт, степенями 2 или числами Фибоначчи.
- Воспользоваться более эффективной структурой данных, чем связанный список списков — возможно, бинарным деревом или даже массивом, если диапазон размеров невелик.
- Предоставить функцию `Flush()`, которая при нехватке памяти удаляет все содержимое списков.
- В функции `Allocate()` при отсутствии в списке свободного места заданного размера выделить память под массив блоков этого размера вместо одного блока.

Подсчет ссылок

Подсчет ссылок основан на простой идее — мы следим за количеством указателей, ссылающихся на объект. Когда счетчик становится равным 0, объект удаляется. Звучит просто, не правда ли? В определенных условиях все действительно просто, но подсчет ссылок обладает довольно жесткими ограничениями, которые снижают его практическую ценность.

Базовый класс с подсчетом ссылок

Начнем с абстрактного базового класса, от которого можно создать производный класс с подсчетом ссылок. Базовый класс содержит переменную, в которой хранится количество вызовов функции `Grab()` за вычетом количества вызовов функции `Release()`.

```
class RefCount {
private:
    unsigned long count;    // Счетчик ссылок
public:
    RefCount() : count(0) {}
    RefCount(const RefCount&) : count(0) {}
    RefCount& operator=(const RefCount&)
        { return *this; }  // Не изменяет счетчик
    virtual ~RefCount() {}  // Заготовка
    void Grab() { count++; }
    void Release()
    {
        if (count > 0) count --;
        if (count == 0) delete this;
    }
};
```

Пока клиентский код правильно вызывает функции `Grab()` и `Release()`, все работает абсолютно надежно. Каждый раз, когда клиент получает или копирует адрес объекта, производного от `RefCount`, он вызывает `Grab()`. Когда клиент гарантирует, что адрес больше не используется, он вызывает `Release()`. Если счетчик падает до 0 — бац! Нет больше объекта!

Недостаток такой методики очевиден — она слишком полагается на соблюдение всех правил программистом. Можно сделать получше.

Укзатели с подсчетом ссылок

Давайте усовершенствуем базовый класс `RefCount` и создадим модифицированный шаблон умного указателя для любых классов, производных от `RefCount`.

```
template <class Type>
class CP {    // “Указатель с подсчетом ссылок”
private:
     Type* pointee;
public:
     CP(Type* p) : pointee(p) { pointee->Grab(); }
     CP(const CP<Type>& cp) : pointee(cp.pointee)
          { pointee->Grab(); }
     ~CP() { ponintee->Release(); }
     CP<Type>& operator=(const CP<Type>& cp)
     {
          if (this == &cp) return *this;
          pointee->Release();
          pointee = cp.pointee;
          pointee->Grab();
          return *this;
     }
     Type* operator->() { return pointee; }
};
```

Если весь клиентский код будет обращаться к классам с подсчетом ссылок через этот или аналогичный шаблон, подсчет ссылок осуществляется автоматически. При каждом создании новой копии указателя происходит автоматический вызов `Grab()`. При каждом уничтожении указателя его деструктор уменьшает значение счетчика. Единственная опасность заключается в том, что клиент обойдет умный указатель. С этой проблемой можно справиться с помощью производящих функций целевого класса.

```
class Foo : public RefCount {
private:
     Foo();    // Вместе с другими конструкторами
public:
     static CP<Foo> make();  // Создаем экземпляр
     // Далее следует интерфейс Foo
};
```

Тем самым мы гарантируем, что доступ к `Foo` будет осуществляться только через указатель с подсчетом ссылок. Обратите внимание: это не ведущий, а самый обычный умный указатель.

Ведущие указатели с подсчетом ссылок

Даже если вы не хотите модифицировать конкретный класс, чтобы сделать его производным от `RefCount` (например, если он имеет критические требования по быстродействию и объему или входит в коммерческую библиотеку классов), не отчаивайтесь. Подсчет ссылок можно переместить в ведущий указатель.

```
template <class Type>
class CMP {   // “Ведущий указатель с подсчетом ссылок”
private:
     Type* pointee;
     unsigned long count;
```

```
public:
    CMP() : pointee(new Type), count(0) {}
    CMP(const CMP<Type>& cmp)
        : pointee(new Type(*(cmp.pointee))), count(0) {}
    ~CMP() { delete pointee; }      // Независимо от счетчика
    CMP<Type>& operator=(const CMP<Type>& cmp)
    {
        if (this == &cmp) return *this;
        delete pointee;
        pointee = new Type(*(cmp.pointee));
        return *this;
    }
    Type* operator->() const { return pointee; }
    void Grab() { count++; }
    void Release()
    {
        if (count > 0) count--;
        if (count <= 0)
        {
            delete pointee;
            delete this;
        }
    }
};
```

В сущности, это равносильно объединению старого шаблона ведущего указателя с базовым классом `RefCount`. Подсчет ссылок уже не выделяется в отдельный класс, но зато нам снова приходится полагаться на правильное поведение программистов — существ, к сожалению, несовершенных.

Дескрипторы с подсчетом ссылок

На сцену выходит нечто новое: дескриптор (handle) с подсчетом ссылок. По отношению к шаблону `CMP` он станет тем же, чем `CP` был для `RefCount`, — то есть он автоматически вызывает функции `Grab()` и `Release()` в своих конструкторах, деструкторе и операторе =.

```
template <class Type>
class CH {    // "Дескриптор с подсчетом ссылок"
private:
    CMP<Type>* pointee;
public:
    CH(CMP<Type>* p) : pointee(p) { pointee->Grab(); }
    CH(const CH<Type>& ch) : pointee(ch.pointee) { pointee->Grab(); }
    ~CH() { pointee->Release(); }
    CH<Type>& operator=(const CH<Type>& ch)
    {
        if (this == &ch) return *this;
        if (pointee == ch.pointee) return *this;
        pointee->Release();
        pointee = ch.pointee;
        pointee->Grab();
        return *this;
    }
```

```
        CMP<Type> operator->() { return *pointee; }
    };
```

Если использовать дескрипторы в сочетании с ведущими указателями, можно выбрать, для каких экземпляров класса следует подсчитывать ссылки, а какие экземпляры должны управляться другим способом.

Трудности подсчета ссылок

Все выглядит так просто; однако без ложки дегтя дело все же не обходится. Подсчет ссылок обладает одним очень распространенным недостатком — зацикливанием. Представьте себе ситуацию: объект A захватил объект B (то есть вызвал для него функцию `Grab()`), а объект B сделал то же самое для объекта A. Ни на A, ни на B другие объекты не ссылаются. Здравый смысл подсказывает, что A следует удалить вместе с B, но они продолжают существовать, поскольку их счетчики ссылок так и не обнуляются. Обидно, да?

Подобное зацикливание возникает сплошь и рядом. Оно может относиться не только к парам объектов, но и целым подграфам. A -> B -> C -> D -> A, но никто за пределами этой группы не ссылается ни на один из этих объектов. Группа словно плывет на «Летучем Голландце», построенном в эпоху высоких технологий и обреченном на вечные скитания в памяти. Существует несколько стратегий борьбы с зацикливаниями. Все они не обладают особой универсальностью, и в вашей конкретной ситуации это может привести к отказу от подсчета ссылок. Как правило, встречаясь с проблемой циклических ссылок, стоит рассмотреть более хитроумные приемы, описанные в двух последних главах. Как видите, мысль об отказе от подсчета ссылок приходит довольно быстро.

Декомпозиция

Предположим, A захватывает B, а затем B захватывает некоторый компонент A:

```
    class A {
    private:
        Foo* foo;
        B* b;
    };
```

Если сделать так, чтобы B выполнял захват в функции `foo`, проблем не возникает. Когда последняя ссылка на A ликвидируется, его счетчик становится равным 0, поскольку B его не увеличивает. Для этого придется проявить некоторую изрядную изобретательность при кодировании, к тому же дизайн сильно зависит от особенностей конкретных объектов, но на удивление часто он решает проблему зацикливания.

Сильные и слабые дескрипторы

Предположим, ссылка A на B создавалась через `Grab()`, а ссылка B на A — нет. В тот момент, когда исчезнет последняя ссылка на A из внешнего мира, подсчет ссылок для обоих объектов пары прекратит их существование. На этой идее основано различие между *сильными (strong)* и *слабыми (weak)* дескрипторами или указателями с подсчетом ссылок. Описанный выше шаблон `CH` будет относиться к сильным дескрипторам, поскольку поддерживает счетчик ссылок. Обычный шаблон дескриптора (без вызова `Grab` и `Release`) будет относиться к слабым. Если спроектировать архитектуру объектов так, чтобы не существовало циклических подграфов, содержащих исключительно сильные дескрипторы, то вся схема подсчета ссылок снова возвращается в игру. Самая распространенная ситуация с таким решением — иерархия целое/часть, в которой пары удаляются при удалении целого. Целые поддерживают сильные ссылки, части — слабые.

Подсчет ссылок и ведущие указатели

Одно из самых распространенных и полезных применений подсчета ссылок заключается в управлении ведущими указателями. В предыдущих главах эта тема упоминалась неоднократно. Дескрипторы живут в стеке и потому автоматически уничтожаются при сборке мусора, выполняемой компилятором. Однако ведущие указатели (по тем же причинам, что и объекты) обычно приходится создавать в куче. Как узнать, когда следует удалять ведущий указатель? Подсчет ссылок упрощает эту задачу.

Проблем с зацикливанием не будет: поскольку ведущий указатель не хранит ссылок на свои дескрипторы, связь является односторонней. Копируемые и передаваемые дескрипторы сохраняют длину в четыре байта без виртуальных функций, а лишь тривиальными подставляемыми функциями. `Grab` и `Release` съедают несколько дополнительных машинных тактов, но это мелочи по сравнению с тем, что вам пришлось бы проделывать для управления ведущима указателями без них. Несколько лишних байт для счетчика в ведущим указателе не играют особой роли; к тому же они выделяются в куче, степень детализации которой обычно заметно превышает четыре байта.

Возможно, вам стоит вернуться к предыдущим главам и подумать, как использовать показанную схему подсчета ссылок везде, где встечаются ведущие указатели. Это станет ключом к нетривиальному управлению памятью в дальнейших главах.

Пространтсва памяти

Все эти фокусы образуют фундамент для дальнейшего строительства, но относить их к архитектуре было бы неверно. Для действительно нетривиального управления памятью понадобятся нетривиальные организационные концепции. В простейшем случае вся доступная память рассматривается как один большой блок, из которого выделяются блоки меньшего размера. Для этого можно либо напрямую обратиться к операционной системе с требованием выделить большой блок памяти при запуске, либо косвенно, в конечном счете перепоручая работу операторным функциям `::operator new` и `::operator delete`.

В двух последних главах мы взглянем на проблему с нетривиальных позиций и поделим доступную память на *пространства (memory spaces)*. Пространства памяти — это концепция; ее можно реализовать на основе практически любой описанной выше блочно-ориентированной схемы управления памятью. Например, в одном пространстве памяти может использоваться система напарников, а в другом — списки свободной памяти. Концепция представлена в следующем абстрактном базовом классе:

```
class MemSpace {
public:
    void* Allocate(size_t bytes) = 0;
    void Deallocate(void* space, size_t bytes) = 0;
};
```

(Если ваш компилятор поддерживает обработку исключений, при объявлении обоих функций следует указать возможность инициирования исключений.) Некоторым пространствам памяти можно не сообщать в функции `Deallocate()` размер возвращаемых блоков; для конкретных схем могут появиться другие функции, но минимальный интерфейс выглядит именно так. Возможно, также будет поддерживаться глобальная структура данных — коллекция всех `MemSpace` (причины рассматриваются ниже). Коллекция должна эффективно отвечать на вопрос: «Какому пространству памяти принадлежит данный адрес?» По имеющемуся адресу объекта вы определяете пространство памяти, в котором он живет.

В реализации пространств памяти могут быть использованы любые методики, описанные в предыдущей главе:

- Глобальная перегрузка операторов `new` и `delete` (обычно не рекомендуется).
- Перегрузка операторов `new` и `delete` на уровне класса.
- Использование оператора `new` с аргументами под руководством клиента.
- Использование оператора `new` с аргументами на базе ведущих указателей.

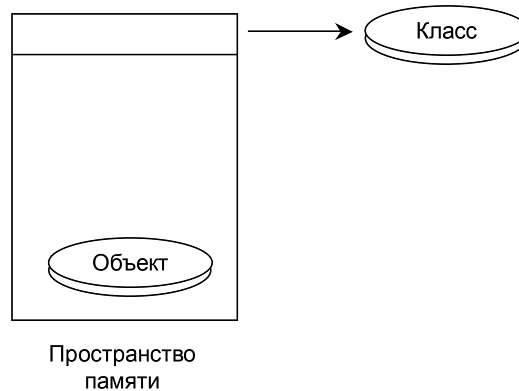
Существует немало причин для деления памяти на пространства. Ниже описаны некоторые распространенные стратегии выбора объектов, которые должны находиться в одном пространстве памяти.

Деление по классам

В предыдущих главах мы говорили об объектах классов, но так и не ответили напрямую на вопрос: как определить класс объекта для имеющегося объекта? Простейшее решение — добавить переменную,

которая ссылается на объект класса. К сожалению, оно связано с заметными затратами как по памяти, так и кода конструктора. Однако существуют два других варианта:

1. Хранить указатель на объект класса в ведущем указателе. Получаем те же затраты, но с улучшенной инкапсуляцией.
2. Выделить все экземпляры некоторого класса в одно пространство памяти и хранить указатель на объект класса в начале пространства (см. рисунок).



Второй вариант существенно снижает затраты при условии, что по адресу объекта можно эффективно определить начало адресного пространства памяти, которому он принадлежит (возможно, с помощью упомянутой выше коллекции пространств памяти).

На первый взгляд кажется, что устраивать такую суету вокруг простейшей проблемы глупо. Почему бы просто не добавить дополнительную переменную, ссылающуюся на объект класса? Приведу по крайней мере две причины:

1. Класс, с которым вы работаете, может входить в коммерческую библиотеку, для которой у вас нет исходных текстов, или его модификация нежелательна по другим причинам.
2. Класс может представлять собой тривиальную объектно-ориентированную оболочку для примитивного типа данных (скажем, `int`). Лишние байты для указателя на объект класса (не говоря уже о v-таблице, которая неизбежно потребуется для виртуальных функций доступа к нему) могут оказаться весьма существенными.

Деление по размеру

Вполне разумно объединить все объекты одинакового размера (или принадлежащие одному диапазону размеров) в одно пространство памяти для оптимизации создания и уничтожения. Многие стратегии управления памятью работают для одних диапазонов лучше, чем для других. Например, при создании множества больших, сильно различающихся по размерам объектов схема со степенями 2 наверняка оставит много неиспользованных фрагментированных блоков. Однако для объектов относительно малых (или близких по размеру к степеням 2) такая схема работает просто замечательно. В крайних случаях все пространство памяти может заполняться объектами одинакового размера. Такие пространства обладают чрезвычайно эффективным представлением и легко управляются.

Деление по способу использования

Еще один возможный вариант — делить объекты в соответствии со способом их использования. Например, для многопоточного приложения-сервера объекты можно разделить по клиентам. Редко используемые объекты могут находиться в одном пространстве, а часто используемые — в другом. Объекты, кэшируемые на диске, могут отделяться от объектов, постоянно находящихся в памяти. Деление может осуществляться как на уровне классов, так и на уровне отдельных объектов.

Деление по средствам доступа

Другая важная причина для разделения по пространствам памяти заключается в том, чтобы хранить по отдельности объекты, доступ к которым осуществляется из стека, из других процессов или чисто

внутренние из кучи. Как будет показано в последних главах, это играет важную роль в схемах уплотнения и сборки мусора.

В двух следующих главах эта методика будет применяться довольно часто. Перемещаемые объекты отделяются от объектов, остающихся на одном месте. Ведущие указатели, доступные из стека, находятся в одном пространстве памяти, а доступные из других процессов — в другом.

Пространства стека и кучи

Наконец, сам стек тоже можно считать разновидностью пространства памяти, а выделяемые в стеке пулы — их частным случаем. Этот подход применялся ранее в этой главе для пулов, локальных по отношению к области действия конкретной функции. Эта особая интерпретация стека упоминается и в двух последующих главах.

Уплотнение памяти

15

Представьте себе обычное управление памятью в C++: вы зажигаете благовония, приносите жертву божествам операционной системы и удаляете объект. Если все идет нормально, объект будет должным образом деинициализирован и уничтожен и никто никогда не попытается им воспользоваться. Ха! Всем известно, что в реальной жизни так не бывает.

Одна из главных достопримечательностей динамических языков — таких как SmallTalk и Lisp — не имеет никакого отношения к самому языку, а лишь к тому, что он удаляет объекты за вас, автоматически и надежно. Выходит потрясающая экономия времени и энергии, не говоря уже о благовониях и жертвах. Можно ли то же самое сделать на C++? Вместо прямолинейного «да» или «нет» я отвечу: «Все зависит от того, сколько труда вы хотите вложить в решение».

В этой главе начинается настоящее веселье — вы увидите, как перемещать объекты в памяти, чтобы свести к минимуму фрагментацию свободной памяти. Надеюсь, при виде моих алгоритмов никто не будет кататься по полу от смеха. Заодно мы подготовим сцену для полноценных алгоритмов сборки мусора, о которых речь пойдет в следующей главе.

Поиск указателей

Помимо подсчета ссылок и нестандартных операторов `new` и `delete` в большинстве стратегий управления памятью сочетаются две методики: определение момента, когда доступ к объекту становится невозможным, для его автоматического уничтожения (сборка мусора) и перемещение объектов в памяти (уплотнение). В свою очередь, эти стратегии зависят от того, что в других языках делается легко и просто, но оказывается дьявольски сложным в C++ — от задачи поиска всех указателей на объекты.

Поиск указателей в программе на C++ чрезвычайно сложен, поскольку компилятор не оставляет никаких инструкций на этот счет. Более того, в C++ программа может получить адрес части объекта, поэтому некоторые указатели могут ссылаться на середину большого объекта.

Мама, откуда берутся указатели?

В C++ существуют невероятно разнообразные способы получения указателей. Одни связаны с конкретным представлением объектов в памяти, другие — с наследованием, третьи — с переменными классов. Конечно, самый очевидный способ — это нахождение адреса. А теперь давайте рассмотрим другие, не столь тривиальные способы.

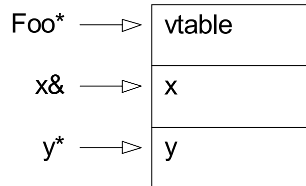
Адреса переменных класса

Имея объект, вы можете получить адрес переменной класса, воспользоваться им или передать другому объекту.

```
class Foo {
private:
     int x;
     String y;
```

```
public:
    int& X() { return x; }  // Ссылка на x
    String* Name() { return &y; }   // Адрес y
};
```

Каждый экземпляр Foo выглядит примерно так, как показано на представленной ниже диаграмме (вообще говоря, все зависит от компилятора, но в большинстве компиляторов дело обстоит именно так):



Как правило, несколько первых байт занимает указатель на v-таблицу для класса данного объекта. За ним следуют переменные класса в порядке их объявления. Если вы получаете адрес переменной класса в виде ссылки или указателя, возникает указатель на середину объекта.

Адреса базовых классов

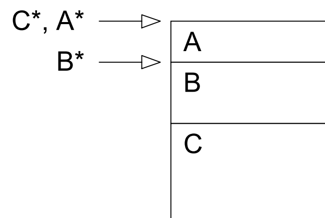
Наследование также может вызвать массу положительных эмоций.

```
class A {...};     // Один базовый класс
class B {...};     // Другой базовый класс
class C : public A, public B {...};       // Множественное наследование
```

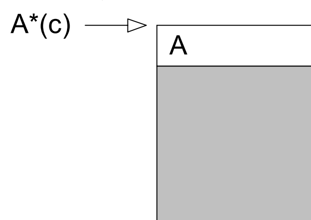
При одиночном наследовании преобразование от derived* к base* (где base — базовый, а derived — производный класс) адрес остается прежним, даже если компилятор полагает, что тип изменился. При множественном наследовании дело обстоит несколько сложнее.

```
C* c = new C;
A* a = c;     // Преобразование от производного к первому базовому классу
B* b = c;     // Преобразование от производного ко второму базовому классу
cout << c << endl;
cout << a << endl;
cout << b << endl;
```

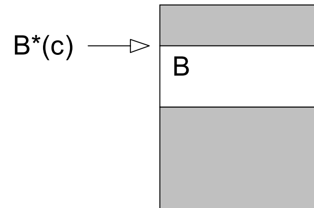
Вроде бы все просто, но в действительности компилятор проделывает довольно-таки хитрый фокус. При преобразовании C* к A* указатель остается прежним. Однако при преобразовании C* к B* компилятор действительно *изменяет адрес*. Это связано с тем, как объект хранится в памяти (структура объектов зависит от компилятора, но сказанное относится ко всем компиляторам, с которыми я работал).



Компилятор строит объект в порядке появления базовых классов, за которыми следует производный класс. Когда компилятор преобразует C* к A*, он словно набрасывает черное покрывало на составляющие B и C и убеждает клиентский код, что тот имеет дело с самым настоящим A.



Размещение v-таблицы в начале объекта приводит к тому, что принадлежащие C реализации виртуальных функций, объявленных в A, останутся доступными, но будут иметь те же смещения, что и для A. Работая с C*, компилятор знает полную структуру всего объекта и может обращаться к членам A, B и C на их законных местах. Но когда компилятор выполняет преобразование ко второму или одному из следующих классов в списке множественного наследования, адрес изменяется — клиентский код будет считать, что он имеет дело с B.



На самом деле v-таблиц две. Одна находится в начале объекта и содержит все виртуальные функции, первоначально объявленные в A или C, а другая — в начале компонента B и содержит виртуальные функции, объявленные в B. Это означает, что преобразование типа от производного к базовому классу в C++ может при некоторых обстоятельствах породить указатель на середину объекта (по аналогии с указателями на переменные класса, о которых говорилось выше). Кроме того, в C++ открывается возможность дурацких фокусов:

```
C* anotherC = C*(void*(B*(c)));
anotherC->MemberOfC();
```

Видите, в чем проблема? Преобразование B*(c) смещает указатель. Затем он преобразуется к типу void*. Далее следует обратное преобразование к C* — и наша программа будет уверена, что C начинается с неверного адреса. Без преобразования к void* все работает, поскольку компилятор может определить смещение B* в C*. В сущности, преобразование от base* к derived* (где base — базовый, а derived — производный класс) выполняется каждый раз, когда клиент вызывает виртуальную функцию B, переопределенную в C. Но когда происходит преобразование от void* к C*, компилятор лишь наивно полагает, что программист действует сознательно.

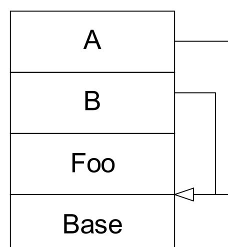
Запомните: каждый программист на C++ за свою карьеру проводит как минимум одну бессонную ночь, пытаясь понять, почему его объект бредит. Потом приходит какой-нибудь гуру, с ходу ставит диагноз «синдром класс-void-класс» — притом так, чтобы слышали окружающие — и разражается злорадным смехом. Впрочем, я отклонился от темы.

Виртуальные базовые классы

Если вы пользуетесь виртуальными базовыми классами, попрощайтесь со всеми схемами уплотнения и сборки мусора, требующими перемещения объектов в памяти. Ниже приведен фрагмент программы и показано, как объект представлен в памяти.

```
class Base {...};
class A : virtual public Base {...};
class B : virtual public Base {...};
class Foo : public A, public B {...};
```

Тьфу. Компилятору так стыдно, что Base приходится реализовывать как виртуальный базовый класс, что он прячет его как можно дальше, под Foo. A и B содержат указатели на экземпляр Base… да, все верно, указатели, то есть непосредственные адреса в памяти. Вы не имеете доступа к этим указателям и, следовательно, не сможете обновить их при перемещении объекта в памяти.



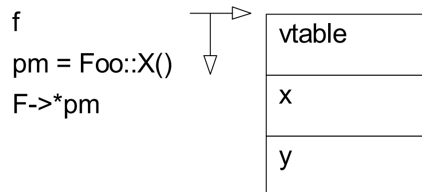
Указатель на переменную класса

Идея указателя на переменную класса заключается в том, что переменную можно однозначно идентифицировать не по ее непосредственному адресу, но по адресу содержащего ее объекта и смещению переменной внутри объекта. Если вы никогда не пользовались указателями на переменные класса, изучите следующий фрагмент как можно внимательнее.

```
class Foo {
private:
    int x;
public:
    static int& Foo::*X() { return &Foo::x; }
};
Foo f = new Foo;              // Создать экземпляр
int& Foo::*pm = Foo::X();    // Вычислить смещение int
int& i = f->*pm;              // Применить смещение к экземпляру
```

Функция `X()` возвращает не ссылку на `int`, а смещение некоторого `int` в экземплярах класса `Foo`. Функция `Foo::X()` объявлена статической, поскольку относится не к конкретному экземпляру, а к классу в целом. Команда `return &Foo::x;` определяет смещение конкретной переменной, `x`. В строке `int& Foo::*pm = Foo::X();` объявляется переменная `pm`, которая содержит смещение переменной `int` класса `Foo`. Она инициализируется смещением, полученным от `Foo::X()`. Наконец, в строке `int& i = f->*pm;` смещение применяется к конкретному экземпляру для вычисления адреса конкретного int. Обратите внимание: значение pm само по себе бесполезно до тех пор, пока вы не примение его к объекту.

Все эти `int&` с таким же успехом можно заменить на `int*`. В любом случае все завершается косвенным получением адреса некоторой части объекта так, словно вы получили явный адрес переменной класса. Указатели на члены классов также могут применяться для косвенных ссылок на функции, а не на переменные класса, но это не относится к нашей теме — управление памятью. К тому же я не хочу взваливать на себя лишнюю головную боль.



Последствия

Все сказанное обладает фундаментальными последствиями для управления памятью. Чтобы переместить объект в памяти, вам придется проследить за тем, чтобы перемещался вмещающий объект верхнего уровня, а не некоторый вложенный объект, адрес которого у вас имеется. Более того, при перемещении объекта придется обновлять *все* указатели — не только на сам объект, но и на все вложенные объекты и базовые классы.

Если вы хотите узнать, существуют ли ссылки на некоторый объект, придется искать указатели не только на начало объекта, но и на все его переменные и базовые классы.

Поиск указателей

Итак, теперь мы знаем, с какими разными указателями нам придется иметь дело. Как же отыскать их все? Чтобы переместить объект, нам придется обновить все указатели на него. Чтобы понять, доступен ли объект, придется собрать все указатели.

Специализированные пространства памяти для указателей

Одно из «силовых» решений — сложить все указатели в одно место, где их будет легко найти. В свою очередь, это подразумевает, что все указатели должны быть умными и храниться в специальных пространствах памяти. Эти пространства должны быть организованы так, чтобы вы могли перебрать их

содержимое (то есть создать итерацию для набора умных указателей). В классах все эти *-указатели заменяются дескрипторами или ссылками на умные указатели, поскольку сами указатели должны находиться в отдельном пространстве. В следующем фрагменте программы P и H представляют собой стандартные указатели и дескрипторы соответственно, за исключением того, что P сохраняет экземпляры указателей в специальном пространстве памяти. Эта методика хорошо подойдет и для невидимых указателей, если для их сохранения в надежном, хорошо известном месте будет использована одна из методик нестандартных пространств памяти. Указатель P обычно является ведущим, но это не обязательно.

```
template <class Type>
class P {     // Указатель
private:
    Type* pointee;
public:
    void* operator new(size_t);      // Использует специальное
                                     // пространство памяти
    void operator delete(void*);     // Использует специальное
                                     // пространство памяти
    // Все для умных указателей
};
template <class Type>
class H {     // Дескриптор
private:
    P<Type>* ptr;
public:
    // Все для дескрипторов
};
class Foo {
private:
    P<Bar>& bar;  // Ссылка на умный указатель на Bar
    // ИЛИ
    H<Bar>& bar;  // Дескриптор Bar
};
```

В первом варианте мы храним ссылку на умный указатель, причем сам указатель, вероятно, хранится где-то в другом месте. Во втором варианте мы используем идиому дескриптора — умного указателя на умный указатель. Сам дескриптор находится в объекте, но указатель, на который он ссылается, — в специальном пространстве указателей, используемом операторами new и delete класса P. Если пространство указателей будет реализовано толково, все указатели можно будет перебирать прямо из него. В это случае задача перемещения объекта несколько упрощается (хотя и не становится простой), поскольку все указатели на него можно найти в пространстве указателей. Полная реализация этой методики приведена ниже в этой главе.

Скрытые коллекции указателей

Другое возможное решение — поддержать скрытые коллекции умных указателей.

```
template <class Type>
class P {
private:
    static P<Type>* head;   // Начало списка MP
    static P<Type>* tail;   // Конец списка
    P<Type>* next;          // Следующий элемент списка
    P<Type>* previous;      // Предыдущий элемент списка
    Type* pointee;
```

```
public:
    P();     // Занести 'this' в список
    P(const P<Type>& p);   // Занести 'this' в список
    ~P();    // Удалить 'this' из списка
    P<Type>& operator=(const P<Type>& p);      // Не изменяя список,
                                               // скопировать p.pointee
    // Все для умных указателей
};
```

Вам придется соблюдать осторожность при выполнении операций со списком в конструкторе копий и операторе =, но во всем остальном реализация достаточно тривиальная. Используя этот шаблон, класс обходится без хранения ссылок на умные указатели; он хранит их непосредственно.

```
class Foo {
private:
    P<Bar> bar;   // Указатель автоматически заносится в скрытую коллекцию
};
```

При конструировании `Foo` вызывается соответствующий конструктор `P`, который автоматически заносит `bar` в скрытую коллекцию. При уничтожении `Foo` вызывает деструктор `P`, который удаляет `bar` из коллекции. Разумеется, вместо двусвязного списка можно воспользоваться другими структурами данных. Кроме того, как вы вскоре убедитесь, для всех этих специализированных указателей стоит создать общий базовый класс и сохранить их все в одной коллекции. В приведенном выше фрагменте для каждого типа указателя создается отдельная коллекция.

Анализ экземпляров

Более радикальный подход — использовать самые обычные указатели и предусмотреть средства для перебора всех указателей, внедренных в экземпляр.

```
class Foo {
private:
    Bar* bar;
};
```

В данной схеме это разрешается, при условии, что какая-то очень сложная архитектура сможет определить `Bar*` по имеющемуся `Foo`. Чтобы реализовать ее, нам понадобится код, который знает структуру каждого экземпляра (а точнее, умеет находить переменные класса, которые являются адресами или могут содержать адреса посредством рекурсии), и возможность точно определить тип указываемого объекта. Например, успешно найденный `bar` не поможет, если вы не знаете, с чем имеете дело — с настоящим Bar или каким-то классом, производным от `Bar`. В производном классе могут появиться дополнительные указатели, отсутствующие в `Bar`.

Мы еще вернемся к этому решению, но если бы это была реклама автомобиля, то в нижней части экрана вы бы увидели предупреждение: «Профессиональный каскадер на закрытом треке. Не пытайтесь повторить в домашних условиях». Тем не менее, во многих ситуациях простые решения не работают, поэтому в следующей главе мы с головой нырнем в эту схему.

Стековые переменные

Конечно, не все указатели являются членами объектов. Некоторые из них — обычные переменные, находящиеся в стеке. Разумеется, решение со специальными пространствами для стековых переменных не подойдет, если только они не являются дескрипторами или ссылками на умные указатели, хранящиеся в другом месте. Скрытые коллекции с тем же успехом будут работать для указателей, хранящихся в стеке или куче — при условии, что вы организуете обработку исключений, которая будет правильно раскручивать стек. Особого обращения требует лишь одна переменная `this`, значение которой задается компилятором, а не вашим кодом, работающим с умными указателями. Стековые переменные значительно усложняют решение с анализом экземпляров, поскольку вам также придется разрабатывать отдельную схему обнаружения или хотя бы предотвращения коллизий.

Дескрипторы, повсюду дескрипторы

Одна из стратегий уплотнения и сборки мусора в C++, которая заимствует кое-что из динамических языков — ссылаться на все объекты только через дескрипторы.

```
class Foo {
private:
    H<Bar> bar;   // Дескриптор Bar
public:
    H<Bar> GetBar() { return bar; }
};
```

Здесь `H` — шаблон дескриптора (вроде тех, которые мы рассматривали в предыдущих главах). Каждый `H<Bar>` представляет собой умный указатель на ведущий указатель на `Bar`. Функции, косвенно открывающие переменные класса (такие как `GetBar()`), возвращают копию дескриптора. Все ведущие указатели (по крайней мере, в этой версии) живут в специальном пространстве памяти, поэтому найти их несложно.

Ниже описывается одна из несложных реализаций уплотнения с применением дескрипторов. Конечно, возможны и другие варианты.

Общее описание архитектуры

В общих чертах наша архитектура строится на следующих принципах:

- Поскольку различные типы объединяются в один набор ведущих указателей, мы воспользуемся абстрактным базовым классом `VoidPtr` для ведущих указателей. Конкретные ведущие указатели будут создаваться по шаблону, производимому от этого базового класса.
- Ведущие указатели находятся в специальном пространстве, обеспечивающем простой перебор указателей.
- Каждый ведущий указатель обеспечивает подсчет ссылок и удаляет себя, когда счетчик переходит от 1 к 0. В свою очередь, его деструктор вызывает деструктор указываемого объекта и, в зависимости от используемых алгоритмов, пытается (или не пытается) вернуть занимаемую объектом память.
- Во всех переменных классов и обычных переменных используются дескрипторы ведущих указателей вместо прямых указателей на другие объекты.
- Память возвращается лишь в процессе уплотнения управляемой части кучи. Иначе говоря, если нам понадобится больше памяти, мы начинаем спускать активные объекты вних по куче, чтобы освободить место наверху. Выделение памяти всегда происходит снизу вверх.

Описана лишь одна из возможных архитектур уплотнения. Мы не пытаемся ни решить проблемы зацикливания, ни удалить объекты, ставшие недоступными, но еще не удаленные. Об этом речь пойдет в следующей главе.

Ведущие указатели

Как и во многих других стратегиях управления памятью, нам придется хранить множество различных ведущих указателей в одной структуре с возможностью перебора. Напрашивается общий абстрактный базовый класс для всех ведущих указателей. К нашим ведущим указателям предъявляются следующие требования:

1. Ведение счетчика ссылок.
2. Хранение их в специальном пространстве памяти с поддержкой перебора.
3. Вызов деструктора указываемого объекта в деструкторе указателя. В зависимости от используемого алгоритма сборки мусора мы одновременно пытаемся (или не пытаемся) вернуть занимаемую объектом память. В нашем примере не стоит беспокоиться о возврате памяти объекта.

Базовый класс VoidPtr

Ниже показан абстрактный базовый класс, удовлетворяющий этим требованиям.

```
class VoidPtrPool;      // Используется для создания, уничтожения
                        // и перебора VoidPtr
class VoidPtr {
friend class VoidPtrPool;
private:
    unsigned long refcount;   // Счетчик ссылок
protected:
    void* address;      // Адрес указываемого объекта
    size_t size;        // Размер указываемого объекта в байтах
    VoidPtr() : address(NULL), size(0), refcount(0) {}
    VoidPtr(void* adr, size_t s) : address(adr), size(s), refcount(0) {}
public:
    static VoidPtrPool* pool;
    virtual ~VoidPtr() { size = 0; address = NULL; }
    void* operator new(size_t)
    {
        if (pool == NULL)
            pool = new VoidPtrPool;
        return pool->Allocate();
    }
    void operator delete(void* space)
        { pool->Deallocate((VoidPtr*)space); }
    void Grab() { refcount++; }
    void Release()
    {
        if (refcount > 0) refcount--;
        if (refcount <= 0) delete this;
    }
};
```

Шаблон ведущего указателя

Наш ведущий указатель представляет собой шаблон, производный от `VoidPtr`. Он существует в основном для того, чтобы реализовать оператор -> и виртуальный деструктор, который знает, какой деструктор должен вызываться для указываемого объекта. Я решил запретить копирование и присваивание. При копировании дескриптора должен копироваться адрес ведущего указателя, а не сам ведущий указатель или указываемый объект. Следовательно, нет особой необходимости поддерживать копирование и присваивание для ведущих указателей. Как обычно, существует множество вариаций на тему конструкторов. В данном случае я выбрал ту, в которой конструктор ведущего указателя создает указываемый объект.

```
template <class Type>
class MP : public VoidPtr {
private:      // Чтобы запретить копирование и присваивание
    MP(const MP<Type>&) {}
    MP<Type>& operator=(const MP<Type>&) { return this; }
public:
    MP() : VoidPtr(new Type, sizeof(Type)) {}
    virtual ~MP() { ((Type*)address)->Type::~TypeOf(); }
```

```
        Type* operator->() { return (Type*)address; }
    };
```

Шаблон дескриптора

Это уже знакомый нам шаблон дескриптора с подсчетом ссылок из предыдущей главы.

```
    template <class Type>
    class Handle {
    private:
        MP<Type>* pointer;
    public:
        Handle() : pointer(new MP<Type>) { pointer->Grab(); }
        Handle(const Handle<Type>& h) : pointer(h.pointer)
            { pointer->Grab(); }
        Handle<Type>& operator=(const Handle<Type>& h)
        {
            if (this == &h) return *this;
            if (pointer == h.pointer) return *this;
            pointer->Release();
            h.pointer->Grab();
            return *this;
        }
        MP<Type>& operator->() { return *pointer; }
    };
```

В программе он используется для переменных, ссылающихся на объекты.

```
    class Bar {
    private:
        H<Foo> foo;
    public:
        void f();
    };
    void Bar::f()
    {
        Handle<Foo> f;      // Эквивалентно Foo* f = new Foo;
        f = foo;            // Использует operator=(Handle<Type>(foo));
        foo = f;            // Использует оператор H<Type>(f)
    }
```

Пул ведущих указателей

Простоты ради мы предположим, что классы, производные от `VoidPtr`, совпадают по размеру с самим `VoidPtr`; иначе говоря, в них не добавляются новые переменные. Наша задача упрощается; `VoidPtrPool` теперь может быть простым связанным списком массивов `VoidPtr`. Структура массива называется `VoidPtrBlock`.

```
    struct VoidPtrBlock {
        VoidPtrBlock* next;     // Следующий блок в списке
        VoidPtr slots[BLOCKSIZE];       // Массив позиций
        VoidPtrBlock(VoidPtrBlock* next_in_list) : next(next_in_list)
        {
            // Организовать новые позиции в связанный список
            for (int i = 0; i < BLOCKSIZE - 1; i++)
```

```
                slots[i].address = &slots[i + 1];
            slots[BLOCKSIZE - 1].address = NULL;
        }
        ~VoidPtrBlock() { delete next; }
    }
    class VoidPtrPool {
    private:
        VoidPtr* free_list;     // Список свободных VoidPtr
        VoidPtrBlock* block_size;       // Начало списка блоков
    public:
        VoidPtrPool() : block_list(NULL), free_list(NULL) {}
        ~VoidPtrPool() { delete block_list; }
        VoidPtr* Allocate();
        void Deallocate(VoidPtr* vp);
    };
    VoidPtr* VoidPtrPool::Allocate()
    {
        if (free_list == NULL)    // Выделить дополнительный блок
        {
            block_list = new VoidPtrBlock(block_list);
            // Добавить в список
            block_list->slots[BLOCKSIZE - 1].address = free_list;
            free_list = &block_list->slots[0];
        }
        VoidPtr* space = (VoidPtr*)free_list;
        free_list = (VoidPtr*)space->address;
        return space;
    }
    void VoidPtrPool::Deallocate(VoidPtr* p)
    {
        vp->address = free_list;
        free_list = (VoidPtr*)vp->address;
        vp->size = 0;
    }
```

В общем, ничего хитрого. При выделении нового блока мы организуем его позиции связанный список и водружаем поверх списка свободных указателей. Если список свободных указателей пуст, а нам потребовался еще один ведущий указатель, мы выделяем новый блок, а затем берем указатель из списка свободных, в котором к этому времени появились вакансии.

Итератор ведущих указателей

Для перебора всех ведущих указателей мы создадим класс итератора с именем `VoidPtrIterator`. `VoidPtrPool` возвращает итератор, перебирающий все активные указатели (то есть все указатели, не присутствующие в списке свободных). Он будет объявлен как чисто абстрактный базовый класс, поскольку в следующей главе тот же интерфейс будет использован для перебора указателей, внедренных в объекты.

```
    class VoidPtrIterator {
    protected:
        VoidPtrIterator() {}
    public:
        virtual bool More() = 0;
```

```
        virtual VoidPtr* Next() = 0;
    };
```

Сам итератор работает весьма прямолинейно. Он просто перебирает блоки в цикле и ищет указатели с ненулевым значением переменной `size`.

```
    class VoidPtrPoolIterator : public VoidPtrIterator {
    private:
        VoidPtrBlock* block;
        int slot;     // Номер позиции в текущем блоке
        virtual void Advance()  // Найти следующую используемую позицию
        {
            while (block != NULL)
            {
                if (slot >= BLOCKSIZE)
                {
                    block = block->next;
                    slot = 0;
                }
                else if (block->slots[slot].size != 0)
                    break;
                slot++;
            }
        }
    public:
        VoidPtrPoolIterator(VoidPtrBlock* vpb)
            : block(vpb), slot(0), { Advance(); }
        virtual bool More() { return block != NULL; }
        virtual VoidPtr* Next()
        {
            VoidPtr* vp = &block->slots[slot];
            Advance();
            return vp;
        }
    };
```

Кроме того, мы добавим в `VoidPtrPool` следующую функцию:

```
    VoidPtrIterator* iterator()
        { return new VoidPtrPoolIterator(this); }
```

Наконец, нам пришлось объявить `VoidPtrPoolIterator` другом `VoidPtr`, чтобы в программе можно было обратиться к его переменной `size`. Забегая вперед, скажу, что в главе 16 мы воспользуемся этим итератором для других целей; поэтому функция `Advance()` и объявлена виртуальной, чтобы производные классы могли добавить свою собственную фильтрацию. Если найденная позиция имеет нулевое значение `size`, мы пропускаем ее. Во всем остальном работа сводится к простому перебору в массивах, образующих блоки указателей.

Вариации

Перед тем как описывать сами алгоритмы уплотнения, давайте рассмотрим другие варианты исходной постановки задачи. Они не оказывают принципиального влияния на архитектуру или алгоритмы, а только на идиомы C++ в их реализации.

Невидимые ведущие указатели

Чтобы не использовать шаблоны дескрипторов и ведущих указателей, можно было организовать множественное наследование ведущих указателей от VoidPtr и гомоморфного базового класса. Иначе говоря, ведущие указатели становятся невидимыми, как объяснялось в предыдущих главах. В игру вступают все механизмы, сопутствующие идиоме невидимых указателей (например, производящие функции).

```
class Foo {
public:
     static Foo* make();     // Возвращает пару указатель-указываемый объект
     virtual void Member1() = 0;
     // И т.д. для открытого интерфейса
};
// В файле foo.cpp
class FooPtr : public Foo, public VoidPtr {
public:
     FooPtr(Foo* foo, size_t size) : VoidPtr(foo, size) {}
     virtual ~FooPtr() { delete (Foo*)address; }
     virtual void Member1()
          { ((Foo*)address)->Member1(); }
     // И т.д. для остальных открытых функций
};
class RealFoo : public Foo { ... };
Foo* Foo::make()
{
     return new FooPtr(new RealFoo, sizeof(RealFoo));
}
// В клиентском коде
class Bar {
private:
     Foo* foo;     // На самом деле невидимый указатель
public:
     Bar() : foo(Foo::make()) {}
};
```

Такое решение улучшает инкапсуляцию применения ведущих указателей. Кроме того, оно позволяет производящим функциям решить, какие экземпляры должны управляться подобным образом, а какие — с помощью обычных невидимых указателей или даже вообще без указателей. Все прекрасно работает, пока вы соблюдаете осторожность и выделяете в VoidPtrPool достаточно места для FooPtr. Помните, что из-за множественного наследования по сравнению с VoidPtr размер увеличивается, по крайней мере, на v-таблицу.

Объекты классов

Возможен и другой вариант — потребовать, чтобы все объекты происходили от общего класса-предка, способного возвратить объект класса для данного объекта или, по крайней мере, размер объекта. В этом случае вам уже не придется хранить в указателе объект экземпляра, поскольку его можно будет получить у объекта класса. Если вы готовы смириться с некоторым насилием в отношении типов в дескрипторах, это также позволит вам избежать шаблонов второго уровня, используемых для главных указателей. Вместо void* в VoidPtr можно будет хранить CommonBase* (где CommonBase — общий базовый класс). Мы избавляемся от переменной size, от необходимости иметь шаблон, производный от VoidPtr, и от виртуального деструктора в VoidPtr, и как следствие — от 4-байтового адреса v-таблицы. С другой стороны, если управляемые объекты уже содержат v-таблицы и не принуждают вас к применению множественного наследования, дополнительных затрат не будет.

Оптимизация в особых ситуациях

Если адрес переменной класса получать не требуется, ее можно хранить в виде внедренного объекта. Впрочем, как показывает следующий фрагмент, ситуация не всегда находится под контролем разработчика класса:

```
void f(int);
class Foo {
private:
    int x;    // Адрес получать не нужно, поэтому храним непосредственно
public:
    void F() { f(x); }
};
```

Вышядит вполне безопасно, не правда ли? А теперь предположим, что автор функции `f()` привел ее интерфейс к следующему виду:

```
void f(int&);
```

И вот вся тщательно спроектированная оптимизация обрушивается вам на голову. У внедренных объектов есть еще одна проблема: вы должны проследить не только за тем, чтобы никогда не получать адрес объекта, но и за тем, чтобы никогда не получать адресов рекурсивно внедренных членов.

```
class Bar {
private:
    Foo foo;
};
```

Допустим, вы сможете доказать, что ни одна из функций `Bar` не получает адрес `foo`. Но вам придется сделать следующий шаг и проследить еще и за тем, чтобы все функции `Foo` тоже были безопасными. Та же логика относится и к базовым классам. Важно понимать, что такая оптимизация должна осуществляться на уровне всей программы, а не только проектируемого класса.

Алгоритм Бейкера

Один из алгоритмов уплотнения жертвует неимоверным количеством (а точнее, половиной) памяти в интересах скорости. Процесс уплотнения понемногу вплетается в обычную работу программы. Этот алгоритм называется *алгоритмом Бейкера (Baker's Algorithm)*.

Пул памяти делится на две половины, A и B. В любой момент времени одна из этих половин является *активной* (то есть в ней создаются новые объекты). Память выделяется снизу вверх, а в момент удаления объекта не делается никаких попыток вернуть занимаемую им память. Время от времени все активные объекты копируются из одной половины памяти в другую. В процессе копирования автоматически происходит уплотнение нижней части новой активной половины. Активным называется объект, для которого в стане ведущих указателей найдется ссылающийся на него ведущий указатель (в нашем случае `VoidPtr`).

Пространства объектов

Половины представлены в виде пространств памяти для создания объектов. Класс `HalfSpace` изображает одну половину, а `Space` — всю память, видимую клиентам.

Класс HalfSpace

Каждая половина по отдельности выглядит как обычное пространство памяти со специализированной функцией `Allocate()`. Парная функция `Deallocate()` не понадобится.

```
class HalfSpace {
private:
    unsigned long next_byte;  // Следующий выделяемый байт
    unsigned char bytes[HALFSIZE];
public:
```

```
    HalfSpace() : next_byte(0) {}
    void* Allocate(size_t size);
    void Reinitialize() { next_byte = 0; }
};
void* HalfSpace::Allocate(size_t size)
{
    // Выровнять до границы слова
    size = ROUNDUP(size);
    if (next_byte + size >= HALFSIZE)
        return NULL;  // Не хватает памяти
    void* space = &bytes[next_byte];
    next_byte += size;
    return space;
}
```

Класс Space

Общий пул представляет собой совокупность двух половин. Он также имеет функцию `Allocate()`, которая в обычных ситуациях просто поручает работу активной половине. Если в активной половине не найдется достаточно памяти, происходит переключение половин и копирование активных объектов в другую половину функцией `Swap()`. Эта схема основана на предыдущем материале — специализированном пуле `VoidPtr` со средствами перебора.

```
class Space {
private:
    HalfSpace A, B;
    HalfSpace* active;
    HalfSpace* inactive;
    void Swap();  // Переключить активную половину, скопировать объекты
public:
    Space() : active(&A), inactive(&B) {};
    void* Allocate(size_t size);
};
void* Space::Allocate(size_t size)
{
    void* space = active->Allocate(size);
    if (space != NULL) return space;
    Swap();
    Space = active->Allocate(size);
    if (space == NULL)
        // Исключение - нехватка памяти
    return space;
}
void Space::Swap()
{
    if (active == &A)
    {
        active = &B;
        inactive = &A;
    }
    else
```

```
    {
        active = &A;
        inactive = &B;
    }
    active->Reinitialize();
    // Перебрать все VoidPtr и найте активные объекты
    VoidPtrIterator* iterator = VoidPtr::pool->iterator();
    while (iterator->More())
    {
        VoidPtr* vp = iterator->Next();
        if (vp->address >= inactive &&
            vp->address < inactive + sizeof(*inactive))
        {
            void* new_space = active->Allocate(vp->size);
            if (new_space == NULL)
                // Исключение - нехватка памяти
            memcpy(new_space, vp->address, vp->size);
            vp->address = new_space;
        }
    }
    delete iterator;
}
```

Все существенное происходит в цикле `while` функции `Space::Swap()`. Каждый объект в предыдущей, ранее активной половине копируется в новую активную половину. Вскоре вы поймете, зачем мы проверяем, принадлежит ли адрес старой половине.

Оператор new

Конечно, у нас появляется перегруженный оператор `new`, который использует эту структуру.

```
void* operator new(size_t size, Space* space)
{
    return space->Allocate(size);
}
```

Ведущие указатели

Наконец, ведущие указатели должны использовать это пространство при создании объектов.

```
template <class Type>
class BMP : public VoidPtr {
private: // Запретить копирование и присваивание указателей
    BMP(const MP<Type>&) {}
    BMP<Type>& operator=(const BMP<Type>&) { return *this; }
public:
    BMP() : VoidPtr(new(object_space) Type, sizeof(Type)) {}
    virtual ~BMP() { ((Type*)address->Type::~Type(); }
    Type* operator->() { return (Type*)address; }
};
```

Здесь `object_space` — глобальная переменная (а может быть, статическая переменная класса `VoidPtr`), которая ссылается на рабочее пространство `Space`.

Последовательное копирование

Функция Swap() вызывается в произвольные моменты и, скорее всего, будет работать в течение некоторого времени. В работе программы возникают непредсказуемые затяжки, а это бесит пользователей едва ли не больше, чем аппаратные сбои. К счастью, алгоритм Бейкера легко модифицировать, чтобы копирование выполнялось поэтапно в фоновом режиме.

На мосту Бей-Бридж в Сан-Франциско постоянно работает бригада маляров. Она начинает красить мост с одного конца и через пару лет успешно докрашивает до другого. К этому времени можно начинать красить заново, в другую сторону. Работа идет постоянно, не считая редких перерывов из-за землетрясений или демонстраций протеста. В сущности, именно так алгоритм Бейкера превращается в схему последовательного уплотнения.

Начинаем следующий заход на класс Space. Функция Swap() делится на две части, одна из которых переключает активные половины, а другая многократно вызывается и при каждом вызове копирует по одному объекту.

```
class Space {
private:
    VoidPtrIterator* iterator;      // Информация о копировании
    HalfSpace A, B;
    HalfSpace* active;
    HalfSpace* inactive;
    void Swap();  // Переключить активную половину, скопировать объекты
public:
    Space() : active(&A), inactive(&B), iterator(NULL) { Swap(); }
    void* Allocate(size_t size);
    void Copy1();
};
void* Space::Allocate(size_t size)
{
    void* space = active->Allocate(size);
    if (space != NULL)
        // Исключение - нехватка памяти
    return space;
}
void Space::Swap()
{
    if (active == &A)
    {
        active = &B;
        inactive = &A;
    }
    else
    {
        active = &A;
        inactive = &B;
    }
    active->Reinitialize();
    delete iterator;
    iterator = VoidPtr::pool->iterator();
}
void Space::Copy1()
```

```
{
     if (!iterator->More())
          Swap();  // Начать работу в другую сторону
     else
     {
          VoidPtr* vp = iterator->Next();
          if (vp->address >= inactive &&
              vp->address < inactive + sizeof(*inactive))
          {
               void* new_space = active->Allocate(size);
               if (new_space == NULL)
                    throw(OutOfMemory());
               memcpy(new_space, vp->address, vp->size);
               vp->address = new_space;
          }
     }
}
```

Функцию `Copy1()` необходимо вызывать как можно чаще, однако делать это можно в ходе нормальной работы программы. Новые объекты размещаются в активной половине, смешиваются со скопированными объектами, но это не приносит вреда. Поскольку перед копированием мы убеждаемся, что объект в данный момент находится в неактивном пространстве, созданные в активной половине объекты остаются без изменений.

Внешние объекты

Предположим, адрес объекта пришлось передать системной функции, которая ничего не знает ни о дескрипторах, ни о ведущих указателях. Такому объекту лучше оставаться на своем месте, пока системная функция не завершит свою работу!

```
SystemCall(&aString);   // aString не следует перемещать до тех пор,
                        // пока его адрес остается в распоряжении системы
```

Прежде всего, совершенно неочевидно, как получить адрес объекта, поскольку рассматривавшиеся до настоящего момента ведущие указатели и дескрипторы не предоставляли прямого доступа к адресам объектов. Но даже если предположить, что такая способность была добавлена, приходится действовать осторожно. Первое побуждение — включить в ведущий указатель флаг, показывающий, что объект не должен перемещаться. Но тем самым вы швырнете гнилой помидор в алгоритм уплотнения; вам придется тщательно обходить этот объект, чтобы не скопировать что-нибудь поверх него. Более удачный выход — убрать объект из сжимаемого пространства на все время, пока он должен оставаться на фиксированном месте.

```
class Space {
public:
     void Externalize(VoidPtr* vp)
     {
          void* space = ::operator new(vp->size);
          memcpy(space, vp->address, vp->size);
          vp->address = space;
     }
     void Internalize(VoidPtr* vp)
     {
          void* space = Allocate(vp->size);
          memcpy(space, vp->address, vp->size);
          ::operator delete(vp->address);
          vp->address = space;
```

```
        }
    }
```

Функция `Externalize()` перемещает объект за пределы сжимаемого пространства; `Internalize()` возвращает его обратно. Алгоритм `Copy1()` будет нормально работать, поскольку не пытается перемещать объекты вне неактивной половины.

Этот способ также может применяться для передачи адреса переменной класса или `this` (см. ниже) некоторой функции класса или глобальной функции. Допустим, вам потребовалось организовать взаимодействие своих классов с коммерческой библиотекой, которая понятия не имеет о ваших хитроумных правилах уплотнения.

Помимо необходимости узнавать, когда внешний код перестал пользоваться вашим объектом, этот вариант может вызвать проблемы и при частой передаче адресов внешним функциям, поскольку копирование целого объекта из пространства памяти и обратно может обходиться довольно дорого.

Алгоритм Бейкера: уход и кормление в C++

Практическое использование описанных выше алгоритмов требует нескольких жестких ограничений. Алгоритм Бейкера для объектов C++ напоминает котенка, которого ваш ребенок приносит в дом и клянется «всегда-всегда» кормить и заботиться. Другими словами, все совершенно искренне клянутся соблюдать правила, а вам лучше надевать передник и идти за тряпкой.

Очереди операций и указатель this

Если на момент вызова `Copy1()` существует указатель `this`, то объект, на который он ссылается, может переместиться из одной половины в другую. При этом `this` будет радостно ссылаться на старую копию. Мы позаботились обо всех остальных стековых переменных и превратили их в дескрипторы. Теперь, чтобы получить доступ к объекту, им приходится разворачиваться на 180º и действовать через ведущий указатель. Возможно, силовое решение, которое работает, хотя и ненамного лучше — потребовать, чтобы функция `Copy1()` всегда вызывалась в самом конце функций класса:

```
class Foo {
public:
    void Fn()
    {
        // Код, который делает нечто осмысленное
        VoidPtr::pool->Copy1();
    }
};
```

Разумеется, все будет нормально лишь в том случае, если объект вызывавший `Fn()`, ну будет использовать свой указатель `this` после возвращения из `Fn()`. Не знаю, как вы, а лично я предпочитаю спать спокойно и не думать о том, как один из 2435 программистов, работающих с моей библиотекой классов, придумает способ все испортить.

Более достойное решение — сделать так, чтобы функция `Copy1()` вызывалась из некоторого цикла событий верхнего уровня. На самом деле нежелательно, чтобы в момент вызова `Copy1()` функции исчезающих объектов находились в стеке. В результате получается архитектура, которую я называю *опосредованной (inside-out)*, — функция класса не выполняет работу сама, а создает *объект-операцию (operational object)* и направляет его в некоторую главную очередь. Это распространенное решение встречается во многих библиотеках классов.

```
class Operation {
friend void MainLoop();
private:
    static Queue<Operation> OperationQ;
public:
    virtual void DoSomething() = 0;
    void Post() { OperationQ.Push(this); }
```

```
};
void MainLoop()
{
    Operation* op;
    while ((op = Operation::OperationQ.Pop()) != NULL)
    {
        op->DoSomething();
        object_space->Copy1();
    }
}
```

Если теперь объект захочет выполнить какое-нибудь действие, он не выполняет его сам, а создает класс, производный от `Operation`, и заносит его в очередь. Если обработка связана с итерациями, объект `Operation` продолжает направлять себя в очередь в конце каждого вызова `DoSomething()` до завершения итераций. Приведу краткий набросок традиционного подхода и опосредованной архитектуры:

```
// Традиционный способ сделать что-то
void Foo::SomeOperation()
{
    for (...)
        OnePass();
}
```

Если операция занимает много времени, перед вами возникают два неудобных варианта: не выполнять сборку мусора и уплотнение до завершения `Foo::SomeOperation()`, а следовательно, утратить многие преимущества от управления памятью; или косвенно вызвать `Copy1()` во время вызова `Foo::SomeOperation()`, а это небезопасно. Очередь операций предоставляет другое решение проблемы:

```
// Опосредованная архитектура с очередями операций
class FooSomeOperation : public Operation }
public:
    virtual void DoSomething();
    {
        // Выполнить один проход
        if (еще не готово)
            this->Post(); // Послать заново для следующего захода
        delete this;
    }
};
void Foo::SomeOperation()
{
    Operation* op = new FooSomeOperation(args);
    op->Post();
    // op->DoSomething выполняет работу
}
```

Теперь функция `Copy1()` заведомо не будет вызвана в момент нахождения в стеке `FooSomeOperation::DoSomething()`. Подобные очереди операций так часто приносят пользу, что являются едва ли не стандартной возможностью объектно-ориентированных библиотек классов. Сколько библиотек — столько и вариаций (скажем, назначение приоритетов операций или возможность блокирования одних операций до завершения других), но во всех разновидностях встречается одна общая черта — максимальное освобождение стека на время периодического выполнения вспомогательных операций.

Адреса переменных класса

Аналогичная проблема возникает и при получении адреса переменной класса, даже если это происходит в функциях класса, которым мы управляем. Именно по этой причине мы и потребовали, чтобы везде применялись дескрипторы. Благодаря опосредованной методике проблем не возникает — при условии, что вы получаете адрес переменной класса, используете и забываете про него в течение одного цикла. Впрочем, от хлопот, связанных с опосредованной архитектурой, можно и отказаться. Если вы абсолютно уверены, что адрес не сохранится до следующего вызова функции `Copy1()`, то можете избирательно снимать требование обязательного применения дескрипторов.

Множественное наследование

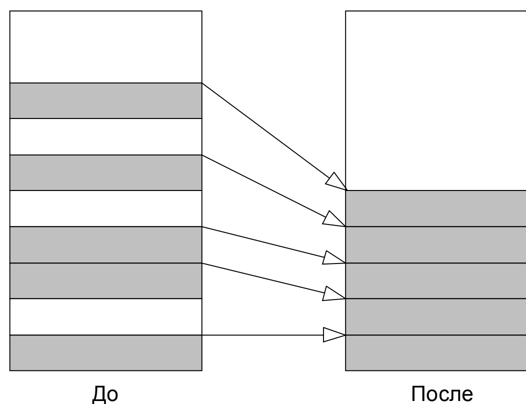
Множественное наследование безопасно при условии соблюдения всех приведенных выше рекомендаций по уходу и кормлению указателя `this`. То обстоятельство, что `this` пляшет в памяти при вызове функций второго и третьего базового класса, не вызовет новых проблем — это все та же проблема с `this`, только замаскированная. Конечно, вы никогда не должны возвращать адрес объекта, преобразованного к базовому классу, но передача `this` тоже небезопасна.

Неустойчивые объекты

Объекты, адреса которых (в отличие от адресов их ведущих указателей) передаются за пределы вашей зоны контроля — скажем, при вызове системной функции — необходимо сначала вывести из очищаемого пространства. Самое простое решение — создать объект-операцию, которая перемещает свой объект и вызывает системную функцию, когда оказывается в новом безопасном месте.

Уплотнение на месте

Очевидный недостаток алгоритма Бейкера заключается в напрасной потере половины памяти. Существует и другой, мене очевидный недостаток — при каждом проходе все объекты копируются из одного места памяти в другое. Такое копирование может отрицательно повлиять на быстродействие программы. Обе проблемы решаются в другом алгоритме, который называется *уплотнением на месте (compaction in place)*. Вместо двух половин существует единое пространство, а в процессе уплотнения все объекты смещаются вниз. На следующей диаграмме показано состояние памяти до и после уплотнения.



Копирование объектов должно происходить в правильном порядке, снизу вверх, в противном случае объекты будут накладываться друг на друга. Этого можно добиться двумя способами: отсортировать ведущие указатели перед началом перебора или изначально хранить их в отсортированном порядке. Хранить ведущие указатели в двусвязном списке, отсортированном по адресу указываемого объекта, довольно просто — при условии, что вы готовы потратить лишнюю пару слов для указателей на следующий и предыдущий элемент. Шаблон ведущего указателя и дескрипторы аналогичны тем, которыми мы пользовались до настоящего момента. Базовый класс `VoidPtr` был усовершенствован для хранения экземпляров в связанном списке.

Базовый класс VoidPtr

Память под объекты всегда выделяется снизу вверх. Если новые объекты `VoidPtr` всегда будут добавляться в конец связанного списка, то список всегда будет отсортирован по возрастанию адресов

указываемых объектов. Конструкторы (см. далее) напрямую работают с переменной VoidPtrPool::tail. Деструктор исключает экземпляр из списка. Во всем остальном класс VoidPtr остался прежним. Ниже показаны изменения в VoidPtr.

```
class VoidPtr {
private:
    // Новые переменные для ведения списка
    VoidPtr* next;          // Следующий элемент списка
    VoidPtr* previous;      // Предыдущий элемент списка
protected:
    // Изменившиеся конструкторы
    VoidPtr() : address(NULL), size(0), refcount(0),
        next(NULL), previous(NULL) {}
    VoidPtr(void* addr, size_t s) : address(addr), size(s), refcount(0),
        next(NULL), previous(pool->tail->previous)
    {
        pool->tail->next = this;
        pool->tail = this;
    }
public:
    // Измененный деструктор
    virtual ~VoidPtr()
    {
        if (size != 0)// Активный указатель - исключить из списка
        {
            if (previous != NULL)
                previous->next = next;
            if (next != NULL)
                next->previous = previous;
            if (pool->tail == this)
                pool->tail = previous;
        }
        size = 0;
        address = NULL;
    }
};
```

Пул ведущих указателей

Изменения в пуле ведущих указателей VoidPtrPool также весьма тривиальны.

```
class VoidPtrPool {     // Как и прежде, плюс следующее
friend class VoidPtr;   // Обеспечивает доступ к tail
private:
    // Новые переменные для ведения списка
    VoidPtr head;       // Фиктивный VoidPtr, который ссылается
                        // на список активных указателей
    VoidPtr* tail;      // Конец списка
public:
    // Новая версия конструктора
    VoidPtrPool() : block_list(NULL), free_list(NULL), tail(&head) {}
};
```

Класс VoidPtrPool идентичен тому, который использовался в алгоритме Бейкера, с добавлением связанного списка активных VoidPtr.

Итератор ведущих указателей

Итератор ведущих указателей устроен элементарно. Он просто перебирает элементы списка от начала к концу — иначе говоря, от нижних адресов памяти к верхним.

```
class VoidPtrPoolIterator : public VoidPtrIterator {
private:
    VoidPtr* next;
public:
    VoidPtrIterator(VoidPtr* first) : next(first) {}
    virtual bool More() { return next != NULL; }
    virtual VoidPtr* Next()
    {
        VoidPtr* vp = next;
        next = next->next;
        return vp;
    }
};
VoidPtrIterator* VoidPtrPool::iterator()
{
    return new VoidPtrPoolIterator(&head.next);
}
```

Алгоритм уплотнения

Алгоритм уплотнения выглядит так просто, что его можно было бы и не приводить.

```
class Space {
private:
    unsigned long next_byte;
    unsigned char bytes[SPACESIZE];
public:
    Space() : next_byte(0) {}
    void* Allocate(size_t size);
    void Compact();
};
void* Space::Allocate(size_t size)
{
    // Выровнять на границу слова
    size = ROUNDUP(size);
    if (next_byte + size > SPACESIZE)
    {
        Compact();
        if (next_byte + size > SPACESIZE)
            // Исключение - нехватка памяти
    }
    void* space = &bytes[next_byte];
    next_byte += size;
    return space;
}
```

```
void Space::Compact()
{
    next_byte = 0;
    VoidPtrIterator* iterator = VoidPtrPool::iterator();
    while (iterator->More())
    {
        VoidPtr* vp = iterator->Next();
        void* space = Allocate(vp->size);
        if (space < vp->address)   // Проверить, что объект поместится
    }
    delete iterator;
}
```

Оптимизация

Существует много возможностей повысить эффективность этой схемы для рядовых классов. Один из простейших видов оптимизации — хранение *нижнего уровня*, который определяет самый нижний удаленный объект. Нижний уровень представляет собой самый нижний `VoidPtr` из активного списка, ниже которого удаленных объектов нет. Он хранится в виде переменной класса `VoidPtr*` вместе с началом и концом списка. Деструктор `VoidPtr` проверяет, находится ли адрес указываемого объекта ниже текущего нижнего уровня; если да, он заменяет нижний уровень новым значением. Уплотнение начинается с нижнего уровня, поскольку ниже него перемещать ничего не требуется. Иначе говоря, мы начинаем не с начала списка, а где-то с середины — с нижнего уровня.

Этот прием особенно полезен, если учитывать специфику уплотнения на месте. Было замечено, что чем старше становится объект, тем меньше вероятность того, что он будет удален в ближайшее время. Старые объекты в этой схеме группируются в нижней части пула. Возникает большой блок объектов, которые практически не перемещаются, поскольку нижний уровень всегда находится над ними.

Другой прием оптимизации — объединять несколько смежных объектов в одной операции перемещения. Экономия не так уж велика, но несколько тактов все же удается сэкономить.

Последовательное уплотнение на месте

Алгоритм уплотнения на месте нетрудно приспособить для условий последовательного уплотнения в фоновом режиме. Все просто: мы сохраняем `VoidPtrIterator` в виде переменной класса `Space` и используем его, чтобы смещать вниз один объект при каждом вызове функции `Copy1()`. Реализация проста и прямолинейна, необходимо лишь соблюдать осторожность с удалением во время уплотнения. Помните, что в процессе перебора списка `VoidPtr` удаляется один из его элементов. Это простой частный случай проблемы надежности итераторов, которую мы обсуждали в главе 8.

Все, что говорилось об алгоритме Бейкера в контексте внешних объектов, в равной степени относится и к уплотнению на месте. Адреса, на которые ссылаются `VoidPtr`, следует проверять по интервалам адресов в пространстве уплотнения, и объекты могут свободно перемещаться из пространства памяти и в него.

Глава получилась очень длинной, а подобная схема уплотнения редко применяется на практике. Подробности оставляю читателю в качестве самостоятельных упражнений.

Перспективы

Программистам на C++ обычно не так уж часто приходится беспокоиться об уплотнении, поэтому все сказанное стоит воспринимать со здоровым скепсисом. Если вам удастся организовать нормальный подсчет ссылок, уплотнение может оказаться приятным дополнением, но в целом эта глава готовит почву для приемов сборки мусора, рассмотренных в следующей главе. Кроме того, она помогает очертить внешние границы языка, поскольку C++ по своей сути не рассчитан на свободное перемещение объектов в памяти.

В следующей главе мы вернемся к стратегии «дескрипторов повсюду». Конечно, организовать универсально уплотнение памятью для производных коммерческих классов невозможно. Но по крайней мере вы увидите, как далеко можно зайти в C++, прежде чем поднять руки, сведенные судорогой от долгого сидения за клавиатурой.

Сборка мусора

16

Вы открыли последнюю главу книги. Если до сих пор вы ее не отбросили — примите мои восхищение и сочувствие. Вероятно, я на всю жизнь отбил у вас вкус к любым реальным проектам, кроме программ управления ядерными реакторами и полетов на Марс. Итак, если кому-нибудь из читателей потребуется организовать в C++ сборку мусора, эта глава укажет им путь. А для остальных она станет очередной интеллектуальной пробежкой по обширным полям идиом и управления памятью в C++.

Доступность

Первое, что от нас потребуется — определить, какие объекты доступны, а какие нет. Сразу же возникает вопрос: доступны *откуда*? В этом разделе описаны основные варианты перемещения внутрь от конкретного периметра. За ними следуют две конкретные архитектуры сборки мусора, в которых используются описанные приемы.

Периметр

Любая методика сборки мусора, не связанная с подсчетом ссылок, должна начинаться с некоторого внешнего периметра графа объектов. Прочем, даже определить этот параметр порой оказывается непросто.

Стековые переменные

Если стряхнуть с программы на C++ всю объектно-ориентированную шелуху, у вас останутся фрагменты кода. В этих фрагментах создаются локальные переменные, которые чаще всего используются для обращений к объектам.

```
void f()
{
    Foo* foo = new Foo;     // Объект foo доступен
}
```

Стековые переменные могут непосредственно (то есть без участия другого объекта) обратиться к объекту несколькими способами:

- Указатель `this` является неявным указателем на объект, доступным из кода функции класса.
- Переменная может содержать указатель (*) или ссылку (&) на объект.
- Переменная может быть объектом.
- Переменная может содержать информацию, необходимую для обращения к объекту.

Кроме того, через стековую переменную к объекту можно обратиться и косвенно. Например, если `this` содержит переменную `Bar*`, то `Bar` будет косвенно доступен из любой функции `this`.

Внешние указатели

Если адрес объекта или его переменной передается системной функции, параллельному процессу или куда угодно за пределы вашего тщательно продуманного кода сборки мусора, такой объект также становится непосредственно доступным.

```
class String {
private:
    char* str;
public:
    operator char*() { return str; }
};
strcpy(aString, bString);   // Использует оператор char*
```

Для вызова strcpy(char*, char*) используется оператор преобразования. Во время выполнения strcpy обе строки непосредственно доступны из кода функции strcpy, которые вами не контролируются. И это еще относительно неплохо, поскольку strcpy можно считать атомарной операцией. Хуже, если вы передадите ссылку на функтор, скажем, функции базы данных как объектоориентированную функцию обратного вызова (callback). Вы можете заниматься своими делами, пока не зазвенит звонок, но до тех пор не смейте уничтожать функтор!

Индексированные коллекции

Весьма специфический случай. Предположим, коллекция объектов индексируется очень большим целым числом.

```
template <class Type>
class Array {
public:
    Type* operator[] (LargeInt);
};
```

Конечно, можно заявить, что все объекты коллекции доступны, если доступна сама коллекция, но такое решение не всегда удовлетворительно. Довольно часто требуется узнать, можно ли к объекту коллекции обратиться по индексу; то есть имеет ли какой-нибудь объект, кроме самой коллекции, индекс объекта X? Если нет — значит, адрес X имеется только у коллекции, и от него желательно избавиться. Мы не будем углубляться в эту тему и лишь мельком обратим внимание на проблему. Чаще всего она возникает при кэшировании объектов на диске и в распределенных объектых системах.

Внутри периметра

После того как будут определены все непосредственно доступные объекты, возникает следующая проблема: идентифицировать объекты, к которым возможны косвенные обращения внутри периметра. Теоретически каждый объект может обратиться к любому другому объекту. Тем не менее, объекты ссылаются друг на друга ограниченным числом способов.

Переменные классов

Один объект может быть внедрен в другой как переменная класса, или же переменная класса может представлять собой указатель или ссылку на другой объект.

Аргументы функций классов

Один объект может получить доступ к другому через аргументы своих функций. На самом деле это лишь частный случай рассмотренных выше стековых переменных.

```
void Foo::f(Bar* b)
{
    b->member_of_Bar();
}
```

Базовые классы

Базовый класс в C++ интерпретируется так, словно он является внедренным объектом. Как было показано в предыдущей главе, это особенно справедливо для множественного наследования и виртуальных базовых классов. По адресу одного объекта вы можете сослаться на несколько разных

логических объектов внутри него — одни являются переменными класса, а другие — базовыми классами. Как правило, адреса этих объектов отличны от того объекта, которому они принадлежат.

Анализ экземпляров

Алгоритмы сборки мусора обычно начинают свою работу с периметра. Для каждого объекта периметра составляется список объектов, которые он содержит и на которые ссылается. Затем для каждого такого объекта составляется новый список и т.д. Рекурсивный перебор продолжается до тех пор, пока удается находить новые объекты. Для этого нам понадобятся некоторые средства, которые позволяют для данного объекта найти все его внедренные объекты и указатели/ссылки.

В SmallTalk и других динамических языках описание структуры экземпляра является задачей объекта класса. В C++ существует несколько вариантов. Первые два решения (см. ниже) вполне практичны, а третье — отчаянная мера, которая подходит только для профессиональных каскадеров на закрытых треках.

Виртуальные функции

Если все объекты происходят от одного общего базового класса, в этом базовом классе можно объявить виртуальную функцию для перечисления указателей и внедренных объектов. Эта функция переопределяется в каждом классе, который добавляет новые переменные или объединяет базовые классы посредством множественного наследования.

Объекты классов

Вы также можете создать свои собственные объекты классов, как было показано в предыдущих главах, и научить их перечислять внедренные объекты и указатели в экземплярах.

Осведомленные указатели

В крайнем случае можно воспользоваться умными указателями и обращаться к ним с просьбой описать объект.

```
class Foo {
private:
     Bar* bar;
};
class PFoo {  // Умный указатель на Foo
private:
     Foo* foo;
public:
     FunctionThatEnumeratesPointersInFoo();
};
```

Почему я называю этот случай крайним? Вы рискуете тем, что указатель неверно определит тип объекта, на который он ссылается. Конечно, если `PFoo` — ведущий указатель, мы знаем, что foo действительно является `Foo*`, но что делать с `bar`? Как узнать, что это действительно `Bar`, а не что-то производное от него? Если `Bar` не имеет только что упоминавшейся самоописывающей виртуальной функции и не возвращает объект класса, остается одно — повсюду раскидать умные указатели и надеяться на лучшее.

```
class Bar {
};
class Pbar {  // Умный указатель на Bar
};
class Foo {
private:
     Pbar bar;
};
```

```
class PFoo {  // Умный указатель на Foo
private:
    Foo* foo;
public:
    FunctionThatEnumeratesPointersInFoo();
};
```

Теперь мы начинаем с одного умного указателя, `PFoo`, и рекурсивно находим другой, `PBar`. Каждый из этих умных указателей разбирается в особенностях строения объекта, на который он ссылается. В этом они превзошли умные указатели, поэтому я называю их *осведомленными (ingenious)*, хотя циник, вероятно, назвал бы их *нерассуждающими*.

Перебор графа объектов

В дальнейшем мы воспользуемся методикой виртуальных функций из приведенного выше списка, хотя материал с таким же успехом применим и к объектам классов. Перечисление реализуется двумя основными способами: с применением рекурсивных функций и функторов, а также с применением итераторов.

Рекурсивные функции и функторы

Первая естественная реакция: организовать механизм косвенного вызова, создать функцию или функтор, вызываемые для каждого доступного объекта, и подождать, пока закончится рекурсивный перебор.

```
class Functor {    // 'Функция' обратного вызова
public:
    virtual void Apply(MotherOfAllClasses*) = 0;
};
class MotherOfAllClasses {
public:
    // Применить fn к каждому доступному объекту
    virtual void EachObject(Functor& fn);
};
```

Функция `EachObject()` вызывает `fn.Apply(this)`, а затем вызывает `EachObject()` для каждого внедренного объекта или объекта, на который указывает переменная класса. Кроме того, `EachObject()` вызывает `Base::EachObject()` для каждого базового класса, таким образом члены базового класса тоже включаются в перечисление. В зависимости от алгоритма в `MotherOfAllClasses` можно включить бит признака, показывающий, что объект был рассмотрен ранее. Впрочем, как мы вскоре убедимся, иногда без этого можно обойтись.

Итераторы

Более удачное решение — организовать возвращение объектом итератора для всех внедренных объектов (включая базовые классы), в том числе и тех, на которые непосредственно ссылаются его переменные.

```
class MOAOIterator {    // "MotherOfAllObjectsIterator"
public:
    virtual bool More() = 0;
    virtual MotherOfAllObjects* Next() = 0;
};
class MotherOfAllObjects {
public:
    virtual MOAOIterator* EachObject();
};
```

Конечно, на этот раз потребуется более хитроумный код, чем в варианте с виртуальными функциями из последнего раздела. Тем не менее, методика «итераторы всюду» обладает одним громадным преимуществом: она позволяет выполнять действия последовательно. Вариант с рекурсивными функциями не позволяет каждую миллисекунду или около того делать передышку и давать поработать другому коду. При использовании итераторов, если соблюдать осторожность, это не проблема. Далее мы будем использовать именно этот вариант.

Сборка мусора по алгоритму Бейкера

Наверное, вам хочется знать, зачем нужен алгоритм Бейкера, не правда ли? В предыдущей главе я выдал его за алгоритм уплотнения, но что толку уплотнять память, если для этого приходится жертвовать 50 процентами ее общего объема? Теперь мы узнаем настоящую прелесть алгоритма Бейкера — его применение в архитектурах сборки мусора.

На данный момент мы не будем беспокоиться об объектах, доступных извне, и сосредоточим внимание на периметре стековых переменных.

Поскольку на этот раз мы занимаемся сборкой мусора, нет причин полагаться во всем на подсчет ссылок. Тем не менее, подсчет ссылок продолжает играть важную роль: он применяется для подсчета дескрипторов в стеке, ссылающихся на конкретный ведущий указатель. Ведущий указатель, у которого счетчик ссылок больше 0, непосредственно доступен из стека, а следовательно, входит в периметр. Мы воспользуемся сильными дескрипторами для стековых переменных и слабыми дескрипторами для ссылок из одного объекта на другой через переменные класса. `VoidPtr` и другие структуры данных из предыдущей главы остаются без изменений, за одним исключением: `VoidPtr::Release()` не удаляет ведущий указатель при обнулении счетчика. Запомните: нулевой счетчик ссылок означает не то, что объект вообще недоступен, а лишь то, что он недоступен непосредственно из стека.

Шаблон слабого дескриптора

Слабый дескриптор устроен просто.

```
template <class Type>
class WH {
friend class Handle<Type>;
private:
	BMP<Type>* pointer;
	WH() : pointer(new BMP<Type> (new(object_space) Type)) {};
	BMP<Type>& operator->() { return *pointer; }
};
```

Он используется в переменных классов, которые ссылаются на другие объекты.

```
class Foo {
private:
	WH<Bar> bar;  // При конструировании создает Bar + MP<Bar>
};
```

Шаблон сильного дескриптора

Шаблон сильного дескриптора идентичен шаблону слабого, за исключением того, что он поддерживает счетчик ссылок для указателя.

```
template <class Type>
class SH {
private:
	BMP<Type>* pointer;
public:
	SH() : pointer(new BMP<Type>(new Type)) { pointer->Grab(); }
	SH(const SH<Type>& h) : pointer(h.pointer) { pointer->Grab(); }
```

```
        SH(const WH<Type>& h) : pointer(h.pointer) { pointer->Grab(); }
        operator WH<Type>() { return WH<Type>(pointer); }
        SH<Type>& operator=(const SH<Type>& h)
        {
            if (this == &h) return *this;
            if (pointer == h.pointer) return *this;
            pointer->Release();
            h.pointer->Grab();
            return *this;
        }
        BMP<Type>& operator->() { return *pointer; }
    };
```

Шаблон используется для обычных переменных (а не для переменных класса), ссылающихся на объекты. Благодаря конструктору, принимающему `H<Type>`, и операторной функции `operator H<Type>()` он также может использоваться в операциях присваивания с участием переменных классов, то есть слабых дескрипторов.

```
    class Bar {
    private:
        WH<Foo> foo;
    public:
        void f();
    };
    void Bar::f()
    {
        SH<Foo> f;     // Эквивалентно Foo* f = new Foo;
        f = foo;       // Использует operator=(SH<Type>(foo));
        foo = f;       // Использует operator WH<Type>(f);
    }
```

Итераторы ведущих указателей

Помните `VoidPtrIterator`? `VoidPtrPool` возвращает один итератор для перебора всех указателей с ненулевыми счетчиками ссылок. Все остается без изменений, однако счетчик ссылок теперь интерпретируется по-другому. Раньше ненулевой счетчик ссылок означал, что объект не следует уничтожать. Теперь он имеет более узкое значение: объект доступен непосредственно из стека. Все эти объекты сохраняются, поскольку они находятся на периметре, но мы также сохраним объекты с нулевыми счетчиками ссылок, если они доступны косвенно.

Для объектов внутри периметра мы должны перебрать дескрипторы каждого объекта периметра, а затем рекурсивно двигаться внутрь до тех пор, пока не будут перебраны все доступные объекты. Для этого нам придется анализировать объекты одним из описанных выше способов. В данном примере будет использовано сочетание виртуальных функций/итераторов. Для этой цели можно слегка переработать старый интерфейс `VoidPtrIterator`.

```
    class VoidPtrIterator {
    protected:
        VoidPtrIterator() {}
    public:
        virtual bool More() = 0;
        virtual VoidPtr* Next() = 0;
    };
```

Теперь пул должен поддерживать два типа итераторов. Один итератор перебирает указатели, находящиеся на периметре (то есть имеющие ненулевые счетчики ссылок). Второй — указатели на

объекты, находящиеся в указанной половине. Итератор VoidPtrPool::iterator() из главы 15 заменяется следующим:

```
// Включить в класс VoidPtrPool
class VoidPtrPool {
public:
    VoidPtrIterator* Reachable()
        { return new ReachableIterator(this); }
    VoidPtrIterator* InRange(void* low, void* high)
        { return new RangeIterator(this); }
};
```

Указатели периметра

Один из типов итераторов, возвращаемых пулом, перебирает непосредственно доступные указатели (имеющие ненулевой счетчик ссылок). В сущности, перед нами тот же VoidPtrPoolIterator с одной изменившейся строкой — теперь Advance() пропускает позиции с нулевым счетчиком ссылок. Класс реализован как производный от VoidPtrPoolIterator.

```
class ReachableIterator : public VoidPtrPoolIterator {
protected:
    virtual void Advance()    // Найти следующую используемую позицию
    {
        do
            VoidPtrPoolIterator::Advance();
        while (block != NULL && block->slots[slot].refcount == 0);
    }
public:
    ReachableIterator(VoidPtrBlock* vpb) : VoidPtrPoolIterator(vpb) {}
};
```

Недоступные указатели

В конце цикла мы должны пройтись по ведущим указателям и найти все те, которые продолжают ссылаться на неактивную половину. Это и будут недоступные объекты. Задачу решает следующий итератор, в котором используется очередное тривиальное переопределение VoidPtrPoolIterator.

```
class InRange : public VoidPtrPoolIterator {
private:
    void* low;     // Нижний адрес диапазона
    void* high;    // Верхний адрес диапазона
    virtual void Advance()  // Найти следующую используемую позицию
    {
        do
            VoidPtrPoolIterator::Advance();
        while (block != NULL &&
            (block->slots[slot].address < low ||
            block->slots[slot].address >= high));
    }
public:
    InRange(VoidPtrBlock* vpb, void* low_addr, void* high_addr)
        : VoidPtrPoolIterator(vpb), low(low_addr), high(high_addr) {}
};
```

Перебор указателей в объектах

Каждый объект возвращает другой итератор VoidPtrIterator, который перебирает указатели, доступные непосредственно из объекта. Для каждого класса этот итератор должен быть своим. Далее показан пример.

```
class MotherOfAllObject {   // Базовый класс для всех остальных
public:
    virtual VoidPtrIterator* Pointers() = 0;
};
template <class Type>
class VoidPtrArrayIterator : public VoidPtrIterator {
private:
    VoidPtr* ptrs[Entries];
    int next;     // Следующая позиция в переборе
public:
    VoidPtrArrayIterator() : next(0)
    {
        for (int i = 0; i < Entries; i++)
            ptrs[i] = NULL;
    }
    VoidPtr*& operator[](uint slot) { return ptrs[slot]; }
    virtual bool More() { return next < Entries; }
    virtual VoidPtr* Next() { return ptrs[next++]; }
};
// Пример класса и итератора
class Foo {
private:
    WH<Bar> bar;
public:
    virtual VoidPtrIterator* Pointers()
    {
        new VoidPtrArrayIterator<1>* iterator = new VoidPtrArrayIterato<1>;
        iterator[0] = bar.Pointer();
        return iterator;
    }
};
```

VoidPtrArrayIterator сделан на скорую руку и в реальном проекте его использовать не стоит, но по крайней мере он демонстрирует общий принцип. Конечно, его следует дополнить проверками диапазонов и инициированием исключений, если будет затребован VoidPtr* для NULL. Foo::Pointers() показывает общий принцип использования VoidPtrArrayIterator. Для каждого класса мы изменяем размер массива, чтобы он совпадал с количеством WH<Widget> и добавляем для каждого дескриптора по одной строке вида iterator(index++) = widget.Pointer(). Этот шаблон справляется со всеми простыми случаями, в которых нам не приходится беспокоиться о базовых классах. Если Foo имеет базовые классы, придется организовать вложение итераторов для его собственных указателей и указателей базовых классов.

Перебор указателей

Настал момент собрать все воедино в алгоритме перебора всех доступных объектов. Встречая объект, который в данный момент находится в неактивной половине, мы копируем его в активную половину и изменяем адрес в ведущем указателе на новую копию. Если найденный объект уже находится в активной половине, предполагается, что он уже был скопирован, поэтому мы не тратим время на

дальнейшие манипуляции с ним. Объекты, не принадлежащие ни одной из половин, мы пока игнорируем.

Интерфейс Space слегка отличается от того, который использовался для уплотнения. Вместо одного итератора приходится поддерживать стек итераторов, поскольку мы перемещаемся по графу объектов. Кроме того, появилась новая функция Scavenge(), которая вызывается в конце каждого прохода по половине. Предполагается, что у нас уже имеется готовый шаблон стека Stack.

```
template <class Type>
class Stack {
public:
     Push(Type*);
     Type* Pop();  // Возвращает NULL для пустого стека
};
class Space {
private:
     VoidPtrIterator* iterator;      // Итератор верхнего уровня
     Stack<VoidPtrIterator> iterator_stack;
     HalfSpace A, B;
     HalfSpace* active;
     HalfSpace* inactive;
     void Scavenge();   // Уничтожить недоступные объекты
     void Swap();       // Переключить активную половину
public:
     Space() : active(&a), inactive(&B), iterator(NULL) { Swap(); }
     void* Allocate(size_t size)
     {
          void* space = active->Allocate(size);
          if (space == NULL) throw(OutOfMemory());
          return space;
     }
     void Copy1();
};
```

Три ключевые функции — Scavenge(), Swap() и Copy1() — ниже рассматриваются более подробно.

Scavenge

Функция Scavenge() вызывается после одного полного цикла. Она перебирает все ведущие указатели и ищет объекты, оставшиеся в неактивной половине. Эти объекты недоступны. Для каждого объекта она удаляет указатель, который, в свою очередь, вызывает деструктор объекта.

```
void Space::Scanvege()
{
     VoidPtrIterator* vpi =
          VoidPtr::pool->InRange(inactive, inactive + sizeof(*inactive));
     while (vpi->More()) {
          VoidPtr* vp = vpi->Next();
          delete vp;    // Вызывает деструктор указываемого объекта
     }
     delete vpi;
}
```

Swap

Функция `Swap()` переключает активную половину. Сначала она вызывает `Scavenge()` в завершение предыдущего цикла, а потом сбрасывает все в исходное состояние, чтобы при следующем вызове функции `Copy1()` копирование пошло в обратную сторону.

```
void Space::Swap()
{
    Scavenge();   // Уничтожить объекты в неактивной половине
    if (active == &A)
    {
        active = &B;
        inactive = &A;
    }
    else
    {
        active = &A;
        inactive = &B;
    }
    active->Reinitialize();
    iterator = VoidPtr::pool->iterator();
}
```

Copy1

Функция `Copy1()` рассматривает один объект. Если объект находится в неактивной половине, он копируется в активную. Если объекта нет, то в рамках текущей задачи мы предполагаем, что он находится в активной половине, а следовательно, был перемещен ранее.

```
void Space::Copy1()
{
    if (!iterator->More())
    {
        // Перебор закончен, удалить итератор и вытолкнуть из стека
        delete iterator;
        iterator = iterator_stack.Pop();
        if (iterator == NULL)      // Готово!
            Swap();   // Начинаем двигаться в другую сторону
    }
    else
    {
        VoidPtr* vp = iterator->Next();
        if (vp->address >= &inactive &&
            vp->address < &inactive + sizeof(*insactive))
        {
            // Объект доступен и его нужно переместить
            void* new_space = active->Allocate(vp->size);
            if (new_space == NULL)
                // Исключение - нехватка памяти
            memcpy(new_space, vp->address, vp->size);
            vp->address = new_space;
            iterator_stack.Push(iterator);
            iterator = vp->address->Pointers();
```

```
            }
            // Иначе перемещение уже состоялось
        }
    }
```

Оптимизация

Только что описанный алгоритм способен резко тормозить программу при каждом запуске функции `Scavenge()`. Для оптимизации по отдельности или вместе могут использоваться два следующих подхода:

1. Функция `Scavenge()` работает поэтапно, а не как единая операция. Для этого вам придется модифицировать `Copy1()`, чтобы ее последовательный аналог вызывался до завершения сборки мусора.
2. Ведение отдельных списков используемых ведущих указателей для каждой половины вместо перебора содержимого `VoidPtrPool` для их поиска. При перемещении каждого объекта из неактивной половины в активную ведущий указатель также перемещается из одного списка в другой. В конце цикла уплотнения для неактивной половины остается список лишь тех ведущих указателей, которые должны быть уничтожены.

Вряд ли эти варианты оптимизации стоит применять в реальных проектах — особенно второй, поскольку ведение списков потребует значительно больших затрат памяти и быстродействия.

Внешние объекты

Объекты, которые существуют за пределами пространства сборки мусора, слегка усложняют нашу задачу. Объекты внутри пространства могут ссылаться на эти «внешние» объекты. Само по себе это не вызовет проблем, поскольку перемещаются только объекты неактивной половины. Проблемы возникают в ситуациях, когда внешние объекты ссылаются на внутренние. Вероятно, они будут использовать дескрипторы, но это заметно повысит сложность алгоритма сборки мусора/уплотнения. Потребуются следующие изменения:

1. Каждый внешний объект также должен обладать средствами перебора указателей и соблюдать правило «дескрипторы повсюду», по крайней мере для ссылок на внутренние объекты.
2. Каждый внешний объект во время очередного прохода должен уметь помечать себя как просмотренный.
3. Если объект в функции `Copy1()` является внешним и непомеченным, он помечается, а его итератор заносится в стек, но сам объект при этом не перемещается.

Множественные пространства

Когда базовая структура будет налажена, объекты в пространстве сборки мусора можно без особых проблем объединить с объектами, управляемыми другими средствами. Главное — чтобы все объекты сотрудничали в процессе перебора указателей. Способ управления объектом легко определяется по его адресу.

Сборка мусора и уплотнение на месте

Решения из предыдущей главы, которые помогли нам превратить схему «дескрипторы повсюду» в уплотнение на месте, можно применить и в данной схеме. Это позволит организовать сборку мусора на месте и обойтись без копирования объектов в памяти. Существуют два варианта этой схемы: с уплотнением и без. В обоих случаях используется алгоритм пометки и удаления — на первом проходе определяются доступные объекты, а на втором происходит сборка мусора и при необходимости — уплотнение. Алгоритм предполагает, что в класс `VoidPtr` добавлен специальный «бит пометки»:

1. Снять пометку со всех `VoidPtr`, отсутствующих в списке свободных указателей.
2. Пометить все `VoidPtr` с ненулевым счетчиком ссылок; то есть пометить объекты, доступные непосредственно из стека.

3. Для каждого только что помеченного VoidPtr пометить все VoidPtr, внедренные в объекты, на которые они ссылаются. При этом используются те же итераторы, что и для алгоритма Бейкера.
4. Повторять шаг 3, пока удается находить новые помечаемые объекты.
5. Удалить все VoidPtr, не помеченные и не находящиеся в списке свободных; в свою очередь, это приведет к вызову деструкторов указываемых объектов. Если вы не собираетесь выполнять уплотнение, следует вернуть память, занимаемую этими объектами.
6. Если уплотнение выполняется, перебрать все помеченные VoidPtr в порядке возрастания адресов указываемых объектов и сместить объекты вниз для уплотнения фрагментированного пространства.

Сделать все это поэтапно несколько сложнее, но если действовать внимательно, возможно и это. Главное — помнить, что объект, ставший недоступным, доступным уже не станет. Объект, который был доступен в начале прохода, но стал недоступным во время него, можно не уничтожать. Память этого объекта будет возвращена во время следующей прогулки по памяти.

Нужно ли вызывать деструкторы?

Нужно ли вызывать деструкторы объектов, ставших недоступными? На этот вопрос трудно дать однозначный ответ. Хотите ли вы, чтобы они вызывали функции других объектов (доступных или нет)? Предполагается, что деструкторы не удаляют другие объекты; с этой целью мы и организовали сборку мусора, поэтому на долю деструкторов остается не так уж много. С другой стороны, иногда в своих деструкторах объекты делают что-то другое — например, освобождают системные ресурсы или закрывают файлы. В общем, у меня нет готового ответа. Решайте сами в зависимости от ситуации.

Только для профессиональных каскадеров

Наверняка вы заглянули в этот раздел хотя бы из любопытства, не правда ли? А может, вы стоите в книжном магазине и думаете, стоит ли покупать эту книгу, и вдруг при просмотре оглавления вам в глаза бросилось интригующее название. Да ладно, признавайтесь — я и сам такой.

Вместе с настоящими сорвиголовами, которые привыкли жить на грани риска, мы посмотрим, как организовать управление памятью для традиционных классов (в отличие от классов, построенных по принципу «дескрипторы повсюду»). Вероятно, приведенный ниже материал понадобится лишь очень немногим читателям, да и те должны очень хорошо программировать на C++. Ну, а если вы все еще раздумываете над тем, стоит ли покупать книгу — купите и прочитайте несколько сотен предыдущих страниц.

Ниже описаны некоторые концепции сборки мусора, которые не перемещают объекты в памяти и требуют никаких особых правил программирования (за исключением первой концепции). Я ограничиваюсь общими набросками, поскольку код сильно зависит от структур данных, выбранных для реализации архитектуры. В конце концов, превращение идей в программный код — право тех, кто на это способен.

Концепции «матери всех объектов»

Начнем с решений, построенных на идее «матери всех объектов» (Mother Of All Objects, MOAO). Чтобы не возвращать итератор для VoidPtr, виртуальная функция может возвращать итератор для void*& или MOAO*&. Выглядит вполне разумно, пока вы не остановитесь и спросите себя — а почему мы отказались от «дескрипторов повсюду»? Скорее всего, из-за того, что не могли в достаточной степени управлять ими. Возможно, вы унаследовали (шутка из области C++) библиотеку классов, созданную кем-то другим, и не захотели переписывать ее по принципу «дескрипторы, одни дескрипторы и ничего, кроме дескрипторов». Может, вы считаете, что ваши клиенты и коллеги попросту не поймут столь сложной архитектуры. А может, вам не хочется превращать C++ в некое подобие SmallTalk, хотя бы в области межобъектных ссылок. Какими бы причинами вы ни руководствовались, нелогично отказываться от «дескрипторов повсюду» и оставлять другие требования — производить все от общего базового класса, перебирать указатели и плясать вокруг адресов переменных и базовых классов. Давайте-ка лучше займемся тем, что достойно настоящих программистов.

Материал, изложенный далее, делится на четыре темы:

1. Организация памяти.
2. Поиск периметра.
3. Перебор внутри периметра.
4. Сборка мусора.

Организация памяти

Существует несколько ключевых вопросов, на которые вы должны уметь быстро отвечать. А для этого необходимо, чтобы память находилась в более-менее организованном состоянии:

1. Известен некий участок памяти. Хранится ли в нем адрес или что-то другое — скажем, номер банковского счета?
2. Известен адрес. Ссылается ли он на объект или просто на случайное место в памяти?
3. Известен адрес объекта. К чему он относится — к вмещающему объекту или же к переменной или базовому классу другого объекта?

Блоки памяти

Управляемый блок памяти начинается с короткого заголовка, в котором хранится следующая информация:

- физический размер блока;
- признак использования блока;
- логический размер блока.

Первоначально вся память представляет собой один большой блок. Когда блок делится, он всегда делится пополам. Рекурсивное деление продолжается, пока не будет найден блок, размер которого равен минимальной степени 2, достаточной для хранения создаваемого объекта. В процессе удаления по начальному адресу блока и его размеру можно легко определить его парный блок; это обеспечивает эффективное объединение смежных свободных блоков.

А теперь ответим на вопросы, перечисленные выше.

Является ли значение адресом?

Является ли некоторая четрехбайтовая (на большинстве компьютеров) величина адресом памяти? Будем считать, что является, если она указывает внутрь всего управляемого пространства (то есть исходного, неразделенного блока).

Является ли адрес адресом объекта?

Будем считать, что является, если адрес лежит в логическом диапазоне используемого блока. Логический диапазон начинается после заголовка и завершается на его логическом размере. Наименьший блок, содержащий данный адрес, находится с помощью поиска в бинарном дереве памяти. Если адрес находится за пределами управляемой памяти и указывает на неиспользуемый блок или на заголовок блока, он не может быть адресом объекта.

Ссылается ли адрес на объект верхнего уровня?

Если точка, на которую ссылается адрес, расположена сразу же после заголовка используемого блока, то адрес ссылается на объект верхнего уровня. Если адрес ссылается на некоторую внутреннюю точку объекта, он соответствует переменной класса или базовому классу вмещающего объекта.

Быстродействие

Если управляемая память имеет длину N байт и вы никогда не выделяете менее 2^M байт, то ответы на все три вопроса потребуют не более N-M просмотров заголовков блоков. Например, если N=20 (один мегабайт), а M=4 (минимальный размер блока равен 16 байтам), потребуется не более 16 попыток. Это

не так уж мало, поэтому важно найти оптимальный размер блока — большие блоки увеличивают фрагментацию, но сокращают количество просмотров.

Поиск периметра

Снятие ограничения «дескрипторы повсюду» означает, что будет разрешен код наподобие следующего:

```
class Foo {
private:
    Bar* bar;
};
Foo* f = new Foo;
```

Кроме того, это означает, что будут разрешены указатели на базовые классы (помните дурацкие фокусы с `this`?) и указатели на переменные классов. Кончено, становится намного сложнее определить, что доступно, а что — нет, начиная с поиска периметра. Рассмотрим два варианта.

Умные указатели

Как и прежде, самое надежное — хранить умные указатели в стеке, даже если они и не являются дескрипторами. Для перебора этих указателей можно воспользоваться скрытой коллекцией. Конструктор умного указателя заносит его в коллекцию, а деструктор — удаляет.

Перебор стека

Возможно, это звучит довольно странно, однако периметр можно определить приближенно, с ошибкой в консервативную сторону (то есть с «запасом»). Достаточно просто просканировать стек и найти в нем значения, соответсвующие адресам объектов. Всегда существует вероятность, что там найдется переменная с телефоном тетушки Милли из Небраски, которая по чистой случайности совпадает с адресом некоторого объекта в памяти. Это называется *имитацией указателя (pointer aliasing)*. В результате объект помечается как доступный, хотя в действительности он недоступен. Обычно это не имеет вредных последствий, разве что несколько неиспользуемых байт не будут возвращены в систему. Подумайте хорошенько — случайный «адрес» в стеке должен не только ссылаться на нужное место в памяти, но и быть *единственным* указателем на недоступный объект. В общем, особенно переживать не стоит.

Пометка объектов

Итак, вы определили, что стековая величина ссылается на допустимый объект. Теперь необходимо пометить этот объект. Бит пометки должен быть частью заголовка блока, поэтому единственная хитрость заключается в том, как эффективно найти наименьший содержащий блок. Для этого придется перебирать дерево памяти до тех пор, пока не будет найден заголовок наименьшего блока.

Перебор внутри периметра

После того как вы определите периметр одним из перечисленных выше способов, возникает следующая задача — пройтись по всем объектам внутри периметра. И снова существуют два основных варианта: анализ объекта или интерпретация всех значений как потенциальных указателей.

Анализ объекта

Программу можно видоизменить, чтобы в перебор включались только указатели внутри каждого объекта. При этом можно использовать решение с виртуальными функциями, объектами классов или даже заставить умные указатели организовать перебор указателей в тех объектах, на которые они ссылаются. В любом случае вам придется основательно потрудиться над модификацией кода ваших классов.

Силовое решение

Второй вариант — просканировать весь логический размер каждого помеченного объекта в поисках потенциальных адресов объектов. Мы делаем то же самое, что делалось раньше для стека, и

сталкиваемся со знакомой проблемой имитации указателей — раздражающей, но безвредной. Каждый раз, когда будет найдено значение, соответствующее адресу некоторого объекта, этот объект помечается и включается в рекурсию.

Внешние объекты

При управлении несколькими пространствами памяти можно встретить объекты, находящиеся не в главном пространстве, в котором происходит сборка мусора, а в другом пространстве по вашему выбору. Тот факт, что объект является внешним, не снимает с вас ответственности — он вполне может ссылаться обратно, в управляемое пространство. Если это пространство не было рассчитано на эффективный перебор указателей (то есть не имеет заголовков объектов), дальше выкручивайтесь сами.

Сборка мусора

Итак, к концу фазы пометки вы определили доступные объекты. Что же дальше? Без дескрипторов и ведущих указателей уплотнение неоправдано, поскольку не существует единого места, в котором можно было бы обновить адрес перемещаемого объекта. Теоретически можно сделать второй проход по памяти и обновить все указатели тем же способом, который использовался при пометке доступных объектов. Прежде чем это делать, закупите побольше акций производителей мощных RISC-компьютеров: работы у них прибавится. Более практичное решение — организовать сборку мусора на месте.

Если вы не можете гарантировать, что все объекты происходят от общего предак, деструкторы лучше не вызывать (а если можете, то зачем использовать такую извращенную и ненадежную архитектуру?). Вполне может оказаться, что вы имеет дело с `int` или `char*`; никто не гарантирует, что у вашего объекта есть v-таблица! Не забывайте о том, что C++ — это все-таки не Lisp и не SmallTalk.

Последовательная сборка мусора

Алгоритмы пометки и удаления довольно трудно реализовать в последовательном варианте, но при должном внимании возможно и это. К сожалению, подробности выходят за рамки этой книги, но они относятся не к C++, а к выбранным вами конкретным алгоритмам.

Итоговые перспективы

В двух последних главах я попытался показать, как сделать на C++ то, для чего он не предназначен. В методиках управления памятью сочетается все, о чем говорилось в книге, от простейших умных указателей и гомоморфизма до объектов классов и подсчета ссылок. Но имеет ли все сказанное какое-нибудь практическое значение или является высокоинтеллектуальным развлечением?

Во-первых, лучший способ понять границы возможностей C++ и разобраться в его идиомах — залезть в дебри управления памятью. Даже если в ваших проектах это не нужно, хорошее понимание языковых ограничений и представления объектов в памяти только пойдет вам на пользу. В конце концов, это повысит вашу квалификацию в отладке, поскольку вы будете досконально понимать, как объекты хранятся в памяти.

Во-вторых, в один прекрасный день перед вами может возникнуть задача: организовать серьезное управление памятью по промышленным стандартам. Когда эта беда произойдет, вы будете к ней готовы. И помните, что эти главы не содержат конкретных решений, а лишь показывают, как реализуются на C++ алгоритмы, выкопанные пыли академических изданий. Все описанные приемы пригодятся, но мы лишь мимоходом коснулись этой обширной темы.

В-третьих, представьте себе вечеринку по C++. Вы ждете, когда окружающие придут в хорошее расположение духа, берете мартини и произносите ключевую фразу: «Помню, летом 95-го делали мы один проект на C++, и возникла задача: реализовать схему сборки мусора с уплотнением…» Развлекайтесь!

ПРИЛОЖЕНИЕ

Java против C++

При виде ажиотажа, поднятого вокруг Java, невольно возникает вопрос: а не является ли это обычной рекламной шумихой, единственная цель которой — заставить вас купить очередной язык, обновить компьютер и приобрести кучу книг? Верятно, вам приходилось видеть, как обозреватели в вашем любимом компьютерном журнале называют Java «новой версией» C++. Если это действительно так, то стоит ли вам, знатоку C++, беспокоиться об этом «новом» языке?

Java — это просто *диалект* C++. Кое-кто называет Java «вычищенным» вариантом C++, из которого убраны некоторые редко используемые и нелогичные возможности. Выходит, вы почти что знаете Java, не открыв ни одной книги. Тем не менее, у Java есть определенные аспекты, способные поколебать вашу уверенность. В этом приложении рассматриваются некоторые отличия между языками.

Забудьте о ручном управлении памятью, благодаря которому на C++ можно писать приложения, превосходящие Java-аналоги по быстродействию и более эффективно расходующие память. Разработчики Java ликвидировали ручное выделение и освобождение памяти (пример 1), стремясь снизить вероятность ошибок при кодировании.

Пример 1

```
int* pt = new int;
delete pt;
```

Арифметические операции с указателями в Java отсутствуют (см. пример 2). Массивы Java представляют собой настоящие массивы, а не указатели, как в C++. Используемая в Java модель указателей фактически ликвидирует синтаксис указателей C++. Изменения были внесены для предотвращения случайных нарушений памяти и порчи данных из-за ошибочных смещений в арифметических операциях с указателями.

Пример 2

```
char* na = “Bob Smith”
na++;
```

Java предотвращает утечки памяти за счет ее автоматического освобождения — своего рода автоматическая сборка мусора.

Кроме того, размер встроенных типов данных в Java не зависит от компилятора или типа компьютера, как в C++. Типы данных имеют фиксированный размер — скажем, `int` в Java всегда является 32-разрядным числом (табл. 1). Компилятор Java генерирует инструкции байт-кода, которые эффективно преобразуются в набор машинных команд.

Таблица 1. Типы данных Java

Тип	Размер
`int`	4 байта
`short`	2 байта
`long`	8 байт
`float`	4 байта
`double`	8 байт

Кроме того, вы не встретите еще некоторых знакомых конструкций. Разработчики Java ликвидировали еще две конструкции, связанные с управлением памятью — структуры (см. пример 3) и объединения. Java не поддерживает этих синтаксических средств C++.

Пример 3

```
struct name {
     char fname[20];
     char lname[30];
}
```

Одна из задач Java заключалась в том, чтобы предотвратить динамические ошибки за счет ликвидации источника распространенных ошибок в C++. Среди таких ошибок — оператор присваивания (=), перепутанный с оператором равенства (==). В примере 4 показана распространенная ошибка C++, предотвращаемая компилятором Java.

Пример 4

```
if (value = 10)
```

Вы провели бесчисленные часы за разработкой изощренной иерархии множественного наследования и теперь желаете перенести ее в Java? Вас ждет некоторое разочарование. Принципиальное отличие между Java и C++ заключается в том, что Java не поддерживает множественного наследования из-за сложностей в управлении иерархиями. Тем не менее, в Java существуют интерфейсы, которые обладают преимуществами множественного наследования без тех затруднений, которые с ним связаны.

Остерегайтесь коварства Java! Этот язык полон ловушек для программистов на C++. Например классы Java похожи на C++. Тем не менее, все функции в Java (в том числе и `main`) должны принадлежать некоторому классу. В соответствии с требованиями Java для `main` необходимо создать класс-оболочку (см. пример 5). В Java нет функций классов, а есть методы, поэтому `main` — метод, а не функция

Пример 5

```
public class ShellClass
{
     public static void main(Strings[] args)
     {
     }
}
```

Работа со строками в Java несколько запутанна. В C++ строка представляет собой массив символов, и вы можете модифицировать отдельные символы в строке. В Java дело обстоит иначе. Строки Java больше напоминают указатель `char*`. Строковые объекты Java удобны для программистов, поскольку они автоматически выделяют и освобождают память. Это происходит в операторе присваивания, в конструкторе и деструкторе.

Методы Java похожи на функции классов C++, но все же не идентичны им. Например, в Java нет глобальных функций и прототипов функций. Компилятор Java работает в несколько проходов, что

позволяет использовать методы до их определения. Более того, функции нельзя передать адрес переменной, поскольку аргументов-указателей и ссылок в Java не существует.

Некоторые части Java узнаются с первого взгляда. Например, объектные переменные Java аналогичны объектым указателям C++ (см. пример 6). Объекты Java находятся в куче, а объект, содержащий объектную переменную другого объекта, на самом деле указывает на другой объект в куче.

Пример 6

```
// Java
MyObject ob1;
// C++
MyObject* ob1;
```

Методы Java должны определяться внутри класса. Внешнее определение, как в C++, не допускается. Фрагмент, показанный в примере 7, работает в C++, но не в Java. Хотя методы определяются внутри класса, это не значит, что они автоматически становятся подставляемыми (inline) функциями.

Пример 7

```
class Person
{
};

void Person::Raise()
{
    salary *= 1000
}
```

Стоит ли бросать C++ и переходить на Java? Трудно сказать. Java заслуживает пристального внимания при разработке приложений для Internet, корпоративных или внешних сетей. Библиотека Java содержит все средства, необходимые для работы с протоколами TCP/IP, HTTP и FTP. Благодаря этому обратиться к сетевому объекту по URL так же просто, как и в локальной файловой системе.

Язык Java архитектурно нейтрален, поскольку копилятор генерирует объектный код и делает Java-приложения независимыми от реализации. Это особенно важно для Internet-приложений. Однако в Java вам не удастся использовать средства управления памятью C++, чтобы выжать все возможное быстродействие для данной платформы. Так приходится расплачиваться за управление памятью, переданное в распоряжение Java.

Как только речь заходит о многопоточности, архитектурная нейтральность Java исчезает. Многопоточный код архитектурно нейтрален, но для реализации многопоточности Java прибегает к услугам операционной системы, поскольку реализация программных потоков существенно различается на разных платформах.

Произведет ли Java революцию в языках программирования? Станет ли это концом C++? Не спешите выбрасывать свой компилятор C++. При разработке приложений для Internet, а также корпоративных и внешних сетей Java оказывается более простым и удобным языком, чем C++. И все же вопрос о том, удастся ли Java стать действительно всесторонним языком и переманить на свою сторону программистов, остается открытым.