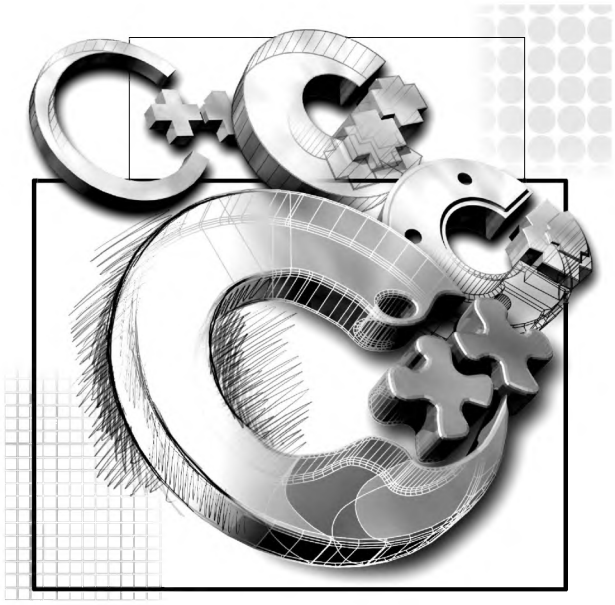


Серия «Для программистов»

Дизайн и эволюция C++

Бьерн Страуструп

ДМК издательство

Москва

ББК 32.973.26-018.1
С80

Страуструп Б.
С80 Дизайн и эволюция C++: Пер. с англ. – М.: ДМК Пресс. – 448 с.: ил.
(Серия «Для программистов»).

ISBN 5-94074-005-7

В книге, написанной создателем языка C++ Бьерном Страуструпом, представлено описание процесса проектирования и разработки языка программирования C++.

Здесь изложены цели, принципы и практические ограничения, наложившие отпечаток на структуру и облик C++, обсужден дизайн недавно добавленных в язык средств: шаблонов, исключений, идентификации типа во время исполнения и пространств имен. Автор анализирует решения, принятые в ходе работы над языком, и демонстрирует, как правильно применять «реальный объектно-ориентированный язык программирования».

Книга удобно организована, поучительна, написана с юмором. Описание ключевых идей даст начинающему пользователю ту основу, на которой позже он выстроит свое понимание всех деталей языка. Опытный программист найдет здесь обсуждение принципиальных вопросов проектирования, что позволит ему лучше понять язык, с которым он работает.

ББК 32.973.26-018.1

Права на издание книги были получены по соглашению с Addison Wesley Longman, Inc. и Литературным агенством Мэтлок (Санкт-Петербург).

ISBN 0-201-54330-3 (англ.)
ISBN 5-94074-005-7 (рус.)

Содержание

Предисловие 13
Благодарности 15
Обращение к читателю 16

Часть I 29

Глава 1. Предыстория C++ 30
1.1. Язык Simula и распределенные системы 30
1.2. Язык C и системное программирование 33
1.3. Немного об авторе книги 33

Глава 2. Язык C with Classes 36
2.1. Рождение C with Classes 36
2.2. Обзор языковых возможностей 38
2.3. Классы 39
2.4. Эффективность исполнения 41
2.4.1. Встраивание 42
2.5. Модель компоновки 43
2.5.1. Простые реализации 46
2.5.2. Модель размещения объекта в памяти 48
2.6. Статический контроль типов 49
2.6.1. Сужающие преобразования 50
2.6.2. О пользе предупреждений 51
2.7. Почему C? 52
2.8 Проблемы синтаксиса 54
2.8.1. Синтаксис объявлений в языке C 54
2.8.2. Тэги структур и имена типов 56
2.8.3. Важность синтаксиса 58
2.9. Производные классы 58
2.9.1. Полиморфизм без виртуальных функций 59
2.9.2. Контейнерные классы без шаблонов 60
2.9.3. Модель размещения объекта в памяти 61
2.9.4. Ретроспектива 62
2.10. Модель защиты 62

2.11. Гарантии времени исполнения ... 65
2.11.1. Конструкторы и деструкторы ... 65
2.11.2. Распределение памяти и конструкторы ... 66
2.11.3. Функции call и return ... 67
2.12. Менее существенные средства ... 67
2.12.1. Перегрузка оператора присваивания ... 67
2.12.2. Аргументы по умолчанию ... 68
2.13. Что не реализовано в C with Classes ... 69
2.14. Рабочая обстановка ... 70

Глава 3. Рождение C++ ... 73
3.1. От C with Classes к C++ ... 73
3.2. Цели C++ ... 74
3.3. Компилятор Cfront ... 76
3.3.1. Генерирование C-кода ... 77
3.3.2. Синтаксический анализ C++ ... 79
3.3.3. Проблемы компоновки ... 80
3.3.4. Версии Cfront ... 80
3.4. Возможности языка ... 82
3.5. Виртуальные функции ... 82
3.5.1. Модель размещения объекта в памяти ... 85
3.5.2. Замещение и поиск подходящей виртуальной функции ... 87
3.5.3. Сокрытие членов базового класса ... 87
3.6. Перегрузка ... 88
3.6.1. Основы перегрузки ... 89
3.6.2. Функции-члены и дружественные функции ... 91
3.6.3. Операторные функции ... 93
3.6.4. Перегрузка и эффективность ... 94
3.6.5. Изменение языка и новые операторы ... 96
3.7. Ссылки ... 96
3.7.1. Lvalue и Rvalue ... 98
3.8. Константы ... 99
3.9. Управление памятью ... 101
3.10. Контроль типов ... 103
3.11. Второстепенные возможности ... 104
3.11.1. Комментарии ... 104
3.11.2. Нотация для конструкторов ... 104
3.11.3. Квалификация ... 105
3.11.4. Инициализация глобальных объектов ... 106
3.11.5. Предложения объявления ... 109
3.12. Языки C и C++ ... 111
3.13. Инструменты для проектирования языка ... 114

3.14. Книга «Язык программирования C++» ... 116
3.15. Статья «Whatis?» ... 117

Глава 4. Правила проектирования языка C++ ... 120
4.1. Правила и принципы ... 120
4.2. Общие правила ... 121
4.3. Правила поддержки проектирования ... 125
4.4. Технические правила ... 128
4.5. Правила поддержки низкоуровневого программирования ... 132
4.6. Заключительное слово ... 134

Глава 5. Хронология 1985–1993 гг. ... 135
5.1. Введение ... 135
5.2. Версия 2.0 ... 136
5.2.1. Обзор возможностей ... 137
5.3. Аннотированное справочное руководство ... 138
5.3.1. Обзор ARM ... 139
5.4. Стандартизация ANSI и ISO ... 140
5.4.1. Обзор возможностей ... 143

Глава 6. Стандартизация ... 144
6.1. Что такое стандарт? ... 144
6.1.1. Детали реализации ... 145
6.1.2. Тест на реалистичность ... 146
6.2. Работа комитета ... 146
6.2.1. Кто работает в комитете ... 148
6.3. Как велась работа ... 148
6.3.1. Разрешение имен ... 149
6.3.2. Время жизни объектов ... 153
6.4. Расширения ... 157
6.4.1. Критерии рассмотрения предложений ... 159
6.4.2. Текущее состояние дел ... 161
6.4.3. Проблемы, связанные с полезными расширениями ... 162
6.4.4. Логическая непротиворечивость ... 163
6.5. Примеры предлагавшихся расширений ... 164
6.5.1. Именованные аргументы ... 164
6.5.2. Ограниченные указатели ... 168
6.5.3. Наборы символов ... 169

Глава 7. Заинтересованность и использование ... 174
7.1. Рост интереса к C++ ... 174
7.1.1. Отсутствие маркетинга C++ ... 175
7.1.2. Конференции ... 175

7.1.3. Журналы и книги ... 176
7.1.4. Компиляторы ... 177
7.1.5. Инструментальные средства и среды программирования ... 177
7.2. Преподавание и изучение C++ ... 178
7.3. Пользователи и приложения ... 183
7.3.1. Первые пользователи ... 183
7.3.2. Сферы применения C++ ... 184
7.4. Коммерческая конкуренция ... 184
7.4.1. Традиционные языки ... 185
7.4.2. Современные языки ... 186
7.4.3. Как выдержать конкуренцию ... 187

Глава 8. Библиотеки ... 189
8.1. Введение ... 189
8.2. Проектирование библиотеки C++ ... 189
8.2.1. Альтернативы при проектировании библиотеки ... 190
8.2.2. Языковые средства и построение библиотеки ... 190
8.2.3. Как работать с разнообразными библиотеками ... 191
8.3. Ранние библиотеки ... 192
8.3.1. Библиотека потокового ввода/вывода ... 193
8.3.2. Поддержка параллельности ... 196
8.4. Другие библиотеки ... 198
8.4.1. Базовые библиотеки ... 199
8.4.2. Устойчивость и базы данных ... 200
8.4.3. Библиотеки для численных расчетов ... 200
8.4.4. Специализированные библиотеки ... 201
8.5. Стандартная библиотека ... 201

Глава 9. Перспективы развития языка C++ ... 203
9.1. Введение ... 203
9.2. Оценка пройденного пути ... 203
9.2.1. Достигнуты ли основные цели C++? ... 204
9.2.2. Является ли C++ логически последовательным языком? ... 204
9.2.3. Основная недоработка языка ... 207
9.3. Всего лишь мост? ... 208
9.3.1. Мост нужен надолго ... 208
9.3.2. Если C++ – это ответ, то на какой вопрос? ... 209
9.4. Что может сделать C++ более эффективным ... 213
9.4.1. Стабильность и стандарты ... 213
9.4.2. Обучение и приемы ... 213
9.4.3. Системные вопросы ... 213
9.4.4. За пределами файлов и синтаксиса ... 214
9.4.5. Подведение итогов и перспективы ... 215

Часть II ... 217

Глава 10. Управление памятью ... 218
10.1. Введение ... 218
10.2. Отделение распределения памяти и инициализации ... 219
10.3. Выделение памяти для массива ... 220
10.4. Размещение объекта в памяти ... 221
10.5. Проблемы освобождения памяти ... 222
10.5.1. Освобождение памяти для массивов ... 224
10.6. Нехватка памяти ... 225
10.7. Автоматическая сборка мусора ... 226
10.7.1. Необязательный сборщик мусора ... 226
10.7.2. Как должен выглядеть необязательный сборщик мусора? ... 228

Глава 11. Перегрузка ... 230
11.1. Введение ... 230
11.2. Разрешение перегрузки ... 230
11.2.1. Детальное разрешение ... 231
11.2.2. Управление неоднозначностью ... 233
11.2.3. Нулевой указатель ... 236
11.2.4. Ключевое слово overload ... 238
11.3. Типобезопасная компоновка ... 239
11.3.1. Перегрузка и компоновка ... 239
11.3.2. Реализация компоновки в C++ ... 240
11.3.3. Анализ пройденного пути ... 241
11.4. Создание и копирование объектов ... 244
11.4.1. Контроль допустимости копирования ... 244
11.4.2. Управление распределением памяти ... 244
11.4.3. Управление наследованием ... 245
11.4.4. Почленное копирование ... 246
11.5. Удобство нотации ... 248
11.5.1. «Умные» указатели ... 248
11.5.2. «Умные» ссылки ... 249
11.5.3. Перегрузка операторов инкремента и декремента ... 252
11.5.4. Перегрузка ->* ... 254
11.5.5. Перегрузка оператора «запятая» ... 254
11.6. Добавление в C++ операторов ... 254
11.6.1. Оператор возведения в степень ... 254
11.6.2. Операторы, определяемые пользователем ... 257
11.6.3. Составные операторы ... 258
11.7. Перечисления ... 259
11.7.1 Перегрузка на базе перечислений ... 261
11.7.2. Тип Boolean ... 261

Глава 12. Множественное наследование ... 263
12.1. Введение ... 263
12.2. Базовые классы ... 264
12.3. Виртуальные базовые классы ... 265
12.3.1. Виртуальные базовые классы и виртуальные функции ... 267
12.4. Модель размещения объекта в памяти ... 270
12.4.1. Размещение в памяти объекта виртуального базового класса ... 272
12.4.2. Виртуальные базовые классы и приведение типов ... 273
12.5. Комбинирование методов ... 274
12.6. Полемика о множественном наследовании ... 276
12.7. Делегирование ... 279
12.8. Переименование ... 280
12.9. Инициализаторы членов и базовых классов ... 282

Глава 13. Уточнения понятия класса ... 284
13.1 Введение ... 284
13.2. Абстрактные классы ... 284
13.2.1. Абстрактные классы и обработка ошибок ... 284
13.2.2. Абстрактные типы ... 286
13.2.3. Синтаксис ... 288
13.2.4. Виртуальные функции и конструкторы ... 288
13.3. Константные функции-члены ... 291
13.3.1. Игнорирование const при приведении типов ... 291
13.3.2. Уточнение определения const ... 292
13.3.3. Ключевое слово mutable и приведение типов ... 293
13.4. Статические функции-члены ... 294
13.5. Вложенные классы ... 295
13.6. Ключевое слово inherited ... 297
13.7. Ослабление правил замещения ... 299
13.7.1. Ослабление правил аргументов ... 301
13.8. Мультиметоды ... 303
13.8.1. Когда нет мультиметодов ... 305
13.9. Защищенные члены ... 307
13.10. Улучшенная генерация кода ... 308
13.11. Указатели на функции-члены ... 309

Глава 14. Приведение типов ... 311
14.1. Крупные расширения ... 311
14.2. Идентификация типа во время исполнения ... 312
14.2.1. Зачем нужен механизм RTTI ... 313
14.2.2. Оператор dynamic_cast ... 313
14.2.3. Правильное и неправильное использование RTTI ... 319

14.2.4. Зачем давать «опасные средства» ... 321
14.2.5. Оператор typeid() ... 322
14.2.6. Модель размещения объекта в памяти ... 326
14.2.7. Простой ввод/вывод объектов ... 327
14.2.8. Другие варианты ... 329
14.3. Новая нотация для приведения типов ... 333
14.3.1. Недостатки старых приведений типов ... 334
14.3.2. Оператор static_cast ... 335
14.3.3. Оператор reinterpret_cast ... 337
14.3.4. Оператор const_cast ... 339
14.3.5. Преимущества новых приведений типов ... 340

Глава 15. Шаблоны ... 343
15.1. Введение ... 343
15.2. Зачем нужны шаблоны ... 344
15.3. Шаблоны классов ... 346
15.3.1. Аргументы шаблонов, не являющиеся типами ... 347
15.4. Ограничения на аргументы шаблонов ... 348
15.4.1. Ограничения за счет наследования ... 349
15.4.2. Ограничения за счет использования ... 350
15.5. Устранение дублирования кода ... 351
15.6. Шаблоны функций ... 353
15.6.1. Выведение аргументов шаблона функции ... 354
15.6.2. Задание аргументов шаблона функции ... 355
15.6.3. Перегрузка шаблона функции ... 357
15.7. Синтаксис ... 360
15.8. Методы композиции ... 361
15.8.1. Представление стратегии реализации ... 362
15.8.2. Представление отношений порядка ... 363
15.9. Соотношения между шаблонами классов ... 365
15.9.1. Отношения наследования ... 365
15.9.2. Преобразования ... 367
15.9.3. Шаблоны-члены ... 368
15.10. Инстанцирование шаблонов ... 369
15.10.1. Явное инстанцирование ... 371
15.10.2. Точка инстанцирования ... 372
15.10.3. Специализация ... 378
15.10.4. Нахождение определений шаблонов ... 381
15.11. Последствия введения шаблонов ... 383
15.11.1. Отделение реализации от интерфейса ... 384
15.11.2. Гибкость и эффективность ... 384
15.11.3. Влияние на другие компоненты C++ ... 385

Глава 16. Обработка исключений ... 387
16.1. Введение ... 387
16.2. Цели и предположения ... 388
16.3. Синтаксис ... 389
16.4. Группировка ... 390
16.5. Управление ресурсами ... 391
16.5.1. Ошибки в конструкторах ... 393
16.6. Возобновление или завершение? ... 394
16.6.1. Обходные пути для реализации возобновления ... 397
16.7. Асинхронные события ... 398
16.8. Распространение на несколько уровней ... 399
16.9. Статическая проверка ... 399
16.9.1. Вопросы реализации ... 401
16.10. Инварианты ... 402

Глава 17. Пространства имен ... 403
17.1. Введение ... 403
17.2. Для чего нужны пространства имен ... 404
17.2.1. Обходные пути ... 404
17.3. Какое решение было бы лучшим? ... 406
17.4. Решение: пространства имен ... 408
17.4.1. Мнения по поводу пространств имен ... 410
17.4.2. Внедрение пространств имен ... 411
17.4.3. Псевдонимы пространства имен ... 412
17.4.4. Использование пространств имен для управления версиями ... 413
17.4.5. Технические детали ... 415
17.5. Классы и пространства имен ... 421
17.5.1. Производные классы ... 421
17.5.2. Использование базовых классов ... 423
17.5.3. Исключение глобальных статических объявлений ... 424
17.6. Совместимость с C ... 425

Глава 18. Препроцессор C ... 427

Алфавитный указатель ... 431

Предисловие

Кто не пашет, должен писать.

Мартин А. Хансен

На второй конференции ACM по истории языков программирования (HOPL-2) меня попросили написать статью по истории C++. Мысль показалась мне разумной, предложение – лестным, поэтому я приступил к работе. Чтобы работа получилась более полной и неискаженной, я обратился за помощью к друзьям. Вслед за этим молва разнесла известие о готовящемся проекте. Не обошлось без преувеличений – и однажды я получил по электронной почте письмо с вопросом, где можно купить мою новую книгу о дизайне C++. Оно и положило начало этой серьезной работе.

Обычно в книгах по программированию и его языкам объясняется, чту представляет собой язык и как им пользоваться. Но многих интересует вопрос, почему язык оказался именно таким и как он создавался. Именно об этом, применительно к C++, и говорится в данной книге. В ней рассказывается, как шла эволюция языка от первоначального проекта до нынешнего состояния, объясняются основные задачи и цели проектирования, заложенные в языке идеи и ограничения, описывается развитие языковых концепций.

Естественно, ни C++, ни представления о его дизайне и применении не изменялись сами собой. В действительности эволюционировали сознание пользователей и их представления о том, какие перед ними стоят цели и какие инструменты необходимы для решения. А значит, в книге говорится о наиболее важных вопросах, которые снимались с помощью C++, а также о людях, решавших эти задачи и оказавших влияние на развитие языка, и их воззрениях.

C++ – все еще сравнительно молодой язык. Некоторые из обсуждаемых в книге проблем пока неизвестны широкой публике. Для полного осознания всех последствий описываемых решений потребуется еще немало лет. В этой книге представлена только моя точка зрения на то, почему появился C++, что он собой представляет и каким ему следовало бы быть. Надеюсь, что данная работа будет способствовать дальнейшей эволюции этого языка и поможет людям понять, как лучше всего использовать его.

Основное внимание в книге уделено общим целям дизайна, практическим ограничениям. Ключевые проектные решения, относящиеся к языковым средствам, излагаются в историческом контексте. Я прослеживаю эволюцию языка от C with Classes (C с классами) к версиям 1.0 и 2.0 и далее к проводимой в настоящее время комитетом ANSI/ISO работе по стандартизации. В книге анализируются и такие аспекты: резкое увеличение количества приложений по C++, возросший

интерес пользователей к языку, усиление коммерческой активности, появление большого количества компиляторов, инструментальных средств, сред программирования и библиотек. Подробно обсуждаются связи между языками C++ и Simula. Взаимодействие C++ с другими языками затрагивается лишь мимоходом. Дизайн основных языковых средств, к которым можно отнести классы, наследование, абстрактные классы, перегрузку, управление памятью, шаблоны, обработку исключений, идентификацию типов во время исполнения и пространства имен, рассматривается довольно подробно.

Основная цель книги – дать программистам на C++ правильное представление о фундаментальных концепциях языка и побудить их к экспериментам с теми его возможностями, о которых они не подозревали. Книга будет полезна как опытным программистам, так и студентам и, возможно, поможет решить, стоит ли тратить время на изучение C++.

Благодарности

Я выражаю глубокую признательность Стиву Клэмиджу (Steve Clamage), Тони Хансену (Tony Hansen), Лорейн Джуль (Lorraine Juhl), Питеру Джулю (Peter Juhl), Брайану Кернигану (Brian Kernighan), Ли Найту (Lee Knight), Дугу Леа (Doug Lea), Дугу Макилрою (Doug MacIlroy), Барбаре Му (Barbara Moo), Йенсу Палсбергу (Jens Palsberg), Стиву Рамсби (Steve Rumsby) и Кристоферу Скелли (Christopher Skelly) за то, что они прочли все черновые варианты этой книги. Их конструктивные замечания привели к кардинальным изменениям в содержании и организации работы. Стив Бурофф (Steve Buroff), Мартин Кэролл (Martin Caroll), Шон Корфилд (Sean Corfield), Том Хагельскяер (Tom Hagelskjaer), Рик Холлинбек (Rick Hollinbeck), Деннис Манкль (Dennis Mancl) и Стэн Липпман (Stan Lippman) высказали замечания по отдельным главам. Отдельное спасибо Арчи Лахнеру (Archie Lachner), поинтересовавшемуся, где достать эту книгу еще до того, как я всерьез задумался о ее написании.

Естественно, я признателен всем тем, кто участвовал в создании языка C++. В некотором смысле моя книга – дань уважения этим людям; имена некоторых из них упомянуты в разных главах и в предметном указателе. Впрочем, если отмечать конкретных людей, то более всех помогали мне и поддерживали меня Брайан Керниган (Brian Kernighan), Эндрю Кениг (Andrew Koenig), Дуг Макилрой (Doug MacIlroy) и Джонатан Шопиро (Jonathan Shopiro). Они также щедро делились своими идеями на протяжении более десяти лет. Спасибо Кристену Найгаарду (Kristen Nygaard) и Деннису Ричи (Dennis Ritchie) – разработчикам соответственно Simula и C, откуда я позаимствовал многие ключевые компоненты. Со временем я оценил Кристена и Денниса не только как блестящих разработчиков языков, но и как очень порядочных и симпатичных людей.

Бьерн Страуструп

Обращение к читателю

**Писательство – это единственное искусство,
которым нужно овладевать посредством писания.**

Анонимный автор

Введение

C++ проектировался с целью обеспечить средства организации программ, присущие языку Simula, а также необходимую для системного программирования эффективность и гибкость, свойственные C. Предполагалось, что от замысла до его первой реализации пройдет примерно полгода. Так оно и вышло.

Тогда – в середине 1979 г. – я еще не осознавал, насколько эта цель была скромной и в то же время абсурдной. Скромной – потому что не предполагалось вводить какие бы то ни было новшества, абсурдной – из-за слишком жестких временных рамок и драконовских требований к эффективности и гибкости языка. Новшества со временем все же появились, но в вопросах эффективности и гибкости ни на какие компромиссы я не пошел. С годами цели C++ уточнялись, видоизменялись и формулировались более четко, но и сегодня язык в точности отражает первоначально поставленные цели.

Назначение книги, которую вы держите в руках, – документировать эти цели, проследить их эволюцию и представить на суд читателей тот C++, который появился в результате. Я старался уделять равное внимание историческим фактам (именам, местам и событиям) и техническим вопросам дизайна, реализации и применения языка. Я стремился фиксировать не каждое событие, а лишь ключевые идеи и направления развития, которые уже повлияли на определение C++ и, возможно, еще скажутся на его дальнейшей эволюции и практике применения.

Если я упоминаю некоторое событие, то стараюсь не выдавать желаемое за действительное. Там, где это уместно, приводятся цитаты из различных статей, иллюстрирующие, как разные цели, принципы или свойства языка представлялись в то время. Я не пытаюсь давать оценку прошлому с позиций сегодняшнего дня. Воспоминания о событиях и замечания по поводу последствий принятых тогда решений отделены от основного текста и помечены. Я вообще питаю отвращение к историческому ревизионизму и стараюсь этого избегать. Приведу в пример свое старое высказывание: «Я пришел к выводу, что система типов в языке Pascal не просто бесполезна– это смирительная рубашка, которая создает проблем больше, чем решает, заставляя меня жертвовать чистотой дизайна ради удовлетворения причуд компилятора». Так я думал в то время, и данное мнение существенно повлияло на эволюцию C++. Было ли это резкое осуждение Pascal справедливым и стал бы я сегодня

(по прошествии более десяти лет) судить точно так же, не имеет значения. Я не могу притвориться, что этого не было (чтобы пощадить чувства приверженцев Pascal или избавить себя от чувства неловкости и лишних споров) или как-то пригладить свое высказывание (приведя более полную и сбалансированную точку зрения), не исказив при этом историю C++.

Когда я говорю о людях, которые внесли вклад в дизайн и эволюцию C++, то по возможности уточняю, в чем именно их заслуга и когда это происходило. Но тут есть опасность. Поскольку память моя несовершенна, я вполне мог что-то забыть. Приношу свои извинения. Я называю имена тех, кто способствовал принятию того или иного решения относительно C++. Увы, не всегда это именно тот человек, который впервые столкнулся с определенной проблемой или задумался о ее решении. Досадно, однако вообще отказаться от упоминания имен было бы еще хуже. Не стесняйтесь сообщать мне обо всем, что могло бы внести ясность в этот вопрос.

Когда описываешь исторические события, всегда задумываешься об объективности. Неизбежную предвзятость я старался компенсировать получением информации о событиях, в которых сам не принимал участия, и беседами с другими участниками событий. Еще я попросил нескольких человек, имеющих отношение к эволюции C++, прочесть эту книгу. Их имена приведены в конце предисловия. Ну и, кроме того, статья, представленная на второй конференции по истории языков программирования [Stroustrup, 1993] и содержащая основные исторические факты, упоминаемые в этой книге, была тщательно переработана и освобождена от излишней предвзятости.

Как читать эту книгу

В части I описывается дизайн, эволюция, практика применения и процесс стандартизации C++ в относительной хронологической последовательности. Такая схема выбрана потому, что решения, принимавшиеся в ранние годы, выстраиваются в четкую, логичную временную цепочку. В главах 1, 2 и 3 описываются истоки C++ и его эволюция от C with Classes к версии 1.0. В главе 4 излагаются правила, по которым C++ развивался в течение этого периода и позже. Главы 5, 6 посвящены соответственно хронологии разработки после выхода версии 1.0 и процессу стандартизации C++ под эгидой ANSI/ISO. О перспективе развития языка говорится в 7 и 8 главах, где анализируются приложения, инструментальные средства и библиотеки. И, наконец, в главе 9 представлены ретроспективный взгляд и некоторые общие мысли о перспективах развития C++.

В части II описывается разработка C++ после выхода версии 1.0. Язык развивался в тех направлениях, которые были определены уже ко времени выпуска версии 1.0: добавление желательных свойств (к примеру, шаблонов и обработки исключений) и принципов их дизайна. После появления версии 1.0 хронологический порядок событий был уже не так важен для разработки C++. Определение языка в основных чертах осталось бы таким же, как сейчас, даже если бы последовательность реализации расширений была бы иной. То, в каком порядке решались задачи и как к языку добавлялись новые свойства, представляет лишь исторический интерес. Строго хронологическое изложение помешало бы логически естественному

представлению идей, поэтому в основу построения части II положено описание важнейших свойств языка. Отдельные главы части II не зависят друг от друга, так что читать их можно в любом порядке. Глава 10 посвящена управлению памятью, 11 – перегрузке, 12 – множественному наследованию, 13 – уточнениям концепции класса, 14 – приведению типов, 15 – шаблонам, 16 – обработке исключений, 17 – пространствам имен, 18 – препроцессору C.

Люди ждут от книги по дизайну и эволюции языка программирования разного. Не найдется двух человек, имеющих одинаковое мнение о степени детализации, необходимой при обсуждении некоторой темы. Так, все рецензии на варианты статьи, представленной на конференции HOPL-2, строились по одинаковому шаблону: «Статья слишком длинная... пожалуйста, уделите больше внимания темам x, y и z». И пока одни рецензенты говорили: «Исключите философскую дребедень, пусть будет побольше технических деталей», другие требовали: «Избавьте от этих утомительных деталей, мне интересна философия проектирования».

Чтобы выйти из этого положения, я написал книгу внутри книги. Если вас не интересуют детали, то для начала пропустите все подразделы, имеющие номера x.y.z, где x – номер главы, а y – номер раздела. А из оставшегося читайте то, что вам интереснее. Впрочем, читать можно и последовательно, от первой до последней страницы. Правда, при этом вы рискуете увязнуть в деталях. Я вовсе не хочу сказать, что они несущественны. Напротив, никакой язык программирования нельзя понять, рассматривая только его общие принципы. Без конкретных примеров никак не обойтись. Но, если изучать только детали и не представлять общей картины, можно окончательно запутаться.

Стремясь облегчить участь читателя, я поместил в часть II обсуждение почти всех новых свойств языка, которые принято считать «продвинутыми». Поэтому о базовых конструкциях говорится в I части. Почти вся информация о нетехнических аспектах эволюции C++ сосредоточена здесь же. Те, кто не любит «философских материй», могут пропустить главы с 4 по 9 и сразу перейти к техническим деталям, описанным в части II.

Я предвижу, что кое-кто захочет использовать эту книгу как справочный материал, а многие прочтут лишь отдельные главы, не обращая внимания на то, что им предшествует. Такой подход тоже возможен, поскольку я постарался сделать главы по возможности замкнутыми и, как опытный программист на C++, не поскупился на перекрестные ссылки и составил подробный предметный указатель.

Пожалуйста, имейте в виду, что в книге я не пытаюсь строго определить возможности языка C++. Здесь излагаются лишь подробности, необходимые для описания того, как появилось на свет то или иное средство. Я не ставлю себе целью научить программированию или проектированию с использованием C++. Тех, кому это интересно, отсылаю к работе [2nd].

Хронология C++

Приведенная ниже хронологическая таблица поможет вам лучше представить то, о чем говорится дальше.

Таблица 1

Год	Месяц	Событие
1979	май	Начало работы на C with Classes
	октябрь	Готова первая реализация C with Classes
1980	апрель	Первый внутренний документ по C with Classes в Bell Labs [Stroustrup, 1980]
1982	январь	Первая опубликованная статья по C with Classes [Stroustrup, 1982]
1983	август	Готова первая реализация C++
	декабрь	C++ получил имя
1984	январь	Вышло первое руководство по C++
1985	февраль	Первая распространяемая версия C++ (Release E)
	октябрь	Cfront Release 1.0 (первая коммерческая версия)
	октябрь	Вышла книга *The C++ Programming Language* [Stroustrup, 1986]
1986	август	Статья «Whatis?» [Stroustrup, 1986b]
	сентябрь	Конференция OOPSLA (положившая начало безудержной рекламе объектно-ориентированного программирования, крутящейся вокруг Smalltalk)
	ноябрь	Первая коммерческая версия Cfront для ПК (Cfront 1.1, Glockenspiel)
1987	февраль	Cfront Release 1.2
	ноябрь	Первая конференция USENIX, посвященная C++ (Санта Фе, штат Нью-Мексико)
	декабрь	Первая версия GNU C++ (1.13)
1988	январь	Первая версия Oregon Software C++
	июнь	Первая версия Zortech C++
	октябрь	Первый симпозиум разработчиков C++ в рамках USENIX (Эстес Парк, штат Колорадо)
1989	июнь	Cfront Release 2.0
	декабрь	Организационная встреча комитета ANSI X3J16 (Вашингтон, округ Колумбия)
1990	март	Первое техническое совещание комитета ANSI X3J16 (Сомерсет, штат Нью-Джерси)
	май	Первая версия Borland C++
	май	Вышла книга *The Annotated C++ Reference Manual* [ARM]
	июль	Одобрены шаблоны (Сиэтл, штат Вашингтон)
	ноябрь	Одобрен механизм обработки исключений (Пало Альто, штат Калифорния)
1991	июнь	Вышло второе издание The C++ Programming Language [2nd]
	июнь	Первая встреча рабочей группы ISO WG21 (Лунд, Швеция)
	октябрь	Cfront Release 3.0 (включающая шаблоны)
1992	февраль	Первая версия DEC C++ (включающая шаблоны и обработку исключений)
	март	Первая версия Microsoft C++
	май	Первая версия IBM C++ (включающая шаблоны и обработку исключений)
1993	март	Одобрена идентификация типов во время исполнения (Портленд, штат Орегон)
	июль	Одобрены пространства имен (Мюнхен, Германия)
1994	август	В комитете ANSI/ISO зарегистрирован предварительный стандарт

Пользователь превыше всего

Эта книга написана для пользователей C++, то есть для программистов и проектировщиков. Я старался избегать запутанных и понятных лишь немногим экспертам предметов, стремясь увидеть C++, его средства и эволюцию глазами пользователя. Узкоспециальные аспекты языка обсуждаются только тогда, когда связаны с вопросами, непосредственно касающимися пользователей. В качестве примеров можно привести рассмотрение правил разрешения имен в шаблонах (см. раздел 15.10) и периода существования временных объектов (см. пункт 6.3.2).

Специалисты по языкам программирования, языковые пуристы и разработчики найдут в этой книге немало поводов для критики, но я стремился дать общую картину, а не точные и исчерпывающие разъяснения по поводу каждой мелочи. Если вам нужны технические детали, обратитесь к книге *The Annotated C++ Reference Manual* [ARM], где дано определение языка, или к книге *Язык программирования C++* (второе издание) [2nd][1], или к рабочим материалам комитета ANSI/ISO по стандартизации.

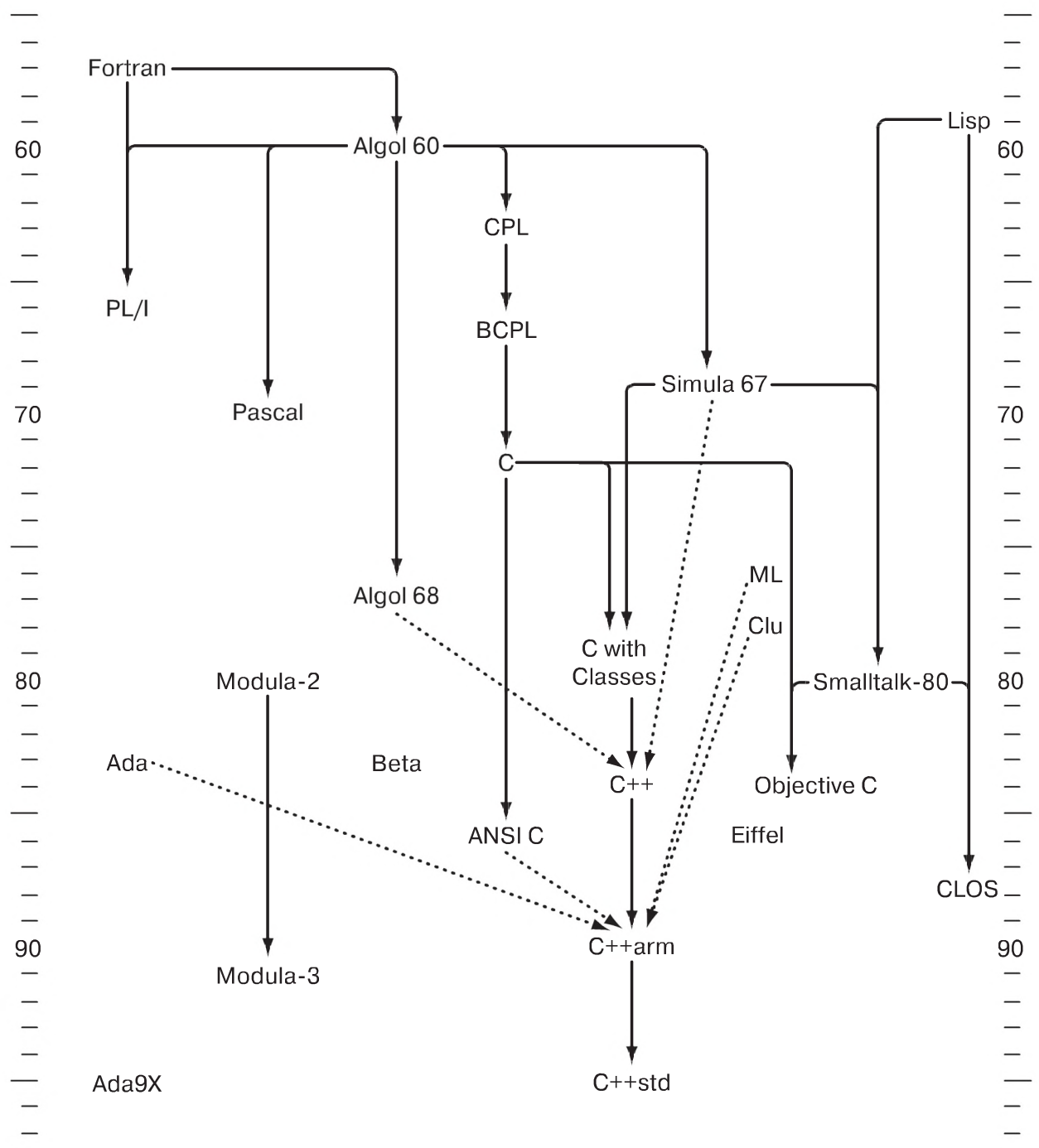
Языки программирования

Несколько рецензентов просили меня сравнить C++ с другими языками. Но я решил этого не делать. Еще раз выскажу свою неизменную точку зрения: сравнение языков редко бывает осмысленным, а еще реже честным. Для хорошего сравнительного анализа основных языков программирования требуется больше времени, чем можно себе позволить, нужно иметь опыт работы во многих прикладных областях, быть беспристрастным и объективным. Времени у меня нет, и вряд ли я, как создатель C++, могу рассчитывать, что многие поверят в мою беспристрастность по этому поводу.

Еще меня беспокоит некий феномен, который я неоднократно наблюдал, когда другие пытались заняться честным сравнением языков. Авторы изо всех сил стараются быть объективными, но обязательно делают упор на каком-то одном виде приложений, одном стиле или определенной культуре программирования. А уж если один язык более известен, чем другой, то в оценке чуть ли изначально возникает некая предубежденность: дефекты хорошо известного языка признаются несущественными, для них предлагаются простые способы обхода, тогда как аналогичные изъяны в других языках объявляются фундаментальными проблемами. Но ведь часто бывает, что в менее популярном языке давно известны методы решения подобных задач, только автор сравнительного обзора о них не знает или считает неудовлетворительными, поскольку в знакомом ему языке они работать не будут.

Кроме того, самая свежая информация о хорошо известном языке доступна. Говоря о языке менее популярном, авторы подчас вынуждены полагаться на устаревшие сведения. Если языки вообще уместно сопоставлять, то сравнивать язык X на основе определения трехлетней давности с языком Y в том виде, в каком он

[1] В настоящее время уже вышел русский перевод третьего издания этой книги (Б. Страуструп «Язык программирования C++», издательство «Бином», 1999). – *Прим. перев.*



существует в данный момент, нечестно и бессмысленно. Поэтому, как уже было сказано выше, в отношении отличных от C++ языков я ограничусь общими высказываниями и небольшими замечаниями на узкие темы.

Чтобы прояснить историческое место C++, на рисунке ниже показано, когда появились на свет другие языки программирования, часто упоминаемые в связи с данным.

Данная диаграмма ни в коей мере не претендует на полноту и отражает лишь важнейшие влияния на C++. В частности, воздействие концепции классов из Simula на диаграмме отражено явно неполно: на языки Ada [Ichbiah, 1979] и Clu [Liskov, 1979] Simula повлияла не очень сильно, а на Ada9X [Taft, 1992], Beta [Madsen, 1993], Eiffel [Meyer, 1988] и Modula-3 [Nelson, 1991] – весьма заметно. Диаграмма не отражает влияние C++ на другие языки. Сплошными линиями обозначено воздействие на структуру языка, пунктирными – на его специфические

средства. Под датой появления языка в большинстве случаев понимается дата первой версии. Например, Algol68 [Woodward, 1974] датирован 1977 г., а не 1968 г.

Анализируя отзывы на статью, представленную на конференции HOPL-2, и многие другие источники, я пришел к выводу: нет согласия по поводу того, что же такое язык программирования и какой главной цели он должен служить. Как его определить? Это инструмент для составления компьютерных заданий? Средство для общения программистов? Способ представить высокоуровневые проекты? Система обозначений для записи алгоритмов? Возможность выразить отношения между концепциями? Средство для проведения экспериментов? Механизм управления компьютеризованными устройствами? По-моему, язык программирования общего назначения должен отвечать всем вышеперечисленным критериям и удовлетворять потребности самых разных групп пользователей. Единственное, чем язык быть не может, – это простым набором «изящных штучек».

Разнобой мнений также отражает существующие точки зрения на то, что такое информатика и как следует проектировать языки. Является ли информатика частью математики или это, скорее, отрасль инженерного проектирования? Или архитектуры? Или искусства? Или биологии? Или социологии? Или философии? А может, информатика заимствует методы и подходы, применяемые во всех этих дисциплинах? Я склоняюсь к последнему предположению.

Отсюда следует, что проектирование языка отличается от более строгих и формальных наук вроде математики и философии. Чтобы стать полезным, универсальный язык программирования должен быть эклектичным и учитывать множество прагматических и социальных факторов. В частности, каждый язык проектируется для решения определенного вида задач в определенное время в соответствии с представлениями определенной группы людей. С появлением новых требований исходный дизайн начинает расширяться, чтобы соответствовать современному пониманию задач. Эта точка зрения прагматична, но не беспринципна. Я твердо убежден, что все успешные языки программирования именно развивались, а не были просто спроектированы на базе первоначально заложенных принципов. Принципы лежат в основе исходного дизайна и определяют последующую эволюцию языка, однако и сами они не остаются застывшими.

Библиография

В этом разделе перечислены работы, на которые даются ссылки во всех главах книги.

Таблица 2

[2nd]	see [Stroustrup,1991].
[Agha,1986]	Gul Agha: *An Overview of Actor languages*. ACM SIGPLAN Notices. October 1986.
[Aho,1986]	Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA. 1986. ISBN 0-201-10088-6.
[ARM]	see [Ellis.1990].
[Babcisky,1984]	Karel Babcisky: *Simula Performance Assessment*. Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience and Assessment. Canterbury, Kent, UK. September 1984.

Таблица 2 (продолжение)

[Barton, 1994]	John J. Barton and Lee R. Nackman: *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples.* Addison-Wesley, Reading, MA. 1994. ISBN 0-201-53393-6.
[Birtwistle,1979]	Graham Birtwistle, Ole-Johan Dahl, Bjцrn Myrhaug, and Kristen Nygaard: *SIMULA BEGIN.* Studentlitteratur, Lund, Sweden. 1979. ISBN 91-44-06212-5.
[Boehm,1993]	Hans-J. Boehm: *Space Efficient Conservative Garbage Collection.* Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation. ACM SIGPLAN Notices. June 1993.
[Booch,1990]	Grady Booch and Michael M. Vilot: *The Design of the C++ Booch Components.* Proc. OOPSLA'90. October 1990.
[Booch,1991]	Grady Booch: *Object-Oriented Design.* Benjamin Cummings, Redwood City, CA. 1991. ISBN 0-8053-0091-0.
[Booch,1993]	Grady Booch: *Object-Oriented Analysis and Design with Applications, 2nd edition.* Benjamin Cummings, Redwood City, CA. 1993. ISBN 0-8053-5340-2.
[Booch,1993b]	Grady Booch and Michael M. Vilot: *Simplifying the C++ Booch Components.* The C++Report. June 1993.
[Budge, 1992]	Ken Budge, J.S. Perry, and A.C. Robinson: *High-Performance Scientific Computation using C++.* Proc. USENIX C++ Conference. Portland, OR. August 1992.
[Buhr,1992]	Peter A. Buhr and Glen Ditchfield: *Adding Concurrency to a Programming Language.* Proc. USENIX C++ Conference. Portland, OR. August 1992.
[Call, 1987]	Lisa A. Call, et al.: *CLAM - An Open System for Graphical User Interfaces.* Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
[Cameron,1992]	Don Cameron, et al.: A *Portable Implementation of C++ Exception Handling.* Proc. USENIX C++ Conference. Portland, OR. August 1992.
[Campbell,1987]	Roy Campbell, et al.: *The Design of a Multiprocessor Operating System.* Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
[Cattell,1991]	Rich G.G. Cattell: *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, Reading, MA. 1991. ISBN 0-201-53092-9.
[Cargill,1991]	Tom A. Cargill: *The Case Against Multiple Inheritance in C++.* USENIX Computer Systems. Vol4, no 1, 1991.
[Carroll,1991]	Martin Carroll: *Using Multiple Inheritance to Implement Abstract Data Types.* The C++Report. April 1991.
[Carroll,1993]	Martin Carroll: *Design of the USL Standard Components.* The C++ Report. June 1993.
[Chandy,1993]	K. Mani Chandy and Carl Kesselman: *Compositional C++: Compositional Parallel Programming.* Proc. Fourth Workshop on Parallel Computing and Compilers. Springer-Verlag. 1993.
[Cristian,1989]	Flaviu Cristian: *Exception Handling.* Dependability of Resilient Computers, T. Andersen, editor. BSP Professional Books, Blackwell Scientific Publications, 1989.
[Cox, 1986]	Brad Cox: *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, MA. 1986.
[Dahl,1988]	Ole-Johan Dahl: Personal communication.
[Dearle, 1990]	Fergal Dearle: *Designing Portable Applications Frameworks for C++.* Proc. USENIX C++Conference. San Francisco, CA. April 1990.

Таблица 2 (продолжение)

[Dorward,1990]	Sean M. Dorward, et al.: *Adding New Code to a Running Program.* Proc. USENIX C++ Conference. San Francisco, CA. April 1990.
[Eick,1991]	Stephen G. Eick: *SIMLIB - An Object-Oriented C++ Library for Interactive Simulation of Circuit-Switched Networks.* Proc. Simulation Technology Conference. Orlando, FL. October 1991.
[Ellis,1990]	Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA. 1990. ISBN 0-201-51459-1.
[Faust, 1990]	John E. Faust and Henry M. Levy: *The Performance of an Object-Oriented Threads Package.* Proc. ACM joint ECOOP and OOPSLA Conference. Ottawa, Canada. October 1990.
[Fontana,1991]	Mary Fontana and Martin Neath: *Checked Out and Long Overdue: Experiences in the Design of a C++ Class Library.* Proc. USENIX C++ Conference. Washington, DC. April 1991.
[Forslund,1990]	David W. Forslund, et al.: *Experiences in Writing Distributed Particle Simulation Code in C++.* Proc. USENIX C++ Conference. San Francisco, CA. April 1990.
[Gautron,1992]	Philippe Gautron: *An Assertion Mechanism based on Exceptions.* Proc. USENIX C++ Conference. Portland, OR. August 1992.
[Gehani,1988]	Narain H. Gehani and William D. Roome: *Concurrent C++: Concurrent Programming With Class(es).* Software—Practice & Experience. Vol 18, no 12, 1988.
[Goldberg,1983]	Adele Goldberg and David Robson: *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, Reading, MA. 1983. ISBN 0-201-11371-6.
[Goodenough,1975]	John Goodenough: *Exception Handling: Issues and a Proposed Notation.* Communications of the ACM. December 1975.
[Gorlen,1987]	Keith E. Gorlen: *An Object-Oriented Class Library for C++ Programs.* Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
[Gorlen,1990]	Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico: *Data Abstraction and Object-Oriented Programming in* C++. Wiley. West Sussex. England. 1990. ISBN 0-471-92346-X.
[Ньbel,1992]	Peter Ньbel and J.T. Thorsen: *An Implementation of a Persistent Store for* C++. Computer Science Department. Aarhus University, Denmark. December 1992.
[Ichbiah,1979]	Jean D. Ichbiah, et al.: *Rationale for the Design of the ADA Programming Language.* SIGPLAN Notices Vol 14, no 6, June 1979 Part B.
[Ingalls,1986]	Daniel H.H. Ingalls: *A Simple Technique for Handling Multiple Polymorphism.* Proc. ACM OOPSLA Conference. Portland, OR. November 1986.
[Interrante,1990]	John A. Interrante and Mark A. Linton: *Runtime Access to Type Information.* Proc. USENIX C++ Conference. San Francisco 1990.
[Johnson, 1992]	Steve C. Johnson: Personal communication.
[Johnson,1989]	Ralph E. Johnson: *The Importance of Being Abstract.* The C++ Report. March 1989.
[Keffer,1992]	Thomas Keffer: *Why* C++ *Will Replace Fortran.* C++ Supplement to Dr. Dobbs Journal. December 1992.
[Keffer,1993]	Thomas Keffer: *The Design and Architecture of Tools. h++.* The C++ Report. June 1993.

Таблица 2 (продолжение)

[Kernighan,1976]	Brian Kernighan and P.J. Plauger: *Software Tools.* Addison-Wesley, Reading, MA. 1976. ISBN 0-201-03669.
[Kernighan,1978]	Brian Kernighan and Dennis Ritchie: *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ. 1978. ISBN 0-13-110163-3.
[Kernighan,1981]	Brian Kernighan: *Why Pascal is not my Favorite Programming Language.* AT&T Bell Labs Computer Science Technical Report No 100. July 1981.
[Kernighan,1984]	Brian Kernighan and Rob Pike: *The UNIX Programming Environment.* Prentice-Hall, Englewood Cliffs, NJ. 1984. ISBN 0-13-937699-2.
[Kernighan,1988]	Brian Kernighan and Dennis Ritchie: *The C Programming Language (second edition).* Prentice-Hall, Englewood Cliffs, NJ. 1988. ISBN 0-13-110362-8.
[Kiczales,1992]	Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow: *The Art of the Metaobject Protocol.* The MIT Press. Cambridge, Massachusetts. 1991. ISBN 0-262-11158-6.
[Koenig,1988]	Andrew Koenig: *Associative arrays in* C++. Proc. USENIX Conference. San Francisco, CA. June 1988.
[Koenig,1989]	Andrew Koenig and Bjarne Stroustrup: C++; *As close to C as possible - but no closer.* The C++ Report. July 1989.
[Koenig,1989b]	Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++. Proc. «C++at Work» Conference. November 1989.*
[Koenig,1990]	Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++ (revised).* Proc. USENIX C++ Conference. San Francisco, CA. April 1990. Also, Journal of Object-Oriented Programming. July 1990.
[Koenig, 1991]	Andrew Koenig: *Applicators, Manipulators, and Function Objects.* C++Journal, vol. 1,#1. Summer 1990.
[Koenig, 1992]	Andrew Koenig: *Space Efficient Trees in C++.* Proc. USENIX C++ Conference. Portland, OR. August 1992.
[Krogdahl,1984]	Stein Krogdahl: *An Efficient Implementation of Simula Classes with Multiple Prefixing.* Research Report No 83. June 1984. University of Oslo, Institute of Informatics.
[Lea,1990]	Doug Lea and Marshall P. Cline: *The Behavior of C++ Classes.* Proc. ACM SOOPPA Conference. September 1990.
[Lea, 1991]	Doug Lea: Personal Communication.
[Lea, 1993]	Doug Lea: *The GNU C++ Library.* The C++ Report. June 1993.
[Lenkov,1989]	Dmitry Lenkov: C++ *Standardization Proposal.* #X3J 11/89-016.
[Lenkov,1991]	Dmitry Lenkov, Michey Mehta, and Shankar Unni: *Type Identification in C++.* Proc. USENIX C++ Conference. Washington, DC. April 1991.
[Linton,1987]	Mark A. Linton and Paul R. Calder: *The Design and Implementation of InterViews.* Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
[Lippman,1988]	Stan Lippman and Bjarne Stroustrup: *Pointers to Class Members in C++.* Proc. USENIX C++ Conference. Denver, CO. October 1988.
[Liskov,1979]	Barbara Liskov, et al.: *CLU Reference manual.* MIT/LCS/TR-225. October 1979.
[Liskov, 1987]	Barbara Liskov: *Data Abstraction and Hierarchy.* Addendum to Proceedings of OOPSLA'87. October 1987.
[Madsen, 1993]	Ole Lehrmann Madsen, et al.: *Object-Oriented Programming in the Beta Programming Language.* Addison-Wesley, Reading, MA. 1993. ISBN 0-201-62430.

Таблица 2 (продолжение)

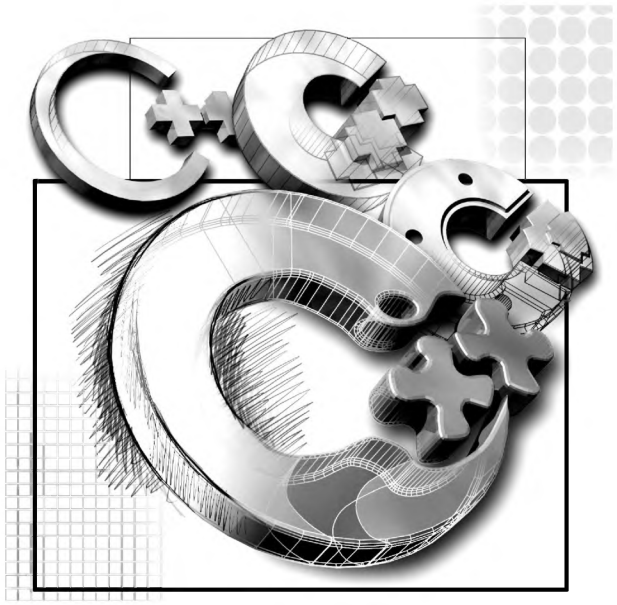
[McCluskey, 1992]	Glen McCluskey: *An Environment for Template Instantiation.* The C++ Report. February 1992.
[Meyer, 1988]	Bertrand Meyer: *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, NJ. 1988. ISBN 0-13-629049.
[Miller,1988]	William M. Miller: *Exception Handling without Language Extensions.* Proc. USENIX C++ Conference. Denver CO. October 1988.
[Mitchell,1979]	James G. Mitchell, et.al.: *Mesa Language Manual.* XEROX PARC, Palo Alto, CA. CSL-79-3. April 1979.
[Murray, 1992]	Rob Murray: A *Statically Typed Abstract Representation for* C++ *Programs.* Proc. USENIX C++ Conference. Portland, OR. August 1992.
[Nelson,1991]	Nelson, G. (editor): *Systems Programming with Modula-3.* Prentice-Hall, Englewood Cliffs, NJ. 1991. ISBN 0-13-590464-1.
[Rose,1984]	Leonie V. Rose and Bjarne Stroustrup: *Complex Arithmetic in* C++. Internal AT&T Bell Labs Technical Memorandum. January 1984. Reprinted in AT&T C++ Translator Release Notes. November 1985.
[Parrington,1990]	Graham D. Parrington: *Reliable Distributed Programming in* C++. Proc. USENIX C++ Conference. San Francisco, CA. April 1990.
[Reiser, 1992]	John F. Reiser: *Static Initializers: Reducing the Value-Added Tax on Programs.* Proc. USENIX C++ Conference. Portland, OR. August 1992.
[Richards, 1980]	Martin Richards and Colin Whitby-Strevens: *BCPL - the language and its compiler.* Cambridge University Press, Cambridge, England. 1980. ISBN 0-521-21965-5.
[Rovner,1986]	Paul Rovner: *Extending Modula-2 to Build Large, Integrated Systems.* IEEE Software Vol 3, No 6, November 1986.
[Russo,1988]	Vincent F. Russo and Simon M. Kaplan: *A C++ Interpreter for Scheme.* Proc. USENIX C++Conference. Denver, CO. October 1988.
[Russo, 1990]	Vincent F. Russo, Peter W. Madany, and Roy H. Campbell: C++ and Operating Systems Performance: A Case Study. Proc. USENIX C++ Conference. San Francisco, CA. April 1990.
[Sakkinen, 1992]	Markku Sakkinen: *A Critique of the Inheritance Principles of C++.* USENIX Computer Systems, vol 5, no 1, Winter 1992.
[Sethi, 1980]	Ravi Sethi: *A case study in specifying the semantics of a programming language.* Seventh Annual ACM Symposium on Principles of Programming Languages. January 1980.
[Sethi, 1981]	Ravi Sethi: *Uniform Syntax for Type Expressions and Declarators.* Software - Practice and Experience, Vol 11. 1981.
[Sethi, 1989]	Ravi Sethi: *Programming Languages - Concepts and Constructs.* Addison-Wesley, Reading, MA. 1989. ISBN 0-201-10365-6.
[Shopiro,1985]	Jonathan E. Shopiro: *Strings and Lists for C++.* AT&T Bell Labs Internal Technical Memorandum. July 1985.
[Shopiro, 1987]	Jonathan E. Shopiro: *Extending the C++ Task System for Real-Time Control.* Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
[Shopiro,1989]	Jonathan E. Shopiro: *An Example of Multiple Inheritance in* C++: A *Model of the Iostream Library.* ACM SIGPLAN Notices. December 1989.
[Schwarz,1989]	Jerry Schwarz: *Iostreams Examples.* AT&T C++ Translator Release Notes. June 1989.

Таблица 2 (продолжение)

[Snyder,1986]	Alan Snyder: *Encapsulation and Inheritance in Object-Oriented Programming Languages.* Proc. OOPSLA'86. September 1986.
[Stal,1993]	Michael Stal and Uwe Steinmьller: *Generic Dynamic Arrays.* The C++ Report. October 1993.
[Stepanov,1993]	Alexander Stepanov and David R. Musser: *Algorithm-Oriented Generic Software Library Development.* HP Laboratories Technical Report HPL-92-65. November 1993.
[Stroustrup,1978]	Bjarne Stroustrup: On Unifying Module Interfaces. ACM Operating Systems Review Vol 12 No 1. January 1978.
[Stroustrup,1979]	Bjarne Stroustrup: Communication and Control in Distributed Computer Systems. Ph.D. thesis, Cambridge University, 1979.
[Stroustrup,1979b]	Bjarne Stroustrup: An Inter-Module Communication System for a Distributed Computer System. Proc. 1st International Conf. on Distributed Computing Systems. October 1979.
[Stroustrup,1980]	Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language.* Bell Laboratories Computer Science Technical Report CSTR-84. April 1980. Revised, August 1981. Revised yet again and published as [Stroustrup, 1982].
[Stroustrup,1980b]	Bjarne Stroustrup: A Set of C Classes for Co-routine Style Programming. Bell Laboratories Computer Science Technical Report CSTR-90. November 1980.
[Stroustrup,1981]	Bjarne Stroustrup: *Long Return: A Technique for Improving The Efficiency of Inter-Module Communication.* Software Practice and Experience. January 1981.
[Stroustrup,1981b]	Bjarne Stroustrup: *Extensions of the C Language Type Concept.* Bell Labs Internal Memorandum. January 1981.
[Stroustrup,1982]	Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language.* ACM SIGPLAN Notices. January 1982. Revised version of [Stroustrup, 1980].
[Stroustrup, 1982b]	Bjarne Stroustrup: *Adding Classes to C: An Exercise in Language Evolution.* Bell Laboratories Computer Science internal document. April 1982. Software: Practice & Experience, Vol 13. 1983.
[Stroustrop,1984]	Bjarne Stroustrup: *The* C++ *Reference Manual.* AT&T Bell Labs Computer Science Technical Report No 108. January 1984. Revised, November 1984.
[Stroustrup,1984b]	Bjarne Stroustrup: Operator Overloading in C++. Proc. IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment. September 1984.
[Stroustrup,1984c]	Bjarne Stroustrup: Data Abstraction in C. Bell Labs Technical Journal. Vol 63, No 8. October 1984.
[Stroustrup,1985]	Bjarne Stroustrup: An Extensible I/O Facility for C++. Proc. Summer 1985 USENIX Conference. June 1985.
[Stroustrup,1986]	Bjarne Stroustrup: The C++ Programming Language. Addison-Wesley, Reading, MA. 1986. ISBNO-201-12078-X.
[Stroustrup,1986b]	Bjarne Stroustrup: What is Object-Oriented Programming? Proc. 14th ASU Conference. August 1986. Revised version in Proc. ECOOP'87, May 1987, Springer Verlag Lecture Notes in Computer Science Vol 276. Revised version in IEEE Software Magazine. May 1988.
[Stroustrup,1986c]	Bjarne Stroustrup: An Overview of C++. ACM SIGPLAN Notices. October 1986.
[Stroustrup,1987]	Bjarne Stroustrup: Multiple Inheritance/or C++. Proc. EUUG Spring Conference, May 1987. Also, USENIX Computer Systems, Vol 2 No 4. Fall 1989.

Таблица 2 (окончание)

[Stroustrup,1987b]	Bjarne Stroustrup and Jonathan Shopiro: A Set of C classes for Co-Routine Style Programming. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
[Stroustrup,1987c]	Bjarne Stroustrup: The Evolution of C++: 1985-1987. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
[Stroustrup,1987d]	Bjarne Stroustrup: Possible Directions for C++. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
[Stroustrup,1988]	Bjarne Stroustrup: Type-safe Linkage for C++. USENIX Computer Systems, Vol 1 No 4. Fall 1988.
[Stroustrup,1988b]	Bjarne Stroustrup: Parameterized Types for C++. Proc. USENIX C++ Conference, Denver, CO. October 1988. Also, USENIX Computer Systems, Vol 2 No 1. Winter 1989.
[Stroustrup,1989]	Bjarne Stroustrup: Standardizing C++. The C++ Report. Vol 1 No 1. January 1989.
[Stroustrup,1989b]	Bjarne Stroustrup: The Evolution of C++: 1985-1989. USENIX Computer Systems, Vol 2 No 3. Summer 1989. Revised version of [Stroustrup, 1987c].
[Stroustrup,1990]	Bjarne Stroustrup: On Language Wars. Hotline on Object-Oriented Technology. Vol 1, No 3. January 1990.
[Stroustrup,1990b]	Bjarne Stroustrup: Sixteen Ways to Stack a Cat. The C++ Report. October 1990.
[Stroustrup,1991]	Bjarne Stroustrup: The C++ Programming Language (2nd edition). Addison-Wesley, Reading, MA. 1991. ISBN 0-201-53992-6.
[Stroustrup,1992]	Bjarne Stroustrup and Dmitri Lenkov: Run-Time Type Identification for C++. The C++ Report. March 1992. Revised version: Proc. USENIX C++ Conference. Portland, OR. August 1992.
[Stroustrup,1992b]	Bjarne Stroustrup: How to Write a C++ Language Extension Proposal. The C++ Report. May 1992.
[Stroustrup, 1993]	Bjarne Stroustrup: The History of C++: 1979-1991. Proc. ACM History of Programming Languages Conference (HOPL-2). April 1993. ACM SIGPLAN Notices. March 1993.
[Taft,1992]	S. Tucker Taft: *Ada 9X: A Technical Summary.* CACM. November 1992.
[Tiemann,1987]	Michael Tiemann: *«Wrappers:» Solving the RPC problem in GNU* C++. Proc. USENIX C++ Conference. Denver, CO. October 1988.
[Tiemann,1990]	Michael Tiemann: *An Exception Handling Implementation for* C++. Proc. USENIX C++ Conference. San Francisco, CA. April 1990.
[Weinand,1988]	Andre Weinand, et al.: *ET++ - An Object-Oriented Application Framework in* C++. Proc. OOPSLA'88. September 1988.
[Wikstrцm,1987]	Eke Wikstrцm: *Functional Programming in Standard ML.* Prentice-Hall, Englewood Cliffs, NJ. 1987. ISBN 0-13-331968-7.
[Waldo,1991]	Jim Waldo: *Controversy: The Case for Multiple Inheritance in* C++. USENIX Computer Systems, vol 4, no 2, Spring 1991.
[Waldo,1993]	Jim Waldo (editor): The Evolution of C++. A USENIX Association book. The MIT Press, Cambridge, MA. 1993. ISBN 0-262-73107-X.
[Wilkes,1979]	M.V. Wilkes and R.M. Needham: *The Cambridge CAP Computer and its Operating System.* North-Holland, New York. 1979. ISBN 0-444-00357-6.
[Woodward, 1974]	P.M. Woodward and S.G. Bond: *Algol 68-R Users Guide.* Her Majesty's Stationery Office, London. 1974. ISBN 0-11-771600-6.



Часть I

Глава 1. Предыстория C++

Глава 2. Язык C with Classes

Глава 3. Рождение C++

Глава 4. Правила проектирования языка C++

Глава 5. Хронология 1985–1993 гг.

Глава 6. Стандартизация

Глава 7. Заинтересованность и использование

Глава 8. Библиотеки

Глава 9. Перспективы развития языка C++

В части I описываются истоки C++ и его эволюция от C with Classes, а также принципы, определявшие развитие языка на протяжении этого периода и в последующее время. Здесь приводится хронология событий после выхода версии 1.0 и рассказывается о стандартизации. Также обсуждаются сферы применения C++ и другие вопросы.

Глава 1. Предыстория C++

Давным-давно, когда правило Зло!

Кристен Найгаард

1.1. Язык Simula и распределенные системы

Предыстория C++ – за пару лет до того, как мне пришла в голову мысль добавить к C некоторые возможности из Simula, – важна потому, что в это время выкристаллизовались критерии, позднее положенные в основу C++. Я работал над докторской диссертацией в лаборатории вычислительной техники Кембриджского университета в Англии. Тема – изучение альтернативных способов построения распределенных систем. Работа велась на базе Кембриджского компьютера CAP с его экспериментальной и постоянно эволюционирующей операционной системой [Wilkes, 1979]. Подробности этой работы и ее результаты [Stroustrup, 1979] к C++ отношения не имеют. Существенными оказались интерес к построению программного обеспечения из четко определенных модулей и тот факт, что основным средством для экспериментирования стал большой симулятор, который я написал для моделирования работы программ в распределенной системе.

Первая версия симулятора была написана на Simula [Birtwistle, 1979] и работала на компьютере IBM 360/165, установленном в вычислительном центре Кембриджского университета. Возможности Simula почти идеально подходили для моих целей. Особенно поразил тот факт, что концепции языка помогали мне размышлять над существом своей задачи. Концепция класса позволила отобразить понятия из предметной области на языковые конструкции настолько естественно, что мой код оказался понятнее, чем все, что я видел в других языках. То, что классы Simula могли использоваться в качестве сопрограмм, позволило мне легко выразить параллельность, присущую моему приложению. Например, объект класса `computer` было совсем просто заставить работать псевдопараллельно с другими объектами того же класса. Варианты понятий прикладного уровня выражались через иерархию классов. Так, виды компьютеров можно было представить как классы, производные от `computer`, а типы механизмов межмодульных коммуникаций – как классы, производные от класса `IPC`. Правда, этот прием я применял нечасто: для моего симулятора важнее было использовать классы с целью представления параллельности.

Во время написания и начальной отладки программы я смог в полной мере оценить выразительную мощь системы типов в Simula и способность компилятора находить ошибки типизации. Я заметил, что ошибки эти почти всегда являлись свидетельством невнимательности или изъяна проектирования. Последнее было

важнее всего и помогало мне больше, чем примитивные «сильно» типизированные языки, с которыми доводилось работать раньше. Так я пришел к уже звучавшему выше выводу, что система типов в языке Pascal не просто бесполезна – это смирительная рубашка, которая создает больше проблем, нежели решает, заставляя меня жертвовать чистотой дизайна ради удовлетворения причуд компилятора. Обнаруженный контраст между строгостью Pascal и гибкостью Simula оказался чрезвычайно важен при разработке C++. Концепция классов в Simula представлялась мне ключевым фактором, и с той поры я считаю, что при проектировании программ следует сосредотачивать свое внимание именно на классах. Я работал с Simula и прежде (во время учебы в университете города Аархус в Дании), но был приятно удивлен следующим: чем больше программа, тем очевидней польза от возможностей Simula. Механизмы классов и сопрограмм, а также исчерпывающий контроль типов гарантировали, что число ошибок увеличивается с ростом программы не более чем линейно (что было неожиданно). Напротив, программа работала, скорее, как набор очень маленьких программок, нежели как большой монолит, и поэтому писать, понимать и отлаживать ее было проще.

Однако реализация самого языка Simula не масштабировалась в той же степени, что и моя программа. В результате весь проект чуть не закончился крахом. В то время я пришел к выводу, что реализация Simula (в отличие от самого языка) была ориентирована на небольшие по объему программы, а для больших не приспособлена [Stroustrup, 1979]. На связывание отдельно скомпилированных классов уходила масса времени: на компиляцию 1/30 части программы и связывание ее с остальными, уже откомпилированными модулями тратилось больше времени, чем на компиляцию и связывание всей программы как монолита. Думаю, что, вероятнее всего, это была проблема используемого компоновщика, а не самого языка Simula, но это слабое утешение. Кроме того, производительность программы была такой низкой, что не оставляла надежд получить от симулятора хоть сколько-нибудь полезные данные. Плохие показатели производительности были обусловлены языком и его реализацией, а не приложением. Проблема накладных расходов является в Simula фундаментальной и неустранимой. Она коренится в некоторых особенностях языка и их взаимодействиях: проверке типов во время исполнения, гарантированной инициализации переменных, поддержке параллельности, сборке мусора для объектов, созданных пользователем, и записей активации процедур. Измерения показали: более 80% времени тратится на сборку мусора, хотя управление ресурсами брала на себя моделируемая система, так что мусор вообще не появлялся. Современные реализации Simula (15 лет спустя) стали лучше, но, по моим сведениям, увеличения производительности на порядок так и не достигнуто.

Чтобы не бросать проект, я переписал симулятор на BCPL и запускал его на экспериментальном компьютере CAP. Опыт кодирования и отладки на BCPL [Richards, 1980] оставил у меня неприятные воспоминания. Язык C по сравнению с BCPL – язык очень высокого уровня. Ни контроля типов, ни поддержки во время исполнения в BCPL здесь нет и в помине. Однако получившийся симулятор работал достаточно быстро и с его помощью я получил целый ряд полезных результатов. Они прояснили многие вопросы и легли в основу нескольких статей по операционным системам [Stroustrup, 1978, 1979b, 1981].

Расставшись с Кембриджем, я поклялся себе никогда больше не приступать к решению задачи, располагая такими неподходящими инструментами, как те, с которыми я намучился при проектировании и реализации симулятора. Для истории C++ важную роль сыграло составленное мной представление о «подходящем» инструменте для проектов такого масштаба, как большой симулятор, операционная система и аналогичные задачи системного программирования. Вот эти критерии:

- хороший инструмент должен предоставлять средства организации программ, подобные имеющимся в Simula: классы, форму их иерархии, поддержку параллельности и сильный (т.е. статический) контроль типов, основанный на классах. Эти критерии представлялись мне тогда (да и теперь) существенными для поддержки процесса проектирования, а не для реализации программы;
- необходимо, чтобы он генерировал программы, работающие так же быстро, как написанные на BCPL, и обладал способностью BCPL объединять раздельно откомпилированные модули в единую программу. Должно быть простое соглашение о связях, чтобы удалось объединять модули, написанные на разных языках, таких как C, Algol68, Fortran, BCPL, ассемблер и т.д. Иначе программист будет вынужден бороться с ограничениями, присущими какому-то одному языку;
- инструмент должен обеспечивать переносимую реализацию. Мой опыт показал, что «правильная» реализация, остро необходимая мне сейчас, будет готова «не раньше следующего года», да и то на компьютере, которого у меня нет. Отсюда следует, что должно быть несколько источников реализации инструмента (никакой монополист не сможет поддерживать всех пользователей «редких» машин, а также бедных студентов). Не должно быть также сложной системы поддержки времени исполнения, которую трудно переносить, и допустима лишь очень ограниченная зависимость инструмента от операционной системы.

Эти критерии еще не были четко сформулированы, когда я покидал Кембридж. Некоторые из них окончательно оформились лишь в ходе последующего осмысления собственного опыта, приобретенного при создании симулятора и программ, которые я писал в течение еще пары лет, а также опыта других людей. C++ в том виде, какой он принял к моменту выхода версии 2.0, полностью отвечает данным критериям; серьезные проблемы, с которыми я столкнулся при проектировании шаблонов и обработке исключений, связаны с отходом от некоторых из этих принципов. Я полагаю, что самой важной особенностью сформулированных выше правил является их слабая связь с нюансами конкретных языков программирования. Вместо этого они налагают определенные ограничения на решение.

Когда я работал в Кембридже, лабораторию вычислительной техники возглавлял Морис Уилкс (Maurice Wilkes). Помощь во всех технических вопросах мне оказывали мой руководитель Дэвид Уилер (David Wheeler) и Роджер Нидэм (Roger Needham). Мои знания в области операционных систем и интерес к модульности и межмодульным коммуникациям способствовали развитию C++.

Например, модель защиты в C++ базируется на концепции предоставления и передачи прав доступа; различие между инициализацией и присваиванием возникло благодаря размышлениям о способности переноса (transferring capabilities); концепция `const` берет начало от механизмов защиты от чтения/записи в аппаратных устройствах; механизм обработки исключений появился в связи с работой над отказоустойчивыми системами, выполненной группой под руководством Брайана Рэнделла (Brian Randell) в Ньюкасле в 70-х гг.

1.2. Язык C и системное программирование

На языке C я начал работать в Лондоне в 1975 г. и оценил его преимущества по сравнению с другими языками, которые принято называть языками для системного программирования, машинно-ориентированными или низкоуровневыми. Из таких языков мне были известны PL 360, Coral, Mary и другие, но в основном BCPL. Я не только пользовался BCPL, но однажды и реализовал его путем трансляции в промежуточный микрокод – O-код, так что хорошо представлял себе низкоуровневые аспекты, связанные с эффективностью языков такого класса.

Защитив диссертацию в Кембридже и получив работу в Bell Labs, я еще раз изучил C по книге Кернигана [Kernighan, 1978]. В то время я не был экспертом по C и рассматривал его в основном как самый современный и известный пример языка системного программирования. Лишь позже, приобретая собственный опыт и беседуя с коллегами – Стью Фельдманом (Stu Feldman), Стивом Джонсоном (Steve Johnson), Брайаном Керниганом и Деннисом Ричи, – я начал по-настоящему понимать C. Таким образом, общее представление о языках системного программирования значило для формирования C++ по меньшей мере столько же, сколько конкретные технические детали C.

Я довольно хорошо знал Algol68 [Woodward,1974] по работе над небольшими проектами в Кембридже и видел связь между конструкциями этого языка и C. Иногда мне казалось полезным рассматривать конструкции C как частные случаи более общих конструкций Algol68. Любопытно, что я никогда не рассматривал Algol68 в качестве языка системного программирования (несмотря на то что сам пользовался написанной на нем операционной системой). Подозреваю, причина здесь в том, что я считал очень важными переносимость, простоту связывания с программами на других языках и эффективность исполнения. Как-то раз я сказал, что Algol68 с классами, как в Simula, – это язык моей мечты. Однако в качестве практического инструмента C казался мне лучше, чем Algol68.

1.3. Немного об авторе книги

Говорят, что структура системы отражает структуру организации, в которой она была создана. В общем и целом я поддерживаю это мнение. Из него также следует, что если система есть плод работы одного человека, то она отражает склад его личности. Оглядываясь назад, я думаю, что на общую структуру C++ мое мировоззрение наложило такой же отпечаток, как и научные концепции, лежащие в основе отдельных его частей.

Я изучал математику, в том числе прикладную, поэтому защищенная в Дании кандидатская диссертация была посвящена математике и информатике. В результате я научился любить красоту математики, но предпочитал смотреть на нее, как на инструмент решения практических задач. Я искренне сочувствовал студенту, которого Евклид, по преданию, выгнал за вопрос «Но для чего нужна математика?» Точно так же мой интерес к компьютерам и языкам программирования носит в основном прагматический характер. Компьютеры и языки программирования можно оценивать как произведения искусства, но эстетические факторы должны дополнять и усиливать их полезные свойства, а не подменять их.

Больше 25 лет я увлекаюсь историей. Немалое время посвятил и изучению философии. Отсюда вполне осознанный взгляд на истоки моих интеллектуальных пристрастий. Если говорить о философских течениях, то мне, скорее, ближе эмпирики, чем идеалисты; мистиков я просто не понимаю. Поэтому Аристотеля я предпочитаю Платону, Юма – Декарту, а перед Паскалем склоняю голову. Всеобъемлющие «системы», такие, как у Платона или Канта, пленяют меня, но кажутся фундаментально порочными, поскольку, по-моему, они очень далеки от повседневного опыта и особенностей конкретного индивидуума.

Почти фанатичный интерес Кьеркегора к личности и его тонкое понимание психологии кажутся мне куда интереснее грандиозных схем и заботы обо всем человечестве, присущих Гегелю или Марксу. Уважение к группе, не подразумевающее уважения к ее членам, я не считаю уважением вовсе. Корни многих решений для C++ – в моем нежелании принуждать пользователей делать что бы то ни было жестко определенным образом. Из истории мы знаем, что вина за многие ужасные трагедии лежит на идеалистах, которые пытались заставить людей «делать так, чтобы им было хорошо». Кроме того, я считаю, что идеалисты склонны игнорировать неудобный опыт и факты, противоречащие их догмам или теории. Всякий раз, когда те или иные идеалы вступают в противоречие, а иногда и в тех ситуациях, где ученые мужи пришли к единодушному согласию, я предпочитаю давать программисту выбор.

Мои литературные вкусы еще раз подтверждают нежелание принимать решение только на основе теории и логики. В этом смысле C++ во многом обязан таким романистам и эссеистам, как Мартин А. Хансен, Альбер Камю и Джордж Оруэлл, которые никогда не видывали компьютера, и таким ученым, как Дэвид Грис, Дональд Кнут и Роджер Нидэм. Часто, испытывая искушение запретить какую-то возможность, которая лично мне не нравилась, я останавливался, ибо не считал себя вправе навязывать свою точку зрения другим людям. Я знаю, что многого можно добиться относительно быстро, если последовательно придерживаться логики и безжалостно выносить приговор «неправильному, устаревшему и нелогичному образу мыслей». Но при такой модели становятся очень велики человеческие потери. Для меня намного дороже принятие того факта, что люди думают и действуют по-разному.

Я предпочитаю медленно – иногда очень медленно – убеждать людей попробовать новые приемы и принять на вооружение те, которые отвечают их нуждам и склонностям. Существуют эффективные методы «обращения в другую веру» и «совершения революции», но я их боюсь и сильно сомневаюсь, что они так уж

эффективны в длительной перспективе и по большому счету. Часто, когда кого-то легко удается обратить в религию X, последующее обращение в религию Y оказывается столь же простым, а выигрыш от этого эфемерный. Я предпочитаю скептиков «истинно верующим». Мелкий, но неопровержимый факт для меня ценнее большинства теорий, а продемонстрированный экспериментально результат важнее груды логических аргументов.

Такой взгляд на вещи легко может привести к фаталистическому принятию status quo. В конце концов, нельзя приготовить омлет, не разбив яиц, – многие люди просто не хотят меняться, если это неудобно для их повседневной жизни, или стремятся хотя бы отложить перемены «до понедельника». Вот тут-то надо проявить уважение к фактам... и толику идеализма.

Состояние дел в программировании, как и вообще в мире, далеко от идеала, и многое можно улучшить. Я проектировал C++, чтобы решить определенную задачу, а не для того, чтобы что-то кому-то доказать, и в результате язык оказался полезным. В основе его философии лежала убежденность, что улучшений можно добиться путем последовательных изменений. Конечно, хотелось бы поддерживать максимальный темп изменений, улучшающих благосостояние людей. Но самое трудное – понять, в чем же состоит прогресс, разработать методику постепенного перехода к лучшему и избежать эксцессов, вызванных чрезмерным энтузиазмом.

Я готов упорно работать над внедрением в сознание идей, которые, по моему глубокому убеждению, принесут пользу людям. Более того, я считаю, что ученые и интеллектуалы должны способствовать распространению своих идей в обществе, чтобы они применялись людьми, а не оставались игрой изощренного ума. Однако я не готов жертвовать людьми во имя этих идей. В частности, я не хочу навязывать единый стиль проектирования посредством узко определенного языка программирования. Мысли и действия людей настолько индивидуальны, что любая попытка привести всех к общему знаменателю принесет больше вреда, чем пользы. Поэтому C++ сознательно спроектирован так, чтобы поддержать различные стили, а не показать единственный «истинный путь».

Принципы, которыми я руководствовался при проектировании C++, будут детально изложены в главе 4. Здесь вы обнаружите отголоски тех общих идей и идеалов, о которых говорилось только что.

Да, язык программирования – на редкость важная вещь, однако это всего лишь крохотная часть реального мира и поэтому не стоит относиться к нему чересчур серьезно. Необходимо обладать чувством меры и – что еще важнее – чувством юмора. Среди основных языков программирования C++ – богатейший источник шуток и анекдотов. И это неслучайно.

При обсуждении философских вопросов, равно как и возможностей языка легко скатиться на чрезмерно серьезный и нравоучительный тон. Если так произошло со мной, примите извинения, но мне хотелось объяснить свои интеллектуальные пристрастия, и думаю, что это безвредно – ну, почти безвредно. Да, кстати, мои литературные вкусы не ограничиваются только произведениями вышеназванных авторов, просто именно они повлияли на создание C++.

Глава 2. Язык C with Classes

Специализация – это для насекомых.
Р.А. Хайнлайн

2.1. Рождение C with Classes

Работа над тем, что впоследствии стало языком C++, началась с попытки проанализировать ядро UNIX, чтобы понять, как можно было бы распределить эту систему между несколькими компьютерами, соединенными локальной сетью. Этот эксперимент проходил в апреле 1979 г. в Центре исследований по вычислительной технике компании Bell Laboratories в Мюррей Хилл, штат Нью-Джерси. Вскоре было выявлено две подзадачи: как проанализировать сетевой трафик, порожденный распределенностью ядра, и как разбить ядро на отдельные модули. В обоих случаях требовалось выразить модульную структуру сложной системы и типичные способы обмена информацией между модулями. Задача была как раз из тех, к решению которых я зарекся приступать без соответствующих инструментов. Поэтому мне пришлось заняться разработкой подходящего инструментария в соответствии с критериями, выработанными в Кембридже.

В октябре 1979 г. был готов препроцессор, который я назвал Cpre. Он добавлял Simula-подобные классы к C. В марте 1980 г. препроцессор был улучшен настолько, что использовался в одном реальном проекте и нескольких экспериментальных. В моем архиве сохранилось упоминание о том, что к тому времени Cpre работал на 16 системах. Самым важным элементом этих проектов была первая серьезная библиотека на C++, которая поддерживала многозадачное программирование с применением сопрограмм [Stroustrup, 1980b], [Stroustrup, 1987b], [Shopiro, 1987]. Язык, подаваемый на вход препроцессора, получил название «C with Classes».

В период с апреля по октябрь я начал думать не об инструменте, а о языке, но C with Classes все еще рассматривался как расширение C, позволяющее выразить концепции модульности и параллельности. Однако важнейшее решение уже было принято. Хотя поддержка параллельности и моделирования в духе Simula и являлась основной целью C with Classes, язык не содержал никаких примитивов для выражения параллельности. Вместо этого была написана библиотека, поддерживающая необходимые стили параллельности. В ней использовалась комбинация наследования (иерархии классов) с возможностью определять функции-члены класса специального назначения, распознаваемые препроцессором. Обратите внимание на множественное число – «стили». Я считал – и считаю – особенно важным, чтобы с помощью языка можно было выразить несколько вариантов параллельной обработки. Есть множество приложений, которым параллельность необходима,

а преобладающей модели ее поддержки нет. Поэтому в случаях, когда параллельность нужна, ее следует реализовывать в библиотеках или в специализированном расширении так, чтобы одна конкретная форма не запрещала использование других.

Таким образом, на языке предлагались общие механизмы организации программ, а вовсе не поддержка конкретных предметных областей. Именно это и сделало C with Classes, позднее и C++, универсальным языком программирования, а не расширением C для специализированных приложений. Позже вопрос о выборе между поддержкой специализированных приложений и общим механизмом абстракций вставал неоднократно. И всякий раз принималось решение в пользу усовершенствования механизма абстракций. Поэтому в C++ нет ни встроенных типов для комплексных чисел, строк и матриц, ни прямой поддержки параллельности, устойчивости объектов, ни распределенных вычислений, ни сопоставления образцов и манипуляций на уровне файловой системы. Я упомянул лишь малую толику часто предлагавшихся расширений. Но существуют библиотеки, поддерживающие все эти возможности.

Предварительное описание C with Classes было опубликовано в виде технического отчета в Bell Labs в апреле 1980 г. [Stroustrup,1980] и в журнале SIGPLAN Notices [Stroustrup,1982]. Более подробный технический отчет *Adding Classes to the C Language: An Exercise in Language Evolution* [Stroustrup, 1980] напечатан в журнале *Software: Practices and Experience*. В данных работах заложена хорошая идея: описывать только те возможности, которые полностью реализованы и нашли применение. Такой подход соответствовал традициям Центра исследований по вычислительной технике компании Bell Laboratories. Я отошел от подобной тактики только тогда, когда назрела необходимость большей открытости относительно будущего C++, чтобы в свободной дискуссии по поводу эволюции языка могли принять участие многочисленные пользователи, которые не работали в AT&T.

C with Classes специально проектировался для обеспечения лучшей организации программ. Было решено, что с собственно «вычислительной частью» C и так справляется. Я считал принципиальным такой аспект: улучшение структуры не может достигаться за счет падения производительности по сравнению с C. Язык C with Classes не должен был уступать C во времени выполнения, компактности кода и данных. Однажды кто-то продемонстрировал систематическое трехпроцентное падение производительности новой версии по сравнению с C, вызванное наличием временных переменных, которые препроцессор C with Classes использовал в механизме возврата из функции. Недочет был признан неприемлемым, и причину быстро устранили. Аналогично, чтобы обеспечить совместимость формата хранения с C и, следовательно, избежать накладных расходов по памяти, в объекты классов не помещались никакие «служебные» данные.

Другой важной целью работы было стремление избежать ограничений на области использования C with Classes. Идеал (кстати, достигнутый) – C with Classes может применяться везде, где использовался C. Отсюда следовало, что новая версия ни в коем случае не должна была уступать C в эффективности, но эта эффективность не могла достигаться за счет отказа от некоторых, пусть даже уродливых особенностей C. Это соображение (если хотите, принцип) приходилось снова

и снова повторять тем людям (как правило, не работавшим постоянно с C with Classes), которые хотели сделать версию безопаснее, введя статический контроль типов а-ля Pascal. Альтернативу такой «безопасности» – вставку проверок в исполняемый код – решили реализовывать в отладочных средах. В самом языке нельзя было организовывать таких проверок, ибо тогда он безнадежно проиграл бы C по скорости и расходу памяти. Поэтому такого рода контроль в C with Classes включен не был, хотя в некоторых средах разработки для C++ он имеется, но только в отладочном режиме. Кроме того, пользователи могут сами производить проверки во время выполнения там, где сочтут целесообразным (см. раздел 16.10 и [2nd]).

C поддерживает низкоуровневые операции, например манипуляции с битами и выбор между разными размерами целых. Имеются в нем и такие средства (допустим, явное преобразование типов), которые позволяют намеренно обойти систему контроля типов. C With Classes, а затем и C++ сохранили низкоуровневые и небезопасные особенности C. Но в C++ систематически устраняется необходимость использования таких средств за исключением тех мест, где без них никак не обойтись. При этом небезопасные операции выполняются только по явному указанию программиста. Я искренне убежден, что не существует единственно правильного способа написать программу, и дизайнер языка не должен заставлять программиста следовать определенному стилю. С другой стороны, дизайнер обязан всячески поддерживать все разнообразие стилей и способов программирования, доказавших свою эффективность, а равно предоставить такие языковые возможности и инструментальные средства, которые уберегли бы программиста от хорошо известных ловушек.

2.2. Обзор языковых возможностей

Вот сводка тех средств, которые были включены в первую версию 1980 г.:

- классы (раздел 2.3);
- производные классы (но пока без виртуальных функций, раздел 2.9);
- контроль доступа – открытый/закрытый (раздел 2.10);
- конструкторы и деструкторы (раздел 2.11.1);
- функции, вызываемые при вызове и возврате (позже исключены, раздел 2.11.3);
- дружественные (`friend`) классы (раздел 2.10);
- контроль и преобразование типов аргументов функции (раздел 2.6).

В 1981 г. добавлены:

- встраиваемые (inline) функции (раздел 2.4.1);
- аргументы по умолчанию (раздел 2.12.2);
- перегрузка оператора присваивания (раздел 2.12.1).

Поскольку C with Classes был реализован с помощью препроцессора, описывать следовало лишь новые, отсутствовавшие в C возможности, а вся мощь C и так оставалась в распоряжении пользователей. В то время оба эти аспекта были должным образом оценены. Раз C – подмножество языка, то объем работы по поддержке и документированию намного уменьшается. Это особенно важно, поскольку на

протяжении нескольких лет мне приходилось заниматься документированием и поддержкой C with Classes и C++ одновременно с экспериментированием, проектированием и реализацией новых версий. Доступность всех возможностей C гарантировала, что по недосмотру или из-за моих предрассудков не будет введено никаких ограничений, которые лишат пользователей привычных средств. Естественно, переносимость на машины, где C уже имелся, обеспечивалась автоматически. Изначально C with Classes был реализован и эксплуатировался на DEC PDP/11, но вскоре был перенесен на DEC VAX и компьютеры на базе процессора Motorola 68000.

C with Classes все еще рассматривался как диалект C, а не как самостоятельный язык. Даже классы тогда назывались «абстрактными типами данных (abstract data type facility)» [Stroustrup, 1980]. Поддержка объектно-ориентированного программирования была провозглашена лишь после добавления в C++ виртуальных функций [Stroustrup, 1984].

2.3. Классы

Очевидно, самой важной чертой C with Classes, а позже и C++ была концепция класса. Многие ее аспекты видны из следующего примера [Stroustrup, 1980][1]:

```
class stack {
   char   s[SIZE];   /* массив символов */
   char*  min;       /* указатель на конец стека */
   char*  top;       /* указатель на вершину стека */
   char*  max;       /* указатель на начало выделенной области */
   void   new();     /* функция инициализации (конструктор) */
public:
   void   push(char);
   char   pop();
};
```

Класс – это определенный пользователем тип данных. Он содержит описания типов членов класса, то есть представление переменной данного типа (объекта класса), и набор операций (функций) для манипулирования такими объектами. Класс также определяет права доступа к своим членам со стороны пользователей. Функции-члены обычно определяются отдельно:

```
char stack.pop()
{
   if (top <= min) error("стек пуст");
   return *(--top);
}
```

Теперь можно определить и использовать объекты класса `stack`:

```
class stack s1, s2;           /* две переменные типа stack */
class stack * p1 = &s2;       /* p1 указывает на s2 */
```

[1] Я сохранил оригинальный синтаксис и стиль C with Classes. Отличия от C++ и современного стиля кодирования вряд ли у кого-то вызовут затруднения, а некоторым читателям будут интересны. Однако очевидные ляпы исправлены. Также добавлены комментарии, которых в исходном тексте не было.

```
class stack * p2 = new stack; /* p2 указывает на объект класса stack,
                                 размещенный в куче */

s1.push('h');    /* прямое обращение к объекту */
p1->push('s');   /* обращение к объекту по указателю */
```

Из этого примера видны несколько ключевых проектных решений:

- по аналогии с Simula язык C with Classes позволяет задать типы, из которых создаются переменные (объекты), тогда как, скажем, Modula описывает модуль как набор объектов и функций. В C with Classes класс – это тип (см. раздел 2.9), что стало главной особенностью C++. Но если слово `class` определяет в C++ пользовательский тип, то почему я не назвал его `type`? В основном потому, что я не люблю изобретать новую терминологию, а соглашения, принятые в Simula, меня в большинстве случаев устраивали;
- представление объектов пользовательского типа – часть объявления класса, что имеет далеко идущие последствия (см. разделы 2.4 и 2.5). Например, это означает следующее: настоящие локальные переменные пользовательского типа можно реализовать без использования кучи (ее еще называют динамической памятью) и сборки мусора. Отсюда также следует, что при изменении представления объекта все функции, использующие его напрямую (не через указатель), должны быть перекомпилированы. (См. раздел 13.2, где описаны средства C++ для определения интерфейсов, позволяющие избежать такой перекомпиляции.);
- контроль доступа используется еще на стадии компиляции для ограничения доступа к элементам представления класса. По умолчанию имена членов класса могут встречаться только в функциях, упомянутых в объявлении класса (см. раздел 2.10). Члены (обычно функции-члены), описанные в открытом интерфейсе класса – части, следующей за меткой `public:`, – могут использоваться и в других местах программы;
- для функций-членов указывается полный тип (включая тип возвращаемого значения и типы формальных аргументов). На основе данной спецификации производится статическая (во время компиляции) проверка типов (см. раздел 2.6). В то время это было отличием от C, где типы формальных аргументов не задавались в интерфейсе и не проверялись при вызове;
- определения функций обычно выносятся в другое место, чтобы класс больше походил на спецификацию интерфейса, чем на лексический механизм организации исходного кода. Это облегчает раздельную компиляцию функций-членов класса и внешней программы, которая их использует. Поэтому техники компоновки, традиционной для C, достаточно для поддержки C++ (см. раздел 2.5);
- функция `new()` является конструктором, для компилятора она имеет специальный смысл. Такие функции дают определенные гарантии относительно классов (см. раздел 2.11): конструктор – в то время он назывался `new`-функцией – обязательно будет вызван для инициализации каждого объекта своего класса перед его первым использованием;

- имеются указательные и неуказательные типы (они есть и в C, и в Simula). Указатели могут указывать на объекты как встроенных, так и пользовательских типов;
- как и в C, память для объектов может выделяться тремя способами: в стеке (автоматическая память), по фиксированному адресу (статическая память) и из кучи (динамическая память). Но в отличие от C, в C with Classes есть специальные операторы `new` и `delete` для выделения и освобождения динамической памяти (см. раздел 2.11.2).

Дальнейшую разработку C with Classes и C++ можно в значительной мере рассматривать как изучение последствий этих проектных решений, выявление их положительных и отрицательных сторон и устранение проблем, вызванных недостатками. Многие, но далеко не все следствия принятых решений были понятны уже в то время (работа [Stroustrup,1980] датирована 3 апреля 1980 г.). В данном разделе я попытаюсь объяснить, чту же было ясно уже тогда, и сошлюсь на разделы, где рассматриваются более отдаленные последствия и позднейшие реализации.

2.4. Эффективность исполнения

В Simula не может быть локальных или глобальных переменных типа класса, память для каждого объекта класса выделяется динамически оператором `new`. Измерения с помощью кембриджского симулятора убедили меня, что это основная причина неэффективности языка. Позднее Карел Бабчиски (Karel Babcisky) из Норвежского Вычислительного Центра представил данные о производительности Simula, подтвердившие мой вывод [Babcisky, 1984]. Уже по этой причине я хотел иметь глобальные и локальные переменные типа класса.

Кроме того, наличие разных правил для создания и области действия переменных встроенных и пользовательских типов просто неизящно, а в некоторых случаях я видел, как страдает мой стиль программирования от отсутствия в Simula локальных и глобальных переменных типа класса. Не хватало и возможности иметь указатели на встроенные типы в Simula. Эти наблюдения со временем превратились в определенное правило дизайна C++: пользовательские и встроенные типы должны вести себя одинаково по отношению к правилам языка. И сам язык, и инструментальные средства должны обеспечивать для них одинаковую поддержку. Когда формулировался этот критерий, поддержка встроенных типов была гораздо обширнее, но развитие языка C++ продолжалось, так что теперь данные типы поддержаны чуть хуже пользовательских (см. раздел 15.11.3).

В первой версии C with Classes отсутствовали встраиваемые (inline) функции, но вскоре они были добавлены. Основная причина их включения в язык – опасение, что из-за расходов на преодоление «защитного барьера» люди не захотят пользоваться классами для сокрытия представления. Так, в [Stroustrup,1982b] отмечено, что многие делают члены классов открытыми, дабы не расплачиваться за вызов конструктора в простых классах, где для инициализации достаточно одного-двух присваиваний. Толчком для включения в C with Classes встраиваемых

функций послужил проект, в котором для некоторых классов, связанных с обработкой в реальном времени, накладные расходы на вызов функций оказались неприемлемы. Чтобы воспользоваться преимуществами классов в такого рода приложениях, необходимо иметь возможность «бесплатно» преодолевать защиту доступа. Этого можно было добиться только сочетанием представления в объявлении класса с встраиванием вызовов открытых функций.

В связи с этим возникло следующее правило C++: недостаточно просто предоставить возможность, нужно также, чтобы плата за пользование ей была не слишком велика, то есть «приемлема на том оборудовании, которое есть у пользователей», а не «приемлема для исследователей, имеющих доступ к высокопроизводительному оборудованию» или «приемлема через пару лет, когда аппаратные средства подешевеют». C with Classes всегда рассматривался как язык, готовый к применению сейчас или в следующем месяце, а не как исследовательский проект, от которого можно ожидать отдачи через несколько лет.

2.4.1. Встраивание

Встраивание было признано важным для удобства работы с классами. Поэтому вопрос заключался не в том, стоит ли его реализовывать, а в том, кбк это сделать. Два аргумента убедили меня, что только программист должен решать, какие функции компилятор попытается встроить в код. Во-первых, у меня был печальный опыт работы с языками, где решение вопроса о встраивании оставлялось на усмотрение компилятора, поскольку он якобы «лучше знает». Однако на компилятор можно положиться только в том случае, если в него запрограммирована концепция встраивания и его представление об оптимизации по времени и памяти совпадает с моим. Опыт работы с другими языками показал, что встраивание, как правило, «будет реализовано в следующей версии», да и то в соответствии с внутренней логикой языка, которой программист не может эффективно управлять. Кроме того, C (а за ним C with Classes, и C++) организует раздельную компиляцию, так что компилятору всегда доступен только небольшой фрагмент всей программы (см. раздел 2.5). Встраивание функции, исходный код которой неизвестен, возможно только при наличии очень развитой технологии компоновки и оптимизации, но тогда такой технологии не было (нет и сейчас в большинстве сред разработки). Наконец, методы, использующие глобальный анализ кода, в частности автоматическое встраивание без поддержки со стороны пользователя, плохо адаптируются к большим программам. C with Classes проектировался для того, чтобы получать эффективный код при наличии простой переносимой реализации на наиболее распространенных системах. С учетом всего вышесказанного от программиста требуется помощь. Даже сегодня этот выбор кажется мне правильным.

В C with Classes допускалось только встраивание функций-членов. Единственным способом заставить компилятор встроить функцию было помещение ее в объявление класса. Например:

```
class stack {
   /* ... */
   char pop()
   {
      if (top <= min) error("стек пуст");
```

```
        return *--top;
    }
};
```

То, что при этом объявление класса несколько теряет наглядность, не осталось без внимания, но было сочтено правильным, поскольку препятствовало чрезмерному применению встраиваемых функций. Ключевое слово `inline` и возможность встраивать функции, не являющиеся членами, появились позже, уже в C++. Так, в C++ этот пример можно переписать иначе:

```
class stack {// C++
    // ...
    char pop();
};

inline char stack::pop()   // C++
{
    if (top <= min) error("стек пуст");
    return *--top;
}
```

Директива `inline` – лишь совет, который компилятор может игнорировать и часто так и поступает. Это вызвано логической необходимостью, поскольку можно написать рекурсивную встраиваемую функцию, а на этапе компиляции невозможно доказать, что рекурсия не окажется бесконечной. Попытка встраивания такой функции привела бы к зацикливанию компилятора. Придание слову `inline` статуса совета имеет и практическую пользу, поскольку позволяет автору компилятора обработать те случаи, когда встраивание невозможно, и просто отказаться от него.

Для C with Classes, как и для всех его преемников, было необходимо, чтобы встраиваемая функция имела в программе единственное определение. Определение такой функции, как `pop()`, приведенной выше, в разных единицах компиляции привело бы к хаосу, игнорированию системы контроля типов. Но в условиях раздельной компиляции трудно гарантировать, что в большой системе данное правило не нарушено. В C with Classes это не проверялось, и в большинстве реализаций C++ до сих пор нет гарантий, что встраиваемая функция не определена по-разному в разных единицах компиляции. Однако теоретическая проблема не переросла в практическую в основном потому, что встраиваемые функции обычно определяют в заголовочных файлах вместе с классами, а объявления классов в программе также должны быть уникальны.

2.5. Модель компоновки

Вопрос о том, как скомпоновать раздельно откомпилированные фрагменты программы, важен для любого языка программирования и до некоторой степени определяет возможности языка. На разработку C with Classes и C++ во многом повлияли следующие решения:

- раздельная компиляция может осуществляться с использованием стандартных компоновщиков (редакторов связей) для C/Fortran, применяемых на платформах UNIX и DOS;

- компоновка должна быть типобезопасной (не противоречить системе контроля типов);
- компоновка не должна нуждаться в какой бы то ни было базе данных (хотя ее использование в конкретных реализациях для повышения эффективности не запрещается);
- компоновка с фрагментами программы, написанными на других языках, например на C, Fortran или ассемблере, должна быть простой и эффективной.

Чтобы обеспечить непротиворечивость раздельной компиляции в C, используются заголовочные файлы, которые обычно дословно включаются в каждый исходный файл, где соответствующее объявление необходимо. В них помещаются объявления структур данных, функций, переменных и констант. Непротиворечивость обеспечивается за счет того, что в заголовочные файлы помещается вся необходимая информация, доступ к которой производится только путем включения этих файлов. C++ следует этой модели, но только до определенного момента.

В объявлении класса в C++ может (хотя и необязательно, см. раздел 13.2) быть описано размещение объекта. Это разрешено для того, чтобы упростить и сделать эффективным объявление истинно локальных переменных. Рассмотрим такую функцию:

```
void f()
{
   class stack s;
   int c;
   s.push('h');
   c = s.pop();
}
```

Реагируя на объявление класса `stack` (см. разделы 2.3 и 2.4.1), даже простейшая версия C with Classes сможет сгенерировать для этого примера код, где: динамическая память не используется, функция `pop()` встроена так, что ее вызов не связан с накладными расходами, а при обращении к `push()` вызывается невстраиваемая, отдельно скомпилированная функция. В этом отношении C++ напоминает язык Ada.

В то время я полагал, что можно найти какой-то компромисс между двумя подходами:

- отделение объявления интерфейса от реализации (как в Modula-2) в сочетании с подходящим инструментом (редактором связей);
- наличие единого объявления класса в сочетании с инструментом (анализатором зависимостей), который будет рассматривать интерфейс отдельно от деталей реализации с целью определить, в каких случаях нужна повторная компиляция.

Похоже, я недооценил сложность последнего решения, а сторонники первого подхода – его стоимость (с точки зрения переносимости и затрат во время исполнения).

Еще больше я осложнил жизнь пользователям C++, не объяснив должным образом, как можно воспользоваться производными классами для отделения

интерфейса от реализации. Разумеется, я пытался растолковать это (см. пример в [Stroustrup,1986, §7.6.2]), но почему-то не был понят. Думаю, причина неудачи в том, что мне никогда не приходила в голову простая мысль: многие (если не большинство) программистов, работая с C++, думают, что раз можно поместить представление прямо в объявление класса, описывающего интерфейс, то это обязательно нужно сделать.

Я не пытался создать инструменты типобезопасной компоновки для C with Classes, они появились лишь в версии C++ 2.0. Однако я помню разговор с Деннисом Ричи и Стивом Джонсоном о том, что безопасность с точки зрения типов при пересечении границ единиц компиляции должна была стать частью C. Просто не было возможностей гарантировать это для реальных программ, так что пришлось полагаться на инструменты типа `Lint` [Kernighan, 1984].

В частности, Стив Джонсон и Деннис Ричи утверждали, что в C предполагалось обеспечить эквивалентность имен, а не структур. Например, объявления

```
struct A { int x, y; };
struct B { int x, y; };
```

определяют два несовместимых типа `A` и `B`. Далее, объявления

```
struct C { int x, y; };    // в файле 1
struct C { int x, y; };    // в файле 2
```

определяют два разных типа с одним и тем же названием `C`, и компилятор, способный осуществлять сквозную проверку в разных единицах компиляции, должен был бы выдать ошибку «повторное определение». Это правило призвано снять некоторые проблемы при сопровождении программы. Подобные повторяющиеся объявления возникают, по всей вероятности, при копировании текста из одного файла в другой. Но после этой операции объявление вполне может измениться. И если, изменив его в одном файле, не сделать то же самое в другом, программа просто перестанет работать.

На практике C, а за ним и C++ гарантируют, что структуры типа `A` и `B`, приведенные выше, одинаково размещаются в памяти, так что их можно приводить друг к другу и использовать очевидным образом:

```
extern f(struct A*);
void g(struct A* pa, struct B* pb)
{
   f(pa);     /* правильно */
   f(pb);     /* ошибка: ожидается A* */

   pa = pb;             /* ошибка: ожидается A* */
   pa = (struct A*)pb;  /* правильно: явное преобразование */
   pb->x = 1;
   if (pa->x != pb->x) error("плохая реализация");
}
```

Эквивалентность имен – основополагающий принцип системы типов в C++, а правила совместимости размещения в памяти гарантируют возможность явных преобразований, используемых в низкоуровневых операциях. В других языках

для этого используется структурная эквивалентность. Я отдаю предпочтение эквивалентности имен, а не структур, потому что считаю такую модель наиболее безопасной. Поэтому мне было приятно узнать, что такое решение не противоречит C и не усложняет предоставление низкоуровневых услуг.

Так появилось «правило одного определения»: каждая функция (переменная, тип, константа и т.д.) должна иметь в C++ ровно одно определение.

2.5.1. Простые реализации

Желание иметь простую реализацию отчасти было вызвано необходимостью (для разработки C with Classes не хватало ресурсов), а отчасти обусловлено недоверием к излишне изощренным языкам и механизмам. Ранняя формулировка одной из целей проектирования C with Classes звучала так: «для реализации языка должно хватать алгоритмов не сложнее линейного поиска». Любое нарушение этого правила – например в случае перегрузки функций (см. раздел 11.2) – приводило к семантике, которая мне казалась слишком сложной. А зачастую и к сложностям при реализации.

Моей целью – памятуя опыт работы с Simula – было спроектировать язык достаточно простой для понимания, чтобы привлечь пользователей, и довольно простой в реализации, чтобы заинтересовать разработчиков компиляторов. С другой стороны, относительно несложная реализация должна была генерировать код, который не уступал бы C в корректности, скорости и величине. Пользователь, незнакомый с языком на практике и находясь в не очень «дружелюбной» среде разработки, тем не менее должен суметь воспользоваться компилятором в реальных проектах. Только при выполнении обоих этих условий можно было рассчитывать, что C with Classes, а позднее C++ выживут в конкуренции с C. Ранняя формулировка принципа звучала так: «C with Classes должен быть неприхотливым «сорняком» вроде C или Fortran, поскольку мы не можем позволить себе ухаживать за такой «розой», как Algol68 или Simula. Если мы создадим компилятор и на год уедем, то по возвращении хотелось бы увидеть его работающим хотя бы на нескольких системах. Этого не произойдет, если будет необходимо постоянное сопровождение или если простой перенос на новую машину займет больше недели».

Данный лозунг был частью философии – воспитывать в пользователях самостоятельность. Всегда, притом явно, ставилась задача вырастить местных экспертов по всем аспектам работы с C++. Кстати, большинство организаций вынуждено следовать противоположной стратегии – культивировать зависимость пользователей от услуг, приносящих доход центральной службе технической поддержки, консультантам или и тем, и другим одновременно. По моему мнению, здесь заключено фундаментальное отличие C++ от многих других языков.

Решение работать в довольно примитивной – и почти повсеместно доступной – среде, обеспечивающей лишь компоновку в стиле C, породило большую проблему, связанную с тем, что компилятор C++ в любой момент времени имел лишь частичную информацию о программе. Каждое предположение относительно программы может стать неверным, если завтра часть этой программы перепишут на каком-то другом языке (C, Fortran или ассемблере) и свяжут с остальными модулями, возможно, уже после того, как программа запущена в эксплуатацию.

Данная проблема имеет многообразные проявления. Компилятору очень трудно гарантировать, что:

- объект, переменная и т.п. уникальны;
- информация непротиворечива (в частности, типы не конфликтуют);
- объект, переменная и т.п. инициализированы.

Кроме того, в C есть лишь самая минимальная поддержка концепции раздельных пространств имен, так что проблемой становится загрязнение пространства имен из-за того, что части программ создаются разными людьми. Развивая C++, мы пытались ответить на все эти вызовы, не принося в жертву фундаментальную модель и технологию, которые обеспечили переносимость и эффективность; но во времена C with Classes мы просто полагались на заголовочные файлы в стиле C.

С принятием в качестве инструмента компоновщика, применяемого для программ на C, связано появление следующего правила: C++ – это просто еще один язык в системе, а не вся система. Другими словами, он выступает в качестве традиционного языка программирования и принимает фундаментальные различия между языком, операционной системой и другими важными компонентами работы программиста. Это ограничивает языковые рамки, что довольно трудно сделать для таких языков, как Smalltalk или Lisp, задумывавшиеся как всеобъемлющие системы или среды. Очень важно, чтобы часть программы на C++ могла вызывать части, написанные на других языках, и сама могла быть вызвана. Кроме того, раз C++ – просто язык, то он может применять инструменты, написанные для других языков.

Тот факт, что язык программирования и написанный на нем код должны быть лишь шестеренкой внутри большого механизма, принципиально важен для большинства пользователей, работающих над промышленными проектами. Тем не менее многие теоретики, педанты и пользователи из академических кругов, очевидно, не учитывали, насколько важно «мирное сосуществование» одного языка с другими и с системами. Я думаю, что в этом одна из основных причин успеха C++.

C with Classes почти совместим с C на уровне исходных текстов. Однако совместимость никогда не была стопроцентной. Например, `class` и `new` допустимы в C в качестве имен идентификаторов, но в C with Classes и его преемниках они являются ключевыми словами. Однако на уровне компоновки совместимость сохранена. Функции на C можно вызывать из C with Classes. Функции на C with Classes можно вызывать из C, а структуры одинаково размещаются в памяти, следовательно, передача как простых, так и составных объектов между функциями, написанными на разных языках, проста и эффективна. Такая совместимость по компоновке сохранена и в C++, за несколькими простыми и явно обозначенными исключениями, которые программист при желании легко может обойти (см. раздел 3.5.1). С годами я и мои коллеги пришли к выводу, что гораздо важнее совместимость на уровне компоновки, чем на уровне исходных текстов. По крайней мере, это верно, когда идентичный исходный код дает одни и те же результаты как в C, так и в C++, или не компилируется, или не связывается на одном из этих языков.

2.5.2. Модель размещения объекта в памяти

Базовая модель объекта имела огромное значение для дизайна C with Classes. Я всегда точно представлял себе, как располагается объект в памяти, и рассматривал воздействие различных языковых средств на объекты. Без понимания эволюции модели объекта нельзя понять эволюцию C++.

В C with Classes объект представлял собой просто C-структуру. Иными словами, объект

```
class stack {
   char s[10];
   char* min;
   char* top;
   char* max;
   void new();
public:
   void push();
   char pop();
};
```

хранился в памяти так же, как структура

```
struct stack {   /* сгенерированный C-код */
   char s[10];
   char* min;
   char* top;
   char* max;
};
```

то есть

Компилятор может добавлять некоторые заполнители до и после членов структуры для выравнивания, но в остальном размер объекта равен сумме размеров членов. Таким образом, расход памяти минимизируется.

Накладные расходы во время исполнения также сведены к минимуму за счет прямого отображения вызова функции-члена

```
void stack.push(char c)
{
   if (top>max) error("стек пуст");
   *top++ = c;
}

void g(class stack* p)
{
   p->push('c');
}
```

на вызов эквивалентной С-функции в сгенерированном коде:

```
void stack__push(this,c)   /* сгенерированный С-код */
struct stack* this;
char c;
{
   if ((this->top)>(this->max)) error("стек пуст");
   *(this->top)++ = c;
}

void g(p) struct stack* p; /* сгенерированный С-код */
{
   stack__push(p,'c');
}
```

В каждой функции-члене указатель `this` ссылается на объект, для которого вызвана функция-член. По воспоминаниям Стью Фельдмана, в самой первой реализации C with Classes программист не мог напрямую обращаться к `this`. Как только мне указали на этот недостаток, я быстро его устранил. Без `this` или какого-то эквивалентного механизма функции-члены нельзя было бы использовать для манипуляций со связанным списком.

Указатель `this` – это вариация C++ на тему ссылки `THIS` из Simula. Иногда меня спрашивают, почему `this` – это указатель, а не ссылка и почему он называется `this`, а не `self`. Дело в том, что, когда `this` был введен в C with Classes, механизма ссылок в нем еще не было, а терминологию C++ я заимствовал из Simula, а не из Smalltalk.

Если бы функция `stack.push()` была объявлена с ключевым словом `inline`, то сгенерированный код выглядел бы так:

```
void g(p)    /* сгенерированный С-код */
struct stack* p;
{
   if ((p->top)>(p->max)) error("стек пуст");
   *(p->top)++ = 'c';
}
```

Именно такой код написал бы программист, работающий на С.

2.6. Статический контроль типов

У меня не сохранилось ни воспоминаний о дискуссиях, ни заметок, касающихся внедрения статического контроля типов в C With Classes. Синтаксис и семантика, впоследствии одобренные в стандарте ANSI C, просто были включены в готовом виде в первую реализацию C with Classes. После этого в результате серии экспериментов появились более строгие правила, ныне вошедшие в C++. Для меня статический контроль типов после опыта работы с Algol68 и Simula был абсолютной необходимостью, вопрос заключался только в том, в каком виде его реализовать.

Чтобы не возникло конфликта с С, я решил разрешить вызовы необъявленных функций и не проверять для них типы. Конечно, это была «дыра» в системе типов, впоследствии предпринимались попытки устранить основной источник

ошибок и уже в C++ проблему решили путем запрета на вызовы необъявленных функций. Одно наблюдение обрекло на провал все попытки найти компромисс и тем самым обеспечить большую совместимость с C: программисты, изучавшие C with Classes, забывали, как искать ошибки времени исполнения, вызванные несоответствием типов. Привыкнув полагаться на проверку и преобразования типов, которые обеспечивал C with Classes, пользователи уже не могли быстро отыскать глупейшие ляпы, просачивавшиеся в C-программы, где контроля типов не было. К тому же забывалось о мерах предосторожности против таких ошибок, которые опытные программисты на C принимают, не задумываясь. Ведь «в C with Classes такого просто не бывает». Таким образом, число ошибок, вызванных необнаруженным несоответствием типов аргументов, становится меньше, но их серьезность и время, потраченное на их поиск, возрастают. А в результате – недовольство программистов и требование ужесточить систему контроля типов.

Самым интересным экспериментом с «неполным статическим контролем типов» была попытка разрешить вызовы необъявленных функций, но запоминать типы переданных аргументов, с тем чтобы при повторном вызове их можно было проверить. Когда Уолтер Брайт (Walter Bright) много лет спустя независимо открыл этот способ, он назвал его автопрототипированием, вспомнив употребляемый в ANSI C термин прототип, обозначающий объявление функции. Опыт показал, что автопрототипирование позволяло отловить многие ошибки, и поначалу доверие программистов к системе контроля типов повысилось. Однако ошибки в функции, которая вызывалась только один раз, таким способом не обнаруживались, и из-за этого автопрототипирование окончательно подорвало веру в проверку типов и привело к еще худшей навязчивой идее, нежели все, с чем я встречался в C или BCPL.

В C with Classes введена нотация `f(void)` для функции `f`, не принимающей аргументов, вместо нотации `f()` в C, обозначающей функцию с любым числом аргументов без проверки типов. Однако вскоре пользователи убедили меня, что нотация `f(void)` неизящна, а объявление с помощью `f()` функции, которая может принимать аргументы, далеко от интуитивно очевидного. Поэтому после некоторых экспериментов было решено оставить нотацию `f()` для функции, не принимающей никаких аргументов, ибо именно этого ожидает неискушенный пользователь. Чтобы отважиться на такой разрыв с традициями C, мне потребовалась поддержка со стороны Дуга Макилроя и Денниса Ричи. Только после того как они охарактеризовали нотацию `f(void)` словом «гадость», я решил придать `f()` очевидную семантику. Однако и по сей день правила контроля типов в C гораздо слабее, чем в C++, а комитет по стандартизации ANSI C одобрил «отвратительную `f(void)`», впервые появившуюся в C with Classes.

2.6.1. Сужающие преобразования

Еще одной ранней попыткой усилить контроль типов в C with Classes был запрет на неявные преобразования, приводящие к потере информации. Как и многие другие, я не раз попадался в ловушку, иллюстрируемую следующими примерами (конечно, в реальных программах найти такую ошибку будет потруднее):

```
void f()
{
   long int lng = 65000;
```

```
    int i1 = lng; /* i1 становится отрицательным (-536)  */
                  /* на машинах с 16-разрядными целыми   */
    int i2 = 257;
    char c = i2;  /* отбрасывание: c становится равным 1 */
                  /* на машинах с 8-разрядным char       */
}
```

Я решил попробовать запретить все преобразования, не сохраняющие значение, то есть потребовать явного оператора преобразования в случаях, когда больший объект копируется в меньший:

```
void g(long lng, int i)/* эксперимент */
{
    int i1 = lng;     /* ошибка: преобразование понижает тип */
    i1 = (int)lng;    /* отбрасывание для 16-разрядных целых */

    char c = i;       /* ошибка: преобразование понижает тип */
    c = (char)i;      /* отбрасывание                        */
}
```

Эксперимент с треском провалился. Все С-программы, которые я просмотрел, содержали множество присваиваний значений типа `int` переменным типа `char`. Разумеется, коль скоро программы работали, большинство таких присваиваний было безопасно: либо значение было маленьким и не обрезалось, либо отбрасывание старших разрядов предполагалось или, по крайней мере, в данном контексте считалось безвредным. Пользователи C with Classes вовсе не желали такого отхода от С. Я все еще пытаюсь найти какой-то способ разрешить эти проблемы (см. раздел 14.3.5.2).

2.6.2. О пользе предупреждений

Я рассматривал возможность проверки присваиваемых значений во время исполнения, но за это пришлось бы заплатить серьезным уменьшением скорости и увеличением размера кода. К тому же, на мой взгляд, диагностировать ошибку было уже поздновато. Проверки преобразований во время исполнения – да и любые другие проверки на данной стадии – были отнесены к разряду «идей о будущей поддержке в отладочной среде». Вместо этого я применил способ, позже ставший стандартным для обхождения с теми недостатками С, которые, на мой взгляд, были слишком серьезными, чтобы игнорировать их, и слишком укоренившимися в С, чтобы их устранять. Я заставил препроцессор C with Classes (а позже и свой компилятор C++) выдавать предупреждения:

```
void f(long lng, int i)
{
    int i1 = lng;     // неявное преобразование: предупреждение
    i1 = (int)lng;    // явное преобразование: нет предупреждения

    char c = i;       // слишком устойчивая идиома: нет предупреждения
}
```

И тогда, и сейчас безусловные предупреждения выдавались для преобразований `long→int` и `double→int`, поскольку я не вижу причин для признания их законными. Это просто результат определенного недоразумения – арифметика с плавающей точкой появилась в C до введения явных преобразований. Такие предупреждения устраивали пользователей. Более того, меня самого и многих других они не раз выручали. С преобразованием же `int→char` я ничего не мог сделать. И по сей день такие преобразования компилятор AT&T C++ пропускает без предупреждений.

Я решил выдавать безусловные предупреждения только в случаях, когда «вероятность ошибки больше 90%», причем был учтен опыт компилятора C и верификатора Lint, которые чаще, чем хотелось бы, выдавали «ложные» предупреждения относительно того, что не мешало программе корректно работать. Поэтому программисты либо вообще игнорировали предупреждения компилятора, либо все же смотрели на них, но добрых чувств явно не испытывали. Итак, предупреждения используются для того, чтобы как-то компенсировать сложности, которые не удается устранить путем изменения языка из-за требования совместимости с C, а также для облегчения перехода с C на C++. Вот пример:

```
class X {
   // ...
}

g(int i, int x, int j)
      // предупреждение: class X определен как тип значения,
      // возвращаемого g() (вы не забыли ';' после '}' ?)
      // предупреждение: j не используется
{
   if (i = 7) {  // предупреждение: присваивание константы в условии
      // ...
   }
      // ...
   if (x&077 == 0) {    // предупреждение: выражение == как операнд
                        // для &
      // ...
   }
}
```

Даже первая версия Cfront (см. раздел 3.3) выдавала такие предупреждения. Они появились в результате осознанного проектного решения, а не были добавлены потом.

Много позже первое из этих предупреждений превратилось в ошибку – был введен запрет на определение новых типов в типах возвращаемых значений и аргументов.

2.7. Почему C?

На презентациях C with Classes меня часто спрашивали: «Почему Вы использовали C? Почему не Pascal?» Один из вариантов ответа можно найти в работе [Stroustrup, 1986c]:

«C, конечно, не самый чистый язык из всех существующих и не самый простой для применения, так почему же так много людей им пользуются? Вот причины:

- гибкость. C применим к большинству предметных областей и позволяет использовать почти любую технику программирования. В языке нет внутренних ограничений, не дающих написать программу определенного вида;
- эффективность. Семантика языка «низкоуровневая», то есть фундаментальные понятия C непосредственно отображаются на фундаментальные понятия традиционного компьютера. Следовательно, можно без особого труда эффективно воспользоваться аппаратными ресурсами из C-программы;
- доступность. Возьмите любой компьютер, от крохотной микромашины до огромной супер-ЭВМ. Почти наверняка для него найдется компилятор C достойного качества, и он будет поддерживать стандарт языка и библиотек. Библиотеки и инструментальные средства также имеются, поэтому программисту редко приходится проектировать новую систему с нуля;
- переносимость. Конечно, написанная на C программа не переносится на другую машину или операционную систему автоматически. Может быть, осуществить данную процедуру даже не очень просто. Но обычно это получается, и перенос даже очень больших программных систем, принципиально машинно-зависимых, технически и экономически реален.

По сравнению с этими важными преимуществами недостатки C – например, причудливый стиль объявлений или небезопасность некоторых языковых конструкций – второстепенны и уже не столь существенны. При проектировании «лучшего C» необходимо компенсировать проблемы, встречающиеся при написании, отладке и сопровождении кода, не жертвуя при этом достоинствами C. C++ сохраняет все преимущества C и совместим с ним, пусть даже и не идеален и имеет несколько усложненные язык и компилятор. Однако проектирование языка с нуля не обязательно гарантирует его совершенство, а компиляторы C++ не уступают компиляторам C по скорости и качеству кода и превосходят их в области обнаружения и диагностики ошибок».

Во время работы над C with Classes я еще не сумел бы дать такую отшлифованную формулировку, но она правильно отражает суть того, что я и тогда считал важным в C и не хотел терять в C with Classes. Pascal считался игрушечным языком [Kernighan, 1981], поэтому более простым казалось добавить контроль типов к C, чем включать в Pascal возможности, необходимые для системного программирования. В то время я очень боялся совершить ошибки, в результате которых дизайнер – из-за неправильно понятого патернализма или просто по незнанию – создает язык, непригодный для реальной работы в важных областях. Прошедшие десять лет показали, что, взяв за основу C, я сумел остаться в основном русле системного программирования, как и намеревался. Плата – сложность языка – оказалась велика, но не непомерна.

В качестве альтернативы C и источника идей для C++ я рассматривал [Stroustrup, 1984c] такие языки, как Modula-2, Ada, Smalltalk, Mesa [Mitchell, 1979] и Clu. Но лишь только C, Simula, Algol68 и в одном случае BCPL оставили заметный след в C++ образца 1985 г. Из Simula я позаимствовал классы, из Algol68 – перегрузку операторов (см. раздел 3.6), ссылки (см. раздел 3.7) и возможность объявлять переменные в любом месте блока (см. раздел 3.11.5), а у BCPL – обозначение // для комментариев (см. раздел 3.11.1).

Были и другие причины избегать резких расхождений со стилем C. Достаточно серьезным вызовом мне представлялось уже само объединение сильных сторон C как

языка системного программирования с возможностями Simula по организации программ. Добавление нетривиальных возможностей из других источников легко могло бы привести к появлению конгломерата типа «чего изволите» вместо целостного языка. Приведу цитату из работы [Stroustrup, 1986]:

«Перед языком программирования стоят две взаимосвязанные задачи: дать программисту средство для описания действий, которые нужно выполнить, и предоставить набор понятий, пользуясь которыми он мог бы размышлять о том, что надо сделать. В идеале для решения первой задачи нужен язык, «близкий к машине», на котором все важнейшие аппаратные особенности выражаются просто и эффективно и в то же время достаточно понятно для программиста. Язык C в основном для этого и проектировался. Для решения второй задачи идеально подошел бы язык, «близкий к решаемой проблеме», в котором непосредственно и без искажений выражались бы понятия предметной области. Возможности, добавленные к C в C++, проектировались исходя именно из этой цели».

Как и в предыдущем случае, я бы не сумел дать такую формулировку во время ранних этапов работы над C with Classes, но общая идея понятна. Отход от известных и проверенных временем методов программирования на C и Simula следовало отложить до приобретения достаточного опыта работы с C with Classes и C++. Еще предстояло провести немало экспериментов. Я верю и верил, что проектирование языка – не просто следование заранее определенным принципам, а искусство, требующее опыта, экспериментирования и компромиссов. Включение в язык важной особенности должно быть обдуманным действием, основанным на опыте. Новой возможности необходимо сочетаться с уже имеющимися и не противоречить представлениям о том, как может использоваться язык. На развитие C++ после 1985 г. влияли Ada (шаблоны, см. главу 15; исключения, см. главу 16; пространства имен, см. главу 17), Clu (исключения, см. главу 16) и ML (исключения, см. главу 16).

2.8. Проблемы синтаксиса

Мог ли я устранить наиболее досадные недостатки синтаксиса и семантики языка C до того, как C++ стал общедоступным языком программирования? Мог ли я добиться этого, не отбросив в сторону некоторые полезные свойства, затрагивающие пользователей C with Classes, «утонувших» в своей среде, которую часто противопоставляют идеальному миру, или сделать этот язык несовместимым с другими, что неприемлемо для программистов на C, желающих перейти на C with Classes?

Думаю, нет. В некоторых случаях я пытался устранить недостатки, но вынужден был отказаться от внесенных изменений, получив от возмущенных пользователей множество нареканий.

2.8.1. Синтаксис объявлений в языке C

Больше всего в C мне не нравился синтаксис объявлений. Наличие одновременно префиксных и постфиксных операторов объявлений – источник многих недоразумений. Например:

```
int *p[10];  /* массив из 10 указателей на int или */
             /* указатель на массив из 10 int?     */
```

Разрешение опускать спецификатор типа (по умолчанию считается, что это `int`) также приводит к сложностям. Например:

```
                /* стиль C (предлагалось запретить): */
static a;       /* неявно: тип 'a' равен int */
f();            /* неявно: возвращает int    */
                // предлагалось в C with Classes:
static int a;
int f();
```

Негативная реакция пользователей на изменения в этой области была очень сильна. Они настолько ценили «краткость» C, что отказывались пользоваться языком, который требовал явно писать спецификаторы типов. Я отказался от изменения. Не думаю, что здесь вообще был выбор. Разрешение неявного `int` – источник многих проблем в грамматике C++. Замечу, что давление шло со стороны пользователей, а не административных органов или кабинетных экспертов по языкам. Наконец, спустя десять лет, комитет ANSI/ISO по стандартизации C++ (см. главу 6) решился запретить неявный `int`. А значит, еще через десяток лет мы, возможно, от него и избавимся. Но с помощью инструментальных средств или предупреждений компилятора отдельные пользователи уже сейчас могут начать защищаться от вызванных неявным `int` недоразумений, например, таких:

```
void f(const T); // константный аргумент типа T или
                 // аргумент с именем T типа const int?
                 // (на самом деле первое)
```

Однако синтаксис определения функции с указанием типов аргументов внутри скобок использовался в C with Classes и C++ и позднее был одобрен комитетом ANSI C:

```
f(a,b) char b;   /* определение функции в стиле K&R   */
{
   /* ... */
}

int f(int a, char b)    // определение функции в стиле C++
{
   // ...
}
```

Я рассматривал и возможность введения линейной нотации для объявлений. В C применяется прием, когда объявление имени имитирует его использование, а в результате получаются объявления, которые трудно читать и записывать, и шансы, что человек или программа спутают объявление и выражение, растут. Неоднократно отмечалось: проблема с синтаксисом объявлений в C заключается в том, что оператор объявления `*` («указатель на») – префиксный, а операторы объявления `[]` («массив») и `()` («возвращающая функция») – постфиксные. Это заставляет использовать скобки для устранения двусмысленности:

```
                /* стиль C                          */
int* v[10];     /* массив указателей на int         */
int (*p)[10];   /* указатель на массив int          */
```

Вместе с Дугом Макилроем, Эндрю Кенигом, Джонатаном Шопиро и другими я решил ввести постфиксный оператор «указатель на» -> как альтернативу префиксному *:

```
                    // радикальное изменение:
v: [10]->int ;      // массив указателей на int
p: ->[10]int;       // указатель на массив int

                    // менее радикальное изменение:
int v[10]->;        // массив указателей на int
int p->[10];        // указатель на массив int
```

Менее радикальный вариант имел то преимущество, что постфиксный оператор объявления -> мог сосуществовать с префиксным оператором * в течение переходного периода. А по его завершении оператор объявления * и лишние скобки можно было бы просто убрать из языка. Польза от этой схемы заметна: скобки потребовались бы только для записи объявления функции, поэтому возможность путаницы и грамматические тонкости просто исчезли бы (см. также [Sethi, 1981]). Если бы все операторы объявления были постфиксными, то объявления могли бы читаться слева направо. Например,

```
int f(char)->[10]->(double)->;
```

означало бы функцию `f`, возвращающую указатель на массив указателей на функции, возвращающие указатель на `int`. Попробуйте написать это на стандартном C/C++! К сожалению, я не довел данную идею до конца и так и не сделал полную реализацию. Пользователи вынуждены строить определения сложных типов с помощью `typedef`:

```
typedef int* DtoI(double); // функция, принимающая double
                           // и возвращающая указатель на int
typedef DtoI* V10[10];     // массив из 10 указателей на DtoI
V10 * f(char);             // f принимает char и возвращает указатель
                           // на V10
```

В конце концов, я решил оставить все как есть, обосновав это тем, что любое изменение синтаксиса лишь увеличит (по крайней мере, временно) путаницу. И хотя старый синтаксис – сущий подарок для любителей придираться по пустякам и желающих высмеивать C, для программистов на C он большой проблемы не составляет. Все же я не уверен, что поступил правильно. Ведь от нелогичного синтаксиса C страдаю и я, и другие разработчики компиляторов C++, составители документации, разработчики инструментальных средств. Пользователи, конечно, могут изолировать себя от таких проблем, довольствуясь только малым, простым для понимания подмножеством синтаксиса объявлений C/C++ (см. раздел 7.2), собственно, так они и поступают.

2.8.2. Тэги структур и имена типов

Для удобства пользователей в C++ было введено одно синтаксическое упрощение, потребовавшее, правда, дополнительной работы для разработчиков компиляторов и вызвавшее некоторые проблемы совместимости с C. В C имени

структуры, так называемому тэгу, всегда должно предшествовать ключевое слово `struct`. Например:

```
struct buffer a; /* в C 'struct' необходимо */
```

В C with Classes меня это не устраивало, поскольку получалось, что синтаксически определенные пользователем типы можно было отнести ко «второму сорту». Но, потерпев неудачу с другими попытками изменения синтаксиса, я не хотел нарываться на неприятности еще раз и внес это изменение (когда C with Classes превратился в C++) только по настоятельному требованию Тома Каргилла. Имя структуры, объединения и класса в C++ – просто имя типа и не требует специального обозначения:

```
buffer a;     // C++
```

«Битвы на поле совместимости с C» продолжались несколько лет (см. также раздел 3.12). Например, следующая запись допустима в C:

```
struct S { int a; };
int S;
void f(struct S x)
{
   x.a = S;       // S - переменная типа int
}
```

Она также допустима и в C with Classes, и в C++, но в течение многих лет мы искали формулировку, которая в целях совместимости допускала бы такой немного странный, но безвредный код в C++. Ведь, если бы это было разрешено, следовало запретить следующее:

```
void g(S x)       // ошибка: S - переменная типа int
{
   x.a = S;
}
```

Насущная необходимость решить данную проблему следовала из того, что в некоторых заголовочных файлах для системы UNIX, прежде всего в `stat.h` встречались и структура, и переменная или функция с одним и тем же именем. Такие вопросы совместимости немаловажны, а для языковых пуристов это особо лакомый кусочек. К сожалению, пока не найдено удовлетворительное (и как правило, тривиальное) решение и данным вопросам приходится уделять слишком много внимания и сил. А когда решение найдено, проблема совместимости становится просто скучной, поскольку в ней нет никакой интеллектуальной ценности, а есть лишь голый практицизм. Принятое в C++ решение проблемы множественных пространств имен в C состоит в том, что одним именем может обозначаться как класс, так и переменная или функция. Таким образом, имя относится не к классу, если только оно явно не уточнено одним из ключевых слов `struct`, `class` или `union`.

Споры с упрямыми старыми пользователями C, так называемыми экспертами, а также реальные проблемы совместимости с C были и остаются одним из самых трудных и вызывающих разочарование аспектов разработки C++.

2.8.3. Важность синтаксиса

Я полагаю, что большинство людей уделяет слишком много внимания синтаксису в ущерб вопросам контроля типов. Однако для дизайна C++ критичными всегда были проблемы типизации, недвусмысленности и контроля доступа, а вовсе не синтаксиса.

Не хочу сказать, что синтаксис не имеет никакого значения. Напротив, он исключительно важен. Удачный синтаксис помогает программисту при изучении новых концепций и позволяет избегать глупых ошибок, поскольку записать их труднее, чем их правильные альтернативы. Однако синтаксис надо проектировать так, чтобы он соответствовал семантическим понятиям языка, а не наоборот. Отсюда следует, что центральной темой при обсуждении языка должен быть вопрос о том, что можно выразить, а не как это сделать.

Тонким нюансом в дискуссиях о совместимости с C было то, что опытные C-программисты давно свыклись с тем, как та или иная операция осуществлялась раньше, и потому были совершенно нетерпимы к несовместимостям, которые требовались для поддержания такого стиля программирования, который в дизайне C не был заложен. И наоборот, программисты, не писавшие на C, обычно недооценивают значимость синтаксиса для C-программистов.

2.9. Производные классы

Концепция производного класса – это адаптация префиксной нотации классов, принятой в Simula, и уже поэтому она схожа с концепцией подкласса в Smalltalk. Названия «производный» и «базовый» были выбраны потому, что я никак не мог запомнить, что есть sub, а что – super, и заметил, что такие сложности возникают не только у меня. Вот еще одно наблюдение: для многих тот факт, что подкласс обычно содержит больше информации, чем его суперкласс, кажется неестественным. Изобретая термины «производный» и «базовый», я отошел от своего принципа не вводить новых названий, если существуют старые. В свою защиту скажу лишь, что я никогда не замечал путаницы по поводу этих терминов и что запомнить их очень просто даже тем, что не имеет математического образования.

В C with Classes для концепции производного класса не было никакой поддержки во время исполнения. В частности, отсутствовало понятие виртуальной функции, которое было в Simula, а позже вошло в C++. Я просто сомневался (думаю, не без оснований) в своей способности научить людей ими пользоваться и, более того, в способности убедить, что для типичных применений виртуальная функция так же эффективна по скорости и по памяти, как обычная. Многие пользователи, имеющие опыт работы с Simula или Smalltalk, до сих пор в это не верят, пока им не объяснишь подробно, как это сделано в C++, а некоторые и после испытывают сомнения.

Даже и без виртуальных функций производные классы в C with Classes были полезны для построения новых структур данных из старых и для ассоциирования операций с получающимися типами. Так, с их помощью удалось определить классы связанного списка и задачи (task).

2.9.1. Полиморфизм без виртуальных функций

Даже в отсутствие виртуальных функций пользователь мог работать с объектами производного класса, а устройство базового класса трактовать как деталь реализации. Например, если имеется класс вектора с элементами, проиндексированными начиная с 0 и без проверки выхода за границы диапазона

```
class vector {
   /* ... */
   int get_elem(int i);
};
```

то на его основе можно построить вектор элементов с индексами из заданного диапазона с контролем выхода за его границы:

```
class vec : vector {
   int hi, lo;
public:
   /* ... */
   new(int lo, int hi);
   get_elem(int i);
};
int vec.get_elem(int i)
{
   if (i < lo || hi < i) error("выход за границы диапазона");
   return vector.get_elem(i - lo);
}
```

Можно было вместо этого ввести в базовый класс явное поле типа и использовать его совместно с явным приведением типов. Первая стратегия применялась, когда пользователь видел только конкретные производные классы, а система видела все их базовые классы. Вторая стратегия полезна тогда, когда базовый класс реализовывал по сути дела одну вариантную запись для множества производных классов.

Например, в работе [Stroustrup, 1982b] приведен следующий уродливый код для извлечения объекта из таблицы и использования имеющегося в нем поля типа:

```
class elem { /* свойства, хранящиеся в таблице */ };
class table { /* табличные данные и функции поиска */ };

class cl_name * cl;     /* cl_name - производный от elem */
class po_name * po;     /* po_name - производный от elem */
class hashed * table;   /* hashed - производный от table */

elem * p = table->look("carrot");

if (p) {
   switch (p->type) { /* поле типа type в объектах класса elem */
   case PO_NAME:
      po = (class po_name *) p;      /* явное преобразование типа */
```

```
        ...
        break;
    case CL_NAME:
        cl = (class cl_name *) p;      /* явное преобразование типа */
        ...
        break;
    default:
        error("неизвестный тип элемента");
    }
}
else
    error("carrot не определено");
```

При проектировании C with Classes и C++ было потрачено много усилий ради того, чтобы программисту не приходилось писать такой код.

2.9.2. Контейнерные классы без шаблонов

В то время в моих программах и размышлениях важное место занимал вопрос о том, как с помощью комбинирования базовых классов, явных преобразований типов и (изредка) макросов можно получить обобщенные контейнерные классы. Например, в работе [Stroustrup, 1982b] продемонстрировано, как из списка объектов типа `link` построить список, который может содержать объекты одного типа:

```
class wordlink : link
{
    char word[SIZE];
public:
    void clear(void);
    class wordlink * get(void);
        { return (class wordlink *) link.get(); };
    void put(class wordlink * p) { link.put(p); };
};
```

Поскольку каждый объект `link`, который с помощью функции `put()` помещается в список, должен принадлежать классу `wordlink`, то обратное приведение к `wordlink` объекта типа `link`, извлекаемого из списка функцией `get()`, безопасно. Отмечу использование закрытого (по умолчанию, в отсутствие ключевого слова `public` в спецификации базового класса `link`; см. раздел 2.10) наследования. Если разрешить использовать `wordlink` как просто `link`, то была бы поставлена под угрозу безопасность работы с типами.

Для обеспечения обобщенных типов использовались макросы. Вот как об этом написано в [Stroustrup, 1982b]:

«В примере класса `stack`, упомянутом во введении, явно определялся стек символов. Иногда это оказывается слишком частным случаем. А что, если нужен еще и стек длинных целых? А если класс `stack` предполагается использовать в библиотеке, когда тип элемента, помещаемого в стек, заранее неизвестен? В таких случаях объявление класса `stack` и ассоциированных с ним функций следует написать так, чтобы при создании стека тип элементов можно было передать параметром так же, как размер стека.

Для этого в языке нет прямой поддержки, но того же эффекта можно достигнуть с помощью стандартного препроцессора C. Например:

```
class ELEM_stack {
   ELEM * min, * top, * max;
   void new(int), delete(void);
public:
   void push(ELEM);
   ELEM pop(void);
};
```

Это объявление можно поместить в заголовочный файл и с помощью макросов расширять для каждого типа ELEM, который предполагается использовать:

```
#define ELEM long
#define ELEM_stack long_stack
#include "stack.h"
#undef ELEM
#undef ELEM_stack

typedef class x X;
#define ELEM X
#define ELEM_stack X_stack
#include "stack.h"
#undef ELEM
#undef ELEM_stack

class long_stack ls(1024);
class long_stack ls2(512);
class X_stack xs(512);
```

Конечно, этот способ далек от совершенства, зато прост».

Это был один из самых ранних и самых грубых методов. Для использования в реальных программах он не годился, так как был причиной слишком многих ошибок, поэтому вскоре я определил несколько «стандартных» макросов, «склеивающих лексемы», и порекомендовал применять их для реализации обобщенных классов [Stroustrup, 1986, §7.3.5]. В конечном итоге отсюда взяли свое начало шаблоны C++ и техника их совместного использования с базовыми классами для выражения общих свойств инстанцированных шаблонов (см. раздел 15.5).

2.9.3. Модель размещения объекта в памяти

Реализация производного класса заключалась в объединении его членов и членов базового классов. Например, если есть объявления

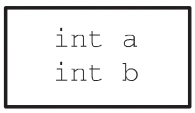
```
class A {
   int a;
public:
   /* функции-члены */
};
```

```
class B : public A {
   int b;
public:
   /* функции-члены */
};
```

то объект класса B будет представлен структурой

```
struct B {       /* сгенерированный C-код */
   int a;
   int b;
};
```

то есть



Конфликты между именами членов базового и производного классов устранялись самим компилятором, который присваивал им подходящие индивидуальные имена. Вызовы обрабатывались так, как если бы никакого наследования не было. По сравнению с C вовсе не возникало дополнительного расхода времени или памяти.

2.9.4. Ретроспектива

Было ли разумно избегать виртуальных функций в C with Classes? Да, язык и без них был полезен, а их отсутствие позволило на некоторое время отложить длительные дебаты о полезности, правильном использовании и эффективности. За это время были разработаны такие языковые механизмы и методы программирования, которые оказались кстати даже при наличии более мощных механизмов наследования. Эти механизмы и методы использовались как противовес стремлению некоторых программистов применять исключительно наследование, как будто больше в языке ничего не было (см. раздел 14.2.3). В частности, классы использовались для реализации таких конкретных типов, как `complex` и `string`, и интерфейс классов завоевал популярность. Класс `stack`, реализованный как интерфейс к более общему классу `dequeue`, – это пример наследования без виртуальных функций.

Нужны ли были виртуальные функции в C with Classes для решения тех задач, на которые он был нацелен? Да, поэтому они и были включены в первое из важнейших расширений при переходе к C++.

2.10. Модель защиты

Перед тем как приступить к созданию C with Classes, я работал с операционными системами. Происхождение механизмов защиты в C++ следует искать в концепции защиты в Кембриджском компьютере CAP и аналогичных системах, а не в каких-то работах по языкам программирования. Единицу защиты составляет

класс, и фундаментальное правило гласит, что нельзя предоставить доступ к классу самому себе. Только явные объявления, помещенные в объявление класса (предположительно его создателем), могут разрешить доступ. По умолчанию вся информация закрыта.

Для предоставления доступа к члену его следует поместить в открытой (public) части объявления класса либо объявить функцию или класс дружественными с помощью ключевого слова `friend`. Например:

```
class X {
   /* представление */
public:
   void f();         /* функция-член, имеющая доступ к представлению */

   friend void g(); /* глобальная функция, имеющая доступ к  */
                    /* представлению                        */
};
```

Первоначально дружественными могли быть только классы, то есть всем функциям-членам одного класса предоставлялся доступ ко всем членам другого класса, в объявлении которого первый класс имел спецификатор `friend`. Но позже было сочтено, что удобно предоставлять такой доступ и отдельным функциям, в частности, глобальным (см. также раздел 3.6.1).

Объявление дружественности виделось как механизм, аналогичный доменам защиты (protection domain), предоставляющим право чтения-записи другим объектам. Это явная часть объявления класса. Поэтому я никогда не усматривал в повторяющихся утверждениях о том, что объявление `friend`, дескать, «нарушает инкапсуляцию», ничего, кроме невежества и путаницы в терминологии.

Даже в первой версии C with Classes модель защиты применялась как к базовым классам, так и к членам. Это означает, что производный класс мог наследовать базовому открыто или закрыто. Введение открытого и закрытого наследования базовых классов примерно на пять лет опережает дебаты по поводу наследования интерфейса и реализации [Snyder, 1986], [Liskov, 1987]. Если вы хотите наследовать только реализацию, пользуйтесь в C++ закрытым наследованием. Открытое наследование дает пользователям производного класса доступ ко всему интерфейсу, предоставляемому базовым классом. При закрытом наследовании базовый класс можно считать деталью реализации, даже его открытые члены недоступны иначе как через интерфейс, предоставленный производным классом.

Для того чтобы иметь «полупрозрачные области действия», был предложен механизм, разрешающий доступ к отдельным открытым членам из закрыто наследуемого базового класса [Stroustrup, 1982b]:

```
class vector {
   /* ... */
public:
   /* ... */
   void print(void);
};
```

```
class hashed : vector  /* vector - закрыто наследуемый базовый  */
                       /* класс для hashed */
{
   /* ... */
public:
   vector.print; /* полупрозрачная область действия */
                 /* другие функции из класса vector не */
                 /* могут применяться к объектам класса hashed */
   /* ... */
};
```

Синтаксически для того, чтобы сделать недоступное имя доступным, нужно просто назвать его. Это пример абсолютно логичного, минимального и недвусмысленного синтаксиса. Но при этом он малопонятен; почти любой другой вариант был бы лучше. Указанная синтаксическая проблема ныне решена с помощью using-объявлений (см. раздел 17.5.2).

В книге [ARM] следующим образом резюмирована концепция защиты в C++:

- защита проверяется во время компиляции и направлена против случайных, а не преднамеренных или явных попыток ее преодоления;
- доступ предоставляется классом;
- контроль прав доступа выполняется для имен и не зависит от типа именованной сущности;
- единицей защиты является класс, а не отдельный объект;
- контролируется доступ, а не видимость.

Все это было верно и в 1980 г., хотя теперь терминология слегка изменилась. Последний пункт проще объяснить на примере:

```
int a;        // глобальная переменная a

class X {
private:
   int a;     // член X::a
};
class XX : public X {
   void f() { a = 1; } // какое a?
};
```

Если бы контролировалась видимость, то член `X::a` был бы невидим и `XX::f()` ссылалась бы на глобальную переменную `a`. На самом деле в C with Classes и C++ считается, что глобальная `a` скрыта за недоступным `X::a`, поэтому `XX::f()` приводит к ошибке компиляции из-за попытки получить доступ к недоступному члену `X::a`. Почему я ввел именно такое определение и был ли прав? Я не очень хорошо помню ход своих мыслей, а в сохранившихся записях об этом ничего нет. Припоминаю только один разговор на эту тему. Для приведенного выше примера принятое правило гарантирует, что `f()` всегда будет ссылаться на одно и то же, какой бы доступ ни был объявлен для `X::a`. Если бы ключевые слова `public` и `private` контролировали видимость, а не доступ, то изменение `public` на `private` без предупреждения изменило бы семантику программы (вместо `X::a`

использовалось бы глобальное a). Я больше не считаю этот аргумент решающим (если и считал раньше), но такой ход оказался полезным, поскольку позволяет программисту добавлять и удалять во время отладки public и private, не меняя смысла программы. Любопытно, была ли эта сторона определения C++ осознанным решением. Может быть, это просто побочный результат препроцессорной технологии, использовавшейся во времена C with Classes, который не пересматривался при реализации настоящего компилятора во время перехода к C++ (см. раздел 3.3).

Другой аспект механизма защиты в C++, свидетельствующий о влиянии операционных систем, – это отношение к нарушению правил. Я думаю, что любой компетентный программист может обойти любое правило, которое не поддержано аппаратно, так что не стоит даже пытаться защититься от мошенничества [ARM]:

«Механизмы контроля доступа в C++ обеспечивают защиту от случайности, а *не* от преднамеренного обмана. Любой язык программирования, который поддерживает прямой доступ к памяти, обязательно оставляет любой элемент данных открытым для сознательного «жульничества», нарушающего явные правила типов, сформулированные для этого элемента».

Задача механизма защиты – гарантировать, что любое такое действие в обход системы типов выражено явно, и свести к минимуму необходимость в таких нарушениях.

Пришедшая из операционных систем концепция защиты от чтения/записи превратилась в C++ в понятие о const (см. раздел 3.8).

За прошедшие годы было много предложений о том, как предоставить доступ к единице, меньшей, чем целый класс. Например:

```
grant X::f(int) access to Y::a, Y::b, and Y::g(char);
```

Мне не нравились все эти предложения, потому что более скрупулезный контроль не дает никакой дополнительной защиты. Любая функция-член может модифицировать любой член-данные класса, поэтому функция, которой предоставлен доступ к функции-члену, может косвенно модифицировать любой член. Я считал и считаю, что усложнение спецификации, реализации и использования не перевешивает преимуществ более явного контроля доступа.

2.11. Гарантии времени исполнения

Описанные выше механизмы контроля доступа просто предотвращают неразрешенный доступ. Другой вид гарантий предоставляется такими «специальными функциями-членами», как конструкторы, которые компилятор распознает и неявно вызывает. Идея заключалась в том, чтобы дать программисту средства формулировать гарантированные условия (их еще называют инвариантами), на выполнение которых другие функции-члены могли бы рассчитывать (см. также раздел 16.10).

2.11.1. Конструкторы и деструкторы

В то время я часто объяснял эту концепцию так: функция new (конструктор) создает среду, в которой работают другие функции-члены, а функция delete

(деструктор) эту среду уничтожает и освобождает выделенные для нее ресурсы. Например:

```
class monitor : object {
   /* ... */
public:
   new()  { /* создать защелку для монитора */ }
   delete()  { /* освободить и удалить защелку */ }
   /* ... */
};
```

См. также разделы 3.9 и 13.2.4.

Откуда возникла концепция конструктора? Подозреваю, что ее изобрел я сам. Я был знаком с механизмом инициализации объекта класса в Simula. Однако я смотрел на объявление класса прежде всего как на определение интерфейса, так что хотел избежать помещения в него кода. Поскольку в C with Classes вслед за C было три класса хранения, то какой-то вид функций инициализации почти неизбежно должен был распознаваться компилятором (см. раздел 2.11.2). Скоро выяснилось, что было бы полезно иметь несколько конструкторов. Это наблюдение послужило причиной появления механизма перегрузки в C++ (см. раздел 3.6).

2.11.2. Распределение памяти и конструкторы

Как и в C, память для объектов можно выделять тремя способами: из стека (автоматическая память), по фиксированному адресу (статическая память) и из свободной памяти (куча или динамическая память). В любом случае для созданного объекта должен быть вызван конструктор. В C размещение объекта в свободной памяти требует только вызова функции распределения. Например:

```
monitor* p = (monitor*)malloc(sizeof(monitor));
```

Очевидно, что для C with Classes этого недостаточно, так как нельзя гарантировать вызов конструктора. Поэтому я ввел оператор, отвечающий одновременно за выделение памяти и инициализацию:

```
monitor* p = new monitor;
```

Я назвал этот оператор `new`, поскольку так назывался аналогичный оператор в Simula. Оператор `new` вызывает некую функцию для выделения памяти, а затем конструктор для инициализации этой памяти. Такая комбинированная операция называется порождением или просто созданием объекта; она создает объект в неинициализированной области памяти.

Нотация, вводимая оператором `new`, очень удобна. Однако объединение выделения и инициализации памяти в одной операции без явного механизма извещения об ошибках на практике привело к некоторым сложностям. Обработка ошибок, происшедших в конструкторе, редко бывает конструктивно важной, однако с введением исключений (см. раздел 16.5) было найдено общее решение и этой проблемы.

Чтобы свести к минимуму необходимость повторной компиляции, в Cfront оператор `new` для классов, имеющих конструктор, был реализован просто как

вызов этого конструктора. Стало быть, конструктор должен выполнить и выделение памяти, и ее инициализацию. При таком подходе, если единица трансляции выделяет память для всех объектов класса X с помощью `new` и не вызывает встраиваемых функций из X, то ее не надо перекомпилировать в случае изменения размера или представления X. Единица трансляции – это принятый в ANSI C термин для обозначения исходного файла после обработки препроцессором. Иными словами, это то, что передается компилятору. Такой подход показался очень удобным для уменьшения числа компиляций моих моделирующих программ. Однако в сообществе пользователей C with Classes и C++ важность данной оптимизации была осознана много позже (см. раздел 13.2).

Оператор `delete` был введен как парный к `new`, точно так же как функция `free()` является парной по отношению к `malloc()` – см. разделы 3.9 и 10.7.

2.11.3. Функции call и return

Интересно, что в первой реализации C with Classes была возможность, которая в C++ исчезла, хотя меня часто просят ее вернуть. Можно было определить функцию, которая неявно вызывалась при каждом вызове любой функции-члена (кроме конструктора), и другую функцию, которая вызывалась перед каждым возвратом из любой функции-члена (кроме деструктора). Эти функции назывались соответственно `call` и `return`. Я использовал их для синхронизации в классе монитора, входившего в первоначальный вариант библиотеки для поддержки многозадачности [Stroustrup, 1980b]:

```
class monitor : object {
   /* ... */
   call();   { /* установить защелку */ }
   return(); { /* снять защелку */ }
   /* ... */
};
```

Данные функции аналогичны методам `:before` и `:after` в языке CLOS. Они были исключены из языка, поскольку никто (кроме меня) ими не пользовался и мне так и не удалось убедить публику, что у них имеются важные применения. В 1987 г. Майк Тиман (Mike Tiemann) предложил альтернативное решение [Tiemann, 1987], которое назвал «обертками» (wrappers), но на семинаре разработчиков USENIX в Эстес Парк было решено, что с этой идеей связано слишком много проблем, и включать ее в C++ не стали.

2.12. Менее существенные средства

В C with Classes были включены две не очень важные особенности: перегрузка оператора присваивания и аргументы по умолчанию. Они были предтечами механизма перегрузки в C++ (см. раздел 3.6).

2.12.1. Перегрузка оператора присваивания

Вскоре было отмечено, что классы с нетривиальным представлением, такие как `string` и `vector`, нельзя нормально копировать, поскольку принятая в C семантика

копирования (побитовое) для таких типов не годилась. Для них копирование по умолчанию сводилось к разделяемому несколькими объектами представлению, а не к созданию настоящих копий. В качестве решения я позволил программисту самому определять семантику копирования [Stroustrup, 1980]:

«К сожалению, стандартное почленное (как для `struct`) присваивание не всегда идеально. Типичный объект класса – это только корень информационного дерева, а копирование одного корня без учета ветвей нежелательно. Точно так же простая перезапись объекта класса может привести к хаосу.

Решение этих проблем – в изменении семантики присваивания для объектов класса. Это можно сделать, объявив функцию-член класса `operator=`. Например:

```
class x {
public:
   int a;
   class y * p;
   void operator= (class x *);
};

void x.operator= (class x * from)
{
   a = from->a;
   delete p;
   p = from->p;
   from->p = 0;
}
```

Таким образом, для объектов класса x мы определили разрушающее чтение, а не операцию копирования, подразумеваемую стандартной семантикой».

В работе [Stroustrup,1982] приведен вариант примера, в котором проверяется условие `this==from`, чтобы корректно обработать присваивание самому себе. Как видите, я учился этой технике на собственных ошибках.

Если в классе был определен оператор присваивания, то он использовался для всех операций копирования. Во время инициализации объект сначала инициировался значением по умолчанию с помощью функции `new` (конструктора) без аргументов, а потом выполнялось присваивание. Это было сочтено неэффективным и привело к появлению в C++ копирующих конструкторов (см. раздел 11.4.1).

2.12.2. Аргументы по умолчанию

Интенсивное использование конструкторов по умолчанию – результат наличия операторов присваивания, определенных пользователем, – привело к появлению аргументов по умолчанию [Stroustrup, 1980]:

«Список аргументов по умолчанию добавлен к механизму классов уже на очень поздней стадии, чтобы противостоять распространению практики передачи идентичных «списков стандартных аргументов» для объектов класса через аргументы функции, аргументов для объектов класса, являющихся членами другого класса, а также аргументов для базового класса. Предоставление в таких случаях списка аргументов показалось достаточно разумным, чтобы побороть антипатию к включению еще одного «средства». Они позволяли сделать объявление `class` более кратким и похожим на объявление `struct`».

Вот пример:

«Можно объявить список аргументов по умолчанию для функции `new()`. Затем этот список будет использоваться всякий раз, как объект класса создается без указания аргументов. Так, при наличии объявления

```
class char_stack
{
   void new(int=512);
   ...
};
```

объявление

```
class char_stack s3;
```

допустимо, при этом `s3` инициализируется вызовом `s3.new(512)`».

При наличии общего механизма перегрузки функций (см. раздел 3.6 и главу 11) аргументы по умолчанию становятся логически избыточными и в лучшем случае обеспечивают небольшое удобство нотации. Однако списки аргументов по умолчанию существовали в C with Classes много лет, прежде чем в C++ появилась перегрузка.

2.13. Что не реализовано в C with Classes

Ряд возможностей, которые позднее были включены в C++ или обсуждаются до сих пор, рассматривались уже во время работы над C with Classes. Среди них – виртуальные функции, статические члены, шаблоны и множественное наследование. Однако «у всех этих обобщений есть свои области применения, но для проектирования, реализации, документирования и обучения каждой «особенности» языка требуются время, силы и опыт... Концепция базового класса – это инженерный компромисс, как и вообще концепция класса в C [Stroustrup, 1982b]».

Я специально упомянул, что приобретение опыта было необходимым. На этом разговор о вреде включения избыточных возможностей и о пользе прагматического подхода можно считать исчерпанным.

Возможность включения автоматического сборщика мусора несколько раз рассматривалась и до 1985 г., но была признана неподходящей для языка, который использовался в системах реального времени, и для программирования низкоуровневых системных задач, вроде написания драйверов аппаратных устройств. В те дни сборщики мусора были не такими совершенными, как сейчас, а нынешние мощности процессоров и объем памяти просто несравнимы с прежними. Личный опыт работы с Simula и отчеты о других системах, где имеется сборка мусора, убедили меня, что эта технология неприемлема в системах, которые разрабатывали я и мои коллеги. Если бы в определении C with Classes (и даже C++) требовалось наличие автоматической сборки мусора, то язык был бы более изящным, но «мертворожденным».

Обсуждалась и прямая поддержка параллельности, но я отказался от этой идеи в пользу библиотечной реализации (см. раздел 2.1).

2.14. Рабочая обстановка

C with Classes был спроектирован и реализован в рамках исследовательского проекта в Центре исследований по вычислительной технике компании Bell Labs. Для такой работы этот центр предоставлял – и до сих пор предоставляет – уникальную обстановку. Здесь мне предложили «заняться чем-нибудь любопытным», выделили достаточные компьютерные ресурсы, всячески поощряли общение с интересными и знающими людьми и дали год на то, чтобы представить на рассмотрение плоды своих трудов.

Одной из составляющих корпоративной культуры Центра было предубеждение против «больших замыслов», требующих более одного-двух человек, против «грандиозных планов», в том числе передачи непроверенных бумажных проектов для реализации другим людям, и против разделения на проектировщиков и кодировщиков. Если вам это по душе, то и в Bell Labs, и в других местах вы найдете немало возможностей для приложения сил. Однако в Центре исследований по вычислительной технике прямо требовалось, чтобы вы (если не заняты теоретическими исследованиями) лично реализовывали задуманное, воплощали свои идеи и находили пользователей, которым это пригодилось бы. Обстановка очень способствовала такой работе, в Bell Labs было достаточно творческих людей, хватало и вопросов, ждущих своего решения. Поэтому я считал себя вправе написать так [Stroustrup, 1986]:

> «Никогда не существовало дизайна C++ на бумаге; проектирование, документирование и реализация проходили одновременно. Естественно, внешний интерфейс C++ написан на C++. Не было никогда ни «проекта C++», ни «комитета по проектированию C++».

Только после признания данного языка начали возникать обычные организационные структуры, но даже тогда именно я официально отвечал за справочное руководство, за мной было последнее слово по поводу всего, что в него включалось. Так продолжалось до начала 1990 г., когда эта обязанность была передана комитету ANSI по C++. Как председатель рабочей группы по расширениям в составе комитета по стандартизации, я по-прежнему несу прямую ответственность за все новшества в C++ (см. раздел 6.4). С другой стороны, лишь несколько первых месяцев я мог заниматься дизайном просто из эстетических соображений и вносить в язык произвольные изменения. Позже каждое языковое средство необходимо было формально реализовывать, а любое изменение или добавление требовали не просто согласия, а чаще всего энергичного одобрения со стороны основных пользователей C with Classes, а потом и C++.

Поскольку никакого гарантированного сообщества пользователей не было, то сам язык и компилятор могли выжить только при условии, если бы понравились пользователям настолько, чтобы те устояли перед естественной тягой к проверенным временем языкам и не поддались рекламным обещаниям относительно новых. Даже мельчайшая несовместимость должна была компенсироваться куда большей выгодой для пользователей, поэтому серьезные несовместимости не вводились в язык даже в начале его существования. Поскольку пользователь склонен считать серьезной любую несовместимость, я оставлял только такие, без которых было никак не обойтись. И только при переходе от C with Classes к C++ было решено пойти на то, что многие старые программы перестанут работать.

Отсутствие формальной организационной структуры, крупномасштабной поддержки – финансовой, кадровой, маркетинговой – более чем компенсировалось неформальной помощью со стороны моих коллег и единомышленников из Центра исследований по вычислительной технике и той защитой от нетехнических требований со стороны организаций, курирующих разработки, которую обеспечила мне администрация Центра. Если бы не помощь и дружеская критика сотрудников Центра, дизайн C++ уступил бы веяниям моды и интересам отдельных групп, а его реализация увязла бы в бюрократической трясине. Важно и то, что администрация Bell Labs создала такую обстановку, где не было нужды утаивать свои идеи ради личной карьеры. Наоборот, дискуссии всегда велись свободно и каждый извлекал из идей и мнений других пользу для себя. Жаль, что даже внутри компании Центр исследований по вычислительной технике – скорее, исключение, чем правило.

C with Classes развивался в ходе обсуждений с сотрудниками Центра и первыми пользователями как внутри него, так и в других подразделениях Bell Labs. Я спроектировал меньшую часть C with Classes и C++, рождением большей мы обязаны многим другим специалистам. Немало идей отвергалось, если они были чересчур сложными, малополезными, слишком трудными для реализации или для обучения, недостаточно эффективными с точки зрения быстроты действия или памяти, несовместимыми с C или просто странными. Реализовывалось лишь очень немногое, да и то после обсуждения не менее чем с двумя специалистами. Обычно в ходе работы, тестирования и использования идея видоизменялась. Новый вариант предлагался более широкой аудитории, затем еще немного модифицировался и, наконец, пролагал дорогу к официальной версии C with Classes, поставляемой лично мной. Важным инструментом проектирования считалось составление руководства, поскольку если что-то не удавалось легко объяснить, то и поддерживать было бы слишком утомительно. Я об этом никогда не забывал, потому что в то время, образно говоря, сам был центром технической поддержки.

Значительное влияние на меня оказал Сэнди Фрэзер (Sandy Fraser), начальник моего отдела. Так, я считаю, что именно он уговорил меня отказаться от определения классов в духе Simula, где включается полное определение каждой функции, и перейти к стилю, подразумевающему, что определения функций находятся в другом месте, а за счет этого усилить роль объявления класса как интерфейса. Многое в C with Classes было сделано для того, чтобы упростить построение симуляторов, которые применялись в работе Сэнди Фрэзера по проектированию сетей. Первым реальным приложением C with Classes как раз и был такой симулятор сети. Еще одним из пользователей C with Classes, оказавшим немалое влияние на его развитие, был Садхир Агравал (Sudhir Agrawal), также занимавшийся моделированием сетей. В решении вопросов дизайна и реализации C with Classes часто принимал участие Джонатан Шопиро, который занимался «потоковой машиной баз данных».

Дискуссии по общим вопросам языков программирования, не касающимся конкретных прикладных задач, я вел с Деннисом Ричи, Стивом Джонсоном и в особенности с Дугом Макилроем. Влияние последнего на разработку C и C++ нельзя переоценить. Не помню ни одного по-настоящему важного проектного решения в C++, которое я бы обстоятельно не обсудил с этим человеком. Естественно, не

всегда мы соглашались друг с другом, но и по сию пору я очень неохотно принимаю решения наперекор мнению Макилроя. Почему-то выходит так, что он всегда оказывается прав и при этом обладает бездной знаний и терпения.

Поскольку C with Classes и C++ проектировались в основном на доске, то обсуждение всегда фокусировалось на «типичных» проблемах: небольших примерах, иллюстрирующих характерные особенности большого класса задач. Поэтому хорошее решение небольшого примера очень помогало при написании программ для реальных аналогичных задач. В C with Classes самым главным считался класс `task`, положенный в основу библиотеки для поддержки многозадачности в духе Simula. Другими важными классами были `queue`, `list` и `histogram`. Классы `queue` и `list` базировались на идее, заимствованной из Simula, о классе `link`, от которого пользователи производили собственные классы.

При таком подходе есть опасность создать язык и инструментарий для изящного решения небольших специально подобранных примеров, которые нельзя масштабировать для построения законченных систем или больших программ. Но такая нежелательная возможность нейтрализовалась тем простым фактом, что C with Classes с самого начала должен был быть самоокупаемым. А стало быть, он не мог выродиться во что-то изящное, но бесполезное.

Работая в тесном контакте с пользователями, я привык обещать только то, что был в состоянии сделать. Я не мог себе позволить надавать столько обещаний, чтобы организация в конце концов сочла разумным выделить достаточно ресурсов для разработки, сопровождения и маркетинга «продукта».

Глава 3. Рождение C++

Никакие узы не связывают крепче,
чем узы наследования.

Стивен Джей Гоулд

3.1. От C with Classes к C++

В 1982 г. мне стало ясно, что C with Classes стал средним языком и останется таким до самой своей кончины. Средний язык я определил следующим образом: продукт достаточно полезен, чтобы окупить себя и своего разработчика, но не настолько привлекателен, чтобы окупить организацию формальной технической поддержки и разработки. Стало быть, продолжение работы над C with Classes и его реализацией на базе препроцессора C обрекало меня на бесконечное сопровождение. Из этой ситуации я видел только два выхода:

- прекратить поддержку C with Classes, перестать работать с пользователями, а себя освободить для новой деятельности;
- на основе опыта работы над C with Classes разработать новый – лучший – язык, который привлек бы достаточно пользователей, чтобы окупить организацию центра поддержки и разработки, и самому опять же заняться чем-то другим. В то время необходимым минимумом я считал 5 тыс. пользователей.

Третья возможность – увеличение числа пользователей за счет маркетинговых мероприятий (читай, хвастливой рекламы) – мне никогда не приходила в голову. На деле же случилось так, что взрывной рост популярности C++ (так был назван новый язык) потребовал всего моего времени, так что и по сей день я не сумел переключиться на что-нибудь значительное.

Думается, что своим успехом C++ обязан тому, что здесь были четко выдержаны поставленные цели проектирования: язык действительно помог структурировать большой класс программ гораздо лучше, чем C, не жертвуя при этом эффективностью и не требуя такой культурной перестройки, которая сделала бы его неприемлемым для организаций, неохотно идущих на большие перемены. Его успех был частично ограничен недостатком новых по сравнению с C возможностей, а также технологией реализации (использование препроцессора). В C with Classes не было предусмотрено достаточной поддержки для людей, готовых потратить значительные усилия в надежде получить адекватный результат. C with Classes был шагом в правильном направлении, но только одним маленьким шажком. Придя к таким выводам, я начал проектировать улучшенный и расширенный язык,

который должен был прийти на смену C with Classes. На этот раз я решил реализовать его в виде нормального компилятора.

Новый язык поначалу продолжал именоваться C with Classes, но после вежливой просьбы руководства получил название C84. Причиной переименования было то, что пользователи стали называть C with Classes сначала «новым C», а потом и просто C. Из-за этого настоящий C низвели до «чистого C», «простого C» и «старого C». Последнее название было особенно обидным, поэтому естественная вежливость и желание избежать путаницы заставили меня искать новое имя.

Имя C84 продержалось всего несколько месяцев отчасти из-за того, что было некрасиво и слишком официально, а отчасти потому, что путаница все равно возникла бы, если бы люди стали опускать «84». К тому же и Ларри Рослер (Larry Rosler), редактор комитета ANSI X3J11 по стандартизации C, попросил меня придумать другое название. Он объяснил, что «стандартизованные языки часто официально именуют, прибавляя к собственному названию языка год принятия стандарта, поэтому всех будет сбивать с толку то, что надмножество (C84, известный также под именем C with Classes, позже C++) имеет номер меньший, чем подмножество (C, возможно, C85, позже ANSI C)». Это замечание показалось мне весьма разумным, хотя Ларри оказался излишне оптимистичен на счет даты принятия стандарта C, и я призвал сообщество пользователей C with Classes подыскать другое название.

Я выбрал для нового языка имя C++, поскольку оно было коротким, имело симпатичные интерпретации и не содержало прилагательных к «C». «++» в зависимости от контекста можно прочесть как «следующий» или «увеличить», хотя обычно произносят «плюс плюс». Предложил это название Рик Маскитти (Rick Mascitti). Впервые я употребил сочетание «C++» в декабре 1983 г. в процессе работы над окончательным редактированием статей [Stroustrup,1984] и [Stroustrup,1984c].

Буква «C» в C++ имеет длинную историю. Разумеется, это название языка, созданного Деннисом Ричи. Непосредственным предшественником C был язык B – интерпретируемый потомок BCPL, который спроектировал Кен Томпсон. Создателем BCPL был Мартин Ричардс из Кембриджского университета, а работал он над ним во время пребывания в Массачусетском технологическом институте. В свою очередь BCPL – это Basic CPL, а CPL – название довольно большого (для своего времени) и изящного языка программирования, совместной разработки кембриджского и лондонского университетов. До подключения к проекту лондонцев «C» было первой буквой от «Cambridge», а потом официально расшифровывалось как «Combined». Неофициально же «C» означало «Christopher», поскольку именно Кристофер Стрейчи (Christopher Strachey) вдохновлял работы по CPL.

3.2. Цели C++

На протяжении периода с 1982 по 1984 гг. перед разработчиками C++ ставились все более честолюбивые и вместе с тем более определенные цели. Я начал смотреть на C++ как на язык, отличный от C. Предметом кропотливой работы

стали библиотеки и инструментальные средства. По этой причине, а также из-за того, что разработчики инструментальных средств в Bell Labs начали проявлять интерес к C++, а я сам полностью погрузился в работу над новым компилятором Cfront, необходимо было дать ответ на несколько ключевых вопросов:

- кто будет пользоваться языком?
- на каких системах будут работать пользователи?
- как прекратить заниматься написанием инструментальных средств?
- как на определении языка скажутся ответы на первые три вопроса?

Что касается первой неясности, я полагал так: пользователями будут мои друзья в Bell Labs и я сам, затем язык более широко распространится в AT&T, после идеями и инструментарием заинтересуются университеты, наконец, AT&T и другие смогут зарабатывать на продаже того набора инструментов, который получится. В какой-то момент первая, до некоторой степени экспериментальная реализация, сделанная мной, перестанет эксплуатироваться и будет заменена компилятором промышленного качества, разработанным AT&T и другими организациями.

Это было разумно с практической и экономической точек зрения. Первая реализация (Cfront) не имела бы полного набора инструментальных средств, была бы переносимой и дешевой, поскольку в таком продукте нуждались и именно его могли себе позволить я, мои коллеги и многие университеты. Возможности для создания лучшего инструментария и специализированных сред появились бы позже. Более развитые инструменты, предназначенные преимущественно для промышленных пользователей, необязательно должны были быть дешевыми, и плата за них окупила бы организацию технической поддержки, без чего невозможно представить широкое распространение языка. Как раз это и есть ответ на третий вопрос: «Как мне перестать заниматься написанием инструментальных средств?». В основных чертах данная стратегия сработала. Но почти все детали оказались не такими, как я рассчитывал.

Чтобы ответить на второй вопрос, я просто огляделся по сторонам – на каких системах реально использовался C with Classes? Практически на любых: от таких маленьких, что на них и компилятор-то не мог работать, до мейнфреймов и суперкомпьютеров. Среди ОС были и такие, о которых я даже не слыхивал. Отсюда следовали выводы, что высокая степень переносимости и возможность выполнять кросс-компиляцию абсолютно необходимы и что я не могу делать никаких предположений о размере и быстродействии машин, на которых будет работать сгенерированный код. Однако для построения компилятора потребовались какие-то допущения о системе, используемой для разработки программ. Я предположил, что скорость действия будет не менее 1 MIPS (миллион команд в секунду), а память – не менее 1 Мб. Предположение достаточно рискованное, поскольку в то время многие потенциальные пользователи работали на разделяемой PDP11 или другой системе, не слишком мощной или эксплуатируемой в режиме разделения времени.

Я не предвидел революции, вызванной пришествием персональных компьютеров, но получилось так, что Cfront, превзошедший намеченные показатели производительности, мог работать (правда, на пределе) даже на IBM PC/AT. Это

доказало, что C++ может быть эффективным языком на ПК, и разработчикам коммерческого программного обеспечения есть чем заняться.

Размышляя над четвертым вопросом, я пришел к следующим выводам. В языке не должно быть средств, требующих особо изощренного компилятора или поддержки во время исполнения, надо использовать существующие компоновщики, а сгенерированный код должен быть эффективен (по сравнению с C), начиная с самых первых версий.

3.3. Компилятор Cfront

Компилятор Cfront для языка C84 был спроектирован и реализован в период между весной 1982 г. и летом 1983 г. Первый пользователь за пределами Центра исследований по вычислительной технике, Джим Коплиен (Jim Coplien), получил копию в июле 1983 г. Джим работал в группе, занимавшейся экспериментальными исследованиями в области коммутации сетей, в отделении Bell Labs в Напервилле, штат Иллинойс. До этого Коплиен некоторое время пользовался C with Classes.

За указанный период я спроектировал C84, написал черновик справочного руководства (опубликованного 1 января 1984 г. [Stroustrup, 1984]), спроектировал библиотеку `complex` для работы с комплексными числами, которую мы реализовали вместе с Леони Роуз (Leonie Rose) [Rose, 1984], спроектировал и реализовал вместе с Джонатаном Шопиро первый класс `string` для работы со строками, занимался сопровождением и переносом на другие платформы C with Classes, поддерживал пользователей C with Classes и готовил их к переходу на C84. В общем, беспокойные выдались полтора года.

Cfront был (и остается) традиционным компилятором неполного цикла (front-end compiler). Он проверяет синтаксис и семантику языка, строит внутреннее представление программы, анализирует и переупорядочивает его и на выходе генерирует файл, подаваемый какому-нибудь генератору кода. Внутреннее представление – это граф, в котором есть по одной таблице символов на каждую область действия. Общая стратегия компиляции такова: читать по одному глобальному объявлению из исходного файла и формировать вывод только тогда, когда объявление полностью проанализировано.

На практике это означает, что компилятору нужно достаточно памяти, чтобы хранить представление всех глобальных имен, типов, а также полный граф одной функции. Несколькими годами позже я выполнил ряд измерений работы Cfront и обнаружил, что потребление памяти стабилизируется на уровне 600 Кб на DEC VAX практически независимо от компилируемой программы. Поэтому в 1986 г. мне удалось выполнить перенос Cfront на PC/AT. К моменту выхода версии 1.0 в 1985 г. Cfront представлял собой примерно 12 тыс. строк кода на C++.

Организация Cfront довольно обычна, разве что в нем используется много таблиц символов вместо одной. Первоначально Cfront был написан на C with Classes (а на чем же еще?), но скоро был переписан на C84, так что самый первый работающий компилятор C++ написан на C++. Даже в первой версии Cfront интенсивно использовались классы и наследование. Однако виртуальных функций в нем не было.

Cfront – это компилятор неполного цикла. В реальных проектах его одного было недостаточно. Ему необходим драйвер, который пропускает исходный файл через препроцессор C (Cpp), затем подает выход Cpp на вход Cfront, а выход Cfront – на вход компилятора C (см. рис. 3.1).

Кроме того, драйвер должен был обеспечить динамическую (во время исполнения) инициализацию. В Cfront 3.0 драйвер стал еще более сложным, поскольку было реализовано автоматическое инстанцирование (см. раздел 15.2) шаблонов [McCluskey, 1992].

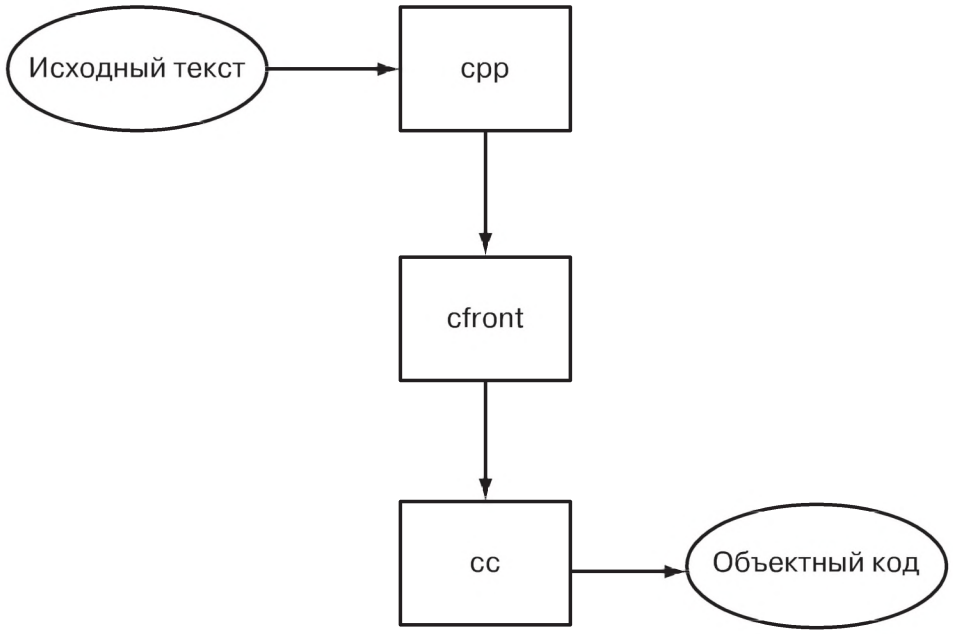


Рис. 3.1

3.3.1. Генерирование C-кода

Самый необычный для своего времени аспект Cfront состоял в том, что он генерировал C-код. Это вызвало много путаницы. Cfront генерировал C-код, поскольку первая реализация должна была быть хорошо переносимой, а я считал, что более переносимого ассемблера, чем C, не найти. Я легко мог бы сгенерировать внутренний формат для кодогенератора или вывести файл на языке ассемблера, но это было не нужно моим пользователям. С помощью ассемблера и внутреннего кодогенератора невозможно было бы обслужить более четверти всех пользователей, и ни при каких обстоятельствах я не смог бы написать, скажем, шесть кодогенераторов, чтобы покрыть запросы 90% пользователей. Поэтому я решил, что использование C в качестве внутреннего формата – это разумный выбор. Позже построение компиляторов, генерирующих C-код, стало распространенной практикой. Так были реализованы языки Ada, Eiffel, Modula-3, Lisp и Smalltalk. Я получил высокую переносимость ценой небольшого увеличения времени компиляции. Вот причины такого замедления:

- время, требующееся Cfront для записи промежуточного C-кода;
- время, нужное компилятору C, чтобы прочитать промежуточный C-код;

- время, «впустую» потраченное компилятором C на анализ промежуточного C-кода;
- время, необходимое для управления всем этим процессом.

Величина накладных расходов зависит в первую очередь от скорости чтения и записи промежуточного C-кода, а это определяется тем, как система управляет доступом к диску. На протяжении нескольких лет я измерял накладные расходы на различных системах и получил данные, что они составляют от 25 до 100% от «необходимого» времени компиляции. Я видел и такие компиляторы, которые не генерировали промежуточный C-код и все же были медленнее, чем сочетание Cfront с C.

Отмечу, что компилятор C используется только как генератор кода. Любое сообщение от компилятора C свидетельствует об ошибке либо в нем, либо в самом Cfront, но не в исходном тексте C++-программы. Любая синтаксическая или семантическая ошибка в принципе должна отлавливаться в Cfront – интерфейсе к компилятору C. В этом отношении C++ и его компилятор Cfront отличаются от языков, построенных как препроцессоры типа Ratfor [Kernighan, 1976] и Objective C [Cox, 1986].

Я подчеркиваю это, поскольку споры на тему, что такое Cfront, шли в течение длительного времени. Его называли препроцессором, так как он генерирует C-код, а для приверженцев C это было доказательством того, что Cfront – простенькая программка наподобие макропроцессора. Отсюда делались ложные выводы о возможности построчной трансляции C++ в C, о невозможности символической отладки на уровне C++ при использовании Cfront, о том, что генерируемый Cfront код должен быть хуже кода, генерируемого «настоящими компиляторами», что C++ – не «настоящий язык» и т.д. Естественно, такие необоснованные обвинения меня раздражали, особенно когда они подавались как критика самого языка C++. В настоящее время несколько компиляторов используют Cfront с локальными кодогенераторами, не прибегая к помощи C. Для пользователя единственным отличием является более быстрая компиляция.

Забавная особенность ситуации заключается в том, что мне не нравится большинство препроцессоров и макрообработчиков. Одна из целей C++ – полностью устранить необходимость в препроцессоре C (см. раздел 4.4 и главу 18), поскольку я считаю, что он провоцирует ошибки. В основные задачи, стоявшие перед Cfront, входило обеспечение в C++ рациональной семантики, которую нельзя было реализовать с помощью имевшихся в то время компиляторов C. Компиляторам не хватало информации о типах и областях действия, для того чтобы обеспечить такое разрешение имен, которое было необходимо C++. Язык C++ проектировался в расчете на интенсивное использование технологии компиляции, а не на поддержку во время исполнения или детальную проработку разрешения имен в выражениях самим программистом (как приходится делать в языках без механизма перегрузки). Следовательно, C++ не мог быть скомпилирован с помощью обычной технологии препроцессирования. В свое время я рассмотрел и отверг такие альтернативы семантике языка и технологии компиляции. Непосредственный предшественник C++ – Cpre – являлся довольно традиционным препроцессором,

которому были неизвестны синтаксические конструкции, области действия и правила контроля типов C. Это явилось источником многих проблем как в определении, так и при использовании языка. Я твердо решил не сталкиваться больше с такими трудностями в пересмотренном языке и его реализации. C++ и Cfront проектировались вместе. Определение языка и технология компиляции, несомненно, были связаны друг с другом, но не так тривиально, как думают некоторые.

3.3.2. Синтаксический анализ C++

В 1982 г., в начале планирования Cfront я хотел воспользоваться методом рекурсивного спуска, поскольку опыт написания таких синтаксических анализаторов у меня уже был. Они легко обеспечивают хорошую диагностику ошибок. Кроме того, мне импонировала идея иметь под рукой всю мощь универсального языка программирования, когда в синтаксическом анализаторе необходимо принимать те или иные решения. Но, будучи добросовестным молодым ученым, я спросил совета у экспертов. В то время в Центре исследований по вычислительной технике как раз работали Эл Ахо (Al Aho) и Стив Джонсон. Они-то, главным образом Стив, и убедили меня, что:

- вручную синтаксические анализаторы уже никто не пишет;
- это пустая трата времени;
- при таком подходе неизбежно получится нечто непонятное, что даже сопровождать невозможно;
- применяемая в подобном анализаторе техника восстановления после ошибок несистематична, а значит, и ненадежна.

Правильным приемом, по мнению вышеупомянутых специалистов, было бы использование LALR(1)-генератора синтаксических анализаторов, поэтому я принял на вооружение написанный Элом и Стивом YACC [Aho,1986].

Для большинства проектов такой выбор был бы идеальным. Это годилось бы почти для любого проекта, связанного с написанием экспериментального языка с нуля. Но, оглядываясь назад, я понимаю, что именно для C++ этот выбор оказался роковой ошибкой. C++ был не экспериментальным языком, а почти совместимым надмножеством C, а в то время еще никому не удалось написать для C LALR(1)-грамматику. Такая грамматика для ANSI C была построена Томом Пеннелло (Tom Pennello) спустя примерно полтора года, то есть слишком поздно. Даже созданный Джонсоном PCC – лучший из всех тогдашних компиляторов C – был не совсем корректен в некоторых деталях, которые для синтаксического анализатора C++ являлись весьма существенными. В частности, в PCC неправильно обрабатывались лишние скобки, поэтому `int(x);` не воспринималось как объявление `x`. Более того, из моих наблюдений следовало, что пользователи тяготеют к разным стратегиям разбора. Мое пристрастие к нисходящему разбору многократно проявлялось в форме написания конструкций, которые очень трудно ложатся на грамматику, поддерживаемую YACC. До сего дня Cfront построен на базе синтаксического анализатора YACC, определенным образом дополненного на уровне лексического разбора. С другой стороны, для C++ все же можно написать эффективный и достаточно изящный анализатор, использующий метод рекурсивного спуска. Так построены некоторые современные компиляторы C++.

3.3.3. Проблемы компоновки

Как уже говорилось выше, я решил обходиться традиционными компоновщиками со всеми их ограничениями. Однако одно ограничение оказалось совершенно нетерпимым и в то же время настолько глупым, что, будь у меня достаточно терпения, я поднял бы заинтересованных людей на борьбу с ним. Дело в том, что большинство тогдашних компоновщиков не позволяло использовать длинные внешние имена. Обычно число символов ограничивалось восемью, а K&R C гарантировал для внешних имен только шесть символов и один регистр. ANSI/ISO C также унаследовал это ограничение. Принимая во внимание, что имя функции-члена включает имя класса и что при компоновке необходимо было каким-то образом отразить тип перегруженной функции (см. раздел 11.3.1), я понимал, что выбор у меня невелик.

Рассмотрим примеры:

```
void task::schedule() { /*...*/ } // 4+8 символов

void hashed::print() { /*...*/ } // 6+5 символов

complex sqrt(complex); // 4 символа плюс 'complex'
double sqrt(double);   // 4 символа плюс 'double'
```

Чтобы представить данные имена всего шестью символами верхнего регистра, пришлось бы применить какую-то форму сжатия, а это усложнило бы написание инструментальных средств. Конечно, можно воспользоваться хэшированием, но тогда для разрешения конфликтов потребуется какая-нибудь рудиментарная «база данных программы». Таким образом, первый способ крайне неприятен, а второй может стать источником серьезных проблем, поскольку в стандартной модели компоновки программ на C и Fortran никакая «база данных» не предусмотрена.

В связи со всем вышесказанным в 1982 г. я начал выступать в пользу доработки компоновщиков для поддержки длинных имен. Не знаю, мои ли усилия тому причиной, но современные компоновщики разрешают гораздо более длинные имена. В Cfront для реализации безопасного связывания применяются специальные алгоритмы кодирования типов. При этом 32 символа – это маловато, порой не хватает даже 256 (см. раздел 11.3.2). Временно – для архаичных компоновщиков – применялась система хэширования длинных идентификаторов, что было очень неудобно.

3.3.4. Версии Cfront

Первые версии компиляторов C with Classes и C++ люди получали прямо от меня. Таким образом, возможность работать с C with Classes появилась у сотрудников десятков образовательных учреждений, таких как: Стэндфордский университет (декабрь 1981 г., первая поставка Cpre), Калифорнийский университет в Беркли, университет штата Висконсин в Мэдисоне, Калифорнийский технологический институт, университет штата Северная Каролина в Чапел Хилл, Массачусетский технологический институт, Сиднейский университет, университет Карнеги-Мэллон, университет штата Иллинойс в Урбана-Шампейн, Копенгагенский

университет, Лаборатория Резерфорда (Оксфорд), IRCAM, INRIA. Поставки компилятора отдельным образовательным учреждениям продолжались и после того, как был спроектирован и реализован C++. Таким образом компилятор получили следующие университеты: Калифорнийский (Беркли – август 1984 г., первая поставка Cfront), штата Вашингтон (Сент-Луис), Техасский (Остин), Копенгагенский и Нового Южного Уэльса. Было, конечно, множество неофициальных копий, поскольку студенты, как водится, не обращали внимания на бюрократические формальности. Уже тогда распространение версий на индивидуальной основе стало для меня обузой, а для университетского народа, желающего иметь C++, – причиной для раздражения. Поэтому мы с Брайаном Кернигана, начальником моего отдела, и Дейвом Каллманом, отвечавшим в AT&T за C++, решили организовать выпуск Cfront несколько иначе. Идея заключалась в том, чтобы избежать назначения цен, подписания контрактов, организации технической поддержки, рекламы, составления документации, отвечающей корпоративным стандартам и т.д. Вместо этого Cfront и некоторые библиотеки передавались университетам по цене носителей. Данную версию назвали Release E («E» – сокращение от Educational, образовательный). В январе 1985 г. Лаборатория Резерфорда и другие учреждения получили первые ленты.

Версия E заставила меня по-другому взглянуть на вещи. Фактически это было полное фиаско. Я ожидал быстрого роста интереса к C++ в университетах. Однако кривая роста нисколько не изменила своего характера (см. раздел 7.1), зато вместо новых пользователей мы получили поток жалоб от профессоров на то, что C++ не распространялся на коммерческой основе. Сколько раз я слышал такие слова: «Да, я хочу работать с C++ и знаю, что могу получить Cfront бесплатно, но, увы, меня это не устраивает, так как мне необходима некая база, на основе которой я могу давать консультации, а мои студенты – использовать в индустриальных проектах». Вот вам и академическое стремление к чистому знанию. Стив Джонсон, тогдашний начальник отдела разработки C и C++, Дейв Каллман и я снова собрались и выработали план создания коммерческой версии 1.0. Однако практика предоставления компилятора C++ (с исходными текстами и библиотеками) образовательным учреждениям «почти бесплатно», которая началась с версии E, до сих пор жива.

Версии C++ часто именуются по номерам версий Cfront. Версия 1.0 определена в книге «Язык программирования C++» [Stroustrup, 1986]. В версиях 1.1 (июнь 1986 г.) и 1.2 (февраль 1987 г.) в основном исправлялись ошибки, сюда были также добавлены указатели на члены и защищенные члены (см. раздел 13.9).

В версии 2.0, вышедшей в июне 1989 г., компилятор подвергся существенной переработке. В нее же было включено множественное наследование (см. раздел 12.1). По общему признанию, это был большой шаг вперед с точки зрения функциональности и качества. В версии 2.1 (апрель 1990 г.) ошибки преимущественно исправили, и Cfront был (почти) приведен к определению, данному в книге «The Annotated C++ Reference Manual» [ARM] – см. раздел 5.3.

В версии 3.0 (сентябрь 1991 г.) были добавлены шаблоны (см. главу 15) в соответствии с определением ARM. Вариант версии 3.0 с поддержкой исключений (см. главу 16) разработала и в конце 1992 г. выпустила на рынок компания Hewlett-Packard [Cameron, 1992].

Сам я написал первые версии Cfront (1.0, 1.1, 1.2) и сопровождал их. В течение нескольких месяцев перед выпуском версии 1.0 в 1985 г. со мной работал Стив Дьюхерст (Steve Dewhurst). Большую работу по написанию синтаксического анализатора для Cfront версий 1.0, 1.1, 2.1 и 3.0 проделала Лаура Ивс (Laura Eaves). Я много сделал для версий 1.2 и 2.0, но, уже начиная с версии 1.2, достаточное время этой работе уделял Стэн Липпман (Stan Lippman). Большую часть работы над версиями 2.1 и 3.0 проделали Лаура Ивс, Стэн Липпман, Джордж Логотетис (George Logothetis), Джуди Уорд (Judi Ward) и Нэнси Уилкинсон (Nancy Wilkinson). Работой над версиями 1.2, 2.0, 2.1 и 3.0 руководила Барбара Му. Эндрю Кениг организовал тестирование версии 2.0. Сэм Харадхвала (Sam Haradhvala) из компании Object Design Inc. выполнил первую реализацию шаблонов в 1989 г., а Стэн Липпман расширил и улучшил ее для версии 3.0 в 1991 г. Первая реализация обработки исключений для Cfront выполнена компанией Hewlett-Packard в 1992 г. Помимо кода, вошедшего в основные версии Cfront, было создано и много локальных вариантов компилятора C++ на его базе. На протяжении нескольких лет такие компании, как Apple, Centerline (бывшая Saber), Comeau Computing, Glockenspiel, ParcPlace, Sun, Hewlett-Packard и другие поставляли продукты, содержащие модифицированные версии Cfront.

3.4. Возможности языка

При переходе от C with Classes к C++ языку были добавлены следующие возможности:

- виртуальные функции (см. раздел 3.5);
- перегрузка функций и операторов (см. раздел 3.6);
- ссылки (см. раздел 3.7);
- константы (см. раздел 3.8);
- контроль над распределением свободной памяти со стороны пользователя (см. раздел 3.9);
- улучшенный контроль типов (см. раздел 3.10).

Кроме того, из языка изъяли функции `call` и `return` (см. раздел 2.11), поскольку ими никто не пользовался, и внесли некоторые мелкие усовершенствования.

3.5. Виртуальные функции

Самой яркой новой возможностью C++, оказавшей огромное влияние на стиль программирования, стали виртуальные функции. Идея была заимствована из Simula, но модифицирована с целью сделать реализацию проще и эффективнее. Логическое обоснование виртуальных функций изложено в работах [Stroustrup, 1986] и [Stroustrup, 1986b]. Для того чтобы подчеркнуть центральную роль виртуальных функций в программировании на C++, приведу развернутую цитату из [Stroustrup, 1986]:

«Абстрактный тип данных – это «черный ящик». Коль скоро он определен, объемлющая программа уже не может вмешаться в его работу. Единственный способ адаптировать его к нуждам

нового пользователя – изменить определение. Это может быть причиной недостаточной гибкости. Рассмотрим тип shape, предназначенный для систем работы с графикой. Предположим, что система должна поддерживать окружности, треугольники и квадраты и имеются следующие классы:

```
class point { /*...*/ };
class color { /*...*/ };
```

Можно было бы определить класс shape таким образом:

```
enum kind { circle, triangle, square };

class shape {
   point center;
   color col;
   kind k;
   // представление геометрической фигуры
public:
   point where() { return center; }
   void move(point to)  { center = to; draw(); }
   void draw();
   void rotate(int);
   // другие операции
};
```

Поле типа k необходимо операциям типа draw() и rotate() для адекватного определения фигуры (в Pascal-подобном языке можно было бы использовать вариантную запись с тэгом k). Тогда функция draw() определялась бы так:

```
void shape::draw()
{
   switch (k) {
   case circle:
       // нарисовать окружность
       break;
   case triangle:
       // нарисовать треугольник
       break;
   case square:
       // нарисовать квадрат
       break;
   }
}
```

Полная неразбериха! Функция вроде draw() должна «знать» обо всех типах фигур. Поэтому код такой функции растет при добавлении в систему каждой новой фигуры. Определив новую фигуру, необходимо просмотреть и, возможно, модифицировать весь ранее написанный код. Добавить новую фигуру не удастся, если исходный код недоступен. Поскольку добавление новой фигуры затрагивает все важные операции над фигурами, от программиста требуется высокая квалификация, и все равно не исключено, что в старый код будут внесены ошибки. Выбор представления конкретных фигур может серьезно усложниться, если требуется, чтобы оно, по крайней мере, частично соответствовало фиксированному образцу, принятому для обобщенного типа shape.

Проблема здесь в том, что между общими свойствами любой фигуры (она имеет цвет, может быть нарисована и т.д.) и свойствами конкретной фигуры (окружность – это фигура, которая имеет радиус, прорисовывается функцией вычерчивания окружности и т.д.) нет четкого разграничения. Возможность выразить такое разграничение и воспользоваться им составляет суть объектно-ориентированного программирования. Язык, обладающий такими выразительными возможностями, поддерживает объектно-ориентированное программирование. Остальные языки его не поддерживают.

Механизм наследования в Simula предлагает решение, которое я принял на вооружение в C++. Сначала мы описываем класс, в котором определены общие свойства всех фигур:

```
class shape {
   point center;
   color col;
   // ...
public:
   point where() { return center; }
   void move(point to) { center = to; draw(); }
   virtual void draw();
   virtual void rotate(int);
   // ...
};
```

Все функции, для которых можно определить порядок вызова, но нельзя предоставить реализацию для фигуры общего вида, помечены словом `virtual` (общий для Simula и C++ термин, означающий «может быть переопределена в классе, производном от данного»). Имея данное определение, для манипуляций с фигурами можно написать обобщенные функции:

```
void rotate_all(shape** v, int size, int angle)
   // повернуть все элементы вектора v размера size
   // на angle градусов
{
   for (int i = 0; i < size; i++) v[i]->rotate(angle);
}
```

Чтобы определить конкретную фигуру, нужно сформулировать, чту это за фигура и задать ее специфические свойства, в том числе виртуальные функции.

```
class circle : public shape {
   int radius;
public:
   void draw() { /* ... */ };
   void rotate(int) {} // да, это пустая функция
};
```

В C++ говорят, что класс `circle` является производным от `shape`, а класс `shape` – базовый для `circle`. В другой терминологии `circle` и `shape` называются соответственно подклассом и суперклассом».

Дальнейшее обсуждение виртуальных функций и объектно-ориентированного программирования см. в разделах 13.2, 12.3.1, 13.7, 13.8 и 14.2.3.

Насколько я помню, в то время виртуальные функции не вызвали большого интереса. Возможно, я недостаточно хорошо объяснил заложенные в них концепции, но преобладающей реакцией в кругу близких мне людей было безразличие и скептицизм. Общее мнение сводилось к тому, что виртуальные функции – это

просто искалеченные указатели на функцию, а потому избыточны. Более того, иногда мне возражали, что хорошо спроектированной программе не нужны ни расширяемость, ни открытость, обеспечиваемые виртуальными функциями, а путем анализа удается установить, какие невиртуальные функции можно вызывать напрямую. Поэтому, говорили оппоненты, виртуальные функции – простое свидетельство плохо проделанной работы. Разумеется, я с этим не соглашался и добавлял виртуальные функции повсюду.

Я намеренно не предоставил в C++ механизма для явного опроса типа объекта:

«Имеющееся в Simula67 предложение INSPECT сознательно не включено в C++. Это сделано для того, чтобы заинтересовать программистов в использовании модульного дизайна за счет применения виртуальных функций» [Stroustrup,1986].

Предложение INSPECT в Simula – это не что иное, как `switch` по поддерживаемому системой полю типа. Однако я видел в этом решении много минусов и решил, насколько возможно, придерживаться статического контроля типов и виртуальных функций. В конце концов, механизм идентификации типа во время исполнения был-таки добавлен в C++ (см. раздел 14.2). Но я надеюсь, что он все же окажется менее привлекательным, чем предложение INSPECT.

3.5.1. Модель размещения объекта в памяти

Ключевая идея реализации состояла в том, что множество виртуальных функций класса определяет массив указателей на функции, поэтому вызов виртуальной функции – это просто косвенный вызов функции через данный массив. Для каждого класса с виртуальными функциями имеется ровно один такой массив, который обычно называют таблицей виртуальных функций или `vtbl`. Любой объект данного класса содержит скрытый указатель (часто его называют `vptr`) на таблицу виртуальных функций своего класса. Если имеются определения

```
class A {
   int a;
public:
   virtual void f();
   virtual void g(int);
   virtual void h(double);
};

class B : public A {
public:
   int b;
   void g(int);  // замещает A::g()
   virtual void m(B*);
};
class C : public B {
public:
   int c;
   void h(double);  // замещает A::h()
   virtual void n(C*);
};
```

то объект класса C выглядит примерно так, как показано на рис. 3.2.

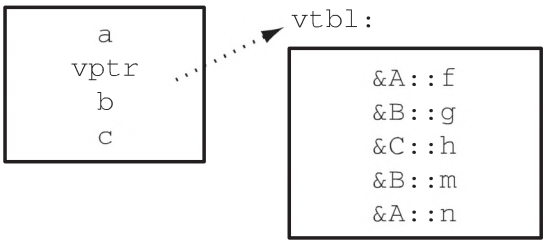


Рис. 3.2

Вызов виртуальной функции преобразуется компилятором в вызов по указателю. Например,

```
void f(C* p)
{
   p->g(2);
}
```

превращается в

```
(*(p->vptr[1]))(p,2);   /* сгенерированный код */
```

Эта реализация не единственно возможная. К ее достоинствам можно отнести простоту и эффективность, к недостаткам – необходимость перекомпиляции при каждом изменении множества виртуальных функций класса.

Таким образом, объект перестает быть просто конгломератом данных-членов. В языке C++ объект класса с виртуальными функциями кардинально отличается от простой структуры в C. Так почему же было решено разделить понятия класса и структуры?

Дело в том, что я хотел иметь единую концепцию: один набор правил размещения в памяти, один набор правил поиска, простые правила разрешения имен и т.д. Единая концепция обеспечивает лучшую интеграцию языковых средств и упрощает реализацию. Я был убежден, что если бы `struct` воспринималось пользователями как «C и совместимость», а `class` – как «C++ и более развитые возможности», то все сообщество раскололось бы на два лагеря, общение между которыми вскоре прекратилось бы. Для меня было очень важно, чтобы при проектировании класса число используемых языковых средств сократилось до минимума. Лишь единая концепция соответствовала моей идее о плавном и постепенном переходе от «традиционного стиля программирования на C» через понятие абстракции данных к объектно-ориентированному программированию.

Оглядываясь назад, я думаю, что такой подход в немалой степени способствовал успеху C++ как инструмента для решения практических задач. На протяжении ряда лет выдвигались самые разные дорогостоящие идеи, которые могли быть реализованы «только для классов», а для структур предлагалось оставить низкие накладные расходы и облегченную функциональность. Но, по-моему, приверженность единой концепции `struct` и `class` избавила нас от включения в классы дополнительных полей и т.п., радикально отличающихся от того, что мы сейчас имеем. Другими словами, принцип «`struct` – это `class`» уберег C++ от превращения в гораздо более высокоуровневый язык с независимым подмножеством низкого уровня, хотя некоторые, возможно, и желали такого развития событий.

3.5.2. Замещение и поиск подходящей виртуальной функции

Виртуальная функция может быть замещена только в производном классе и только функцией с такими же типами аргументов, возвращаемого значения и именем. Это позволило избежать проверки типов во время исполнения и не хранить в выполняемой программе обширную информацию о типах. Например:

```
class Base {
public:
   virtual void f();
   virtual void g(int);
};

class Derived : public Base {
public:
   void f();     // замещает Base::f()
   void g(char); // не замещает Base::g()
};
```

Здесь скрыта очевидная ловушка. Невиртуальная функция `Derived::g()` никак не связана с виртуальной функцией `Base::g()` и скрывает ее. Если вы работаете с компилятором, не выдающим предупреждений по этому поводу, то здесь может возникнуть проблема. Однако обнаружить такую ситуацию компилятору не составляет труда. Cfront 1.0 не давал таких предупреждений, чем вызвал немало нареканий. Уже в версии 2.0 это было исправлено.

Правило точного соответствия типов для замещающих функций позже было ослаблено в отношении типа возвращаемого значения (см. раздел 13.7).

3.5.3. Сокрытие членов базового класса

Имя в производном классе скрывает любой объект или функцию с тем же именем в базовом классе. Несколько лет продолжалась полемика на тему, разумно ли такое решение. Данное правило впервые введено в C with Classes. Я считал, что это естественное следствие обычных правил областей действия. Отстаивая свою точку зрения, я говорил, что противоположная позиция – поместить имена из базового и производного классов в единую область действия – приводит к не менее многочисленным трудностям. В частности, по ошибке можно вызвать из подобъектов функции, изменяющие состояние:

```
class X {
   int x;
public:
   virtual void copy(X* p) { x = p->x; }
};

class XX : public X {
   int xx;
public:
   virtual void copy(XX* p) { xx = p->xx; X::copy(p); }
};
```

```
void f(X a, XX b)
{
   a.copy(&b);   // правильно: скопировать часть b, принадлежащую X
   b.copy(&a);   // ошибка: copy(X*) скрыто за copy(XX*)
}
```

Если разрешить вторую операцию копирования – а так и случилось бы в случае объединения областей действия, – то состояние объекта `b` изменилось бы лишь частично. В большинстве реальных ситуаций это привело бы к очень странному поведению объектов класса `XX`. Я видел, как люди попадались в эту ловушку при использовании компилятора GNU C++ (см. раздел 7.1.4), разрешающего такую перегрузку.

Если функция `copy()` виртуальна, то можно считать, что `XX::copy()` замещает `X::copy()`, но тогда для нахождения ошибки с `b.copy(&a)` потребовалась бы проверка типов во время исполнения, и программистам пришлось бы страховаться от таких неприятностей (см. раздел 13.7.1). Это я понимал уже тогда и боялся, что есть и другие, не осознанные мной проблемы. Вот почему пришлось остановиться на таких правилах, которые казались мне самыми строгими и одновременно наиболее простыми и эффективными в реализации.

Теперь уже я подозреваю, что с помощью правил перегрузки, введенных в версии 2.0 (см. раздел 11.2.2), можно было справиться с этой ситуацией. Рассмотрим вызов `b.copy(&a)`. Переменная `b` по типу точно соответствует неявному аргументу `XX::copy`, но требует стандартного преобразования для соответствия `X::copy`. С другой стороны, переменная `a` точно соответствует явному аргументу `X::copy`, но требует стандартного преобразования для соответствия `XX::copy`. Следовательно, при разрешенной перегрузке такой вызов был бы признан ошибочным из-за неоднозначности.

О явном задании перегрузки функций в базовом и производном классах рассказывается в разделе 17.5.2.

3.6. Перегрузка

Некоторые пользователи просили меня обеспечить возможность перегрузки операторов. Идея выглядела заманчиво, а по опыту работы с Algol68 я знал, как ее реализовать. Однако вводить понятие перегрузки в C++ мне не хотелось по следующим причинам:

- реализовать перегрузку нелегко, а компилятор при этом разрастается до чудовищных размеров;
- научиться правильно пользоваться перегрузкой трудно и столь же трудно корректно определить ее. Следовательно, руководства и учебники тоже разбухнут;
- код, в котором используется перегрузка, принципиально неэффективен;
- перегрузка затрудняет понимание программы.

Если признать верными 3-й и 4-й пункты, то C++ лучше было бы обойтись без перегрузки. А если истинны были бы 1-й или 2-й, то у меня не хватило бы ресурсов для поддержки перегрузки.

Однако если все эти четыре утверждения являлись бы ложными, то перегрузка помогла бы решить множество проблем, возникавших у пользователей C++. Некоторые хотели, чтобы в C++ имелись комплексные числа, матрицы и векторы в духе APL, многим для работы пригодились бы массивы с проверкой выхода за границы диапазона, многомерные массивы и строки. Мне были известны, по крайней мере, два разных приложения, где было бы полезно перегрузить логические операции типа | (или), & (и) и ^ (исключающее или). Список можно было бы продолжать, и с ростом числа пользователей C++ он только расширялся бы. На пункт 4 – перегрузка «затуманивает» код – я отвечал, что некоторые из моих друзей, чье мнение я ценил и чей опыт программирования насчитывал десятки лет, утверждают, что их код от перегрузки стал бы только понятнее. Что с того, если, пользуясь перегрузкой, вы написали невразумительную программу? В конце концов, это можно сделать на любом языке. Важно лишь научиться использовать то или иное средство во благо, а не во вред.

Затем я убедил себя, что перегрузка не обязательно должна быть неэффективной [Stroustrup,1984b], [ARM, §12.1c]. Детали механизма перегрузки были в основном проработаны мной, а также Стью Фельдманом, Дугом Макилроем и Джонатаном Шопиро. Итак, получив ответ на третий вопрос о якобы «принципиальной неэффективности кода, в котором используется перегрузка», нужно было ответить на первый и второй – о сложности компилятора и языка. Сначала я отметил, что пользоваться классами с перегруженными операторами, например `complex` и `string`, было очень легко. Затем написал разделы руководства, дабы доказать, что дополнительная сложность вовсе не так серьезна, как кажется. Этот раздел занял меньше полутора страниц (в 42-страничном руководстве). Наконец, за два часа была написана первая реализация, которая заняла в Cfront всего 18 лишних строк. Таким образом, я достаточно убедительно продемонстрировал, что опасения относительно сложности определения и реализации перегрузки преувеличены. Впрочем, в главе 11 показано, что некоторые проблемы все же имеют место.

Естественно, любой вопрос я решал, сначала доказывая полезность того или иного нововведения, а затем реализуя его. Механизмы перегрузки детально рассмотрены в работе [Stroustrup, 1984b], а примеры их использования в классах – в [Rose, 1984] и [Shopiro, 1985].

Полагаю, перегрузка операторов была для C++ хорошим приобретением. Помимо очевидного использования перегруженных арифметических операторов (`+`, `*`, `+=`, `*=` и т.д.) в численных приложениях, для управления доступом часто перегружают операторы взятия индекса `[]`, применения `()` и присваивания `=`, а операторы `<<` и `>>` стали стандартными операторами ввода/вывода (см. раздел 8.3.1).

3.6.1. Основы перегрузки

Нижеприведенный пример иллюстрирует базовый способ применения перегрузки:

```
class complex {
   double re, im;
public:
   complex(double);
   complex(double,double);
```

```
    friend complex operator+(complex, complex);
    friend complex operator*(complex, complex);
    // ...
};
```

С помощью этого класса простые выражения с комплексными числами преобразуются в вызовы функций:

```
void f(complex z1, complex z2)
{
    complex z3 = z1+z2;  // operator+(z1,z2)
}
```

По умолчанию присваивание и инициализацию определены как почленное копирование (см. раздел 11.4.1).

Проектируя механизм перегрузки, я полагался на преобразования, чтобы уменьшить число перегружаемых функций. Например:

```
void g(complex z1, complex z2, double d)
{
    complex z3 = z1+z2;  // operator+(z1,z2)
    complex z4 = z1+d;   // operator+(z1,complex(d))
    complex z5 = d+z2;   // operator+(complex(d),z2)
}
```

Иными словами, неявное преобразование `double` в `complex` позволяет поддержать «смешанную арифметику» с помощью одной функции сложения комплексных чисел. Для повышения эффективности и точности вычислений можно было бы ввести дополнительные функции.

Мы могли вообще обойтись без неявных преобразований, если бы потребовали, чтобы преобразование всегда выполнялось явно, или предоставили полный набор функций сложения:

```
class complex {
public:
    friend complex operator+(complex, complex);
    friend complex operator+(complex, double);
    friend complex operator+(double, complex);
    // ...
};
```

Язык без неявных преобразований стал бы проще, не было бы места злоупотреблениям ими. К тому же вызов с использованием неявного преобразования обычно менее эффективен, чем вызов функции с точно соответствующими типами аргументов.

Рассмотрим четыре базовых арифметических операции. Чтобы определить полный набор смешанных операций над числами типа `complex` и `double`, требуется 12 арифметических функций. И всего 4 функции плюс один конвертор, если использовать неявные преобразования. Если же число операций и типов больше, то разница между линейным ростом числа функций в случае применения

преобразований и квадратичным, если обходиться без них, становится весьма ощутимой. Я знаю примеры, где приходилось писать полный набор операторов, поскольку не удавалось безопасно определить конверторы. В результате получалось более 100 функций. Возможно, в особых случаях это и приемлемо, но в качестве постоянной практики вряд ли.

Естественно, не все конструкторы определяют разумное и предсказуемое преобразование. Например, у типа `vector` обычно есть конструктор, которому передается целый аргумент, обозначающий число элементов. Если бы предложение `v=7` конструировало вектор из семи элементов и присваивало его `v`, это следовало бы счесть нежелательным побочным эффектом. Я не считал проблему первостепенной. Однако несколько членов комитета по стандартизации C++, в первую очередь Натан Майерс (Nathan Myers), настаивали на необходимости срочного решения. Оно было найдено уже в 1995 г.: в объявление конструктора разрешили включать префикс `explicit`. Конструктор, объявленный как `explicit`, используется только для явного конструирования, но не для неявных преобразований. Например, если в объявлении класса `vector` имеется конструктор `explicit vector(int);`, то предложение `v=7` вызовет ошибку компиляции, тогда как более явное `v=vector(7)` приведет к нормальному конструированию вектора и присваиванию его `v`.

3.6.2. Функции-члены и дружественные функции

Обратите внимание на то, что вместо функции-члена используется глобальная дружественная (`friend`) функция `operator+`. Это обеспечивает симметричность операндов для операции `+`. При использовании функций-членов было бы необходимо разрешать имена:

```
void f(complex z1, complex z2, double d)
{
   complex z3 = z1+z2;  // z1.operator+(z2)
   complex z4 = z1+d;   // z1.operator+(complex(d))
   complex z5 = d+z2;   // d.operator+(z2)
}
```

Тогда потребовалось бы определить, как соединяются объекты типа `complex` с объектами встроенного типа `double`. Для этого понадобилось бы увеличить число функций и модифицировать код в разных местах (в определении класса `complex` и встроенного типа `double`). Вариант был сочтен нежелательным. Я думал о том, чтобы разрешить определение дополнительных операций над встроенными типами, но отверг эту идею, поскольку не хотел нарушать правило, гласящее, что после определения любого типа – встроенного или определенного пользователем – добавление операций с ним уже невозможно. Были и другие причины:

- определение преобразований между встроенными типами и так достаточно запутано, чтобы добавлять еще что-либо;
- обеспечение смешанной арифметики с помощью функций-членов менее прозрачно, чем решение с помощью сочетания глобальной функции и конвертора.

Использование глобальной функции позволяет определить операторы таким образом, что их аргументы логически эквивалентны. Наоборот, определение оператора в виде функции-члена гарантирует, что для первого (левого) операнда никакие преобразования вызываться не будут. Таким образом, правила имеют зеркальное отражение для операторов, где левый операнд – `lvalue`:

```
class String {
   // ...
public:
   String(const char*);
   String& operator=(const String&);
   String& operator+=(const String&);      // добавить в конец
   // ...
};

void f(String& s1, String& s2)
{
   s1 = s2;
   s1 = "asdf";  // правильно: s1.operator=(String("asdf"));
   "asdf" = s2;  // ошибка: String присваивается char*
}
```

Позже Эндрю Кениг заметил, что операторы присваивания, такие как +=, более фундаментальны и эффективны, чем арифметические операторы вроде +. Часто лучше определить только функции типа += и *= в виде членов, а обычные операторы типа + и * добавить позже как глобальные функции:

```
String& String::operator+=(const String& s)
{
   // добавить s в конец *this

   return *this;
}

String operator+(const String& s1, const String& s2)
{
   String sum = s1;
   sum+=s2;
   return sum;
}
```

Заметим, что модификатор `friend` здесь не нужен и что определение бинарного оператора тривиально. Для реализации вызова += вообще не требуются временные переменные, а локальная переменная `sum` в реализации + – это единственный временный объект, с которым сталкивается пользователь. Об остальном позаботится компилятор (см. раздел 3.6.4).

Первоначально я предполагал, что любой оператор можно будет реализовать хоть в виде функции-члена, хоть в виде глобальной функции. Удобно, когда простые операторы доступа уже предоставлены в виде функций-членов, а пользователь может определять собственные операторы как глобальные функции. Применительно

к операторам вроде + и – рассуждения были верны, но в отношении оператора = мы столкнулись с некоторыми проблемами. Поэтому в версии 2.0 требовалось, чтобы оператор = был членом. Такое изменение было несовместимо с предыдущими версиями, и несколько программ перестало работать. Одним словом, решение далось нам нелегко. Ведь если оператор = не был членом, то в программе он мог по-разному интерпретироваться в зависимости от места в исходном коде. Вот пример:

```
class X {
   // нет оператора =
};

void f(X a, X b)
{
   a = b;     // предопределенная семантика =
}

void operator=(X&,X);   // запрещено в 2.0

void g(X a, X b)
{
   a = b;     // определенная пользователем семантика =
}
```

Это могло послужить причиной серьезных недоразумений, особенно если оба присваивания находились в разных исходных файлах. Поскольку для оператора += предопределенной семантики класса не существует, то проблем здесь и не возникает.

Однако даже при первоначальном проектировании C++ я предусмотрел следующее ограничение: операторы `[]`, `()` и `->` должны быть членами. Мне казалось, что оно безобидно и в то же время исключает возможность некоторых ошибок, связанных с тем, что указанные операторы всегда зависят от состояния левого операнда и, как правило, модифицируют его. Впрочем, тут я проявил чрезмерную заботу о пользователях.

3.6.3. Операторные функции

Решив поддержать неявные преобразования и, как следствие, модель смешанных операций, я должен был корректно определить такие преобразования. Один из возможных механизмов дают конструкторы с единственным аргументом. Имея объявление

```
class complex {
   // ...
   complex(double); // преобразует double в complex
   // ...
};
```

можно явно или неявно преобразовать `double` в `complex`. Однако при этом проектировщик класса определит преобразование только к этому классу. Довольно часто возникает необходимость создать новый класс, который вписывается

в уже имеющийся контекст. Например, в библиотеке C существует множество функций, принимающих строковые аргументы типа `char*`. Когда Джонатан Шопиро впервые разрабатывал полнофункциональный класс `String`, он обнаружил, что ему либо придется продублировать все функции следующего вида:

```
int strlen(const char*);   // исходная C-функция
int strlen(const String&); // новая C++-функция
```

либо предоставить преобразование из `String` в `const char*`.

Поэтому я ввел в C++ понятие операторной функции преобразования (конвертора):

```
class String {
   // ...
   operator const char*();
   // ...
};

int strlen(const char*);   // исходная C-функция

void f(String& s)
{
   // ...
   strlen(s);// strlen(s.operator const char*())
   // ...
}
```

На практике пользоваться неявными преобразованиями иногда бывает трудно. Однако написание полного набора операций для смешанных вычислений – тоже не самый легкий путь. Я бы хотел найти лучшее решение, но из всех, что знаю, неявные преобразования – наименьшее зло.

3.6.4. Перегрузка и эффективность

Вопреки наивному предрассудку, между операциями, записанными в виде вызовов функций и в виде операторов, нет фундаментального различия. С эффективностью перегрузки были и остаются актуальными вопросы другого рода: как реализовать встраивание и избежать создания лишних временных объектов.

Сначала я отметил, что код, сгенерированный для выражения типа `a+b` или `v[i]`, идентичен тому, который получается при вызове функций `add(a,b)` и `v.elem(i)`.

Далее я обратил внимание на то, что, пользуясь встраиванием, программист может ликвидировать накладные расходы (время и память) на вызов функции. И наконец, пришел к выводу, что эффективная поддержка такого стиля программирования для больших объектов требует вызова по ссылке (подробнее об этом см. раздел 3.7). Оставалось только решить, как избавиться от лишних копирований в конструкциях вроде `a=b+c`. Генерирование кода вида

```
assign(add(b,c),t); assign(t,a);
```

неудачно по сравнению с кодом

```
add_and_assign(b,c,a);
```

который компилятор генерирует для встроенного типа, а программист может написать явно. В конце концов я продемонстрировал, как сгенерировать код вида

```
add_and_initialize(b,c,t); assign(t,a);
```

При этом остается еще одно «лишнее» копирование. Избавиться от него можно, только когда удается доказать, что операции + и = не зависят от значения, которое присваивается (совмещение имен – aliasing). Более понятное объяснение этой оптимизации – в Cfront ее не было вплоть до версии 3.0 – см. в [ARM]. Полагаю, что первой реализацией, где использовалась подобная техника, был компилятор фирмы Zortech. Уолтер Брайт легко реализовал ее после нашей беседы в 1990 г. после завершения рабочей встречи комитета по стандартизации ANSI C++.

Я считал такую схему приемлемой, поскольку для ручной оптимизации большинства распространенных операций лучше подходят такие операторы, как +=, а также потому, что при инициализации можно принять отсутствие совмещения имен. Позаимствовав из Algol68 идею о том, что объявить переменную допустимо в точке, где она впервые используется (а не в начале блока), я мог поддержать идиомы «только инициализация» или «единственное присваивание», которые более эффективны и менее подвержены ошибкам, чем традиционная техника, когда значение переменной присваивается многократно. Например, можно написать

```
complex compute(complex z, int i)
{
   if ( /* ... */ ) {
      // ...
   }
   complex t = f(z,i);
   // ...
   z += t;
   // ...
   return t;
}
```

вместо

```
complex compute(complex z, int i)
{
   complex t;
   if ( /* ... */ ) {
      // ...
   }
   t = f(z,i);
   // ...
   z = z + t;
   // ...
   return t;
}
```

Еще одна идея о том, как повысить эффективность за счет устранения временных объектов, изложена в разделе 11.6.3.

3.6.5. Изменение языка и новые операторы

Я считал важным ввести перегрузку так, чтобы расширить язык, а не изменить существующую семантику. Иначе говоря, должна быть возможность определять операторы над пользовательскими типами (классами), но не изменять смысл операторов над встроенными типами. Кроме того, я не хотел позволять программисту вводить новые операторы. Мне не нравились загадочность нотации и необходимость сложных стратегий синтаксического анализа вроде тех, что применяются в Algol68. Думаю, в этом вопросе моя сдержанность была оправданной. См. также разделы 11.6.1 и 11.6.3.

3.7. Ссылки

Ссылки были введены в основном для поддержки перегрузки операторов. Дуг Макилрой вспоминает, что однажды я объяснял ему некоторые проблемы, касавшиеся схемы перегрузки операторов. Он употребил слово «ссылка», после чего я, пробормотав «спасибо», выбежал из его кабинета, чтобы на следующий день появиться с практически готовым решением, которое и вошло в язык. Просто Дуг тогда напомнил мне об Algol68.

В языке C аргументы всегда передаются функции по значению, а в случаях, когда передача объекта по значению не годится или слишком неэффективна, программист может передать указатель на объект. При этом нотация должна быть удобной, поскольку нельзя ожидать, что пользователь всегда будет вставлять оператор взятия адреса для больших объектов. Так,

```
a = b - c;
```

это общепринятая нотация, в то время как

```
a = &b - &c;
```

нет. Как бы то ни было, `&b-&c` уже имело в C определенную семантику, и менять ее я не хотел.

После инициализации ссылка уже не может ссылаться ни на что другое. Иными словами, коль скоро она инициализирована, нельзя заставить ее ссылаться на другой объект, то есть нельзя изменять привязку. В прошлом меня неприятно удивил тот факт, что в Algol68 предложение `r1=r2` может означать либо присваивание объекту, на который ссылается `r1`, либо изменение значения самой ссылки `r1` (повторная привязка) в зависимости от типа `r2`. Я хотел уйти от таких проблем в C++.

Если необходимы более сложные манипуляции, всегда можно воспользоваться указателями. Поскольку в C++ есть и ссылки, и указатели, ему не нужны средства для различения операций над самой ссылкой и операций над объектом, на который она ссылается (как в Simula). Не нужен и дедуктивный механизм, применяемый в Algol68.

Однако я сделал одну серьезную ошибку, разрешив инициализировать неконстантную (без спецификатора `const`) ссылку значением, не являющимся `lvalue`. Например:

```
void incr(int& rr) { rr++; }

void g()
{
   double ss = 1;
   incr(ss); // примечание: передано double, ожидалось int
             // исправлено в версии 2.0
}
```

Из-за различия типов `int&` не может ссылаться на переданное значение типа `double`, поэтому генерировалась временная переменная для хранения значения типа `int`, инициализированная значением `ss`. В связи с этим функция `incr` модифицировала данную временную переменную, а вызывающая функция получала неизмененное значение.

Я разрешил инициализировать ссылки не-`lvalue`, чтобы сделать различие между вызовом по значению и по ссылке деталью реализации вызываемой функции, которая вызывающей вообще неинтересна. Для константных ссылок это возможно, для неконстантных – нет. В версии 2.0 определение C++ было соответствующим образом изменено.

Следует отметить, что константную ссылку можно инициализировать не-`lvalue`, а также `lvalue`, для типа которой требуется преобразование. В частности, это позволяет вызывать написанную на Fortran функцию с аргументом-константой:

```
extern "Fortran" float sqrt(const float&);

void f()
{
   sqrt(2);      // вызов с передачей аргумента по ссылке
}
```

Помимо очевидных применений ссылок (для передачи аргументов), мы считали важной возможность использовать ссылки как возвращаемые значения. Это позволило легко написать оператор взятия индекса для класса `String`:

```
class String {
   // ...
   char& operator[](int index);      // оператор взятия индекса
                                     // возвращает ссылку
};

void f(String& s, int i)
{
   char c1 = s[i];  // присвоить результат operator[]
   s[i] = c1;       // присвоить результату operator[]
   // ...
}
```

Возврат ссылки на внутреннее представление `String` предполагает, что пользователи ведут себя ответственно.

3.7.1. Lvalue и Rvalue

Если `operator[]()` реализован так, что возвращает ссылку, то не удастся приписать различную семантику чтению и записи элемента, извлеченного по индексу. Например, в выражении

```
s1[i] = s2[j];
```

мы не можем одновременно произвести действие над строками `s1` и `s2`, где в одном случае значение присваивается, а в другом – читается. Размышляя над задачами реализации строк с разделяемым представлением и доступа к базам данных, мы с Джонатаном Шопиро пришли к выводу о необходимости иметь различную семантику для доступа на чтение и на запись. В обоих случаях чтение – это простая и дешевая операция, тогда как запись – потенциально дорогая и сложная, и для нее может потребоваться копирование структур данных.

Рассматривались две возможности:

- определить различные функции для применения к `lvalue` и `rvalue`;
- заставить программиста использовать вспомогательную структуру данных.

В результате был выбран второй подход, поскольку он помогал избежать расширения языка, а кроме того, мы считали возврат объекта, описывающего положение в контейнерном классе вроде `String`, более общим приемом. Основная идея заключалась в том, чтобы иметь вспомогательный класс, который определяет позицию в контейнерном классе почти как ссылку, но имеет разную семантику для чтения и записи. Например:

```
class char_ref { // идентификация символа в строке
friend class String;
   int i;
   String* s;
   char_ref(String* ss, int ii); { s=ss; i=ii; }
public:
   void operator=(char c);
   operator char();
};
```

Присваивание объекту типа `char_ref` реализовано как присваивание символу, на который он ссылается, а чтение объекта типа `char_ref` – как преобразование в тип `char`, возвращающее значение символа:

```
void char_ref::operator=(char c) { s->r[i]=c; }
char_ref::operator char() { return s->r[i]; }
```

Заметим, что только `String` может создать `char_ref`. Фактическое присваивание реализуется классом `String`:

```
class String {
friend class char_ref;
   char* r;
```

```
public:
   char_ref operator[](int i)
      { return char_ref(this,i); }
   // ...
};
```

При наличии таких определений предложение

```
s1[i] = s2[j];
```

означает

```
s1.operator[](i) = s2.operator[](j)
```

где `s1.operator[](i)` и `s2.operator[](j)` возвращают временные объекты класса `char_ref`. Это в свою очередь означает

```
s1.operator[](i).operator=(s2.operator[](j).operator char())
```

Встраивание во многих случаях делает производительность приема вполне приемлемой, а использование дружественных отношений для ограничения создания объектов типа `char_ref` гарантирует, что мы не получим проблем с «долгоживущими» временными объектами (см. раздел 6.3.2). Данный прием был использован в классе `String`, доказавшем свою полезность. Однако для таких простых случаев, как доступ к отдельным символам, он выглядит усложненным и тяжеловесным. Поэтому я искал альтернативные решения. Одна такая возможность – составные операторы (см. раздел 11.6.3).

3.8. Константы

В операционных системах доступ к некоторой области памяти часто прямо или косвенно контролируется двумя битами: первый дает информацию о том, можно ли пользователю писать в эту область, а второй – можно ли из нее читать. Эта идея показалась мне имеющей прямое отношение к C++, и я подумывал о том, чтобы разрешить для любого типа задавать атрибут `readonly` или `writeonly`. Вот как излагается эта идея во внутренней записке, датированной январем 1981 г. [Stroustrup,1981b]:

«До настоящего времени в C было невозможно указать, что некоторый элемент данных должен быть доступен только для чтения, то есть его значение неизменно. Не было также способа наложить ограничения на то, что функция может делать с аргументами. Деннис Ричи отметил, что если бы существовал модификатор типа `readonly`, то легко можно было получить обе возможности:

```
readonly char table[1024]; /* символы в таблице table нельзя
                              изменять */
int f(readonly int * p)
{
   /* f не может изменять данные, на которые указывает p */
   /* ... */
}
```

Модификатор readonly служит для предотвращения изменения переменной. Он показывает, что из всех способов доступа к переменной законны только те, которые не изменяют ее значения».

Далее в записке говорится:

«Модификатор readonly применим также и к указателям. *readonly интерпретируется как «неизменяемый указатель на». Например:

```
readonly int * p;   /* указатель на int только для чтения */
int * readonly pp;  /* неизменяемый указатель на int */
readonly int * readonly ppp;  /* неизменяемый указатель на */
                              /* int только для чтения */
```

В этих примерах допустимо присвоить новое значение p, но не *p. Можно присвоить значение *pp, но не pp. Ни ppp, ни *ppp значение присвоить нельзя».

В той же записке введено понятие о writeonly:

«Модификатор типа writeonly аналогичен readonly, только запрещает чтение, а не запись. Например:

```
struct device_registers {
   readonly int  input_reg, status_reg;
   writeonly int output_reg, command_reg;
};
void f(readonly char * readonly from,
   writeonly char * readonly to)
/*
   f может получить данные через from,
   сохранить результаты в to,
   но не может изменить ни тот, ни другой указатель
*/
{
   /* ... */
}

int * writeonly p;
```

Здесь ++p недопустимо, так как в качестве побочного эффекта подразумевает чтение предыдущего значения p, но p=0 допустимо».

Это предложение акцентировало внимание на определении интерфейса, а не на включении в C символических констант. Ясно, что значение с модификатором readonly – это символическая константа, но идея была шире. Первоначально я предлагал указатели на readonly, но не readonly-указатели. Благодаря короткой дискуссии с Деннисом Ричи возникла идея о механизме readonly/writeonly, который был реализован и предложен внутренней группе по стандартизации C в стенах Bell Labs под председательством Ларри Рослера. Это мой первый опыт работы над стандартами. Покидая совещание, я имел договоренность (то есть принятое большинством решение), что readonly будет включено в C – да-да, именно в C, а не в C with Classes или C++, следует только переименовать его в const. К сожалению, подобные решения ни к чему не обязывают, вот почему

с нашими компиляторами C ничего не произошло. Позже был образован комитет ANSI C (X3J11), предложение о введении `const` было представлено на его рассмотрение и стало частью ANSI/ISO C.

Между тем я продолжил эксперименты с `const` в C with Classes и обнаружил, что этот модификатор может служить заменой макросам для представления констант, только если глобальные `const`-объекты используются как локальные в своих единицах компиляции. Лишь в этом случае компилятор мог без труда сделать вывод, что значение `const`-объектов действительно не изменится. Знание этого факта позволяет использовать простые `const`-объекты в константных выражениях, не выделяя под них память. В C это правило оказалось непринятым. Например, в C++ можно написать

```
const int max = 14;
void f(int i)
{
   int a[max+1]; // константа max используется в
                 // константном выражении

switch (i) {

   case max:     // константа max используется в
                 // константном выражении
      // ...
   }
}
```

тогда как в C (даже сегодня) необходимо

```
#define max 14
// ...
```

поскольку, в C не разрешается использовать `const`-объекты в константных выражениях. Из-за этого модификатор `const` в C не так полезен, как в C++. Язык C остается зависимым от препроцессора, тогда как программистам на C++ доступны типизированные константы с ограниченной областью действия.

3.9. Управление памятью

Задолго до того как была написана первая программа на C with Classes, я знал, что свободная (динамическая) память в языке с классами будет использоваться гораздо интенсивнее, чем в C. По этой причине в C with Classes и были введены операторы `new` и `delete`. Оператор `new`, который одновременно выделяет память и инициализирует ее, заимствован из языка Simula. Оператор `delete` был необходимым дополнением, поскольку я не хотел, чтобы C with Classes зависел от сборщика мусора (см. разделы 2.13 и 10.7). Изложу аргументацию в пользу оператора `new`. Какая запись вам больше нравится, такая:

```
X* p = new X(2);
```

или такая:

```
struct X * p = (struct X *) malloc(sizeof(struct X));
if (p == 0) error("не хватило памяти");
p->init(2);
```

И где проще сделать ошибку? Отметим, что проверка на нехватку памяти ведется в обоих случаях. Только оператор `new` при выделении памяти делает это неявно и может вызывать написанную пользователем функцию `new_handler` [2nd, §9.4.3]. В то время достаточно часто звучали аргументы против: «нам это ни к чему» и «ведь кто-то мог уже использовать `new` как идентификатор». Разумеется, и то, и другое правильно.

Итак, введение оператора `new` упростило работу со свободной памятью и сделало ее менее подверженной ошибкам. Это способствовало широкому распространению оператора, поэтому из-за функции выделения памяти `malloc()` из библиотеки C, использованной в реализации `new`, значительно снижалась производительность программы. Само по себе это не удивительно; вопрос – что с этим делать? То, что реальные программы тратили 50% времени и даже больше внутри `malloc()`, было абсолютно неприемлемым.

Я пришел к выводу, что очень эффективны операторы распределения и освобождения памяти, определенные в классе. Основная идея заключалась в том, что свободная память, как правило, выделяется и освобождается для большого числа маленьких объектов, принадлежащих небольшому числу классов. Вынесите выделение памяти для таких объектов в отдельную функцию-распределитель, и тогда можно будет сэкономить и время, и память, а заодно уменьшить фрагментацию свободной памяти общего назначения.

Вспоминаю, как излагал Брайану Кернигану и Дугу Макилрою прием «присваивания указателю `this`» (описан ниже) и резюмировал свою речь так: «Это уродливо, как смертный грех, но работает, и если у вас нет лучших предложений, то именно так я это и реализую». Предложений у них не оказалось, так что более красивого решения нам пришлось ждать до выхода версии 2.0, уже в C++ (см. раздел 10.2).

По умолчанию память для объекта выделялась «системой», не требуя от пользователя никаких действий. Чтобы отменить это, программист просто должен был присвоить значение указателю `this`. По определению `this` указывает на объект, для которого вызвана функция-член. Например:

```
class X {
   // ...
public:
   X(int i);
   // ...
};

X::X(int i)
{
   this = my_alloc(sizeof(X));
   // инициализация
}
```

При каждом вызове конструктора `X::X(int)` память будет выделяться с помощью `my_alloc()`. Этот механизм вполне справляется со своей непосредственной задачей и с некоторыми другими тоже, но он слишком низкоуровневый, плохо сочетается с управлением стеком и наследованием, подвержен ошибкам и должен применяться слишком часто, поскольку типичный класс имеет много конструкторов.

Статические и автоматические (с выделением памяти из стека) объекты всегда было возможно создать, и именно для них управление памятью реализуется максимально эффективно. Типичный пример – класс `String`. Его объекты обычно создаются в стеке, следовательно, не требуют явного управления памятью, а необходимая для них свободная память распределяется невидимо для пользователя функциями-членами класса.

Примененная выше нотация для записи конструкторов обсуждается в разделах 3.11.2 и 3.11.3.

3.10. Контроль типов

Правила контроля типов в языке C++ появились как результат опыта работы с C with Classes. Все вызовы функций проверяются во время компиляции. Проверку последних аргументов можно отменить, дав явное указание в объявлении функции. Это необходимо для функции `printf()` из библиотеки C:

```
int printf(const char* ...);  // принимаются любые аргументы после
                              // начальной символьной строки
// ...
printf("date: %s %d 19%d\n", month,day,year); // может, и правильно
```

Было предложено несколько механизмов контроля типов. Полный обход системы контроля типов с помощью многоточия являлся самым радикальным и наименее рекомендуемым. Перегрузка имен функций (см. раздел 3.6.1) и аргументы по умолчанию [Stroustrup,1986] (см. раздел 2.12.2) дали возможность сделать вид, что имеется одна функция, принимающая разные множества аргументов, при этом не приходилось жертвовать безопасностью типов.

Кроме того, я спроектировал систему потокового ввода/вывода, дабы продемонстрировать, что ослабление контроля типов необязательно даже в этом случае (см. раздел 8.3.1). Так, запись

```
cout << "дата: "<<month<<' '<<day<<" 19"<<year<<'\n';
```

является безопасной версией примера, приведенного выше.

Я считал и считаю до сих пор, что контроль типов – это, скорее, практический инструмент, а не самоцель. Важно понимать: устранение из программы всех до единого нарушений согласованности типов не гарантирует ни правильности программы, ни даже того, что она не перестанет работать (причина – использование объекта противоречит его определению). Например, случайный электрический импульс может кардинально изменить значение бита памяти. Считать небезопасное использование типов и крах программы эквивалентными явлениями – безответственно и неправильно. Это то же самое, как не видеть разницы между зависанием

компьютера и такими катастрофами, как крушение самолета, обрыв телефонного кабеля или авария на АЭС.

Надежность системы зависит от всех ее элементов, и нельзя возлагать ответственность за ошибку только на какой-то один. Мы пытаемся проектировать важные системы так, чтобы ни отдельная ошибка, ни даже множество ошибок не приводили к полному краху. Ответственность за целостность системы лежит на людях, которые ее создавали. В частности, безопасное использование типов – не замена тестированию, хотя оно может оказать существенную помощь при подготовке системы к тестированию. Ругать язык программирования за тот или иной сбой в системе, даже чисто программной, – значит не понимать сути предмета (см. также раздел 16.2).

3.11. Второстепенные возможности

При переходе от C with Classes к C++ было добавлено несколько второстепенных возможностей.

3.11.1. Комментарии

Из второстепенных возможностей самой заметной было введение комментариев в стиле BCPL:

```
int a; /* явно завершающийся комментарий в стиле C */
int b; // комментарий в стиле BCPL, действующий до конца строки
```

Допустимы оба вида комментариев, поэтому программист может выбирать тот, что ему больше по душе. Я предпочитаю однострочные комментарии а-ля BCPL. Поводом для введения комментариев, начинающихся с `//`, было то, что изредка я делал глупые ошибки, забывая закончить C-комментарий, а иногда добавление трех лишних символов делало этот комментарий слишком длинным, не помещавшимся на одной строке экрана. Еще я заметил, что `//` удобнее, чем `/*` для комментирования коротких участков кода.

Скоро обнаружилось, что добавление `//` не вполне совместимо с C. Так, выражение

```
x = a//* разделить на */b
```

в C++ означает `x=a`, а в C – `x=a/b`. Но и тогда, и сейчас большинство программистов на C не считают, что на практике такое расхождение может составлять проблему.

3.11.2. Нотация для конструкторов

Название «функция `new`» для конструктора всегда было источником путаницы, поэтому были введены именованные конструкторы. Одновременно появилось разрешение явно использовать конструкторы в выражениях:

```
complex i = complex(0,1);

complex operator+(complex a, complex b)
```

```
{
   return complex(a.re+b.re, a.im+b.im);
}
```

Выражение вида `complex(x,y)` – это явный вызов конструктора класса `complex`.

Чтобы уменьшить число ключевых слов, я решил не пользоваться явным синтаксисом вроде

```
class X {
   constructor();
   destructor();
   // ...
};
```

Вместо этого был выбран декларативный синтаксис, лучше отражающий характер использования конструкторов:

```
class X {
   X();       // конструктор
   ~X();      // деструктор (в C оператор ~ означает дополнение)
   // ...
};
```

Явный вызов конструкторов в выражениях оказался очень полезным, но явился причиной многих проблем при синтаксическом анализе C++-программ. В C with Classes функции `new()` и `delete()` по умолчанию считались `public`. Эта аномалия была устранена таким образом, чтобы конструкторы и деструкторы подчинялись тем же правилам контроля доступа, что и обычные функции. Например:

```
class Y {
   Y();   // закрытый конструктор
   // ...
};

Y a;  // ошибка: нет доступа к закрытому члену Y::Y()
```

В результате появились многие полезные приемы, основанные на идее контроля над операциями путем сокрытия функций, которые их выполняют (см. раздел 11.4).

3.11.3. Квалификация

В C with Classes точка использовалась для обозначения принадлежности к классу, а также для выбора члена конкретного объекта. Подчас из-за этого возникали неоднозначные конструкции. Вот пример:

```
class X {
   int a;
public:
   void set(X);
};

void X.set(X arg) { a = arg.a; };    // пока все корректно
```

```
class X X;    // обычный прием в C:
              // у класса и объекта одно и то же имя

void f()
{
   // ...
   X.a;   // что означает X: класс или объект?
   // ...
}
```

Чтобы справиться с данной проблемой, для обозначения членства в классе была введена нотация `::`, а `.` (точка) оставлена только для обозначения членов объекта. Тогда приведенный выше пример записывается так:

```
void X::set(X arg) { a = arg.a; };

class X X;

void g()
{
   // ...
   X.a;   // объект.член
   X::a;  // класс::член
   // ...
}
```

3.11.4. Инициализация глобальных объектов

Я поставил целью разрешить использование определенных пользователем типов всюду, где допустимы встроенные типы. По опыту работы с Simula было известно, что отсутствие глобальных переменных типа класса – одна из причин снижения производительности. В C++ они были разрешены. Это имело важные и в чем-то неожиданные последствия. Рассмотрим пример:

```
class Double {
   // ...
   Double(double);
};

Double s1 = 2;          // конструирование s1 из 2
Double s2 = sqrt(2);    // конструирование s2 из sqrt(2)
```

В общем случае такую инициализацию нельзя выполнить ни во время компиляции, ни во время связывания. Необходима динамическая инициализация (во время исполнения). Она выполняется в том порядке, в котором объявления встречаются в единице трансляции. Порядок инициализации объектов из разных единиц трансляции не определен. Гарантируется лишь, что статическая инициализация будет выполнена раньше динамической.

3.11.4.1. Проблемы динамической инициализации

Предполагалось, что глобальные объекты будут довольно простыми, и сложная инициализация для них не понадобится. В частности, ожидалось, что глобальные объекты, инициализация которых зависит от глобальных объектов в других единицах трансляции, – это редкость. Такие зависимости я считал следствием плохого проектирования и потому не собирался вводить в язык какую-то специальную поддержку для них. В отношении простых примеров, подобных приведенному выше, я был прав. Такие примеры действительно встречаются на практике и не приводят к сложностям. К сожалению, я обнаружил и другие, более интересные применения динамически инициализируемых глобальных объектов.

В библиотеке часто бывает необходимо выполнить какие-то действия до того, как будут использоваться отдельные функции из нее. С другой стороны, в библиотеке могут быть объекты, которые следовало бы инициализировать предварительно, чтобы пользователь мог сразу применять их, не задумываясь об инициализации. Например, вы же не инициализируете переменные `stdin` и `stdout` из стандартной библиотеки C, за вас это делает стартовый модуль. А закрывает эти потоки библиотечная функция `exit()`. Это очень специфическая ситуация, ничего подобного в других библиотеках нет. Когда я проектировал библиотеку потокового ввода/вывода, то хотел добиться такого же удобства, не вводя в C++ специализированных расширений. Поэтому было решено воспользоваться динамической инициализацией.

И все бы хорошо, вот только приходилось полагаться на деталь реализации, заключавшуюся в том, что `cin` и `cout` конструируются до начала выполнения пользовательского кода и уничтожаются после его завершения. Другие разработчики оказались не такими внимательными. В результате программы аварийно завершались, поскольку `cout` использовался еще до того, как был сконструирован. С другой стороны, часть выходных данных терялась из-за того, что `cout` уничтожался (со сбросом буферов) слишком рано. Другими словами, мы попались на той самой зависимости от порядка выполнения, которую я считал «маловероятной и признаком плохого дизайна».

3.11.4.2. Как обойти зависимость от порядка

Есть два решения данной проблемы. Очевидное – добавить к каждой функции-члену флажок, сигнализирующий, в первый ли раз выполняется фрагмент данного кода. Для этого необходима некая глобальная переменная, инициализированная нулем. Например:

```
class Z {
   static int first_time;
   void init();
   // ...
public:
   void f1();
   // ...
   void fn();
};
```

Каждая функция-член при этом выглядела бы примерно так:

```
void Z::f1()
{
   if (first_time == 0) {
      init();
      first_time = 1;
   }
   // ...
}
```

Накладные расходы для таких простых функций, как вывод одного символа, могут оказаться слишком велики.

В новом дизайне потокового ввода/вывода (см. раздел 8.3.1) Джерри Шварц (Jerry Schwarz) применил хитроумный вариант этого приема [Schwarz, 1989]. Заголовок `<iostream.h>` содержит примерно такой код:

```
class io_counter {
   static int count;
public:
   io_counter()
   {
      if (count++ == 0) { /* инициализировать cin, cout и т.д. */ }
   }

   ~io_counter()
   {
      if (--count == 0) { /* очистить cin, cout и т.д. */ }
   }
};

static io_counter io_init;
```

Теперь каждый файл, включающий заголовок `<iostream.h>`, создает и инициализирует также объект `io_counter`, в результате чего увеличивается счетчик `io_counter::count`. Когда это происходит в первый раз, инициализируются библиотечные объекты. Поскольку заголовок предшествует любому использованию библиотечных средств, правильная инициализация гарантируется. Так как деструкторы вызываются в порядке, обратном вызову конструкторов, этот прием гарантирует также корректную очистку вслед за последним использованием библиотеки.

Таким образом решается проблема зависимости от порядка, причем от поставщика библиотеки требуется добавить лишь несколько почти стандартных строк кода. К сожалению, последствия для производительности бывают весьма серьезными. При использовании таких трюков код динамической инициализации будет присутствовать в большинстве объектных файлов, а значит (если предположить, что применялся обычный компоновщик), вызовы функций инициализации будут разбросаны по всему адресному пространству процесса. В системах с виртуальной памятью это означает, что на этапе начальной инициализации и на этапе

финальной очистки в основную память подгрузится большая часть страниц программы. Такое использование виртуальной памяти нельзя признать разумным, ибо при запуске объемных приложений будет наблюдаться заметная (порядка нескольких секунд) задержка.

Для разработчика инструментальных средств тривиальное решение – модифицировать компоновщик так, чтобы он собирал весь код, выполняемый при инициализации, в одном месте. Сложностей не возникает и в случае, когда система ни в каком виде не поддерживает частичную динамическую загрузку программы в основную память. Однако это слабое утешение для программиста на C++, столкнувшегося с такой проблемой [Reiser, 1992]. Мы имеем здесь нарушение основополагающего принципа, говорящего, что любая возможность C++ должна быть не только полезной, но и не накладной (см. раздел 4.3). Можно ли решить проблему, добавив какое-то новое средство? На первый взгляд, нет, поскольку ни дизайн языка, ни даже официальные комитеты по стандартизации не могут узаконить эффективность языка. В предложениях, которые мне довелось читать, делаются попытки решить проблему зависимости от порядка, которая уже решена приемом, предложенным Джерри, но не касаются возникающего при этом вопроса об эффективности. Подозреваю, что настоящее решение состоит в том, чтобы убедить разработчиков инструментальных средств предотвратить «избиение виртуальной памяти» (virtual memory bashing), вызванное процедурами динамической инициализации. Для этого нужно лишь вставить необходимые слова в стандарт.

3.11.4.3. Динамическая инициализация встроенных типов

В языке C статический объект можно инициализировать лишь слегка расширенным константным выражением, например:

```
double PI = 22/7;       /* правильно */
double sqrt2 = sqrt(2);/* в C это ошибка */
```

Однако в C++ возможны выражения общего вида при инициализации объектов, например:

```
Double s2 = sqrt(2);    // правильно
```

Стало быть, для встроенных типов обеспечивается меньшая поддержка, чем для классов. Это несоответствие было окончательно исправлено в версии 2.0:

```
double sqrt2 = sqrt(2);// правильно в C++ (начиная с версии 2.0)
```

3.11.5. Предложения объявления

В Algol68 есть прекрасная идея о том, что объявление может вводиться в любом месте, а не только в начале блока. В C++ я сделал возможными идиомы «только инициализация» или «единственное присваивание», менее подверженные ошибкам, чем традиционный стиль. Такой подход абсолютно необходим для ссылок и констант, которым нельзя присваивать значения, и более эффективен для типов, инициализация которых по умолчанию – дорогая операция. Например:

```
void f(int i, const char* p)
{
   if (i<=0) error("отрицательный индекс");
   const int len = strlen(p);
   String s(p);
   // ...
}
```

Гарантированная инициализация за счет конструкторов – это другая сторона усилий, направленных на уменьшение числа ошибок из-за переменных.

3.11.5.1. Объявления в циклах for

Одна из самых распространенных причин для введения новой переменной в середине блока – объявление переменной цикла. Например:

```
int i;
for (i=0; i<MAX; i++) // ...
```

Чтобы не разделять объявление переменной и ее инициализацию, объявление было разрешено переносить в заголовок цикла:

```
for (int i=0; i<MAX; i++) // ...
```

К сожалению, я не стал менять семантику так, чтобы ограничить область действия переменной, введенной таким образом, областью действия предложения `for`. Основная причина была в том, чтобы избежать специального исключения из правила, гласящего: «область действия переменной простирается от точки объявления до конца наименьшего объемлющего блока».

Это правило стало предметом многочисленных споров и, в конце концов, было пересмотрено, так чтобы соответствовать правилу для объявлений в условиях (см. раздел 3.11.5.2). Именно имя, введенное инициализатором цикла `for`, выходит из области действия в конце этого предложения.

3.11.5.2. Объявления в условиях

Пользователи, добросовестно старающиеся избежать неинициализированных переменных, сталкиваются с такими примерами:

- переменные, используемые для ввода:

```
int i;
cin>>i;
```

- переменные, используемые в условиях:

```
Tok *ct;
if (ct = gettok()) { /* ... */ }
```

При проектировании механизма идентификации типов во время исполнения в 1991 г. (см. раздел 14.2.2.1), я обнаружил, что вторую причину появления неинициализированных переменных можно устранить, если разрешить объявления в условиях:

```
if (Tok* ct = gettok()) {
   // здесь ct находится в области действия
}

// а здесь ct уже вышла из области действия
```

Это не просто трюк с целью уменьшить число вводимых символов, а функционирование в точном соответствии с правилами локального объявления переменных. Объединив в одном предложении объявление переменной, ее инициализацию и проверку результата инициализации, мы добиваемся компактности, позволяющей устранить ошибку, которая может возникнуть вследствие использования неинициализированной переменной. Ограничив область действия такой переменной предложением, управляемым условием, мы также решили проблему случайного «повторного использования» переменных, которые больше не нужны.

На мысль разрешить объявления в выражениях меня натолкнуло изучение языков на базе выражений, прежде всего Algol68. Так, в Algol68 объявление порождало значение, и этот принцип я положил в основу своего дизайна. Позже стало понятно, что память подвела меня: как раз объявления были одной из немногих конструкций в Algol68, которые значений не порождали! Вот что сказал по этому поводу Чарльз Линдсей: «Даже в Algol68 есть несколько изъянов, которые говорят о том, что этот язык не полностью непротиворечив».

Если бы я проектировал язык с нуля, то пошел бы по пути Algol68 и сделал бы каждое предложение и объявление выражением, имеющим значение. Возможно, неинициализированные переменные были бы полностью запрещены и не разрешалось бы указывать более одного имени в одном объявлении. Однако эти идеи, очевидно, неприемлемы для C++.

3.12. Языки C и C++

После появления названия C++ и написания справочного руководства по языку [Stroustrup, 1984] вопросом, которому придавалось большое значение и вокруг которого завязалась ожесточенная полемика, стала совместимость нового языка с C.

Кроме того, в начале 1983 г. подразделение Bell Labs, занимавшееся разработкой и поддержкой UNIX и выпускавшее серию компьютеров AT&T 3B, заинтересовалось C++ настолько, что решило выделить ресурсы под разработку инструментальных средств для него. Наличие инструментария было необходимо для того, чтобы C++ стал языком, который мог использоваться компанией для создания важных проектов. К сожалению, это означало также, что C++ попадает в поле зрения административных структур, отвечающих за разработку.

Сначала руководство потребовало обеспечить стопроцентную совместимость языка с C. Вообще совместимость – это вполне разумно, но практика программирования не так проста. Во-первых, с каким именно C должен быть совместим C++? У C было множество диалектов, и, хотя на горизонте вырисовывался ANSI C, до появления стабильного определения должны были пройти годы, причем диалекты все равно до конца не устранялись. Помню, я тогда подсчитал – в шутку, но ведь в каждой шутке есть доля правды, – что может существовать примерно

3^{42} диалектов ANSI C. При подсчете учитывалось число неопределенных и зависящих от реализации аспектов, а 3 было выбрано в качестве основания как среднее число вариантов каждого такого аспекта.

Естественно, средний пользователь, жаждущий совместимости с C, понимал под ней совместимость с локальным диалектом. Это виделось важной практической проблемой, которой я и мои друзья уделяли много внимания. Администраторам и продавцам до нее было меньше дела: они либо не вполне понимали технические детали, либо их интерес к C++ объяснялся желанием привязать пользователей к своим программным и аппаратным решениям. С другой стороны, разработчики C++ в Bell Labs независимо от того, в каком подразделении они работали, были «эмоционально преданы концепции переносимости» [Johnson, 1992] и противились требованиям администрации закрепить конкретный диалект C в определении C++.

У вопроса о совместимости была и другая, более важная сторона. Какими параметрами C++ должен отличаться от C, чтобы достичь своих основных целей? И в чем именно C++ должен быть совместим с C для достижения этих же целей? Оба аспекта существенны, и при переходе от C with Classes к C++ пересмотр концепций производился в обоих направлениях. Медленно и мучительно было выработано соглашение о том, что между C++ и ANSI C (когда появится стандарт) не будет неоправданных несовместимостей [Stroustrup, 1986], но тем не менее понятие «вынужденной несовместимости» имеет право на существование. Естественно, вопрос о том, что считать «вынужденной несовместимостью», вызывал много споров и отнимал у меня непропорционально много времени и сил. Впоследствии появилась формулировка – «C++: настолько близко к C, насколько возможно, но не ближе», по названию работы, написанной совместно Эндрю Кенигом и мной [Koenig, 1989]. Один из показателей успеха такой стратегии – все примеры из книги K&R2 [Kernighan, 1988] написаны на C, который является подмножеством C++. Примеры программ из K&R2 тестировались с помощью компилятора Cfront.

Некоторые заключения о модульности и о том, как собирать программу из отдельно скомпилированных частей, были изложены в первой редакции справочного руководства по C++ [Stroustrup, 1984]:

- имена являются закрытыми, если не объявлены открытыми;
- имена локальны в файле, где они определены, если явно не экспортированы из него;
- типы контролируются статически, если такой контроль не подавлен;
- класс составляет область действия (откуда следует, что классы могут быть вложенными).

Первый пункт не влияет на совместимость с C, остальные приводят к некоторым расхождениям:

- имя нелокальной функции или объекта в C по умолчанию видимо в других единицах трансляции;
- в C необязательно объявлять функции перед использованием, и по умолчанию проверка типов при вызове не производится;

- имена структур в C не вкладываются (даже если лексически они вложены);
- в C++ есть только одно пространство имен, тогда как в C с каждым «тэгом структуры» связано отдельное (см. раздел 2.8.2).

Сегодня «баталии по поводу совместимости» кажутся мелкими и скучными, но некоторые из поднятых тогда проблем не решены до сих пор. Полагаю, причина, по которой эти «войны» так затянулись и не принесли окончательного результата, заключается в том, что мы не затрагивали более глубоких вопросов о различии целей C и C++ и видели в совместимости множество несвязанных проблем, к каждой из которых следует подходить индивидуально.

Типичный пример – наименее значимый вопрос о «пространствах имен» потребовал больше всего усилий и, в конце концов, был решен путем компромисса [ARM].

Мне пришлось нарушить концепцию класса как области действия и принять «решение» C, иначе не разрешалось выпускать версию 1.0. При этом я не понимал, что структура в C не составляет области действия, поэтому конструкция

```
struct outer {
   struct inner {
       int i;
   };
   int j;
};

struct inner a = { 1 };
```

вполне допустима. Более того, такого рода код встречается в стандартных заголовочных файлах системы UNIX. Когда данная проблема всплыла, уже ближе к концу известных «баталий», у меня не было времени подробно изучать все последствия этого «решения» C. Гораздо проще было согласиться, чем затевать новые споры. В 1989 г., после многих технических сложностей и недовольства пользователей, вложенные области действия классов снова были введены в C++ [ARM] (см. раздел 13.5).

Ожесточенные споры привели к тому, что для C++ был одобрен усиленный контроль типов при вызове функции (без модификаций). Неявное нарушение статического контроля типов стало первым случаем «вынужденной» несовместимости C и C++. Комитет ANSI C одобрил облегченный вариант правил и нотации C++ в этой области и объявил устаревшими такие виды использования, которые не отвечают правилам C++.

Пришлось принять правило C о том, что глобальные имена по умолчанию видимы в других единицах трансляции. Просто не было поддержки для правила, более жестко ограничивающего область видимости имен. Это означало, что в C++, как и в C, не будет механизма для выражения модульности на уровне выше класса или файла. Это послужило причиной многих жалоб, пока комитет ANSI/ISO не принял пространства имен (см. главу 17) в качестве механизма, позволяющего избежать непреднамеренного совмещения имен. Однако Дуг Макилрой и другие возражали, что C-программисты не оценят язык, где каждый объект и функцию,

которые должны быть доступны из другой единицы трансляции, придется явно объявлять таковыми. Возможно, в то время они были правы и уберегли меня от серьезной ошибки. В любом случае теперь я убежден, что первоначальное решение, предлагавшееся для C++, было не слишком изящным.

Обсуждение вопросов совместимости раскололо пользователей на два противоборствующих лагеря, и члены каждого из них были абсолютно убеждены в справедливости своей точки зрения. Первый лагерь требовал стопроцентной совместимости, часто не осознавая ее последствий. Например, многие из поборников полной совместимости с недоумением узнавали, что это приведет к несовместимости с уже существующим C++, в результате чего перестанут компилироваться десятки миллионов строк написанного на C++ кода. Во многих случаях требование стопроцентной совместимости исходило из предположения, что у C++ мало пользователей. Нередко за этим требованием стояло желание скрыть свое незнание C++ или неприятие некоторых его новых возможностей.

В другом лагере объявляли, что проблемы совместимости с C не существует вовсе, и требовали введения таких возможностей, которые вызвали бы серьезные неудобства у всех, кто хотел писать на смеси языков C и C++. Разумеется, чем более «экстремистскими» были притязания, исходящие из одного лагеря, тем глубже окапывались представители другого, боясь потерять те возможности языка, в которых были заинтересованы. В тех случаях (к счастью, почти всегда), где учитывались истинные интересы людей и реальные примеры использования C и C++, споры обычно заканчивались конструктивным рассмотрением деталей компромиссного решения. На организационном собрании комитета ANSI X3J16 Ларри Рослер, первый редактор комитета ANSI C, объяснил скептически настроенному Тому Пламу (Tom Plum), что «C++ – это C, каким мы пытались, но не смогли его сделать». Возможно, данное утверждение чересчур сильно, но для общего подмножества C и C++ оно недалеко от истины.

3.13. Инструменты для проектирования языка

Рассказывая о дизайне и эволюции C++, я почти не уделил внимания теории и инструментам более сложным, чем обычная классная доска. Для работы над грамматикой я пытался использовать YACC (генератор синтаксических анализаторов LALR(1)-грамматик [Aho, 1986]), но потерпел неудачу из-за особенностей синтаксиса C (см. раздел 2.8.1). Пробовал применить денотационную семантику, тоже неуспешно. Рави Сетхи (Ravi Sethi), занимавшийся этой проблемой, понял, что не может выразить семантику C подобным способом [Sethi, 1980].

Основной трудностью была нерегулярность C и большое число зависящих от реализации и неопределенных аспектов. Много позже комитет по стандартизации ANSI/ISO C++ призвал экспертов по формальным определениям растолковать свои методы и инструментарий и высказать мнение о том, в какой мере формализованный подход к определению C++ будет полезен нам при стандартизации. Я знакомился также с формальными спецификациями языков ML и Modula-2, чтобы понять, поможет ли формальный подход получить более короткое и изящное описание, чем описание на обычном английском языке. Не думаю, что формальное

описание C++ оставило бы разработчикам компиляторов и опытным пользователям меньше шансов на неправильную интерпретацию. Я пришел к выводу, что формальное описание языка, при проектировании которого какой-либо метод формального определения не использовался изначально, не под силу никому, кроме горстки экспертов в этой узкой области.

Однако отказ от надежд на формальную спецификацию оставил меня один на один с неточной и недостаточной терминологией. Что можно было сделать в этой ситуации? Я рассказывал о новых возможностях языка коллегам, чтобы проверить свою логику. Однако вскоре у меня выработалось стойкое пренебрежение ко всем имеющимся аргументам, поскольку убедительно обосновать каждую особенность языка не получалось. С другой стороны, нельзя построить язык, который был бы полезен, включая в него все, что кому-нибудь может пригодиться. Полезных возможностей слишком много, и ни в одном языке они не могут быть объединены так, чтобы язык сохранил при этом внутреннюю целостность. Поэтому всюду, где возможно, я старался экспериментировать.

К сожалению, чистый эксперимент поставить обычно не удается. Невозможно создать две полномасштабных системы, содержащих компилятор, набор инструментальных средств и документацию, а потом попросить одну группу людей пользоваться первой, другую – второй и сравнить результаты. Реализовав то или иное средство, мы с немногими коллегами пытались применить его на практике, и я, как мог, старался проявлять крайнее недоверие ко всем положительным откликам. По мере возможности я старался учитывать мнения опытных программистов. Так я пытался компенсировать фундаментальные ограничения своих опытов. Обычно эксперимент состоял в сравнении реализаций, изучении качества исходного кода небольших примеров и в измерении производительности и потребления памяти на таких примерах. По крайней мере, в процессе проектирования у меня была обратная связь с практиками, поэтому я мог полагаться на экспериментальные результаты, а не только на плоды абстрактных размышлений. Убежден, что проектирование языка – это не упражнение в голом теоретизировании, а тесно связанный с практикой процесс учета различных потребностей, методов и ограничений. Хороший язык не просто хорошо спроектирован, он выращен. Эта работа имеет больше общего с инженерной практикой, социологией, философией, нежели с математикой.

Оглядываясь назад, я жалею, что не нашел способа формализовать правила преобразования типов и сопоставления формальных и фактических аргументов. У меня есть подозрение, что никакой формализм не смог бы справиться с крайней нерегулярностью правил C в отношении встроенных типов и операторов.

У дизайнера языка есть большое искушение предоставить специальные средства там, где пользователям пришлось бы искать обходные пути. Недовольство по поводу отказа добавлять что-то обычно гораздо сильнее, чем жалобы на то, что «добавлена еще одна бесполезная возможность». Это серьезная проблема и для комитетов по стандартизации (см. раздел 6.4), и худший вариант в этом смысле – культ ортогональности. Многие думают, что если от добавления некоей возможности язык становится более ортогональным, то и спорить больше не о чем. Обычно же, вопреки всем благим намерениям относительно ортогональности, определение

комбинации средств все же требует дополнительной работы над справочным руководством и руководством пользователя. Чаще всего реализация комбинации, которую предписывают идеалы ортогональности, оказывается сложнее, чем многие себе представляют. В случае C++ я всегда учитывал, во что (с точки зрения времени и памяти) обойдется ортогональность тем, кто не использует данную комбинацию. Если цену нельзя было, по крайней мере, в принципе свести к нулю, то я очень неохотно шел на добавление новой возможности, какой бы ортогональной она ни была.

Мне казалось, да и сейчас кажется, что многие языки программирования и инструментальные средства дают решения, которые так и влекут за собой неприятности, и я ни в коем случае не хотел допустить, чтобы мою работу постигла та же участь. Поэтому я читаю литературу по языкам программирования, принимаю участие в дискуссиях на эту тему, но в основном ищу идеи для решения проблем, с которыми я и мои коллеги сталкивались в реальных приложениях. В других языках скрыта масса вдохновляющих возможностей, только надо тщательно просеять их, чтобы не поддаться соблазну, утратив чувство меры, и избежать внутренних противоречий. Основными источниками идей для C++ были Simula, Algol68, а позже – Clu, Ada и ML. Ключ к хорошему дизайну – глубокое понимание стоящих перед языком задач, а не включение самых передовых идей.

3.14. Книга «Язык программирования C++»

Осенью 1984 г. сидевший в соседнем с моим кабинете Эл Ахо предложил мне написать книгу по C++, организованную по образу и подобию книги Брайана Кернигана и Денниса Ричи «Язык программирования C» [Kernighan, 1978]. В основу труда должны были лечь мои статьи, внутренние отчеты и справочное руководство по C++. Работа над книгой заняла девять месяцев. Я закончил ее в середине августа 1985 г., а первые экземпляры появились в середине октября. В предисловии упоминаются те, кто внес наибольший вклад в C++: Том Каргилл, Джим Коплиен, Стью Фельдман, Сэнди Фрэзер, Стив Джонсон, Брайан Керниган, Барт Локанти (Bart Locanthi), Дуг Макилрой, Деннис Ричи, Ларри Рослер, Джерри Шварц и Джонатан Шопиро.

«C++ проектировался прежде всего для того, чтобы автору и его друзьям не приходилось программировать на ассемблере, C или современных языках высокого уровня. Основная задача C++ – сделать написание хороших программ более простым и приятным занятием для профессионального программиста», – говорилось вначале.

Объект моей работы – человек, индивидуум (неважно, является ли он членом какого-то коллектива или нет), программист. С годами я становился все более привержен этой идее и еще сильнее подчеркнул ее во втором издании [2nd], где гораздо глубже обсуждались вопросы проектирования и разработки программного обеспечения.

Книга «Язык программирования C++» представляла собой определение языка C++ и введение в него. Она писалась с твердой решимостью не отдавать предпочтение никаким особым приемам программирования в ущерб всем прочим.

Точно так же, как я боялся внести ограничения в язык по невежеству, я не хотел, чтобы эта книга была манифестом моих персональных предпочтений.

3.15. Статья «Whatis?»

После выпуска версии 1.0 и отправки оригинал-макета книги в издательство у меня появилось время вернуться к рассмотрению крупных вопросов и документировать весь ход проектирования. Как раз в это время мне позвонил из Осло Карел Бабчиски, председатель Ассоциации пользователей Simula (ASU), и пригласил выступить с докладом о C++ на конференции ASU 1986 г. в Стокгольме. Конечно, я хотел поехать, но был обеспокоен тем, что презентация C++ на конференции по Simula может быть расценена как вульгарная самореклама и попытка отвратить пользователей от этого уважаемого языка. В конце концов я сказал: «C++ – не Simula, так почему пользователи Simula захотят слушать о нем?» На это Бабчиски ответил: «Да мы же не цепляемся за синтаксис». Это дало мне возможность написать не только о том, что такое C++, но и о том, чем он предположительно станет, и о том, где пришлось отступить от идеалов. В результате появилась статья «What is Object-Oriented Programming» («Что такое объектно-ориентированное программирование»), [Stroustrup, 1986b]. Развернутый вариант работы представлен на первой конференции ECOOP в июне 1987 г. в Париже.

В этой статье впервые изложены те приемы и методы, поддержку которых должен был обеспечить C++. Все предшествующие презентации ограничивались описанием тех возможностей, которые уже реализованы и применяются. В статье «Whatis?» был сформулирован ряд проблем, которые, как я надеялся, сможет решить язык, поддерживающий абстракцию данных и объектно-ориентированное программирование. Здесь также приводились примеры средств, которые я считал необходимыми.

В работе вновь подчеркивалась природа C++ как языка, поддерживающего несколько парадигм:

«Объектно-ориентированное программирование – это программирование с применением наследования. Абстрагирование данных – это программирование с использованием определенных пользователем типов. За немногими исключениями, объектно-ориентированное программирование может и должно быть надмножеством абстрагирования данных. Чтобы эти методы были эффективны, необходима надлежащая поддержка. Поддержка абстрагирования данных требует языковых средств, а для объектно-ориентированного программирования нужна еще и поддержка со стороны среды. Чтобы претендовать на звание языка общего назначения, язык, поддерживающий абстрагирование данных или объектно-ориентированное программирование, должен эффективно работать с распространенной аппаратурой».

Особо говорилось о важности статического контроля типов. Другими словами, модель наследования и контроля C++, скорее, построена на базе Simula, чем Smalltalk:

«Класс в Simula или C++ определяет неизменный интерфейс к множеству объектов, принадлежащих любому производному классу, тогда как в Smalltalk класс определяет начальный набор операций с объектами (любого подкласса). Иначе говоря, в Smalltalk класс – минимальная спецификация и пользователь может попробовать выполнить неуказанные операции, в то время

как в C++ класс – точная спецификация и гарантируется, что компилятор не пропустит операций, не указанных в объявлении класса».

Такой подход оказывает огромное влияние на способ проектирования систем и на выбор языковых средств. Язык с динамическими типами, вроде Smalltalk, упрощает проектирование и реализацию библиотек, позволяя отложить проверку типов до исполнения программы. Например (используем синтаксис C++):

```
void f()  // только динамический контроль типов, это не C++
{
   stack cs;
   cs.push(new Saab900);
   cs.pop()->takeoff(); // Увы! Ошибка времени исполнения.
                        // В классе car нет метода takeoff
}
```

Такое отложенное обнаружение ошибок типизации было сочтено в C++ неприемлемым. Но следовало найти способ добиться удобства обозначений и построения библиотек не хуже, чем в языках с динамическими типами. В качестве решения (будущего) этой проблемы в C++ была представлена концепция параметризованных типов:

```
void g()
{
   stack(plane*) cs;

   cs.push(new Saab37b); // правильно: Saab37b - это объект типа plane
   cs.push(new Saab900); // ошибка: несоответствие типов
                         // передан car, ожидался plane

   cs.pop()->takeoff(); // во время исполнения проверка не нужна
   cs.pop()->takeoff(); // во время исполнения проверка не нужна
}
```

Такие ошибки было решено обнаруживать именно на этапе компиляции потому, что, как было подмечено, C++ часто используется для написания программ, исполняемых в отсутствие программиста. Статический контроль типов мы принципиально считали не просто методом повышения производительности, а лучшим способом в максимально возможной мере гарантировать правильность программы.

Но главная причина, по которой я полагаюсь на статически контролируемые интерфейсы, – моя твердая убежденность, что программа, составленная из статически проверенных частей, может лучше отразить хорошо продуманный дизайн, нежели программа, зависящая от слабо или динамически типизированных интерфейсов. При этом необходимо помнить, что не всякий интерфейс поддается одной лишь статической проверке, а из того, что программа прошла статический контроль, не следует, что в ней не осталось ошибок.

В статье «Whatis?» перечислены три изъяна C++:

- «Ada, Clu и ML поддерживают параметризованные типы. В C++ этого нет. Там, где необходимо, параметризованные классы «имитируются» с помощью макросов. Ясно, что параметризованные классы были бы очень полезны в C++. Встроить соответствующие

средства в компилятор легко, но современная система программирования для C++ недостаточно развита, чтобы поддержать их без заметных расходов и неудобств. Употребление параметризованных типов не должно приводить ни к каким новым накладным расходам по сравнению с обычными»;

- «по мере увеличения размера программы и особенно в случае интенсивного использования библиотек возрастает значимость стандартной обработки ошибок (или, в более общем смысле, «исключительных ситуаций»). В Ada, Algol68 и Clu таким стандартом является механизм обработки исключений. Увы, в C++ его нет. Там, где необходимо, исключения подменяются указателями на функции, «объектами исключений», «ошибочными состояниями» и библиотечными функциями `signal` и `longjmp`. В обычной ситуации этого недостаточно. С помощью этих средств не удается даже сформулировать единый подход к обработке ошибок»;
- «принимая во внимание это объяснение, представляется очевидным, что наследование классом B двух классов A1 и A2 могло бы быть полезным. Этот механизм называется множественным наследованием».

Все три возможности были упомянуты в контексте построения лучших (то есть более общих и более гибких) библиотек. Все они вошли теперь в C++ (шаблоны, см. главу 15; исключения, см. главу 16; множественное наследование, см. главу 12). Отмечу, что добавление множественного наследования и шаблонов рассматривалось в качестве вероятных направлений развития C++ еще в работе [Stroustrup, 1982b]. Там же, как одна из возможностей, упоминалась обработка исключений.

Как обычно, я подчеркнул, что требования к эффективности по скорости и памяти, а равно к возможности сосуществовать с другими языками в широко распространенных системах ограничивают язык, но, с другой стороны, язык, претендующий на звание «универсального», не может отказаться от этих требований.

Глава 4. Правила проектирования языка C++

Если карта не соответствует местности,
доверяй местности.

Заповедь швейцарской армии

4.1. Правила и принципы

Чтобы с языком программирования было полезно и приятно работать, он должен быть создан на основе некоего общего взгляда, определяющего проектирование индивидуальных особенностей. Для C++ такой взгляд оформлен в виде набора правил и ограничений. Я употребляю слово «правила», поскольку понятие «принципы» кажется мне излишне претенциозным в контексте такой дисциплины, как проектирование языков программирования, где настоящих научных принципов не так уж и много. Кроме того, для многих людей словосочетание «научный принцип» прочно ассоциируется с отсутствием исключений, а это вовсе нереально. На самом деле, если правило и практика вступают в конфликт, то, уверен, уступить должно правило. Нижеизложенные правила нельзя применять бездумно, но нельзя и заменить их ни к чему не обязывающими лозунгами. Как проектировщик языка, я видел свою работу в том, чтобы определить, какие задачи следует и можно решать в рамках C++ и как поддержать баланс между различными правилами при проектировании конкретного средства.

Правила служили путеводной нитью при работе над конкретными средствами. Однако общий план усовершенствований был продиктован фундаментальными целями C++ (см. табл. 4.1).

Таблица 4.1

Цели
C++ делает программирование более приятным занятием для серьезных программистов
C++ – это язык программирования общего назначения, который
❑ лучше C
❑ поддерживает абстракции данных
❑ поддерживает объектно-ориентированное программирование

Я разбил все правила на четыре большие группы. Первая содержит общие критерии языка в целом – настолько общие, что к отдельным возможностям они

не относятся. Во вторую группу включены правила, относящиеся к роли C++ как инструмента поддержки проектирования. В третьей группе собраны аспекты, касающиеся технических, формирующих облик языка деталей. А правила из четвертой группы акцентируют внимание на роли C++ как языка для низкоуровневого программирования.

Конечно, формулировки отточены уже задним числом, но эти правила определяли ход моих мыслей еще до завершения работы над первой версией C++ в 1985 г., а многие из них, как упоминалось в предыдущих главах, легли в основу исходной концепции C with Classes.

4.2. Общие правила

Самые общие и важные правила C++ мало связаны с техническими вопросами (см. табл. 4.2). Основной упор в них сделан на интересах сообщества пользователей. Природа C++ в значительной мере определена моим желанием помочь современному поколению системных программистов в решении актуальных задач. Подчеркну, что смысл слова «современный» изменяется со временем, а значит, C++ должен эволюционировать, чтобы соответствовать потребностям своих пользователей; он не может быть определен раз и навсегда.

Таблица 4.2

Общие правила
Эволюция C++ должна определяться реальными задачами
Не устремляться в бесплодную погоню за совершенством
C++ должен быть полезен сейчас
Каждое средство должно иметь достаточно ясную реализацию
Всегда оставлять путь для перехода
C++ – это язык, а не законченная система
Предоставлять всестороннюю поддержку для каждого поддерживаемого стиля программирования
Ничего не заставлять делать насильно

Эволюция C++ должна определяться реальными задачами. В информатике, как и во многих других областях, можно наблюдать, как люди ищут задачу, к которой можно было бы применить свое любимое решение. Я не знаю безошибочного способа уберечься от субъективности при определении того, что же является действительно важным, но точно знаю, что многие возможности, которые мне предлагали включить в язык, не уживаются с общей идеологией C++ и зачастую не нужны программистам, работающим над реальными проблемами.

Единственным мотивом для внесения в C++ каких-либо изменений является продемонстрированная несколькими независимыми программистами недостаточная для решения их задач выразительность языка. Если только возможно, я привлекаю пользователей к поиску решения и воплощению его в жизнь.

Не устремляться в бесплодную погоню за совершенством. Любой язык программирования несовершенен. И никогда не будет идеального языка, поскольку задачи и системы не стоят на месте. Годами шлифовать язык, пытаясь создать идеальную модель, – значит лишать программистов тех преимуществ, которые появились за это время. Тем самым проектировщик языка оставляет себя без обратной связи, и язык может развиться до такой степени, что станет уже никому не нужен. С другой стороны, программисты тратят больше всего времени на модификацию старого кода или попытки сопряжения с ним новых программ. Для работы им нужна стабильность. Если язык реально используется, то в нем неприемлемы радикальные изменения. Даже относительно мелкие модификации трудно внести, не навредив пользователям. Поэтому решение пойти на серьезное усовершенствование должно опираться на надежную обратную связь с аудиторией, и принимать ту или иную поправку нельзя без серьезного рассмотрения вопросов совместимости, перехода на новую версию и обучения. По мере того как язык становится все более зрелым, любым изменениям в нем следует предпочитать альтернативы в виде инструментальных средств, специальных приемов и библиотек.

Не каждую задачу нужно решать на C++, и не каждая проблема, имеющаяся в C++, настолько серьезна, чтобы заниматься ею. Например, не следует включать в сам язык средства для сопоставления с образцом или доказательства теорем, а хорошо известные недостатки C++, связанные с неудачно выбранными приоритетами операций (см. раздел 2.6.2), лучше не пытаться исправлять, а ограничиться предупреждениями компилятора.

C++ должен быть полезен сейчас. Многие занимаются программированием для решения сиюминутных задач, работают на довольно слабых компьютерах с устаревшими ОС и инструментальными средствами. Некоторым программистам не хватает формального образования. У многих нет времени повышать свою квалификацию. Таким образом, C++ должен быть полезен любому, кто обладает средней квалификацией и обычным компьютером.

Хотя временами меня и подталкивали к этому, я никогда не собирался бросить таких пользователей на произвол судьбы и начать ориентироваться только на системы высшего класса и вкусы профессиональных ученых в области информатики.

Смысл этого правила – как и большинства других – изменяется со временем и в какой-то мере с распространением C++. Сейчас доступны более мощные компьютеры, растет и число программистов, знакомых с базовыми концепциями и методами, на которые опирается C++. А когда человек ставит перед собой более амбициозные цели, изменяются и задачи, с которыми он сталкивается. Отсюда следует, что сегодня можно и должно подумать о средствах, для которых требуются больше вычислительных ресурсов и более высокая квалификация программистов. Примерами могут служить обработка исключений (см. главу 16) и идентификация типов во время исполнения (см. раздел 14.2).

Каждое средство должно иметь достаточно ясную реализацию. Для правильной и эффективной реализации любого средства не должно возникать нужды в сложных алгоритмах. В идеале должны существовать очевидные стратегии анализа и генерации кода, достаточно удобные для применения на практике. Большинство возможностей удалось реализовать, проверить и подвергнуть окончательной

ревизии. Там, где не соблюдалась такая схема, например в случае с механизмом инстанцирования шаблонов (см. раздел 15.10), возникали неприятности.

Однако пользователей гораздо больше, чем разработчиков компиляторов. Поэтому всякий раз, когда приходится выбирать между сложностью компилятора и применения, решение должно приниматься в интересах пользователей. Я заслужил право на такое мнение, поскольку в течение многих лет сам занимался сопровождением компилятора.

Всегда оставлять путь для перехода. C++ должен развиваться постепенно, чтобы обслуживать потребности своих пользователей. Также нельзя обойтись и без обратной связи с сообществом. Значит, необходимо внимательно следить за тем, чтобы ранее написанный код продолжал работать. Если несовместимости не избежать, то надо приложить все усилия, чтобы помочь пользователям модернизировать свои программы. Точно так же следует указать путь перехода от неудачных приемов программирования на C к более результативному использованию C++.

Общая стратегия исключения из языка небезопасного, провоцирующего ошибки, да и просто неудачного средства должна заключаться в том, чтобы сначала предложить более удачную альтернативу, затем порекомендовать не пользоваться устаревшим средством и лишь спустя несколько лет (а может быть, и никогда) убрать изжившую себя возможность. Эту стратегию можно поддержать с помощью предупреждений компилятора. Часто не представляется возможным отказаться от некоторого средства или исправить ошибку (причина, как правило, в необходимости сохранять совместимость с C); тогда остаются только предупреждения (см. раздел 2.6.2). Таким образом, реализация C++ может оказаться более безопасной, чем следует из определения языка.

C++ – это язык, а не законченная система. Среда программирования состоит из многих компонентов. Можно объединить их в единую – интегрированную – систему. Другой подход состоит в том, чтобы сохранить классическое разграничение между такими частями среды, как компилятор, компоновщик, библиотеки поддержки исполнения, библиотеки ввода/вывода, редактор, файловая система, база данных и т.д. Языку C++ ближе второй путь. С помощью библиотек, соглашений о вызове и пр. C++ приспосабливается к соглашениям, принятым в конкретной системе, при соблюдении которых возможно взаимодействие с другими языками и использование имеющихся инструментальных средств. Это ключ к переносимости реализации и, что важнее, к возможности обращаться к коду, написанному на других языках. Это также позволяет пользоваться общими инструментами, облегчает совместную работу программистов, предпочитающих разные языки, и дает возможность одному человеку работать на разных языках.

По своему замыслу C++ – это лишь один язык среди многих. Он позволяет разрабатывать инструментальные средства, но не требует никаких конкретных форм. У программиста остается свобода выбора. C++ и сопутствующий инструментарий должны хорошо вписываться в данную систему. Особенно важно это для больших систем и систем с необычными ограничениями. Такие системы, как правило, не очень хорошо поддерживаются, поскольку «стандартные» инструментальные средства ориентированы преимущественно на отдельных программистов и небольшие коллективы, выполняющие «среднюю» работу.

Предоставлять всестороннюю поддержку для каждого поддерживаемого стиля программирования. C++ должен развиваться, чтобы не отставать от потребностей серьезных программистов. Простота, конечно, важна, но нужно принимать во внимание и сложность проектов, в которых используется C++. Удобство сопровождения системы, написанной на C++, и ее производительность считаются более важными, нежели краткость определения языка. В результате язык получается довольно сложным.

Отсюда также следует, что нужно поддерживать много гибридных стилей программирования. Программисты пишут не только классы, укладывающиеся в узкие рамки либо абстрактного типа данных, либо объектно-ориентированного стиля. Иногда приходится писать классы, имеющие признаки того и другого одновременно. Нередки и программы, части которых написаны в разных стилях, отражающих вкусы и потребности автора.

Следовательно, языковые средства нужно проектировать так, чтобы их можно было применять в разных сочетаниях. Отсюда вытекает некоторая ортогональность проектирования C++. Возможность «необычного» использования – важный источник гибкости, заложенной в C++. Так, в этом языке правила контроля доступа, поиска подходящего имени, привязки виртуальных и невиртуальных функций и проверки типов ортогональны. Это делает возможным применение различных приемов, опирающихся на сокрытие информации и наследование классов. Пользователи, предпочитающие видеть лишь жестко определенные стили программирования, называют это «хакерством». С другой стороны, ортогональность не относится к числу первостепенных принципов языка.

C++ – развитый язык, и из этого следует, что программисту придется не только изучить библиотеки и отдельные программы, но и освоить язык и базовые приемы его использования для проектирования. Для большинства пользователей такое смещение акцентов, привыкание к новым методам программирования и включение в свой арсенал прогрессивных средств должно быть постепенным. Лишь немногие могут усвоить все сразу и применить вновь приобретенные навыки на практике (см. раздел 7.2). C++ спроектирован так, чтобы постепенное освоение было возможным и естественным. Общий принцип таков: если не знаешь, то и не страдаешь. Статическая система контроля типов и предупреждения компилятора всегда помогут.

Ничего не заставлять делать насильно. Программисты – толковые ребята. Перед ними ставятся непростые задачи, поэтому им пригодится все, что можно получить от языка программирования и сопутствующих инструментальных средств. Попытка ограничить пользователя, заставив его «делать правильно», заведомо бессмысленна и обречена на провал. Программист найдет способ обойти ограничение, которое считает неприемлемым. Следовательно, язык должен поддерживать широкий диапазон разумных стилей проектирования и программирования, а не навязывать единую концепцию.

Вышесказанное не означает, что все способы программирования равно хороши или что C++ должен поддерживать любой имеющийся стиль. C++ проектировался для прямой поддержки стилей проектирования, базирующихся на статическом контроле типов, абстрагировании данных и наследовании. Однако

наставления на тему о том, как правильно использовать эти средства, сведены к минимуму, и никакие средства не добавлялись в язык и не исключались из него только ради того, чтобы воспрепятствовать некоему логически последовательному стилю программирования.

Я хорошо знаю, что далеко не все ценят свободу выбора и разнообразие. Однако те, кто предпочитает более ограниченную среду, вольны наложить на C++ собственные правила или выбрать язык, предоставляющий программисту меньший выбор.

Многим программистам не нравится, когда им говорят, что «вот тут возможна ошибка», если они точно знают, что ошибки здесь быть не может. Поэтому «потенциальные ошибки» в C++ таковыми не являются. Например, не считается ошибкой объявление, в принципе допускающее неоднозначное использование. Ошибкой будет само неоднозначное использование, а не его возможность. Мой опыт показывает, что такие «потенциальные ошибки» никогда не проявляются, поэтому откладывание диагностического сообщения обычно означает, что оно так и не будет выдано. Это приносит дополнительное удобство и гибкость.

4.3. Правила поддержки проектирования

Перечисленные в таблице 4.3 правила относятся главным образом к роли C++ как инструмента поддержки проектирования на основе абстрагирования данных и объектно-ориентированного программирования. Иными словами, язык в них рассматривается, скорее, как средство организации процесса мышления и выражения идей на достаточно высоком уровне, нежели как «высокоуровневый ассемблер», что характерно для C или Pascal.

Таблица 4.3

Правила поддержки проектирования
Поддерживать устоявшиеся методы проектирования
Предоставлять средства для организации программ
Точно говорить то, что имеется в виду
Все возможности должны быть адекватны
Важнее включить полезную возможность, чем предотвращать неправильное использование
Поддерживать сборку программ из независимо разработанных частей

Поддерживать устоявшиеся методы проектирования. Каждое языковое средство должно укладываться в общую структуру. Наличие таковой позволяет ответить на вопрос, какие возможности были бы желательны. Сам язык тут не помощник, системообразующие принципы закладываются на другом концептуальном уровне. Для C++ это уровень, на котором формулируются идеи о возможностях проектирования программы.

Моей целью было поднять уровень абстракции в системном программировании. Точно так же C заменил ассемблер в роли главной опоры системщика. Все

идеи по поводу новых возможностей рассматриваются в свете того, как они помогают улучшить выразительность языка для описания проектирования. В частности, способствуют ли они более эффективному представлению концепции в виде класса? Это ключ к тому, как в C++ поддерживаются абстракции данных и объектно-ориентированное программирование.

Язык программирования не может и не должен быть полноценным языком проектирования. Последний, разумеется, богаче. Однако язык программирования по возможности напрямую поддерживает некоторые принципы проектирования, что облегчает взаимодействие между проектировщиками и программистами и создание инструментальных средств.

Оценка языка программирования с точки зрения методов проектирования позволяет принимать или отвергать те или иные средства в зависимости от того, насколько удачно они соотносятся с поддерживаемыми стилями проектирования. Один язык не может поддержать все мыслимые стили, но должен быть адаптирован хотя бы к нескольким. Расширение C++ в направлении поддержки методик проектирования, ориентированных на «улучшенный C», абстракцию данных и объектно-ориентированное программирование, помогло избежать искушения сделать язык «всем для всех», и в то же время сохранило постоянный стимул для улучшений.

Предоставлять средства для организации программ. По сравнению с C, C++ помогает организовать программу так, чтобы ее было легко писать, читать и сопровождать. Я считал, что осуществлять вычислительную работу с C очень удобно. Конечно, у меня есть свои мысли, как можно было бы улучшить выражения и предложения C, но я решил сосредоточить усилия на другом. Когда кто-то предлагал новый вид выражений или предложений, я оценивал его по тому, может ли оно улучшить структуру программы или просто слегка упростит некоторые вычисления. За немногими исключениями (например, разрешение включать объявление при первом обращении к переменной, см. раздел 3.11.5), выражения и предложения C не подвергались модификациям.

Точно говорить то, что имеется в виду. Фундаментальная проблема низкоуровневых языков программирования заключается в следующем. Имеется огромный разрыв между тем, что люди говорят друг другу в беседе, и тем, что они могут выразить на данном языке. Базовая структура программы теряется за нагромождением битов, байтов, указателей, циклов и т.д. Основное средство преодоления такого разрыва – сделать язык более декларативным. Почти все предоставляемые C++ средства предполагают объявление чего-либо, а затем использование дополнительных конструкций для проверки непротиворечивости, обнаружения глупых ошибок и генерирования оптимизированного кода.

Там, где нельзя воспользоваться декларативной структурой, может помочь явная нотация. Хорошими примерами служат операторы выделения и освобождения памяти (см. раздел 10.2) и новый синтаксис приведения типов (см. раздел 14.3). Концепция прямого и явного выражения намерений на первых порах формулировалась так: «позволить выразить все важное самим языком, а не с помощью комментариев или трюков с макросами». Отсюда следует, что язык в целом

и система контроля типов в особенности должны быть более выразительными, чем в предшествующих языках общего назначения.

Все возможности должны быть адекватны. Недостаточно просто дать пользователю некоторое средство или порекомендовать способ решения определенной задачи. Предлагаемый вариант не должен требовать слишком больших затрат. В противном случае совет звучит, как издевательство. Конечно, в ответ на вопрос «Как лучше всего добраться до Мемфиса?» можно посоветовать нанять реактивный самолет, но вряд ли это предложение устроит кого-нибудь, кроме миллионеров.

Любое средство включалось в C++, только когда было невозможно добиться той же функциональности с меньшими затратами. Мой опыт показывает, что при наличии выбора между эффективностью и изяществом, большинство программистов предпочтет эффективность. Например, встраиваемые функции были введены для того, чтобы, с одной стороны, уменьшить плату за пересечение границ защиты, с другой – чтобы предложить улучшенную альтернативу макросам. Конечно, в идеале средство должно быть одновременно и изящным, и эффективным. Там же, где это не получалось, средство либо не предоставлялось вовсе, либо – если без него нельзя было обойтись – делался выбор в пользу эффективности.

Важнее включить полезную возможность, чем предотвращать неправильное использование. Плохие программы можно писать на любом языке. Важно уменьшить шансы на случайное неправильное применение тех или иных возможностей. Поэтому я потратил много усилий, чтобы принимаемое по умолчанию поведение конструкций C++ было либо разумно, либо приводило к ошибке компиляции. Например, по умолчанию проверяются все аргументы функции. По умолчанию же все члены класса являются закрытыми. Однако язык, предназначенный для системного программирования, не может помешать решительно настроенному программисту «обмануть» систему, поэтому лучше направить силы на разработку средств, помогающих писать хорошие программы, чем не дающих написать плохую программу. Похоже, что со временем программисты обучаются. Это вариант старого лозунга, пришедшего еще из C, – «доверяй программисту». Различные способы контроля типов и правила ограничения доступа предназначены для того, чтобы разработчик класса мог ясно выразить, чего он ожидает от пользователей с целью предотвращения случайных ошибок, но никак не с целью воспрепятствовать преднамеренному нарушению (см. раздел 2.10).

Поддерживать сборку программ из независимо разработанных частей. Чем сложнее система, чем больше программа, чем ограниченнее ресурсы, тем более обширная поддержка необходима программистам. При проектировании C++ много усилий было потрачено на то, чтобы предоставить поддержку в описанных условиях. Большие и сложные приложения должны составляться из независимых частей, иначе управлять ими будет невозможно.

Для такой цели подойдет все, что позволяет разрабатывать компоненты независимо, а затем использовать их в крупной системе не модифицируя. Эволюция C++ во многом проходила под влиянием этой идеи. Классы работают в данном направлении, а абстрактные классы (см. раздел 13.2.2) явно поддерживают разделение интерфейса и реализации. На самом деле классы можно использовать для выражения широкого спектра стратегий сопряжения [Stroustrup, 1990b]. Исключения

позволяют вынести обработку ошибок из библиотеки (см. раздел 16.1), шаблоны обеспечивают композицию с помощью подстановки типов (см. разделы 15.3, 15.6 и 15.8), пространства имен решают проблему непреднамеренного совмещения имен (см. раздел 17.2), а идентификация типов во время выполнения отвечает на вопрос, что делать, если истинный тип объекта «потерялся» где-то в библиотечных функциях (см. раздел 14.2.1).

Из утверждения «программисту нужна тем большая поддержка, чем крупнее система» следует, что для обеспечения эффективности нельзя полагаться лишь на методы оптимизации кода, которые хорошо работают только для сравнительно небольших программ. Стало быть, решение о размещении объекта в памяти должно приниматься на основе отдельно взятой единицы трансляции, а виртуальные функции должны компилироваться в эффективный код без расчета на глобальную оптимизацию, в ходе которой обрабатываются все единицы трансляции. Это верно, даже если в качестве эталона эффективности берется C. Последующая оптимизация возможна на этапе, когда доступна информация обо всей программе. Например, если в программе появляется вызов виртуальной функции, иногда есть возможность (в отсутствии динамического связывания) определить, какая же функция на самом деле вызывается. В таком случае вызов виртуальной функции можно заменить вызовом обычной или даже встроить такой вызов. Существуют компиляторы C++, способные на такое. Однако для генерирования эффективного кода подобные оптимизации не являются необходимыми, их стоит считать лишь дополнительным удобством, когда эффективность исполнения важнее времени компиляции, а от динамического связывания классов можно отказаться. Даже если такая глобальная оптимизация признается неразумной, вызов виртуальной функции все равно можно оптимизировать, когда она вызывается для объекта известного типа; еще в Cfront версии 1.0 была возможность осуществлять это.

Поддержка больших систем обсуждается в главе 8.

4.4. Технические правила

Следующие правила говорят о способах выражения в C++ (см. табл. 4.4).

Таблица 4.4

Технические правила
Никаких неявных нарушений статической системы типов
Предоставлять для определенных пользователем типов такую же полноценную поддержку, как для встроенных
Локальность – это прекрасно
Избегать зависимости от порядка
Если есть сомнения, выбирать такой вариант средства, которому легче обучить
Синтаксис важен, хотя бывает и нелогичным
Использование препроцессора должно быть исключено

Никаких неявных нарушений статической системы типов. При создании каждого объекта указывается его тип: `double`, `char*` или `dial_buffer`. Если способ

использования объекта противоречит его типу, то говорят о нарушении правил контроля типов. Язык, где такие нарушения в принципе невозможны, называется сильно типизированным. Язык, в котором каждое такое нарушение обнаруживается во время компиляции, называется статически сильно типизированным.

C++ наследовал от C такие свойства, как объединения (unions), приведение типов и массивы, которые не позволяют обнаружить все нарушения во время компиляции. Сегодня в C++ не допускаются неявные нарушения правил контроля типов. Это означает: для того чтобы обойти систему, вам придется явно использовать объединение, приведение типов или массив, явно сказать, что аргумент функции не должен контролироваться, или явно воспользоваться небезопасной C-компоновкой. При любом использовании небезопасных средств на этапе компиляции может выдаваться предупреждение. Еще важнее то, что теперь в C++ имеются средства, которые удобнее эквивалентных небезопасных средств и не уступают им в эффективности. В качестве примера можно привести производные классы (см. раздел 2.9), стандартный шаблон класса массива (см. раздел 8.5), типобезопасную компоновку (см. раздел 11.3) и динамически проверяемые приведения типов (см. раздел 14.2). Из-за требований совместимости с C и уважения к сложившейся практике путь к такому положению был длинным и тернистым; большинству программистов еще только предстоит усвоить более безопасные приемы.

Всюду, где возможно, проверки производятся на этапе компиляции. Если что-то нельзя проверить из-за отсутствия необходимой информации в данной единице трансляции, прилагаются все усилия, чтобы выполнить проверку во время редактирования связей. Чтобы помочь программисту в ситуации, где компилятор и компоновщик оказались бессильны, имеется механизм идентификации типов во время выполнения (см. раздел 14.2) и исключения (см. главу 16).

Предоставлять для определенных пользователем типов такую же полноценную поддержку, как для встроенных. Поскольку определенные пользователем типы – это главное, что есть в программе на C++, поддержка для них должна быть максимально обширной. Поэтому ограничения типа «объекты классов могут распределяться только в свободной памяти» признаны неприемлемыми. Необходимость в настоящих локальных переменных для таких арифметических типов, как `complex`, привела к тому, что поддержка для типов, имеющих значение (конкретных типов), оказалась сравнимой с поддержкой для встроенных типов, а иногда даже превосходящей ее.

Локальность – это прекрасно. Весьма желательно, чтобы написанный код был замкнутым за исключением тех случаев, когда ему нужны внешние сервисы. Хорошо, если за такими сервисами можно обращаться без особых хлопот. И наоборот, когда функции, классы и т.п. предоставляются другим частям программы, хотелось бы не опасаться взаимодействия деталей реализации и постороннего кода.

До этих идеалов языку C далеко, как до небес. Любая глобальная функция или глобальная переменная видна компоновщику, поэтому ее имя будет конфликтовать с такими же именами в других частях программы, если только оно явно не объявлено как `static`. Любое имя может использоваться как имя функции, даже

если оно не объявлено заранее. До сих пор имена структур, объявленных внутри структур, глобальны, это пережиток тех дней, когда глобальными были вообще все имена членов структур. Помимо всего прочего, при обработке макросов препроцессор не принимает во внимание области действия, поэтому любая последовательность символов в программе может быть заменена чем-то иным, если модифицировать заголовочный файл или применить другой флаг компиляции (см. раздел 18.1). Все это дает чрезвычайно мощные средства, если вы хотите изменить семантику локального, на первый взгляд, фрагмента кода или, наоборот, произвести существенную трансформацию путем маленькой «локальной» модификации. В целом это кардинально противоречит моим представлениям о создании и сопровождении сложных программных систем. Поэтому я поставил себе целью обеспечить лучшую защиту от «проникновений извне» и лучший контроль над тем, что «экспортируется» из моего кода.

Классы – первый и самый главный механизм локализации кода и предоставления доступа только через четко определенный интерфейс. Вложенные классы (см. разделы 3.12 и 13.5) и пространства имен (см. главу 17) расширяют понятие локальной области действия и явного предоставления доступа. В любом случае объем глобальной информации в системе резко уменьшается.

Контроль доступа позволяет локализовать доступ без каких бы то ни было издержек во время исполнения, неизбежных при полном разъединении частей программы (см. раздел 2.10). С помощью абстрактных классов можно еще больше отделить друг от друга разные части при минимальных затратах (см. раздел 13.2).

Важно, что, оставаясь внутри класса или пространства имен, допустимо отделить объявление от реализации. Тогда проще разобраться в том, что класс делает, не углубляясь в изучение функций-членов, чтобы понять, как это делается. В объявлении класса допустимы встраиваемые функции, позволяющие добиться локальности в случаях, когда такое разделение неудобно.

И наконец, код проще понять и изменить, когда большая его часть помещается на экране. Здесь очень к месту традиционная краткость C, а возможность объявлять в C++ новые переменные непосредственно перед их использованием (см. раздел 3.11.5) – еще один шаг в том же направлении.

Избегать зависимости от порядка. Зависимость от порядка – причина путаницы и ошибок при реорганизации кода. Все знают, что предложения выполняются в определенном порядке, но на зависимости между глобальными объявлениями и объявлениями членов класса часто не обращают внимания. Правила перегрузки (см. раздел 11.2) и правила использования базовых классов (см. раздел 12.2) специально составлены так, чтобы избежать данной зависимости. В идеале ошибка должна выдаваться, когда к изменению семантики приводит перестановка двух объявлений. Для членов классов это верно, но для глобальных объявлений поддержать данное правило невозможно. Препроцессор C может ввести неожиданные и некорректные зависимости в результате макроподстановок, что приведет к полной неразберихе (см. раздел 18.1).

Ошибка на этапе компиляции все же лучше, чем непонятно как разрешенный конфликт. Характерный пример – правила разрешения неоднозначности для множественного наследования (см. раздел 12.2). Правила разрешения неоднозначности

для перегруженных функций – пример того, как трудно добиться желаемого в условиях ограничений на совместимость и гибкость (см. раздел 11.2.2).

Если есть сомнения, нужно выбирать такой вариант средства, которому легче обучить. Применять данное правило трудно, поскольку могут быть аргументы в пользу логической красоты и в пользу того, чтобы не отступать от хорошо знакомого. Практическому использованию этого принципа можно научиться, если смотреть, насколько написанные тобой учебные и справочные руководства понятны читателям. Одна из целей – облегчить труд преподавателей и персонала службы технической поддержки. Необходимо помнить, что, как уже говорилось, программисты – неглупый народ; нельзя жертвовать важной функциональностью в угоду простоте.

Синтаксис важен, хотя бывает и нелогичным. Важно, чтобы система типов была логически последовательной, а семантика языка – ясной и четко определенной. Роль синтаксиса вторична, и, наверное, программист может работать практически с любым синтаксисом.

Однако с помощью синтаксиса язык взаимодействует с пользователем. Есть пользователи, которые бесконечно привержены определенным синтаксическим конструкциям и отстаивают свое мнение прямо-таки с фанатичным пылом. Я не надеюсь изменить эту ситуацию или ввести новые семантические понятия. Поэтому синтаксис C++ подобран так, чтобы не задевать предрассудки программистов. Однако со временем он будет приведен к более рациональному и регулярному виду. Моей целью было устранить такие пережитки, как неявный `int` (см. раздел 2.8.1), старая форма приведения типов (см. раздел 14.3.1), и в то же время свести к минимуму использование усложненных форм синтаксиса объявлений (см. раздел 2.8.1).

Опыт показывает, что люди привыкли к тому, что каждому новому понятию должно соответствовать новое ключевое слово, поэтому обучить новшеству, для которого ключевого слова нет, оказывается на удивление трудно. Это явление имеет более глубокие корни, чем громко выражаемая нелюбовь к новым ключевым словам. Если предоставить человеку выбор и время для раздумий, то он обязательно примет решение в пользу нового ключевого слова, а не изобретательного обходного пути.

Я стараюсь, чтобы важные операции были очень заметны. Например, одна из проблем, связанных со старым стилем приведения типов, состоит в том, что конструкции приведения почти не выделяются в тексте программы. Кроме того, я предпочитаю, чтобы семантически неудачные операции, например плохие приведения типов, выглядели некрасиво и с синтаксической точки зрения (см. раздел 14.3.3). Обычно я выбираю лаконичный синтаксис.

Использование препроцессора должно быть исключено. Без препроцессора сам C, а за ним и C++ были бы «мертворожденными» языками. Без Cpp они не стали бы достаточно выразительными и гибкими для использования в больших проектах. С другой стороны, именно из-за низкоуровневой семантики Cpp так трудно и дорого создать более развитые и изящно оформленные среды для программирования на C.

Поэтому было необходимо найти замены для всех существенных возможностей Cpp, которые бы гармонично сочетались с синтаксисом и семантикой C++.

Сделав это, мы получили бы более дешевое и намного более удобное окружение для работы с C++. А заодно устранили бы источники многих трудных для обнаружения ошибок. Шаблоны (см. главу 15), встраиваемые функции (см. раздел 2.4.1), константы (см. раздел 3.8) и пространства имен (см. главу 17) – вот шаги в этом направлении.

4.5. Правила поддержки низкоуровневого программирования

Разумеется, вышеупомянутые правила относятся практически ко всем особенностям языка. А те, что приведены в таблице 4.5, применимы к C++ не только как к языку для системного программирования, но и как к средству для выражения высокоуровневого проектирования.

Таблица 4.5

Правила поддержки низкоуровневого программирования
Использовать традиционные компоновщики
Никаких неоправданных несовместимостей с C
Не оставлять места для языка более низкого уровня, чем C++, исключая ассемблер
Правило нулевых издержек: чем не пользуетесь, за то не платите
Если есть сомнения, предоставлять средства для ручного контроля

Использовать традиционные компоновщики. С самого начала были поставлены цели обеспечить простоту переносимости и взаимодействия с модулями, написанными на других языках. Ради этого пришлось настаивать на том, чтобы C++ можно было реализовать при использовании традиционных компоновщиков. Но работать с технологией, которая возникла еще во времена ранних версий Fortran, нелегко. Некоторые особенности C++, прежде всего, безопасную компоновку (см. раздел 11.3) и шаблоны (см. главу 15) можно, конечно, реализовать и с помощью традиционных компоновщиков, но при наличии более развитой поддержки реализация была бы лучше. Подспудно мы хотели, чтобы C++ дал стимул для разработки более совершенных компоновщиков.

Традиционный компоновщик сравнительно легко позволяет обеспечить совместимость с C на уровне редактирования связей. Это важно для безболезненного использования средств операционной системы, библиотек, написанных на C, Fortran и т.д., и для написания кода, который можно поместить в библиотеки, вызываемые из других языков. Традиционный компоновщик важен и при написании программ, используемых на низких уровнях системы, например драйверов устройств.

Никаких неоправданных несовместимостей с C. C – это язык системного программирования, получивший наибольшее признание. Программисты хорошо знают C, на нем написаны миллиарды строк кода, существует целая индустрия, специализирующаяся на предоставлении инструментов и услуг для этого языка. C++ основан на C. Возникает вопрос: «Насколько близки должны быть определения

C++ и C?» Как уже говорилось, C++ не претендует на стопроцентную совместимость с C, так как это противоречило бы таким целям языка, как безопасное использование типов и поддержка проектирования. Но любых несовместимостей, не вступающих в конфликт с этими целями, мы избегали. В большинстве случаев мы шли на несовместимость с C, только если действующее в C правило оставляло «зияющую дыру» в системе контроля типов.

С годами выяснилось, что совместимость с C – одновременно самый большой плюс и самый существенный минус C++. И это неудивительно. Степень совместимости с C останется важнейшим вопросом и в будущем. Со временем совместимость будет становиться все меньшим преимуществом и все большей обузой. В связи с этим следует наметить путь решения вопроса (см. главу 9).

Не оставлять места языкам более низкого уровня, чем C++, исключая ассемблер. Раз высокоуровневый язык должен полностью изолировать программиста от особенностей определенного компьютера, то задача системного программирования должна решаться с помощью какого-нибудь другого языка. Обычно таким языком был C. Но потом C заменил языки высокого уровня в тех областях, где гибкость управления или скорость имели решающее значение. При этом система могла полностью переписываться на C, или для ее поддержки требовался человек, одинаково хорошо владеющий двумя языками. В последнем случае программист оказывался перед трудным выбором: какой язык предпочесть для решения конкретной задачи. Приходилось держать в уме примитивы и принципы обоих языков. В C++ предоставляются как низкоуровневые средства, так и механизмы абстракции, и поддерживается создание гибридных систем, где используется то и другое.

Чтобы остаться жизнеспособным языком для системного программирования, в C++ должны быть заложены возможности напрямую работать с аппаратурой, контролировать размещение структур данных в памяти и иметь примитивные операции и типы данных, отображаемые непосредственно на аппаратные средства. Если этого не будет, то придется использовать C или ассемблер. Задача проектирования языка состоит в изолировании низкоуровневых средств, чтобы прибегать к ним имело смысл лишь тогда, когда есть необходимость в прямом доступе к аппаратуре и операционной системе. Цель – защитить программиста от случайного неправильного использования, не возводя ненужных препятствий.

Правило нулевых издержек: чем не пользуетесь, за то не платите. Обычно сложные языки заслуженно порицают за генерирование большого и медленного кода. Зачастую затраты, необходимые для поддержки развитых возможностей, распространяются и на все остальные. Например, в каждом объекте хранится служебная информация; косвенный доступ к данным выполняется всегда и везде, хотя реально необходим только в некоторых случаях; управляющие конструкции чрезмерно усложнены, чтобы поддержать «развитые абстракции управления». Такая модель была признана неприемлемой для C++.

Неоднократно при принятии тех или иных проектных решений обнаруживалось, что правилу нулевых издержек очень трудно следовать. Виртуальные функции (см. раздел 3.5), множественное наследование (см. раздел 12.4.2), идентификация типов во время выполнения (см. раздел 14.2.2.2), обработка исключений

и шаблоны – в каждом случае средство принималось лишь после того, как я убеждался, что его можно реализовать с нулевыми издержками. Естественно, разработчик может согласиться на компромисс между данным правилом и другим полезным свойством системы, но подходить к этому надо с большой осторожностью, тем более что многие программисты резко и эмоционально реагируют на излишние затраты.

Пожалуй, именно данное правило заставило отказаться от включения в язык многих возможностей.

Если есть сомнения, предоставлять средства для ручного контроля. Я не очень доверяю «передовой технологии» и не склонен думать, что нечто изощренное может быть доступным повсеместно. Хорошее подтверждение этому – встраиваемые функции (см. раздел 2.4.1). Другой пример – инстанцирование шаблонов, где мне следовало быть осторожнее, тогда не пришлось бы потом добавлять механизм для явного контроля (см. раздел 15.10). Управляемое распределение памяти – это пример ситуации, где за счет ручного управления удалось добиться важных преимуществ, но только время покажет, не пострадали ли при этом автоматизированные механизмы (см. раздел 10.7).

4.6. Заключительное слово

Совокупность всех вышеприведенных правил можно считать важнейшей особенностью языка. Убери хоть одно – и, скорее всего, нарушится равновесие, что ударит по интересам какой-то группы пользователей. И то же самое произойдет, если любая из установок будет доминировать над остальными.

Я старался, чтобы мои правила выглядели как утверждения и рекомендации, а не как набор запретов. При таком подходе становится гораздо труднее игнорировать новые идеи. В то же время мой взгляд на C++ как на язык для создания программного обеспечения с акцентом на средства, облегчающие структурирование программ, препятствует тенденции к внесению мелких изменений.

Более подробный перечень вопросов, на которые надо ответить, рассматривая новое языковое средство, приведен в контрольном списке, предложенном рабочей группой по расширениям языка при комитете ANSI/ISO (см. раздел 6.4.1).

Глава 5. Хронология 1985–1993 гг.

Не забывай: на все нужно время.

Пиет Хайн

5.1. Введение

В части II описываются все языковые средства, добавленные в C++. В данной же главе приводится последовательность событий.

Причина, по которой хронология выделена в отдельный раздел, состоит в том, что фактическое время события не очень важно для окончательного определения C++. В общих чертах я знал направление, в котором идет развитие языка, проблемы, требующие решения, и необходимые для этого средства. Но не мог же я осуществить все задуманное в рамках одной крупной ревизии языка. Это заняло бы слишком много времени и оставило бы меня без необходимой обратной связи с пользователями. Поэтому расширения добавлялись к языку постепенно. Фактический порядок был чрезвычайно важен для тогдашних пользователей и существенен для поддержания логической стройности языка. Однако для окончательной формы C++ он не играл такой уж значительной роли. Поэтому хронологически последовательное описание всех расширений лишь затемнило бы логическую структуру языка.

Данная глава посвящена работе, которая привела к созданию версии Cfront 2.0, написанию книги «The Annotated C++ Reference Manual» («Аннотированное справочное руководство по C++») и усилиям по стандартизации.

1986–1989 гг. Версия 2.0 добавила к C++ такие возможности, как абстрактные классы, типобезопасную компоновку и множественное наследование.

1988–1990 гг. В книге «The Annotated C++ Reference Manual» описаны шаблоны и обработка исключений. Тем самым брошен серьезный вызов разработчикам компиляторов и открыт путь к революционным изменениям в способах программирования на C++.

1989–1993 гг. В результате усилий по стандартизации добавлены пространства имен, идентификация типов во время исполнения и многие менее важные средства для программиста на C++.

Во всех трех случаях была проделана большая работа по уточнению определения языка и его упорядочиванию. С моей точки зрения, все представляло собой одно действие, растянувшееся на длительный срок.

5.2. Версия 2.0

К середине 1986 г. абсолютно всем стало понятно направление развития C++. Основные проектные решения были приняты. Следующим этапом развития было внедрение параметризованных типов, множественного наследования и обработки исключений. Требовалось провести много экспериментов и на основе их результатов откорректировать курс, но даже это уже не могло кардинально изменить ситуацию. C++ и раньше-то не отличался эластичностью, а теперь все возможности для радикальных изменений были исчерпаны. Оставалось, однако, много непростой работы. К этому времени во всем мире языком C++ пользовались примерно 2 тыс. человек.

Согласно плану, выработанному Стивом Джонсоном и мной, повседневная работа над инструментарием (главным образом, Cfront) должна была перейти в ведение организации по разработке и технической поддержке. Это позволило бы мне сосредоточиться на новых возможностях и библиотеках. Предполагалось, что сначала AT&T, а затем и другие компании начнут создавать компиляторы и инструментальные средства, которые, в конце концов, сделают Cfront ненужным.

Фактически этот процесс уже начался, но хороший план пошел кувырком из-за нерешительного и неумелого руководства разработкой. Проект разработки нового компилятора C++ отвлек внимание и ресурсы от задач сопровождения и развития Cfront. Планы начать поставку версии 1.3 в начале 1988 г. провалились. В результате пришлось ждать выхода версии 2.0 в июне 1989 г., и, хотя 2.0 была лучше версии 1.2 почти во всех отношениях, в ней все же не были полностью реализованы возможности, упомянутые в статье «Whatis?» (см. раздел 3.15). Частично вследствие этого в версию 2.0 не вошли значительно улучшенные и расширенные библиотеки. Поставка такой библиотеки была вполне возможна, поскольку многое из того, что стало библиотекой Стандартных Компонентов USL[1], к тому времени уже использовалось внутри AT&T. Однако мое желание иметь прямую поддержку шаблонов заслонило все иные варианты. Кроме того, некоторая часть руководителей отдела разработки почему-то верила, что библиотека может быть стандартом и одновременно значительным источником дохода.

Версия 2.0 стала плодом работы группы, в которую входили Эндрю Кениг, Барбара Му, Стэн Липпман, Пэт Филип и я. Барбара занималась координацией, Пат – интеграцией, я и Стэн – кодированием. Мы с Кенигом рассматривали извещения об ошибках и обсуждали детали языка. Эндрю и Барбара занимались тестированием. В результате удалось реализовать все новые возможности и исправить 80% ошибок. Кроме того, я написал большую часть документации. Как всегда, вопросы проектирования языка и сопровождения справочного руководства лежали на моих плечах. Барбара Му и Стэн Липпман возглавили команду, которая позже выпустила версии 2.1 и 3.0.

---

[1] USL образовалась как отделение AT&T, занимающееся технической поддержкой и дистрибуцией UNIX и соответствующих инструментальных средств. Позже она выделилась в отдельную компанию под названием UNIX System Laboratories, а затем была куплена Novell.

Многие из тех, кто оказал влияние на C with Classes и на начальный вариант C++, продолжали разными способами содействовать его развитию. Фил Браун (Phil Brown), Том Каргилл, Джим Коплиен, Стив Дьюхерст, Кит Горлен (Keith Gorlen), Лаура Ивс, Боб Келли (Bob Kelley), Брайан Керниган, Энди Кениг, Арчи Лахнер, Стэн Липпман, Ларри Майка (Larry Mayka), Дуг Макилрой, Пэт Филип (Pat Philip), Дейв Проссер (Dave Prosser), Пегги Куин (Peggy Quinn), Роджер Скотт (Roger Scott), Джерри Шварц, Джонатан Шопиро и Кэти Старк (Kathy Stark) – всех этих людей я с благодарностью упомянул в работе [Stroustrup,1989b]. С наибольшим энтузиазмом язык обсуждали Дуг Макилрой, Энди Кениг, Джонатан Шопиро и я.

Стабильность определения языка и его реализации были признаны самыми важными аспектами [Stroustrup, 1987c]:

«Подчеркнем, что эти модификации языка – лишь расширения. C++ был и остается стабильным языком, на который можно положиться при разработке долгосрочных проектов».

Не менее важной считалась роль C++ как языка общего назначения для промышленного применения [Stroustrup, 1987c]:

«Переносимость... некоторых реализаций C++ – важнейшая цель проектирования. Поэтому мы избегали расширений, которые могут существенно увеличить время, необходимое для переноса, или предъявить повышенные требования к ресурсам, нужным компилятору C++. Этот идеал эволюции языка противостоит внешне благовидным предложениям сделать программирование более удобным:

- **за счет потери эффективности или структурированности;**
- **для новичков – за счет потери единого стиля;**
- **в конкретных предметных областях путем добавления в язык специализированных возможностей;**
- **путем добавления возможностей, облегчающих интеграцию с конкретной средой программирования».**

В версии 2.0 было много улучшений, но ничего радикально нового. В то время я, случалось, объяснял, что «все возможности 2.0, включая множественное наследование, – просто снятие ограничений, которые стали нас слишком сильно стеснять». Это одновременно и преувеличение, и сдержанный отпор общей тенденции придавать излишне большое значение каждой новой возможности. С точки зрения проектирования языка, самым важным аспектом версии 2.0 было то, что улучшилась интеграция отдельных особенностей языка C в структуру языка. Думаю, что для пользователя самыми важными аспектами новой версии были более стабильный компилятор и усовершенствованная техническая поддержка.

5.2.1. Обзор возможностей

Основные возможности версии 2.0 были впервые представлены в статье [Stroustrup, 1987c] и подытожены в пересмотренном варианте этой статьи [Stroustrup, 1989b], которая вошла в состав документации по 2.0:

1. Множественное наследование (см. раздел 12.1).
2. Безопасное связывание (см. раздел 11.3).
3. Улучшенное разрешение имен перегруженных функций (см. раздел 11.2).

4. Рекурсивное определение присваивания и инициализации (см. раздел 11.4.4).
5. Улучшенные средства пользовательского управления памятью (см. разделы 10.2 и 10.4).
6. Абстрактные классы (см. раздел 13.2).
7. Статические функции-члены (см. раздел 13.4).
8. Константные функции-члены (см. раздел 13.3).
9. Защищенные члены с квалификатором `protected` (впервые появились в версии 1.2, см. раздел 13.9).
10. Обобщенные инициализаторы (см. раздел 3.11.4).
11. Инициализаторы членов базовых классов (см. раздел 12.9).
12. Перегрузка оператора `->` (см. раздел 11.5.4).
13. Указатели на члены (впервые появились в версии 1.2, см. раздел 13.11).

Большая часть этих расширений и улучшений отражала опыт эксплуатации C++ и не могла быть добавлена раньше. Естественно, для интеграции средств пришлось много поработать, и очень печально, что этой работе был отдан больший приоритет, чем завершению построения языка в соответствии с описанием, данным в статье «Whatis?» (см. раздел 3.15).

В основном все указанные возможности так или иначе способствовали большей безопасности языка. В Cfront 2.0 проверялась согласованность типов функций через границы единиц трансляции (типобезопасная компоновка), правила разрешения перегрузки не зависели от порядка и больше вызовов, чем раньше, признавались неоднозначными. Определение понятия константности (`const`) стало более полным, указатели на члены «закрыли дыру» в системе контроля типов. Были введены операции явного распределения и освобождения памяти на уровне класса, что сделало ненужной технику «присваивания указателю `this`» (см. раздел 3.9).

Из вышеупомянутых возможностей 1, 3, 4, 5, 9, 10, 11, 12 и 13 уже использовались в Bell Labs во время моей презентации на конференции USENIX в 1987 г. (см. раздел 7.1.2).

5.3. Аннотированное справочное руководство

Приблизительно к 1988 г. стало очевидным, что C++ будет стандартизован [Stroustrup, 1989]. К этому моменту уже появилось несколько независимых реализаций. Настало время составить более точное и полное описание языка. Кроме того, следовало сделать C++ широко доступным. Поначалу о формальной стандартизации никто не думал. Многие из тех, кто непосредственно участвовал в разработке C++, считали, что заниматься стандартизацией до накопления достаточного опыта работы с языком неразумно. Однако в одиночку написать улучшенное справочное руководство я бы не смог. Необходимы были мнения и отклики пользователей. Поэтому я решил переписать справочное руководство по C++ и распространить его черновой вариант среди наиболее авторитетных и опытных пользователей во всем мире.

Примерно в то же время USL – подразделение AT&T, которое занималось коммерческими продажами C++, – пожелало иметь новое, улучшенное справочное

руководство и поручило написать его одной из своих сотрудниц – Маргарет Эллис (Margaret Ellis). В этой связи вполне естественным казалось объединить усилия, совместно выпустить руководство и передать его внешним рецензентам. Мне также представлялось очевидным, что публикация такого труда с некоторой дополнительной информацией будет способствовать принятию нового определения и широкому распространению C++ в целом. Вот так появилась книга «The Annotated C++ Reference Manual» [ARM] – «Аннотированное справочное руководство по C++». Вот несколько строк из нее:

«...чтобы заложить твердые основы для дальнейшей эволюции C++... [и] стать отправной точкой для его формальной стандартизации... собственно справочное руководство по C++ и является полным определением языка, но сжатый стиль такого рода документов оставляет многие вопросы без ответа. Обсуждение того, что не входит в язык, почему те или иные средства определены именно так, а не иначе, и как можно было бы реализовать различные возможности, остается за рамками справочного руководства, но тем не менее представляет интерес для большинства пользователей. Такие замечания приводятся в аннотациях и комментариях.

Комментарии также помогут читателю разобраться во взаимосвязях между разными частями языка. В них подчеркнуты те моменты и последствия, на которые можно было бы не обратить внимания при чтении самого руководства. Примеры и сравнения с языком C делают книгу более легкой для чтения, чем сухое справочное описание».

После ряда мелких стычек с людьми, отвечавшими за выпуск книги, было решено, что мы включим в ARM (так в обиходе называлась «The Annotated C++ Reference Manual») полное определение C++, с шаблонами и обработкой исключений, а не только определение подмножества, реализованного в последней версии AT&T. Тем самым постулировалось, что язык как таковой отличается от любой его конкретной реализации. Об этом заложенном с самого начала принципе приходилось часто напоминать, поскольку пользователи, разработчики компиляторов и продавцы никак не могли его запомнить.

На само справочное руководство были получены рецензии примерно от ста человек из двух десятков организаций. Имена большинства из них указаны в разделе благодарностей ARM. Кроме того, многие внесли свой вклад в содержание ARM. Особо хочу отметить помощь Брайана Кернигана, Эндрю Кенига и Дуга Макилроя. Часть ARM, составляющая собственно справочное руководство, была принята за основу стандарта ANSI C++ в марте 1990 г.

В ARM не рассказывается о приемах, поддерживаемых языковыми средствами: «эта книга не ставит себе целью научить программированию на C++; в ней объясняется, что такое язык, а не как им пользоваться [ARM]». Рассказ об использовании C++ был отложен до выхода второго издания книги «Язык программирования C++» [2nd]. К сожалению, некоторые проигнорировали это предупреждение. В результате C++ часто воспринимают как набор непонятных и не связанных между собой деталей, а потому и не могут написать на нем красивой и легкой для сопровождения программы (см. раздел 7.2).

5.3.1. Обзор ARM

В ARM представлены некоторые не очень существенные возможности, которые были реализованы только в версии 2.1 от AT&T и в более поздних компиляторах других производителей. Самая очевидная возможность – вложенные классы.

К первоначальному определению их областей действия мне настоятельно рекомендовали вернуться внешние рецензенты. К тому же я и сам отчаялся найти для областей действия в C++ непротиворечивые правила, так что пока действуют принятые в C (см. раздел 2.8.1).

Более важными из новых возможностей были шаблоны (см. главу 15) и обработка исключений (см. главу 16). Кроме того, в ARM описывалась независимая перегрузка операций префиксного и постфиксного инкремента (++) – раздел 11.5.3.

В целях совместимости с ANSI C была разрешена инициализация локальных статических массивов.

С этой же целью введен модификатор `volatile`, облегчающий труд разработчиков компиляторов. Я вовсе не уверен, что семантическое сходство служит оправданием для синтаксических параллелей с `const`, однако не вижу причин для попыток изменить решения комитета ANSI C в этом вопросе.

Итак, в ARM были представлены следующие возможности:

- все, что вошло в версию 2.0 (см. раздел 5.2.1);
- шаблоны (см. главу 15);
- исключения (см. главу 16);
- вложенные классы (см. раздел 13.5);
- раздельная перегрузка префиксных и постфиксных операций ++ и – (см. раздел 11.5.3);
- `volatile`;
- локальные статические массивы.

Все описанные в ARM возможности, кроме исключений, получили широкое распространение вместе с версией Cfront 3.0 в октябре 1991 г. Полная реализация возможностей, упомянутых в ARM, впервые появилась в компиляторах фирм DEC и IBM в начале 1992 г.

5.4. Стандартизация ANSI и ISO

С 1990 г. основные усилия для завершения C++ прилагались в комитете по стандартизации ANSI/ISO.

Инициатива начать формальную стандартизацию C++ исходила от компании Hewlett-Packard совместно с AT&T, DEC и IBM. Важную роль в этом сыграл Ларри Рослер из Hewlett-Packard. Именно он связался со мной ближе к концу 1988 г., и между нами состоялся разговор о необходимости формальной стандартизации. Основной проблемой было время. Выражая интересы крупных пользователей, Ларри выступал за срочность, я же хотел оттянуть начало стандартизации, пока не будет накоплен достаточный опыт. Взвесив все не очень ясные технические и коммерческие факторы, мы пришли к выводу, что к стандартизации нужно приступить в течение года, если мы вообще рассчитываем на успех. Как мне помнится, первая техническая встреча комитета ANSI состоялась за три дня до истечения этого срока (в марте 1990 г.).

Предложение по стандартизации для комитета ANSI написал Дмитрий Ленков из компании Hewlett-Packard [Lenkov, 1989]. В нем приведено несколько причин, почему стандартизацию следует начинать немедленно:

- C++ завоевывает массовое признание значительно быстрее, чем большинство других языков;
- задержка процесса приведет к появлению диалектов;
- C++ нуждается в детальном проработанном определении, которое бы полностью описывало семантику каждого языкового средства;
- языку недостает нескольких важных возможностей, в том числе обработки исключений, некоторых особенностей множественного наследования, средств для поддержки параметрического полиморфизма и стандартных библиотек.

В этом предложении была также подчеркнута необходимость сохранения совместимости с ANSI C. Организационное собрание комитета ANSI C++ X3J16 состоялось в декабре 1989 г. в Вашингтоне. На нем присутствовали 40 человек, в том числе все принимавшие участие в стандартизации C, те, кого называли «старой гвардией C», и другие. Дмитрий Ленков был избран председателем, а Джонатан Шопиро – редактором комитета.

Первое техническое заседание состоялось на территории компании AT&T в Сомерсете в марте 1990 г. AT&T удостоилась этой чести не из-за вклада в создание C++, а потому что мы (члены комитета X3J16, присутствовавшие на встрече в Вашингтоне) решили, что планирование места проведения заседаний на первые несколько лет будет зависеть от климатических условий местности в то или иное время года. Так, второе заседание состоялось в помещении Microsoft в Сиэтле в июле, а третье – в Hewlett-Packard в Пало-Альто в ноябре. Таким образом, нам всегда сопутствовала прекрасная погода, а заодно удалось избежать соперничества компаний.

Сейчас в комитет входит более 250 человек, из которых около 70 посещают заседания регулярно. Первоначальной целью комитета была выработка предварительного стандарта, который предполагалось передать на суд общественности в конце 1993 г. или начале 1994 г. Была надежда, что еще через пару лет после этого можно будет принять официальный стандарт (довольно оптимистичный прогноз для стандартизации языка общего назначения). Для сравнения напомню, что стандартизация C заняла семь лет. В тот момент мы планировали опубликовать предварительный стандарт в апреле 1995 г. [1]

На заседаниях комитета ANSI C++ присутствовали и представители других стран (Канада, Дания, Франция, Япония, Швеция, Великобритания). В Лунде (Швеция) в июне 1991 г. собрались члены комитета WG21 по стандартизации C++ при ISO, и обе структуры решили проводить совместные заседания, начав их незамедлительно, прямо в Лунде. Стоит отметить, что большинство участников из других стран уже давно использовали C++.

Перед комитетом по C++ стояли довольно трудные задачи:

- определение языка должно быть точным и полным;
- необходимо принять во внимание совместимость с C;

[1] В июле 1994 г. комитет проголосовал за «CD registration» – так называется первый этап завершающей процедуры ISO. Планирование работы над стандартом – дело нелегкое. В частности, такие «детали», как решение вопроса о том, что представляет собой стандарт и как его принимать, не стандартизованы и меняются, похоже, каждый год.

- нужно рассмотреть расширения, выходящие за пределы текущей практики применения C++;
- должны быть рассмотрены библиотеки.

Ко всему прочему сообщество пользователей C++ было очень неоднородным и совершенно неорганизованным, поэтому комитету по стандартизации пришлось стать его центром. В краткосрочной перспективе именно это и было самой важной его целью:

«Комитет по C++ – это место, где разработчики компиляторов и инструментальных средств, их коллеги и представители могут обсудить проблемы определения языка и – насколько позволяет коммерческая конкуренция – его реализации. Таким образом, комитет уже сослужил добрую службу сообществу: помог сблизить различные реализации, предоставив площадку для дискуссий. В противном случае разработчики компиляторов вместе с немногими коллегами или в одиночестве строили бы догадки относительно тех вопросов, на которые нет ответа в ARM. Возможно, они связались бы со мной, но я не в состоянии один справиться со всеми возникающими проблемами. Отсутствие общения неизбежно ведет к появлению диалектов. Комитет противостоит таким тенденциям».

Стандартизация – непростой процесс. В комитет входили люди, стремившиеся сохранить status quo; мечтавшие вернуться на несколько лет назад; были такие, кто хотел «порвать с проклятым прошлым» и спроектировать совершенно новый язык; те, кого волновал лишь один конкретный класс систем; люди, чьи голоса были куплены их работодателями, и представлявшие лишь самих себя; специалисты, озабоченные в основном теоретическими взглядами на программирование вообще и языки программирования в частности; нетерпеливые и требовавшие принять стандарт немедленно, даже если некоторые детали останутся неразрешенными, и наоборот, те, которых не устраивало ничего, кроме идеального определения языка. Были такие, кто полагал, что C++ – совсем новый язык, у которого и пользователей-то почти нет, и представители организаций, написавших за минувшее десятилетие миллионы строк кода, и т.д. Но в соответствии с правилами стандартизации нам всем предстояло прийти к более или менее одинаковому мнению. Мы должны были достичь консенсуса (обычно под этим словом понимают подавляющее большинство). Это разумные правила, они имеют национальный и интернациональный характер. Все интересы законны, и если позволить большинству ущемлять интересы меньшинства, то получится стандарт, полезный лишь ограниченному кругу пользователей. Поэтому каждому члену комитета предстоит научиться уважать другие, чуждые ему точки зрения и идти на компромиссы. И это вполне соответствует стилю C++.

Совместимость с C – это первый вопрос, на котором мы споткнулись. После длительных, иногда весьма ожесточенных споров было решено, что за стопроцентную совместимость с C бороться не стоит. Но и резкий уход от совместимости с C не годится. C++ – это самостоятельный язык, он не является строгим надмножеством ANSI C, и сделать его таковым нельзя без существенного ослабления гарантий, предоставляемых системой контроля типов и не «поломав» миллионы строк написанного на C++ кода. С другой стороны, значительное уменьшение совместимости с C тоже разрушит существующий код, усложнит создание и сопровождение систем, написанных на смеси этих языков, и сделает

более проблематичным переход от C к C++. Решение, которое часто формулируют как уже упоминавшееся «настолько близко к C, насколько возможно, но не ближе», совпадает с тем, к которому снова и снова приходили все размышлявшие над языком C++ и направлениями его развития (см. раздел 3.12). Проработка деталей данного решения после независимых изменений, которые C++ и ANSI C внесли в исходное определение C, составила заметную часть работы комитета по стандартизации. Основной вклад в это дело внес Томас Плам.

5.4.1. Обзор возможностей

Вот сводка возможностей C++, зафиксированных в ноябре 1994 г. в рабочих документах комитета на заседании, состоявшемся в Вэлли Фордж:

- возможности, описанные в ARM (см. раздел 5.3);
- представление европейского набора символов (см. раздел 6.5.3.1);
- ослабление правила для типа возвращаемого значения в замещающих функциях (см. раздел 13.7);
- идентификация типов во время исполнения (см. раздел 14.2);
- объявления в условиях (см. раздел 3.11.5.2);
- перегрузка, основанная на перечислениях (см. раздел 11.7.1);
- определенные пользователем операторы выделения и освобождения памяти для массивов (см. раздел 10.3);
- опережающие объявления вложенных классов (см. раздел 13.5);
- пространства имен (см. главу 17);
- ключевое слово `mutable` (см. раздел 13.3.3);
- новый синтаксис приведения типов (см. раздел 14.3);
- тип `bool` (см. раздел 11.7.2);
- явное инстанцирование шаблонов (см. раздел 15.10.4);
- явная спецификация аргументов в вызовах шаблонов функций (см. раздел 15.6.2);
- шаблоны-члены (см. раздел 15.9.3);
- шаблоны классов как аргументы шаблонов (см. раздел 15.3.1);
- возможность инициализировать константные статические члены интегральных типов константным выражением внутри объявления класса;
- явные конструкторы (см. раздел 3.6.1);
- статическая проверка спецификаций исключений (см. раздел 16.9).

Глава 6. Стандартизация

Не пытайся перещеголять меня в эксцентричности.

Зафод Библброкс

6.1. Что такое стандарт?

В умах программистов царит путаница относительно того, что представляет собой стандарт и каким он должен быть. Вот один из идеалов: в стандарте полно и точно определяется, какие программы корректны и какова семантика каждой корректной программы. В действительности такое определение не может и не должно быть идеалом для языков, которые спроектированы для работы с самыми разнообразными аппаратными архитектурами. Для подобных языков важно, чтобы определенные аспекты зависели от реализации. Таким образом, стандарт часто описывают как «контракт между программистом и разработчиком компилятора». Он определяет не только «корректный» исходный код, но и свойства, на которые программист может полагаться всегда, и зависящие от реализации. Например, в языках C и C++ разрешается объявлять переменные типа `int`, но стандарт ничего не говорит о размере `int`, кроме того что он не меньше 16 бит.

Можно вести длинные ученые споры о том, что такое стандарт и какая терминология лучше всего подходит для его формулирования. Но важнее уметь отличить корректную программу от некорректной и точно специфицировать аспекты, одинаковые во всех реализациях и допускающие различия. Многие члены комитета акцентируют свое внимание на технических вопросах языка, поэтому основные проблемы, связанные с тем, чту же именно стандартизируется, вынужден решать редактора комитета. К счастью, нашего первого редактора, Джонатана Шопиро, такие материи интересовали. Джонатан ушел с поста редактора, передав его Эндрю Кенигу. Но он и поныне остается членом комитета.

Еще один интересный (то есть трудный) вопрос – до какой степени допустимы реализации, включающие возможности, не специфицированные в стандарте. В конце концов, некоторые расширения действительно необходимы тем или иным группам внутри сообщества пользователей C++. Так, есть компьютеры, аппаратно поддерживающие некоторые механизмы параллельности, имеющие ограничения на адресацию или специализированную векторную аппаратуру. Мы не можем перегружать «C++ для всех» средствами, необходимыми для поддержки этих несовместимых между собой специальных расширений, поскольку зачастую они влекут за собой издержки для всех пользователей. Но было бы неправильно запрещать разработчикам компиляторов для таких узких групп включить необходимые им расширения, строго придерживаясь стандарта во всем остальном. С другой

стороны, я как-то раз видел «расширение», разрешающее доступ к закрытым членам класса из любой функции в программе, то есть разработчик не счел важной реализацию контроля доступа. Такую вольность я не считаю разумной. Сформулировать стандарт так, чтобы расширения первого вида были возможны, а второго – нет, – нетривиальная задача.

Также программа должна иметь возможность распознать нестандартные расширения. Иначе, проснувшись однажды утром, программист может обнаружить, что важнейший код зависит от расширения, предоставленного конкретным компилятором, а стало быть, сменить поставщика так просто уже не удастся. Я вспоминаю, как, будучи еще студентом, был приятно удивлен тем, что на нашем университетском компьютере установили «расширенный Fortran» с некоторыми очень симпатичными возможностями. Но каково же было удивление, когда я понял, что все мои программы могут работать только на машинах серии CDC6000.

Итак, стопроцентное соответствие стандарту программ, написанных на C++, не является ни достижимым, ни желательным идеалом. Программы, соответствующие стандарту, необязательно полностью переносимы, поскольку некоторые аспекты могут зависеть от компилятора. Но большинство из них все же переносимо. Например, абсолютно законная программа на C или C++ может изменить семантику, если полагается на результаты встроенной операции взятия остатка от деления в применении к отрицательным числам.

Кроме того, реальные программы обычно зависят от библиотек, предоставляющих сервисы, которые в некоторых системах просто отсутствуют. Например, программа, написанная под Microsoft Windows, вряд ли будет без изменений работать под X Windows, а программу, в которой используются базовые классы Borland, не удастся без труда перенести на платформу MacApp. Переносимость реальной программы определяется тем, насколько хорошо при ее проектировании были инкапсулированы зависимости от компилятора и окружения, а не только строгим соответствием немногим простым правилам, изложенным в стандарте.

6.1.1. Детали реализации

Чуть ли не каждую неделю поступают просьбы стандартизировать такие аспекты, как расположение в памяти таблицы виртуальных функций, схему кодирования имен при типобезопасной компоновке или отладчик. Однако все это относится к качеству компилятора или к деталям реализации, которые лежат за пределами стандарта. Пользователи желают, чтобы библиотеки, откомпилированные одним компилятором, можно было связать с кодом, откомпилированным другим, чтобы двоичные файлы переносились с одной архитектуры на другую и чтобы отладчик не зависел от того, какой компилятор использовался при создании отлаживаемой программы.

Однако «узаконивание» наборов команд, интерфейсов операционных систем, форматов отладчиков, соглашений о вызове и хранении объектов в памяти находится далеко за рамками возможностей группы по стандартизации языка программирования. Такая универсальная стандартизация, наверное, даже и нежелательна: она затормозит прогресс в области машинных архитектур и операционных систем. Если пользователю нужна полная независимость от аппаратуры, систему следует

строить как интерпретатор, предоставляющий приложениям собственную среду. Но у такого подхода есть свои нюансы: в частности, становится трудно воспользоваться особенностями специализированной аппаратуры и следовать локальным требованиям к стилю. Работая на языке, пригодном для построения серьезных систем, мы должны смириться с тем, что вновь и вновь наивный пользователь посылает в сетевую конференцию сообщение: «Я перенес свой объект с Mac на SPARC, и он перестал работать». Как и переносимость, способность к взаимодействию – это вопрос проектирования и понимания тех ограничений, которые налагает среда. Некоторые программисты не знают, что фрагменты, откомпилированные двумя разными компиляторами C для одной и той же системы, необязательно можно будет связать друг с другом (скорее всего, это не получится). И тем не менее они приходят в ужас, оттого что C++ не гарантирует такое взаимодействие. Это обычное явление, и мы должны просвещать пользователей.

6.1.2. Тест на реалистичность

Помимо многих формальных ограничений, которыми связан комитет по стандартизации, есть одно неформальное, но важное для практики. На ряд стандартов потенциальные пользователи просто не обращают внимания. Например, практически забыты стандарты Pascal и Pascal2. Для большинства программистов на Pascal само название языка означает расширенный диалект, введенный в обиход компанией Borland. Язык, определенный стандартом Pascal, не предоставлял средств, которые пользователи считали необходимыми, а Pascal2 появился тогда, когда утвердился другой, неформальный «промышленный стандарт». Еще одно настораживающее наблюдение: в среде UNIX по-прежнему работают на K&R C; ANSI C только борется за это сообщество. Некоторые пользователи не видят в ANSI/ISO C таких технических преимуществ, которые оправдали бы краткосрочные неудобства при переходе. Даже к не вызывающему возражений стандарту приходится привыкать. Для этого он должен быть принят вовремя и отвечать насущным потребностям. Попытка превратить C++ в «идеальный язык» или выпустить стандарт, не допускающий неверного истолкования никем, даже самым безграмотным человеком, – задача, непосильная ни для комитета (см. раздел 3.13), ни для иной организации, работающей на удовлетворение потребностей большого сообщества пользователей (см. раздел 7.1).

6.2. Работа комитета

Для стандартизации C++ образовано несколько комитетов. Первый и самый крупный – комитет ANSI-X3J16, сформированный Американским национальным институтом стандартизации (ANSI). За этот комитет отвечает Ассоциация производителей компьютеров и оборудования для бизнеса (CBEMA), поэтому он работает по установленным ею правилам. В частности, это означает, что одна компания имеет один голос, а физическое лицо, не работающее ни в одной компании, рассматривается как компания. Член комитета может принимать участие в голосовании, начиная со второго заседания, на котором присутствует. Официально самым важным считается комитет WG-21, сформированный Международной

организацией по стандартизации (ISO). Он работает по международным правилам и принимает окончательное решение о придании стандарту статуса международного. В частности, действует правило «одна страна – один голос». Другие страны, в том числе Великобритания, Германия, Дания, Россия, Франция, Швеция и Япония, теперь имеют собственные национальные комитеты по стандартизации C++, которые направляют запросы, рекомендации и своих представителей на совместные заседания комитетов ANSI/ISO.

Мы не стали принимать никаких решений в обход процедуры голосования, удовлетворяющей правилам как ANSI, так и ISO. Это означает, что комитет функционирует, скорее, как двухпалатный парламент, где основные дебаты проходят в «нижней палате» (ANSI), а затем решение ратифицируется в «верхней палате» (ISO).

Однажды такая процедура привела к отклонению предложения, которое в противном случае было бы принято незначительным большинством голосов. Таким образом национальные представители уберегли нас от ошибки, которая могла бы вызвать разногласия. Вопрос был в том, следует ли в какой-то форме определять для C++ ограничения на трансляцию. Гораздо лучшее решение по этому поводу принято позднее.

ANSI и ISO проводят совместные заседания три раза в год. Чтобы не вносить путаницу, я буду называть две эти структуры словом «комитет». Каждая встреча продолжается неделю, много часов уходит на решение процедурных вопросов. Но гораздо больше времени тратится на всякие недоразумения, неизбежные, когда 70 человек пытаются разобраться, чту же все-таки обсуждается. Несколько дневных и несколько вечерних заседаний носят технический характер: здесь обсуждаются такие вопросы, как международные наборы символов и идентификация типов во время выполнения, а также проблемы, относящиеся собственно к стандартизации, например формальные методики и организация международных структур по стандартизации. Остальное время отведено для встреч рабочих групп и обсуждений представленных ими отчетов.

Сейчас в работе участвуют такие группы:

- по совместимости с C;
- по ядру языка;
- редакторская;
- по окружению;
- по расширениям;
- по вопросам интернационализации;
- по библиотекам;
- по синтаксису.

Ясно, что за три недели в году комитету не справиться с таким обширным спектром вопросов, поэтому большая часть работы проводится в промежутках между встречами. На каждое заседание представляется стопка отпечатанных на обеих сторонах листа документов толщиной около трех дюймов. Документы посылаются в двух бандеролях: одна за пару недель до заседания, чтобы все члены комитета могли с ними ознакомиться, а вторая – через две недели после заседания.

6.2.1. Кто работает в комитете

В комитет по C++ входят люди с разными интересами, ожиданиями и подготовкой. Одни работают на персональных компьютерах, другие – на UNIX-машинах, третьи – на мейнфреймах и т.д. Одни используют C++, другие – нет. Кто-то хочет, чтобы C++ стал более объектно-ориентированным языком (причем «объектной ориентированности» даются очень разные определения), других бы больше устроило, если бы эволюция C остановилась на ANSI C. Многие, но не все работали с C. Некоторые имеют опыт работы над стандартами, но их меньшинство. Есть профессиональные программисты. Одни обслуживают интересы конечных пользователей, другие поставляют на рынок инструментальные средства. Некоторым интересны крупные проекты. Кому-то необходима совместимость с языком C, кому-то – нет.

Если не считать того, что все входящие в комитет – добровольцы, не получающие за свою работу в нем ни копейки (хотя некоторые представляют компании), то трудно выделить какие-то черты, присущие всем без исключения. И это хорошо: только очень разношерстная группа людей может выразить интересы весьма неоднородного сообщества пользователей C++. Правда, из-за этого дискуссии бывают не очень конструктивными и занимают много времени. Меня беспокоит также, что голос пользователей C++ (программистов и проектировщиков) может быть не услышан в хоре языковых пуристов, так называемых проектировщиков языков, бюрократов от стандартизации, разработчиков компиляторов и т.д.

У меня совещательный голос, то есть я представляю свою компанию, но голосует от нее другой человек. Узнать, какие компании представлены в комитете, вы сможете, взглянув на выборку из перечня участников, относящегося к 1990 г.: Amdahl, Apple, AT&T, Bellcore, Borland, British Aerospace, CDC, Data General, DEC, Fujitsu, Hewlett-Packard, IBM, Los Alamos National Labs, Lucid, Mentor Graphics, Microsoft, MIPS, NEC, NIH, Object Design, Ontologics, Prime Computer, SAS Institute, Siemens Nixdorf, Silicon Graphics, Sun, Tandem Computers, Tektronix, Texas Instruments, Unisys, US WEST, Wang, Zortech и др. Надеюсь, вы согласитесь, что отрасль представлена довольно широко.

6.3. Как велась работа

Большая часть работы над стандартами не видна среднему программисту, а когда начинаешь ее описывать, выглядит эта работа скучной и понятной лишь посвященным. Множество усилий тратится на ясное и полное выражение и без того известных всем, но не упомянутых в руководстве возможностей, а также на разрешение вопросов, важных для производителей компиляторов. Ведь это они хотят быть уверены, что правильно поняли то или иное языковое средство. Однако подобные вопросы становятся потом существенными и для программистов, поскольку в основу даже очень тщательно написанной большой программы может быть случайно или намеренно положена возможность, для кого-то не полностью понятная. Если производители компиляторов не договорятся, то программист окажется «заложником» единственного поставщика, а это уже противоречит моим взглядам на то, каким должен быть C++ (см. раздел 2.1).

Чтобы проиллюстрировать возникающие сложности, я подробно остановлюсь на двух вопросах: разрешении имен и продолжительности функционирования временных объектов. Большая часть усилий комитета тратится на решение именно этих проблем.

6.3.1. Разрешение имен

Самые большие сложности в определении C++ связаны с разрешением имен: с каким точно объявлением связано данное употребление имени. Ниже описана лишь одна проблема такого рода. Она относится к зависимости от порядка объявления членов класса. Рассмотрим пример:

```
int x;

class X {
   int f() { return x; }
   int x;
};
```

Какой `x` имеется в виду внутри `X::f()`? А вот другой пример:

```
typedef char* T;

class Y {
   T f() { T a = 0; return a; }
   typedef int T;
};
```

Какое `T` имеется в виду внутри `Y::f()`?

В ARM даются такие ответы: `x` в определении `X::f()` – это `X::x`, а определение класса `Y` содержит ошибку, поскольку смысл `T` изменяется после использования в `Y::f()`.

Эндрю Кениг, Скотт Тэрнер (Scott Turner), Том Пеннелло, Билл Гиббонс (Bill Gibbons) и еще несколько человек потратили много часов, чтобы найти точные, полные, полезные на практике, логически непротиворечивые и совместимые (со стандартом C и существующим кодом C++) решения. Мое участие в такого рода спорах было ограничено: я как раз размышлял о расширениях.

Сложность проистекает из противоречивости целей:

- мы хотим, чтобы при синтаксическом анализе можно было ограничиться одним проходом по тексту;
- изменение порядка следования членов не должно менять семантику класса;
- семантика функции не должна зависеть от того, определена данная функция прямо в объявлении класса или вне его;
- имена, объявленные во внешней области действия, должны быть видимы и во внутренней (так же, как в C);
- правила разрешения имен не должны зависеть от того, к чему относится имя.

Если строго следовать всем этим установкам, то синтаксический анализ будет достаточно быстрым, а пользователям не придется заботиться о соблюдении

правил, поскольку компилятор по большей части способен найти неоднозначные случаи употребления. В том виде, в каком они сформулированы сегодня, эти правила очень удачны.

6.3.1.1. Правила разрешения имен в ARM

В ARM эти проблемы решены без особого успеха. Имена из внешней области действия можно использовать непосредственно, а возникающие при этом зависимости от порядка я попытался минимизировать с помощью двух правил:

- правило переопределения типа: имя типа нельзя переопределять в классе после того, как оно в нем уже использовалось;
- правило переписывания: функции-члены, определенные в объявлении класса, разбираются так, как если бы они были определены в точке, непосредственно следующей за концом объявления этого класса.

Из-за правила переопределения класс `Y` дает ошибку:

```
typedef char* T;

class Y {
   T f() { T a = 0; return a; }
   typedef int T;    // ошибка: T переопределен после
                     // использования
};
```

Правило переписывания говорит, что `class X` следует интерпретировать как:

```
int x;

class X {
   int f();
   int x;
};

inline int X::f() { return x; }      // возвращает X::x
```

К сожалению, не все примеры так просты. Рассмотрим:

```
const int i = 99;

class Z {
   int a[i];
   int f() { return i; }
   enum { i = 7 };
};
```

Этот пример корректен согласно букве правил ARM, но находится в некотором противоречии их идеям. Два использования `i` относятся к разным определениям и дают разные значения. Правило переписывания говорит, что `i` в `Z::f()` – это `Z::i` и равно 7. Однако для использования `i` в качестве индекса никакого правила переписывания нет, поэтому имеется в виду глобальное `i`, равное 99.

Несмотря на то что i используется для определения типа, само оно именем типа не является, поэтому не подпадает под действие правила переопределения. Правила ANSI/ISO говорят, что этот пример некорректен, так как i переопределено после использования.

Вот другой пример:

```
class T {
   A f();
   void g() { A a; /* ... */ }
   typedef int A;
};
```

Предположим, что тип A не был определен вне T. Законно ли объявление T::f()? А определение T::g()? В ARM объявление T::f() считается незаконным, поскольку тип A в этой точке не определен. Это не вызывает противоречия с ANSI/ISO. С другой стороны, согласно ARM, определение g() законно, если правило переписывания интерпретировать в том смысле, что переписывание производится до начала синтаксического анализа, и незаконно, если допустить, что сначала выполняется синтаксический анализ, а затем – переписывание. Проблема заключается в том, является ли A именем типа к моменту начала анализа. Я полагаю, что в ARM выражена первая точка зрения (то есть объявление T::g() законно), но не стал бы утверждать, что это бесспорно.

6.3.1.2. Зачем допускать опережающие ссылки?

Видимо, всех этих проблем можно было бы избежать, если настоять на применении только однопроходных анализаторов. Имя можно использовать только тогда, когда оно объявлено «выше/раньше», а то, что происходит «ниже/позже», не влияет на это объявление. В конце концов, именно это правило применяется в C и в C++. Например:

```
int x;

void f()
{
   int y=x;      // глобальная x
   int x=7;
   int x=x;      // локальная x
}
```

Однако, когда я только проектировал классы и встраиваемые функции, Дуг Макилрой убедительно возразил, что применение данного правила к объявлениям классов может привести к путанице. Например:

```
int x;

class X {
   void f() { int y = x; } // ::x или X::x?
   void g();
   int x;
```

```
    void h() { int y = x; } // X::x
};

void X::g() { int y = x; } // X::x
```

Если объявление X велико, то присутствие различных x часто будет оставаться незамеченным. Более того, если член x используется не очень последовательно, то изменение порядка членов приведет к молчаливому изменению семантики. Вынесение определения функции за пределы объявления класса также может повлечь за собой изменение семантики без предупреждения. Правила переписывания и переопределения защищали от подобных коварных ошибок и обеспечивали некоторую свободу при реорганизации классов.

Эти аргументы применимы не только к классам, но издержки компилятора на обеспечение такой защиты приемлемы лишь для классов, и лишь для классов можно избежать проблем совместимости с C. Кроме того, именно в объявлениях классов переупорядочение делается чаще всего и как раз там оно может дать нежелательные побочные эффекты.

6.3.1.3. Правила разрешения имен ANSI/ISO

С годами мы нашли много примеров, для которых явных правил ARM было недостаточно. Зависимость от порядка в них была совершенно неочевидной и потенциально опасной, а интерпретация правил – неоднозначной. Один из интересных примеров нашел Скотт Тэрнер:

```
typedef int P();
typedef int Q();
class X {
    static P(Q);  // определим Q как P
                  // эквивалентно "static int Q()"
                  // скобки, в которые взят Q, избыточны

                  // Q в этой области действия уже не тип

    static Q(P);  // определим Q как функцию, принимающую аргумент
                  // типа P и возвращающую int.
                  // эквивалентно "static int Q(int())"
};
```

Объявление двух функций с одинаковыми именами в одной и той же области действия допустимо, если типы их аргументов достаточно различаются. При изменении порядка объявления членов мы получим две функции с именем P. Удалим typedef для P или для Q и получим новую семантику.

Заметим, что этот пример, как и многие другие, основан на злосчастном правиле «неявного int», унаследованном от C. Еще более десяти лет назад я пытался избавиться от него (см. раздел 2.8.1). Но, увы, есть примеры и другого рода:

```
int b;
class Z {
    static int a[sizeof(b)];
    static int b[sizeof(a)];
};
```

Данный пример содержит ошибку, поскольку b изменяет смысл после использования. К слову сказать, такие ошибки компилятору обнаружить легко, не то что в случае с P(Q).

В марте 1993 г. в Портленде комитет одобрил следующие правила:

- область действия имени, объявленного в классе, состоит не только из текста, следующего за объявителем, но также из определений всех функций, аргументов по умолчанию и инициализаторов конструкторов, которые встречаются в этом классе (включая и вложенные в него классы). Собственно имя класса в эту область не входит;
- имя, использованное в классе S, должно относиться к одному и тому же объявлению независимо от того, вычисляется оно в своем контексте или в полной области действия S. Полная область действия S состоит из самого класса S, его базовых классов и всех классов, объемлющих S. Часто это правило называют «правилом пересмотра» (the reconsideration rule);
- если изменение порядка следования объявлений членов дает новую, но корректную программу с учетом правил 1 и 2, семантика программы не определена. Часто это называют «правилом переупорядочения» (the reordering rule).

Отмечу, что лишь очень немногие программы затронуло такое изменение правил. Новые установки – это лишь более четкая формулировка первоначальных намерений. На первый взгляд, они требуют многопроходного алгоритма при реализации C++. Но их можно реализовать за один проход по тексту, за которым следуют один или несколько просмотров собранной на этом проходе информации. Производительность при этом почти не меняется.

6.3.2. Время жизни объектов

Для многих операций в C++ приходится создавать временные значения. Например:

```
void f(X a1, X a2)
{
   extern void g(const X&);
   X z;
   // ...
   z = a1+a2;
   g(a1+a2);
   // ...
}
```

Обычно для хранения результата a1+a2 перед присваиванием его переменной z необходим объект (вероятно, типа X). Точно так же нужен объект, в который помещается сумма a1+a2 перед передачей функции g(). Предположим, что в классе X есть деструктор. В какой момент он должен вызываться для временного объекта? Первоначально ответ на этот вопрос звучал так: «в конце блока, точно так же, как для любой другой локальной переменной». Но оказалось, что тут есть две проблемы:

- иногда при этом объект уничтожается слишком рано. Например, g() может поместить указатель на свой аргумент (временный объект, в котором

хранится сумма a1+a2) в стек, а какая-то другая функция попытается затем воспользоваться им, сняв его со стека уже после того, как произошел возврат из f() и, стало быть, временный объект уничтожен;

- иногда временный объект живет слишком долго. Например, что произойдет, если X – тип матрицы размером 1000×1000, а в блоке создается несколько десятков временных матриц? Так можно исчерпать даже довольно большую физическую память и заставить механизм виртуальной памяти постоянно подкачивать страницы.

По-моему, в реальных программах первая проблема встречается редко и общее решение дает автоматическая сборка мусора (см. раздел 10.7). Но вот вторая весьма распространена и серьезна. Некоторые пользователи вынуждены решать ее, заключая в отдельный блок каждое предложение, в котором есть вероятность создания временных объектов:

```
void f(X a1, X a2)
{
   extern void g(const X&);
   X z;
   // ...
   {z = a1+a2;}
   {g(a1+a2);}
   // ...
}
```

Если точка уничтожения расположена в конце блока (в Cfront так оно и есть), то у пользователей хотя бы остается возможность явно обойти проблему. Однако некоторые требовали более удобного ее решения. Поэтому в ARM это правило было ослаблено, и объект мог теперь уничтожаться в любой точке между первым его использованием и концом блока. Но, поскольку в разных компиляторах время жизни объекта устанавливалось по-разному, ситуация не стала лучше. Поэтому невозможно было написать гарантированно переносимую программу, если только не предполагать, что временный объект уничтожается немедленно. Это решение быстро признали неприемлемым, поскольку оно было несовместимо с некоторыми общеупотребительными в C идиомами, например:

```
class String {
   // ...
public:
   friend String operator+(const String&, const String&);
   // ...
   operator const char*(); // C-строка
};

void f(String s1, String s2)
{
   printf("%s", (const char*)(s1+s2));
   // ...
}
```

Для создания С-строки, передаваемой для печати функции printf, используется конвертор из класса String. В типичной реализации конвертор просто возвращает указатель на часть объекта String.

При такой простой реализации конвертора этот пример не будет работать, если применяется «немедленное уничтожение временных объектов». Происходит вот что: создается временный объект для хранения s1+s2, указатель на внутреннюю область этого объекта передается конвертору в С-строку, временный объект уничтожается, а уже после этого указатель на уничтоженный объект передается printf(). Но деструктор временного объекта класса String освободил память, где находилась С-строка.

Такой код распространен широко, и даже те реализации, в которых обычно выдержана стратегия немедленного уничтожения, например GNU C++, стараются в подобных случаях отложить его. Размышления в этом русле натолкнули меня на мысль уничтожать временные объекты в конце предложения, в котором они были сконструированы. Тогда приведенный выше пример был бы не только корректным, но и гарантированно переносимым. Однако при этом перестали бы работать другие, «почти эквивалентные» примеры:

```
void g(String s1, String s2)
{
   const char* p = s1+s2;
   printf("%s",p);
   // ...
}
```

При использовании стратегии «уничтожения в конце предложения» С-строка, на которую указывает p, оказалась бы во временном объекте, представляющем s1+s2, и была бы уничтожена в конце предложения инициализации p.

Дискуссии о продолжительности жизни временных объектов не затихали в комитете примерно два года, пока Дэг Брюк (Dag Brьck) не положил им конец. Перед этим комитет потратил массу времени в спорах о сравнительных достоинствах тех или иных решений. Все соглашались, что ни одно из них не совершенно. Мое мнение, выраженное довольно резко, заключалось в том, что «промедление смерти подобно», поэтому хоть какому-то решению нужно отдать предпочтение. Я думаю, что был выбран наилучший вариант.

В июле 1993 г. Дэг Брюк представил отчет о состоянии дел по этому вопросу, основанный главным образом на работе Эндрю Кенига, Скотта Тэрнера и Тома Пеннелло. В нем были определены семь вариантов установления точки разрушения временного объекта:

- сразу после первого использования;
- в конце предложения;
- в следующей точке ветвления;
- в конце блока (оригинальное правило C++, реализованное в Cfront);
- в конце функции;
- после последнего использования (неявно подразумевается сборка мусора);
- где-то между первым использованием и концом блока (правило ARM).

Оставляю подбор аргументов в пользу каждого варианта в качестве упражнения для читателя. Однако можно найти и серьезные возражения против каждого из указанных подходов. Следовательно, необходимо выбрать вариант с оптимальным сочетанием преимуществ и недостатков.

Мы также рассматривали возможность разрушения временного объекта после его последнего использования в блоке. Но это потребовало бы анализа потока выполнения, а у нас не было уверенности, что любой компилятор сможет провести такой анализ достаточно надежно, чтобы точка «за последним использованием в блоке» означала бы в любой реализации одно и то же. Отмечу, что локального анализа потока недостаточно для предотвращения «преждевременного уничтожения», например потому, что конверторы, возвращающие указатель на внутреннюю область объекта, часто определяются не в той единице трансляции, в которой используются. Попытка запретить такие функции не имела смысла, так как при этом перестало бы работать слишком много программ. Да и контроль над ситуацией был невозможен.

С 1991 г. комитет склонялся к варианту «конца предложения». В обиходе он получил название EOS (end of statement). Оставалось точно определить, что^ же такое «конец предложения». Рассмотрим пример:

```
void h(String s1, String s2)
{
   const char* p;

   if (p = s1+s2) {
      // ...
   }
}
```

Будет ли значение `p` использоваться внутри предложения блока? Иначе говоря, нужно ли уничтожать объект, содержащий `s1+s2`, в конце условия или в конце всего предложения `if`? Ответ: объект, содержащий `s1+s2`, будет уничтожен в конце условия. Было бы абсурдно гарантировать что-то относительно предложения

```
if (p = s1+s2) printf("%s",p);
```

оставляя

```
p = s1+s2;
printf("%s",p);
```

зависящим от реализации.

Как следует обрабатывать ветвления внутри выражения? Например, должно ли гарантированно работать такое предложение:

```
if ((p = s1+s2) && p[0]) {
   // ...
}
```

Да, должно. Но гораздо проще привести готовый ответ, чем объяснить специальные правила для `&&`, `||` и `?:`. Против этого предложения выдвигались возражения,

поскольку чаще всего его невозможно реализовать, не вводя флаги, гарантирующие, что временный объект уничтожается, только когда он появился именно в той ветви, по которой пошло выполнение. Однако разработчики компиляторов ответили на вызов и продемонстрировали, что возникающие затраты чрезвычайно малы.

Таким образом, EOS стало означать «конец полного выражения», где под полным выражением понималось выражение, не являющееся подвыражением другого выражения.

Разрешив уничтожать временные объекты в конце полного выражения, мы «разрушили» некоторые программы, компилировавшиеся ранее Cfront, но все гарантии, предоставленные ARM, были сохранены. Принятое решение отражает желание иметь четко определенное и легко поддающееся объяснению местоположение точки уничтожения. Оно также согласуется с желанием не хранить временные объекты слишком долго. Объекты, которые должны существовать дольше, надо именовать, или использовать приемы, не требующие долгоживущих временных объектов. Например:

```
void f(String s1, String s2)
{
   printf("%s", s1+s2); // правильно

   const char* p = s1+s2;
   printf("%s", p); // не работает, временный объект уничтожен

   String s3 = s1+s2;
   printf("%s", (const char*)s3);    // правильно

   cout << s3;   // правильно

   cout << s1+s2;    // правильно
}
```

6.4. Расширения

Исключительно важным был и остается вопрос о том, как справиться с нескончаемым потоком предложений об изменениях и расширениях языка. Основная роль при его решении отводится рабочей группе по расширениям, в которой я являюсь председателем. Любое предложение гораздо проще принять, чем отвергнуть. Так вы приобретаете друзей, а люди ценят язык, в котором много «полезных штучек». Однако язык, построенный как конгломерат средств, внутренне не связанных между собой, неудачен, поэтому нет никакой возможности принять даже большую часть средств, которые были бы по-настоящему полезны той или иной части сообщества пользователей C++.

На вышеупомянутой встрече в Лунде была очень популярна такая история [Stroustrup,1992b]:

«Мы часто вспоминаем о прекрасном корабле «Vasa». Он был гордостью шведского Военно-морского флота, самым большим и красивым из когда-либо построенных военных судов. К сожалению, во время строительства в проект были внесены существенные изменения, поскольку

в первоначальном варианте не было предусмотрено места для всех статуй и пушек, которые хотели установить на корабле. Не успело судно доплыть до середины Стокгольмской бухты, как налетевший порыв ветра перевернул его, и примерно пятьдесят человек погибли. Корабль подняли со дна, и теперь на него можно посмотреть в музее Стокгольма. Он прекрасен – гораздо красивее, чем был бы оригинальный, выполненный по первоначальному проекту, и уж, конечно, много красивее того, во что бы он превратился, если бы разделил обычную судьбу военных кораблей семнадцатого века. Но вряд ли это может служить утешением для инженеров, строителей и моряков, которые так и не смогли увидеть «Vasa» в деле».

Зачем вообще рассматривать расширения? Ведь X3J16 – это комитет по стандартизации, а не группа проектирования, которая обязана разработать язык «C++++». Да и не может коллектив из 250 человек с переменным составом надеяться на успех в области проектирования языка.

Поначалу группе было поручено заняться шаблонами и обработкой исключений. Но еще до того как комитет приступил к работе, на его представителей посыпались предложения о расширениях и несовместимых изменениях. Пользователи, даже те из них, кто сам ничего не предлагал, ждали, что комитет предложения рассмотрит. Если комитет принимает такие предложения всерьез, а это обычная практика, он становится центром дискуссий по поводу будущего C++. Если же не обращать внимания на чьи-то инициативы, то этот центр просто переместится в другое место, что приведет к появлению несовместимых расширений.

Кроме того, многие любят новые возможности, хотя публично расписываются в преданности минимализму и стабильности. Проектирование языка – само по себе интересное занятие, дебаты вокруг новых возможностей стимулируют воображение и служат хорошим поводом для написания статей и выпуска новых версий. Поэтому я предпочитаю «дать выход пару», извлекая из этого конструктивные преимущества.

Итак, у комитета был выбор: обсуждать расширения, рассматривать диалекты уже после их появления или игнорировать ситуацию. Подобне дилеммы вставали перед разными комитетами по стандартизации, и каждый принимал свое решение. Большинство, в том числе комитеты по языкам Ada, C, Cobol, Fortran, Modula-2 и Pascal-2, решали вопрос в пользу рассмотрения расширений.

Я уверен, что стремление добавить языку то или иное расширение неизбежно, и лучше, чтобы это происходило открыто, на публичном форуме, где есть хоть какие-то формальные правила. Альтернатива – схватка за признание собственных идей через привлечение на свою сторону пользователей на рынке. Такой механизм не способствует спокойным размышлениям, открытым дискуссиям и попыткам удовлетворить потребности всех пользователей. В результате мы получаем язык, распавшийся на диалекты.

Очевидные опасности, присущие рассмотрению указанного вопроса, лучше, чем хаос, который воцарится в противном случае. Большинство членов комитета согласилось со мной, но мы шли к той точке, когда работа над расширениями в том виде, в котором она ведется сейчас, должна была прекратиться, поскольку приближался момент принятия стандарта и все направили усилия на его критическое изучение.

Со временем кто-то переключился на другие языки, кто-то занялся экспериментальной работой, кто-то посвятил себя созданию библиотек. Интересно отметить, что группы по стандартизации, как и любые другие организации, расформировываются с большой неохотой. Часто такая группа возрождается, пусть даже в виде бюрократического механизма для создания стандарта следующего уровня, то есть комитета по проектированию нового языка или диалекта. Примерами подобного явления служат комитеты по языкам Algol, Fortran, Pascal и даже ANSI C. Тем временем я стараюсь обезопасить себя от попыток проектирования чего-либо самим комитетом, посвящая много времени каждому поступившему предложению. Стратегия эта хоть как-то защищает от принятия взаимоисключающих средств, и язык, возможно, не утратит внутреннюю целостность.

Комитет легко может угодить в ловушку и одобрить некоторую возможность просто потому, что кто-то настаивает на ее необходимости. Всегда проще отстаивать полезность того или иного средства, чем доказывать, что его преимущества – возможно, на самом деле очень важные – не навредят внутренней логике, простоте и стабильности языка. Кроме того, в комитете не склонны прислушиваться к аргументам, основанным на экспериментах и опыте. Видимо, когда решения принимаются путем голосования, лучшими аргументами являются те, что легче усваиваются уставшими представителями комитета.

Таким образом, «стандартизация» может стать источником нестабильности. Всегда есть опасность, что принятое изменение будет совершенно случайным или даже никуда не годным. Чтобы не допустить этого, к стандартизации надо приступать на подходящей стадии эволюции языка, когда пути развития четко определены, а диалекты, поддерживаемые солидными коммерческими структурами, еще не успели появиться. Я надеюсь, что для C++ момент был выбран правильно и комитет будет и дальше проявлять необходимую консервативность в отношении нововведений.

Стоит напомнить, что некоторые пользователи обходятся и вовсе без расширений. Поборники новых возможностей как-то забывают, что хорошую программу можно построить и без изощренной языковой поддержки. Никакое отдельное средство не является необходимым для проектирования симпатичной программы, даже такое, без которого мы не мыслим своей работы. Удачную программу можно написать и на C, и на небольшом подмножестве C++. Языковые средства хороши тем, что помогают программисту выразить свои идеи, уменьшают время отладки программы, делают код более понятным и облегчают его сопровождение.

6.4.1. Критерии рассмотрения предложений

Чтобы помочь пользователям понять, какие факторы должны сопутствовать внесению предложения о расширении или изменении C++, рабочая группа по расширениям сформулировала ряд вопросов, которые предположительно задаются соискателю [Stroustrup, 1992b]:

«Этот перечень содержит критерии оценки новых возможностей C++:

- **точность формулировки. Ясно и точно опишите суть изменения и то, как оно отразится на текущем предварительном стандарте справочного руководства по языку:**

  - какие потребуются изменения грамматики?
  - как изменится описание семантики языка?
  - хорошо ли изменение согласуется с остальной частью языка?
- обоснование изменения:
  - зачем нужно изменение?
  - для кого будет полезно?
  - является ли оно изменением общего назначения?
  - затрагивает ли предложение интересы одних групп пользователей C++ больше, чем других?
  - реализуемо ли оно на всех разумных аппаратных платформах и операционных системах?
  - принесет ли ваше предложение пользу на всех разумных аппаратных платформах и операционных системах?
  - какие стили программирования и проектирования поддерживает предполагаемое изменение?
  - каким стилям программирования и проектирования оно мешает?
  - в каких языках есть аналогичные средства (если таковые существуют)?
  - упрощает ли оно проектирование, реализацию и использование библиотек?
- вероятность реализации. Было ли реализовано ваше изменение? Если да, то в какой форме (в точности в той, какую вы предлагаете)? Если нет, почему вы думаете, что опыт аналогичных реализаций или других языков можно перенести на ту форму, в которой вы предлагаете свое средство?
  - как оно отразится на реализации C++?
    - на организации компилятора?
    - на поддержке времени исполнения?
  - была ли реализация полной?
  - использовалась ли реализация кем-то, кроме разработчиков?
- влияние изменения на код:
  - как выглядит код без изменения?
  - что будет с кодом, если игнорировать внесение изменения?
  - предъявляет ли новое средство какие-то требования к инструментальным средствам?
- влияние изменения на эффективность и совместимость с C и C++:
  - как отразится ваше предложение на эффективности исполнения кода, в котором оно
    - используется?
    - не используется?
  - как изменение повлияет на время компиляции и компоновки?
  - как оно отразится на существующих программах?
    - нужно ли перекомпилировать код, который не использует новое средство?
    - повлияет ли изменение на возможность связывания с такими языками, как C или Fortran?
  - изменится ли качество статического или динамического контроля типов в программах на C++?
- простота документирования нового средства и легкость в обучении:
  - начинающих пользователей?
  - опытных пользователей?
- причины, по которым не стоит принимать изменение. Мы обязательно приведем контрдоводы, найти и оценить их – часть нашей работы. Поэтому вы можете сэкономить время, подготовившись к дискуссии по вопросам:
  - отразится ли изменение на старом коде, который не использует новое средство?
  - трудно ли научить людей пользоваться им?

  - потребуются ли дальнейшие расширения?
  - увеличится ли размер компилятора?
  - будет ли нужна обширная поддержка во время исполнения?
- существуют ли:
  - альтернативные способы удовлетворить те же потребности?
  - иные варианты использования предложенного синтаксиса?
  - привлекательные обобщения предложенной схемы?

Разумеется, список представленных вопросов – неполный. Пожалуйста, расширьте его, включив пункты, имеющие отношение к вашему предложению, и исключив те, которые к нему не относятся».

Эти вопросы всегда должны задавать себе проектировщики реальных языков.

6.4.2. Текущее состояние дел

Вот краткое описание ситуации после заседания комитета в Вэлли Фордж в ноябре 1994 г. Предлагать расширения для C++, похоже, стало популярным делом. Среди вариантов:

- расширенные (международные) наборы символов (см. раздел 6.5.3.2);
- различные расширения шаблонов (см. разделы 15.3.1, 15.4, 15.8.2);
- сборка мусора (см. раздел 10.7);
- предложения группы NCEG (например, см. раздел 6.5.2);
- объединения с дискриминантами;
- определенные пользователем операторы (см. раздел 11.6.2);
- непрямые классы (indirect classes);
- перечисления с предопределенными операторами `++`, `<<` и т.д.;
- перегрузка с учетом типа возвращаемого значения;
- составные операторы (см. раздел 11.6.3);
- ключевое слово для нулевого указателя (`NULL`, `nil` и т.д.) – см. раздел 11.2.3;
- пред- и постусловия;
- улучшенные макросы для Cpp;
- повторная привязка ссылок;
- продолжения (continuations).

Есть все же надежда на то, что этому потоку будет положен предел и что одобренные расширения будут удачно интегрированы в язык. До сих пор было принято лишь несколько новых возможностей, а именно:

- обработка исключений – см. главу 16;
- шаблоны – см. главу 15;
- представление европейского набора символов (см. раздел 6.5.3.1);
- ослабление правила для типа возвращаемого значения в замещающих функциях;
- идентификация типов во время исполнения (см. раздел 14.2);
- объявления в условиях (см. раздел 3.11.5.2);
- перегрузка, основанная на перечислениях (см. раздел 11.7.1);
- определенные пользователем операторы выделения и освобождения памяти для массивов (см. раздел 10.3);

- опережающие объявления вложенных классов (см. раздел 13.5);
- пространства имен (см. главу 17);
- ключевое слово `mutable` (см. раздел 13.3.3);
- тип `boolean` (см. раздел 11.7.2);
- новый синтаксис приведения типов (см. раздел 14.3);
- оператор явного инстанцирования шаблона (см. раздел 15.10.4);
- явная спецификация аргументов в вызовах шаблонов функций (см. раздел 15.6.2);
- шаблоны-члены (см. раздел 15.9.3);
- шаблоны классов как аргументы шаблонов (см. раздел 15.3.1);
- возможность инициализировать константные статические члены интегральных типов константным выражением внутри объявления класса;
- явные конструкторы (см. раздел 3.6.1);
- статическая проверка спецификаций исключений (см. раздел 16.9).

Исключения и шаблоны выделяются среди прочих расширений тем, что должны быть проработаны рабочей группой, входили в первоначальное предложение и описаны в ARM. Кроме того, определить и реализовать их в два раза труднее, чем все остальные варианты.

В то же время комитет отклонил много предложений, например:

- несколько предложений о прямой поддержке параллельности;
- переименование унаследованных имен (см. раздел 12.8);
- именованные аргументы (см. раздел 6.5.1);
- несколько предложений по несущественным изменениям правил сокрытия данных;
- ограниченные указатели («родственник noalias») – см. раздел 6.5.2;
- оператор возведения в степень (см. раздел 11.5.2);
- автоматически генерируемые составные операторы;
- определяемые пользователем операторы и `()` (см. раздел 11.5.2);
- вложенные функции;
- двоичные литералы;
- обобщенная инициализация членов в объявлении класса.

Хочу подчеркнуть, что отклоненные возможности не обязательно плохи или бесполезны. На самом деле, ряд предложений, доходящих до комитета, исполнен технически грамотно и принес бы пользу, по крайней мере, некоторой части пользователей C++. Просто большинство идей было недостаточно проанализировано и оформлено для того, чтобы называться предложениями.

6.4.3. Проблемы, связанные с полезными расширениями

Даже то расширение, которое нужно на самом деле, порождает ряд проблем. Оно, как минимум, отвлекает силы разработчика от других важных задач. Например, что произойдет, если разработчику компилятора придется выбирать между реализацией нового средства и оптимизацией кодогенератора? Наверное, в большинстве случаев он отдаст предпочтение новому средству, поскольку оно более заметно для пользователей.

Расширение может быть безупречным, если рассматривать его в отдельности от всего остального, и все же иметь недостатки, если видеть всю картину целиком. При работе над расширениями основной акцент делается на интеграции в язык и на взаимодействии с другими его возможностями. Трудность такой работы и необходимое для нее время постоянно недооцениваются.

Любое новое средство делает все существующие реализации устаревшими. Поэтому пользователям приходится обновлять версии компиляторов, обходиться некоторое время без нового средства или сопровождать две версии системы (для нового и старого компиляторов). Последний вариант обычно приходится выбирать разработчикам библиотек и инструментальных средств.

Документацию и прочую учебную литературу также приходится обновлять и иногда специального оговаривать, каким был язык до введения нового средства, чтобы помочь пользователям, еще не перешедшим на новую версию.

Все это негативные последствия «идеального» расширения. А если оно и изначально противоречиво, то от членов комитета и от сообщества в целом потребуются дополнительные усилия. Если у расширения есть аспекты, которые несовместимы с предыдущими версиями языка, при переходе на новую версию это необходимо учесть, и не исключено, что эффект будет заметен даже тогда, когда новое средство не используется. Классический пример – введение нового ключевого слова. Такая безобидная, на первый взгляд, функция

```
void using(Table* namespace) { /* ... */ }
```

перестала компилироваться после введения пространств имен, поскольку `using` и `namespace` оказались ключевыми словами. Мой опыт, впрочем, показывает, что введение новых ключевых слов создает немного проблем и все они легко решаются. С другой стороны, предложение ввести новое ключевое слово всегда вызывает взрыв негодования, хотя практические сложности можно свести к минимуму за счет выбора такого слова, которое вряд ли будет конфликтовать с именами идентификаторов. Именно по этой причине было выбрано `using`, а не `use`, и `namespace`, а не `scope`. Когда ради эксперимента мы без предварительного уведомления ввели ключевые слова `using` и `namespace` в используемый внутри фирмы компилятор, никто за два месяца их даже не заметил.

Помимо вполне реальных проблем, связанных с принятием и реализацией новых возможностей, даже сам факт их обсуждения может производить негативный эффект: пользователи начинают сомневаться в стабильности языка. Многие неискушенные не отдают себе отчет в том, что изменения тщательно фильтруются, чтобы свести к минимуму влияние на уже написанные программы. С другой стороны, идеалистически настроенные поборники новых возможностей часто не хотят ничего предпринимать, чтобы развеять опасения на счет нестабильности. Кроме того, энтузиасты «улучшений» нередко преувеличивают слабости языка, чтобы выставить свои предложения в более выгодном свете.

6.4.4. Логическая непротиворечивость

На мой взгляд, основная трудность при рассмотрении расширения – обеспечить логическую непротиворечивость C++ и убедить пользователей в том, что

данная цель достигнута. Возможности, включаемые в C++, должны работать согласованно и компенсировать серьезные недостатки программы, синтаксически и семантически укладываться в структуру языка и способствовать разумному стилю программирования. Язык программирования не может быть просто собранием интересных возможностей, поэтому при оценке и включении расширений важно привести их к такому виду, чтобы они стали естественной частью языка. При серьезном анализе некоторых расширений, 95% времени уходит на то, чтобы найти форму оригинальной идеи, которая позволила бы гармонично интегрировать ее в C++. Обычно много сил тратится на выработку путей перехода к новой версии языка для разработчиков компиляторов и пользователей. Даже самое удачное средство приходится отклонять, если им нельзя будет воспользоваться, не удалив значительную часть прежнего кода и старых инструментальных средств.

6.5. Примеры предлагавшихся расширений

Предлагаемые для включения в язык расширения рассматриваются в книге в контексте связанных с ними средств, но некоторые просто некуда отнести, поэтому я привожу их здесь в качестве примеров. Неудивительно, что средства, которым нигде не нашлось места, обычно отклоняются. Какой бы разумной ни казалась возможность сама по себе, к ней следует относиться с большим подозрением, если она не помогает развитию языка в нужном направлении.

6.5.1. Именованные аргументы

Предложение Роланда Хартингера (Roland Hartinger) об именованных аргументах, т.е. о возможности при вызове функции задавать аргументы по имени, с технической точки зрения было близко к совершенству. Поэтому особенно интересна причина, по которой его отклонили: группа по расширениям единодушно решила, что предложение не дает почти ничего нового, приведет к несовместимости с существующим кодом и будет способствовать распространению неудачного стиля программирования. Ниже отражено обсуждение, состоявшееся по этому поводу в рабочей группе. Как обычно, сотни замечаний приходится опускать из-за недостатка места.

Рассмотрим некрасивый, но, к сожалению, вполне реалистичный пример, заимствованный из аналитической статьи Брюса Эккеля (Bruce Eckel):

```
class window {
   // ...
public:
   window(
       wintype=standard,
       int ul_corner_x=0,
       int ul_corner_y=0,
       int xsize=100,
       int ysize=100,
       color Color=black,
       border Border=single,
       color Border_color=blue,
```

```
        WSTATE window_state=open);
    // ...
};
```

Если вы хотите определить окно по умолчанию, все нормально. Определить же окно «почти по умолчанию» нелегко, при этом можно наделать ошибок. Суть предложения заключалась в том, чтобы ввести новый оператор `:=` и использовать его в вызовах функций для именования аргумента. Например,

```
new window(Color:=green,ysize:=150);
```

будет эквивалентно предложению

```
new window(standard,0,0,100,150,green);
```

что с учетом аргументов по умолчанию эквивалентно

```
new window(standard,0,0,100,150,green,single,blue,open);
```

На первый взгляд, это полезное синтаксическое удобство, которое может сделать программы более надежными. Предложение даже было реализовано, при этом не возникло никаких концептуальных проблем. Кроме того, предложенный механизм основывался на опыте других языков, в частности, Ada.

С другой стороны, нет сомнений, что без именованных аргументов вполне можно обойтись, что они не представляют собой фундаментально нового средства, не поддерживают принципиально новую парадигму программирования и не закрывают никакой «дыры» в системе контроля типов. Поэтому ответы на следующие вопросы зависят больше от личных пристрастий и представлений о состоянии сообщества пользователей C++:

- удастся ли с помощью именованных аргументов писать более хороший код?
- не приведут ли эти аргументы к путанице или проблемам при обучении языку?
- породят ли проблемы совместимости?
- должны ли именованные аргументы стать одним из немногих расширений, которые мы одобрим?

Первая сложность, с которой пришлось столкнуться, заключалась в том, что именованные аргументы вводят новый вид зависимости между интерфейсом вызова и реализацией:

- аргументу необходимо иметь одно и то же имя в объявлении и определении функции;
- если именованный аргумент использовался, то изменить его имя в определении функции уже нельзя, не нанеся ущерб пользовательской программе.

Из-за времени перекомпиляции многие не хотят увеличивать степень зависимости между интерфейсом и реализацией. Более того, за этой зависимостью скрыта существенная проблема совместимости. В ряде организаций рекомендуется использовать «длинные, информативные» имена аргументов в заголовочных файлах и «короткие, удобные» в определениях. Например:

```
void reverse(int* elements, int length_of_element_array);

// ...

void reverse(int* v, int n)
{
   // ...
}
```

Как всегда, одни находят такой стиль отвратительным, другие (в том числе я) – вполне разумным. Как бы то ни было, программ в указанном стиле написано очень много. Но, введя именованные аргументы, мы не сможем изменить ни одно имя в стандартно распространяемом заголовочном файле без риска сделать неработоспособным пользовательский код. Поставщикам заголовочных файлов для одних и тех же библиотек, например Posix или X Windows, придется договариваться об именах аргументов.

Но можно сделать так, чтобы в языке не было требования одинаково именовать один и тот же аргумент в объявлениях. Такой вариант казался мне приемлемым. Однако другим он не понравился.

Если требовать проверки того, что имена аргументов одинаковы в разных единицах трансляции, то время компоновки может существенно увеличиться. Если же не организовывать такой проверки, под угрозой окажется безопасность типов, что может стать источником скрытых ошибок.

И потенциальную проблему замедления компоновки, и вполне реальную проблему дополнительных зависимостей можно было бы решить, опустив имена аргументов в заголовочных файлах. Тогда осмотрительный пользователь мог бы не задавать там имена аргументов, но, цитируя Билла Гиббонса, «это может негативно сказаться на удобочитаемости программ на C++».

Я опасался, что именованные аргументы могут замедлить постепенный переход от традиционных приемов программирования к абстракции данных и далее к объектно-ориентированному стилю. В программах, которые, на мой взгляд, лучше всего написаны и удобны для сопровождения, длинные списки аргументов – редкость. Многие пользователи отмечали, что переход к объектно-ориентированному стилю сопровождается укорачиванием списков аргументов: то, что раньше передавалось в виде аргументов или глобальных переменных, теперь становится локальным состоянием объекта. Основываясь на опыте, я ожидаю, что среднее число аргументов будет меньше двух, а функций, имеющих более двух аргументов, почти не останется. Отсюда следует, что именованные аргументы окажутся наиболее полезными как раз в таких программах, которые мы считаем написанными плохо. Так следует ли вводить новое средство, поощряющее именно тот стиль программирования, который мы хотели бы изжить? Такая постановка вопроса, проблема совместимости и некоторые мелкие детали стали причиной единодушного отказа принять предложение об именованных аргументах.

6.5.1.1. Альтернативы именованным аргументам

Но если мы отказываемся от именованных аргументов, то как сократить длину списка аргументов в примере с классом `window`? Кажущаяся сложность уже

уменьшена за счет аргументов по умолчанию. Еще один распространенный прием – дополнительные типы для представления наиболее типичных вариантов:

```
class colored_window : public window {
public:
   colored_window(color c=black)
       : window(standard,0,0,100,100,c) { }
};


class bordered_window : public window {
public:
   bordered_window(border b=single, color bc=blue)
       : window(standard,0,0,100,100,black,b,bc) { }
};
```

Этот прием ограничивает использование функции несколькими наиболее употребительными формами и потому код и поведение программы становятся более однообразными. Другой способ – ввести явные операции для задания установок, отличных от подразумеваемых:

```
class w_args {
   wintype wt;
   int ulcx, ulcy, xz, yz;
   color wc, bc;
   border b;
   WSTATE ws;
public:
   w_args()      // установки по умолчанию
       : wt(standard), ulcx(0), ulcy(0), xz(100), yz(100),
       wc(black), b(single), bc(blue), ws(open) { }
   // отменить умолчания:

   w_args& ysize(int s) { yz=s; return *this; }
   w_args& Color(color c) { wc=c; return *this; }
   w_args& Border(border bb) { b = bb; return *this; }
   w_args& Border_color(color c) { bc=c; return *this; }
   // ...
};

class window {
   // ...
   window(w_args wa);   // скопировать настройки wa
   // ...
};
```

Это дает нам нотационное удобство, в некоторой степени эквивалентное тому, которое обеспечивают именованные аргументы:

```
window w;     // окно по умолчанию
window w( w_args().Color(green).ysize(150) );
```

Существенное преимущество такого способа состоит в том, что он облегчает передачу объектов, представляющих аргументы, между различными частями программы.

Естественно, все эти приемы можно использовать в любом сочетании. В совокупности они позволяют сократить список аргументов и, следовательно, свести на нет потребность в именованных.

Число аргументов можно еще больше сократить, если воспользоваться типом Point, вместо того чтобы передавать пары координат.

6.5.2. Ограниченные указатели

Компилятору Fortran разрешено предполагать, что два массива, переданных функции в качестве аргументов, не перекрываются. Функция C++ не вправе делать таких допущений. В результате подпрограмма, написанная на Fortran, оказывается на 15–30% быстрее в зависимости от качества компилятора и машинной архитектуры. Особенно заметную экономию дают векторные операции на машинах, имеющих специальную векторную аппаратуру (например, Cray).

Учитывая повышенное внимание к эффективности C, все вышесказанное рассматривалось, как досадная оплошность. В комитет ANSI C поступило предложение решить эту проблему с помощью механизма, названного `noalias`. При наличии этого ключевого слова считалось бы, что у указателя нет псевдонимов (то есть никакие два указателя не указывают на одну и ту же область памяти). К сожалению, предложение поступило слишком поздно и в очень непроработанном виде. Это обстоятельство побудило Денниса Ричи написать открытое письмо, начинающееся словами: «никаких `noalias`, это даже не подлежит обсуждению».

Понятно, что после этого сообщество пользователей C и C++ без особой охоты возвращалось к проблеме псевдонимов. Но вопрос был очень важен для пользователей C на платформе Cray, поэтому Майк Холли (Mike Holly) из компании Cray, в конце концов, представил улучшенное предложение по этой проблеме группе по численным расширениям C (NCEG) и комитету по C++. Идея заключалась в том, чтобы дать программисту возможность декларировать отсутствие у указателя псевдонимов, объявив его с ключевым словом `restrict`:

```
void* memcopy(void*restrict s1, const void* s2, size_t n);
```

Поскольку уже сказано, что у `s1` нет псевдонимов, нет необходимости указывать `restrict` еще и для `s2`. Ключевое слово `restrict` применяется к оператору `*` точно так же, как модификаторы `const` и `volatile`. Это предложение должно устранить преимущество Fortran в эффективности, приняв в C правила, действующие в Fortran (правда, лишь выборочно).

Комитет по C++, внимательно относившийся к любому предложению, способному повысить эффективность языка, потратил немного времени на обсуждение данного предложения, но все же оно было отклонено, насколько я помню, единогласно. Вот основные мотивы:

- расширение небезопасно. Если объявить указатель как `restrict`, то компилятору позволено предполагать, что у него нет псевдонимов. Однако

пользователь об этом может и не знать, а сам компилятор убедиться в этом не способен. Из-за интенсивного использования указателей и ссылок в C++ возможно появление большого числа ошибок, и опыт Fortran тут ничего не доказывает;

- альтернативы предложению недостаточно исследованы. Во многих случаях можно было бы провести начальную проверку на перекрытие и при его отсутствии выбрать оптимизированный вариант алгоритма. В других случаях стоит обратиться к специализированным математическим библиотекам (например, BLAS), которые имеют специальную поддержку для векторных операций. Еще предстоит изучить многообещающие альтернативные способы оптимизации. Например, глобальная оптимизация относительно небольших и легко распознаваемых участков кода, где встречаются операции с векторами и матрицами, представляется вполне реализуемой в компиляторах C++ для высокопроизводительных компьютеров;
- расширение зависит от архитектуры. Высокопроизводительные численные расчеты – это специальная область со своими методами, зачастую предполагающая наличие особой аппаратуры. Поэтому лучше ввести нестандартное архитектурно-зависимое расширение или прагму. Если же такого рода оптимизация окажется полезной и более широкому кругу пользователей, то к рассмотрению этого расширения можно будет вернуться.

Принятое решение еще раз подтверждает, что C++ поддерживает абстрагирование за счет общих механизмов, а не специализированные приложения путем реализации специальных средств. Конечно, я хотел бы помочь людям, занимающимся численными расчетами, но как? Идти по стопам Fortran в реализации классических векторных и матричных алгоритмов – это далеко не всегда самый лучший путь. Было бы хорошо, если бы любую расчетную программу можно было написать на C++ без потери эффективности, но если не удастся достичь этого, не принося в жертву систему контроля типов C++, то, видимо, стоит обратиться к Fortran, ассемблеру или архитектурно-зависимым расширениям.

6.5.3. Наборы символов

Язык C ориентирован на американский вариант международного 7-битного набора символов ISO 646-1983, называемого также ASCII (ANSI3.4-1968). Это порождает две проблемы:

- код ASCII содержит некоторые знаки препинания и специальные символы, например `]` или `{`, которых нет во многих национальных наборах символов;
- В коде ASCII отсутствуют символы типа Å и æ, которых нет в английском алфавите.

6.5.3.1. Ограниченные наборы символов

В коде ASCII (ANSI3.4-1968) специальные символы `[`, `]`, `{`, `}`, `|` и `\` занимают позиции, которые по стандарту ISO отведены под буквы. В большинстве европейских национальных наборов символов ISO-646 эти позиции заняты буквами,

которых нет в английском алфавите. Например, в датском наборе символов на этих местах находятся гласные Æ, Å, æ, å, ø и Ø, без которых невозможно написать сколько-нибудь осмысленный датский текст. Это ставит датских программистов перед неприятным выбором: либо приобретать компьютеры, на которых есть полный 8-битный набор символов (например, ISO-8859/1/2), либо не использовать три гласные буквы своего родного языка, либо не программировать на C++. Люди, говорящие на немецком, французском, испанском, итальянском и других языках, сталкиваются с той же проблемой. Кстати, данное недоразумение стало серьезным препятствием на пути признания C в Европе, особенно в коммерческих приложениях (к примеру, банковских системах), поскольку 7-битные национальные наборы символов во многих странах используются повсеместно.

Рассмотрим, например, безобидную, на первый взгляд, программу на ANSI C и C++:

```
int main(int argc, char* argv[])
{
   if (argc<1 || *argv[1] == '\0') return 0;
   printf("Hello, %s\n",argv[1]);
}
```

На стандартном датском терминале или принтере она будет выглядеть так:

```
int main(int argc, char* argvÆÅ )
æ
   if (argc<1 øø *argvÆ1Å == 'Ø0') return 0;
   printf("Hello, %sØn",argvÆ1Å);
å
```

Комитет ANSI C одобрил частичное решение проблемы, определив набор триграфов, с помощью которых можно вводить символы национальных алфавитов:

```
 #    [    {    \    ]    }    ^    |    ~
??=  ??(  ??<  ??/  ??)  ??>  ??'  ??!  ??-
```

Возможно, для обмена программами это и имеет смысл, но удобочитаемость записи остается на прежнем уровне:

```
int main(int argc, char* argv??(??))
??<
   if (argc<1 ??!??! *argv??(1??) == '??/0') return 0;
   printf("Hello, %s??/n",argv??(1??));
??>
```

Разумеется, решить эту проблему можно, купив оборудование, которое поддерживает как национальный язык, так и символы, необходимые C и C++. К сожалению, иногда это оказывается невозможным, да и вообще замена оборудования – процесс медленный. Чтобы помочь программистам, работающим на старом оборудовании, а значит, и продвигать C++, комитет по стандартизации решил предоставить более удобную нотацию.

Показанные в табл. 6.1 слова и диграфы являются эквивалентами операторов, содержащих символы национальных алфавитов.

Таблица 6.1

Ключевые слова		Диграфы	
and	&&	<%	{
and_eq	&=	%>	}
bitand	&	<:	[
bitor	|	:>	]
compl	~	%:	#
not	!	%:%:	##
or	||		
or_eq	|=		
xor	^		
xor_eq	^=		
not_eq	!=		

Я бы предпочел %% для # и <> для !=, но %: и not_eq – лучший компромисс, на котором сошлись комитеты по C и C++.

В этих обозначениях наш пример выглядит следующим образом:

```
int main(int argc, char* argv<: :>)
<%
   if (argc<1 or *argv<:1:>=='??/0') return 0;
   printf("Hello, %s??/n", argv<:1:>);
%>
```

Замечу, что триграфы все-таки необходимы для вставки таких «отсутствующих» символов, как \, в строки и символьные константы.

Введение диграфов и новых ключевых слов вызвало споры. Многие пользователи – в основном, англоговорящие и имеющие большой опыт работы с C – не видели причин усложнять и портить C++ ради тех, кто «не хочет купить приличное оборудование». Мне такая позиция близка, поскольку диграфы и триграфы – не верх изящества, а новые ключевые слова – всегда источник несовместимости. С другой стороны, мне приходилось работать на оборудовании, которое не поддерживало мой родной язык, и я видел, как люди отказывались от C в пользу «языка, в котором нет этих дурацких символов». Также я припоминаю отчет представителя IBM о том, что отсутствие символа ! в кодировке EBCDIC, применяемой на мейнфреймах IBM, – причина частых жалоб. Интересно отметить, что даже в тех случаях, когда расширенные наборы символов имеются, системные администраторы иногда запрещают их употребление.

Думается, что на переходный период, который может продлиться лет десять, диграфы и триграфы – это меньшее из зол. Надеюсь, это поможет C++ завоевать

те области, в которые так и не смог проникнуть C, и поддержать программистов, которые ранее не входили в сообщество C и C++.

6.5.3.2. Расширенные наборы символов

Поддержка ограниченных наборов символов в языке C++ – всего лишь взгляд в прошлое. Более интересная и трудная задача – поддержка расширенных наборов символов, в которых число символов больше, чем в коде ASCII. Здесь есть два важных вопроса:

- как поддержать манипуляции с расширенными наборами символов?
- как позволить употребление расширенных наборов в исходных текстах программ на C++?

Первую проблему комитет по стандартизации C решил путем определения типа `wchar_t` для представления многобайтных символов. Кроме того, был введен тип `wchat_t[]` для многобайтных строк, а функции семейства `printf` расширили для ввода/вывода типа `wchar_t`. В C++ `wchar_t` стал настоящим типом (а не просто синонимом для другого типа, определенного в C с помощью `typedef`), стандартный класс `wstring` был введен для строк символов типа `wchar_t`, и эти типы были поддержаны средствами потокового ввода/вывода.

Таким образом, поддержан только один тип «широких символов». Если программисту нужно больше типов, скажем, для японских символов, строк японских символов, символов иврита или составленных из них строк, то существует, по крайней мере, два альтернативных подхода. Можно отобразить такие символы в общепринятом, достаточно большом наборе (скажем, Unicode) и написать код, работающий с этим набором, а не с `wchar_t`. Допустимо также определить классы для каждого вида символов и строк, например `Jchar`, `Jstring`, `Hchar` и `Hstring`, и добиться, чтобы каждый из них правильно работал. Такие классы должны генерироваться из общего шаблона. Мой опыт показывает, что оба подхода уместны, хотя любое решение, затрагивающее интернационализацию и разные наборы символов, становится наилучшим поводом для споров и эмоциональных оценок.

Не менее сложен вопрос о том, стоит ли – и если да, то как – разрешить расширенные наборы символов в исходных текстах программ на C++. Разумеется, мне хотелось бы употреблять датские названия яблока, дерева, лодки и острова в программах, оперирующих такими понятиями. Разрешить их в комментариях нетрудно, тем более что комментарии на других языках встречаются довольно часто. Допустить же символы из расширенных наборов в именах идентификаторов сложнее. В целом в программах на C и C++ я бы разрешил употреблять датские, японские, корейские, да и любые другие идентификаторы. Технически здесь нет ничего трудного. Локальный компилятор C, написанный Кеном Томпсоном (Ken Thompson), допускает в идентификаторах любые символы из набора Unicode.

С другой стороны, могут возникнуть проблемы, связанные с переносимостью и способностью понимать чужие программы. Технически проблему переносимости вполне можно решить. Однако английский играет важную роль как международный

язык общения программистов и, очевидно, не стоит отказываться от этого без серьезных причин. Для большинства программистов систематическое использование иврита, китайского, корейского и т.д. будет означать непреодолимый барьер для понимания программ, написанных другими людьми. Даже мой родной датский язык для среднего англоговорящего программиста – проблема.

Пока комитет по C++ не принял окончательного решения по этому вопросу, но думаю, что сделать это придется, и любое возможное решение вызовет ожесточенные споры.

Глава 7. Заинтересованность и использование

Одни языки создаются для решения задачи,
другие – для доказательства той или иной точки зрения.

Деннис Ричи

7.1. Рост интереса к C++

C++ проектировался, чтобы служить интересам пользователей. Это не был академический эксперимент по созданию идеального языка программирования. Не являлся C++ и коммерческим продуктом, имеющим целью обогатить своих создателей. Поэтому у языка должен были быть приверженцы (см. табл. 7.1).

Таблица 7.1

Дата	Примерное число пользователей C++
октябрь 1979 г.	1
октябрь 1980 г.	16
октябрь 1981 г.	38
октябрь 1982 г.	85
октябрь 1983 г.	??+2 (без учета Cpre)
октябрь 1984 г.	??+50 (без учета Cpre)
октябрь 1985 г.	500
октябрь 1986 г.	2000
октябрь 1987 г.	4000
октябрь 1988 г.	15000
октябрь 1989 г.	50000
октябрь 1990 г.	150000
октябрь 1991 г.	400000

За двенадцать лет число пользователей C++ в среднем удваивалось каждые семь с половиной месяцев. И это еще осторожные оценки. Определить точное число пользователей C++ всегда было нелегко. Во-первых, такие реализации, как GNU C++ (G++) и Cfront, передаются в университеты, где на них работают студенты (неизвестно, сколько именно). Во-вторых, многие компании – и поставщики,

и потребители – держат в секрете число пользователей и характер их деятельности. Однако у меня всегда было много друзей, коллег и знакомых, готовых сообщить необходимые цифры при условии ответственного обращения с ними, что и позволило мне сделать довольно верную оценку. Приведенные цифры отражают расчеты, сделанные в то время, и с тех пор не корректировались. Для подтверждения того, что они занижены, я могу сослаться на публичное заявление компании Borland – крупнейшего поставщика компилятора C++ – о том, что к октябрю 1991 г. ими было продано 500 тыс. копий компилятора.

В настоящее время число пользователей C++ выросло настолько, что я уже не могу подсчитать их с достаточной степенью достоверности. Опубликованные цифры говорят, что уже к концу 1992 г. было продано свыше 1 млн. копий компиляторов C++.

7.1.1. Отсутствие маркетинга C++

Больше всего меня удивляет тот факт, что первых пользователей C++ обрел без какого-либо традиционного маркетинга (см. раздел 7.4). Главную роль в распространении языка сыграли различные формы электронных коммуникаций. В первые годы дистрибуция (в основном) и поддержка (полностью) осуществлялись по электронной почте. Довольно быстро сами пользователи организовали электронные конференции, посвященные C++. Благодаря интенсивному использованию сетей, информация о языке, приемах программирования на нем и текущем состоянии распространялась очень широко. Ныне это обычное явление, но в 1981 г. было относительно новым. Полагаю, что C++ был первым из основных языков, который пошел по этому пути.

Позже появились и более традиционные формы маркетинга и обмена информацией. В 1986 г., после того как AT&T выпустила Cfront 1.0, некоторые посредники, прежде всего компания Glockenspiel Джона Кэролана (John Carolan) в Ирландии и ее американский дистрибьютер Oasys (позже – часть Green Hills) начали ограниченную рекламную кампанию. А когда появились независимо разработанные компиляторы C++, такие как Oregon Software C++ и Zortech C++, название C++ уже не сходило с рекламных объявлений (примерно с 1988 г.).

7.1.2. Конференции

В 1987 г. Дэвид Йост (David Yost) из USENIX – Ассоциации пользователей UNIX – взял на себя инициативу организации первой конференции, посвященной исключительно C++. Поскольку Дэвид не был уверен, много ли народу проявит интерес к мероприятию, конференция была названа «семинаром». По секрету Йост сказал мне, что, если «зарегистрируется недостаточно участников (менее 30 человек), нам придется свернуть лавочку». На роль координатора программы был выбран Кит Горлен из Национального института здравоохранения. Он собирал электронные адреса интересных проектов, о которых мы были наслышаны, и рассылал отдельным специалистам предложения выступить с докладом. В конце концов было принято 30 докладов, и в ноябре 1987 г. на семинар в Санта-Фе собрались 214 человек.

Состоявшееся мероприятие стало образцом для последующих конференций. В Санта-Фе прозвучали доклады по приложениям, методам программирования и обучения, библиотекам, технике реализации компиляторов, идеи по улучшению языка. Удивляет, что на конференции, организованной USENIX, были сообщения по работе C++ на платформе Apple Macintosh, OS/2, компьютере Connection и по реализации отличных от UNIX операционных систем (например, CLAM [Call, 1987] и Choices [Campbell, 1987]). О библиотеках NIH [Gorlen, 1987] и Interviews [Linton, 1987] также заговорили в Санта-Фе. Там была продемонстрирована ранняя версия разработки, позже ставшей Cfront 2.0 [Stroustrup, 1987c].

Предполагалось, что конференция в Санта-Фе будет семинаром и, несмотря на наличие 200 участников, так оно и случилось. С большой интенсивностью проходили дискуссии. Однако было ясно, что на следующую конференцию съедутся в основном начинающие пользователи и люди, пытающиеся понять, что представляет собой C++. Поэтому не будет глубокой и открытой полемики и ее место займут вопросы обучения и коммерческого распространения языка. По предложению Эндрю Кенига конференцию USENIX по C++ в Денвере было решено назвать «семинаром разработчиков». После конференции докладчики, разработчики компиляторов и др. отправились из Денвера в Эстес Парк на однодневную «живую» дискуссию. Здесь с таким воодушевлением были приняты идеи о статических (см. раздел 13.4) и константных (см. раздел 13.3) функциях-членах, что я решил включить их в версию Cfront 2.0, выпуск которой все еще откладывался AT&T по внутренним причинам. По моей настоятельной просьбе Майк Миллер (Mike Miller) представил доклад [Miller,1988], который положил начало серьезному публичному обсуждению обработки исключений в C++.

Помимо конференций USENIX теперь есть много коммерческих и полукоммерческих конференций, посвященных C и C++, именно C++, а также объектно-ориентированному программированию, в том числе и с применением C++. В Европе конференции организует Ассоциация пользователей C и C++ (ACCU).

7.1.3. Журналы и книги

К середине 1992 г. только на английском языке вышло порядка 100 книг по C++, не считая переводов и оригинальных книг на китайском, датском, французском, немецком, итальянском, японском, русском и других языках. Разумеется, они неодинаковы по качеству. Приятно, что мои собственные книги переведены уже на десять языков.

Первый посвященный C++ журнал – *The C++ Report*, под редакцией Роба Мюррея (Rob Murray) – начал выходить в январе 1989 г. Более объемное, красочное издание *The C++ Journal*, под редакцией Ливлина Сингха (Livleen Singh), появилось весной 1991 г. и стало выходить ежеквартально. Есть еще несколько информационных бюллетеней, издаваемых фирмами-производителями инструментальных средств, а в таких журналах, как *Computer Language*, *The Journal of Object-Oriented Programming* (*JOOP*), *Dr. Dobb's Journal*, *The C User's Journal* и *.EXE* регулярно публикуются статьи или колонки, посвященные C++. Состав изданий, на страницах которых обсуждаются вопросы, относящиеся к C++, и политика их редколлегий меняются довольно быстро.

В электронных конференциях и досках объявлений, таких как comp.lang.c++ в USENET и c.plus.plus в BIX, за несколько лет появились десятки тысяч новых сообщений. Если следить за всем, что пишется о C++, то не хватит и целого рабочего дня.

7.1.4. Компиляторы

Конференция в Санта-Фе (см. раздел 7.1.2) знаменовала начало второй волны реализаций C++. Стив Дьюхерст описал архитектуру компилятора, который он вместе с коллегами разрабатывал в отделении AT&T в Саммите. Майк Болл (Mike Ball) рассказал о некоторых идеях, положенных в основу компилятора TauMetric C++, созданный совместно со Стивом Клэмиджем в Сан-Диего (чаще его называют компилятором Oregon Software C++). Майк Тиман (Mike Tiemann) провел яркую и интересную презентацию с целью доказать, что его GNU C++ сможет делать почти все, так что другие авторы останутся без работы. Новый компилятор C++ от AT&T так никогда и не увидел света, GNU C++ версии 1.13 вышел в декабре 1987 г., а TauMetric C++ – в январе 1988 г.

Вплоть до июня 1988 г. все компиляторы C++ на персональных компьютерах были перенесенными вариантами Cfront. Затем компания Zortech начала поставки компилятора, созданного Уолтером Брайтом в Сиэтле. Появление Zortech C++ открыло реальный доступ к C++ пользователям ПК. Консервативно настроенные программисты решили подождать выхода компилятора Borland C++ в мае 1990 г. или даже выхода Microsoft C++ в марте 1992 г. В феврале 1992 г. компания DEC выпустила свой первый независимо разработанный компилятор, а в мае 1992 г. то же самое сделала IBM. Сейчас на рынке есть около дюжины компиляторов, созданных разными фирмами.

Кроме того, Cfront перенесен практически на все платформы. Большой вклад в это дело внесли такие фирмы, как Sun, Hewlett-Packard, Centerline, ParcPlace, Glockenspiel и Comeau Computing.

7.1.5. Инструментальные средства и среды программирования

C++ проектировался так, чтобы им можно было пользоваться в среде, почти лишенной каких бы то ни было инструментальных средств. Отчасти это было вызвано необходимостью: сначала не хватало ресурсов, позже – финансовых средств. Но это было и сознательным решением, направленным на упрощение реализации и переноса на другие платформы.

Теперь начинают появляться такие среды программирования на C++, которые сопоставимы с разработанными для других объектно-ориентированных языков. Например, ObjectWorks C++ от компании ParcPlace – по сути дела, среда разработки Smalltalk, адаптированная под C++, а Centerline C++ (бывший Saber C++) – интерпретируемая, созданная под влиянием среды для Interlisp. Это дает программистам на C++ возможность пользоваться более дорогими, но и повышающими производительность труда средами, которые раньше были доступны только для других языков. В настоящее время для C++ существует довольно много таких сред. Большинство реализаций C++ для ПК – это компиляторы, интегрированные с редакторами, инструментальными средствами, файловыми системами, стандартными

библиотеками и т.д. MacApp и Mac MPW – соответствующие аналоги на платформе Apple Mac. ET++ – свободно распространяемый компилятор в духе MacApp. Другие примеры – Energize от компании Lucid и Softbench от Hewlett-Packard.

Эти среды, хотя и более сложные, чем использовавшиеся ранее для C, являются лишь примитивными предшественниками гораздо более развитых систем. Современные инструментальные средства обычно акцентируют внимание на синтаксических аспектах языка, на поведении во время исполнения и на текстовом представлении программы. Но чтобы в полной мере проявились достоинства C++, в среде должна использоваться вся система типов. Нужно также, чтобы информация, доступная во время исполнения, ассоциировалась в среде со статической структурой программы. Естественно, такая среда должна масштабироваться под очень большие программы (порядка 500 тыс. строк на C++), для которых наличие инструментальных средств наиболее важно.

Несколько подобных систем находятся в процессе разработки, я принимаю активное участие в одном из таких проектов, [Murray,1992], [Koenig, 1992]. Однако уместным будет предупреждение. С помощью среды программирования поставщик может заманить пользователя в замкнутый мирок средств, библиотек, инструментов и приемов работы, которые с большим трудом переносятся на другие системы. Поэтому у программиста есть риск попасть в сильную зависимость от одного поставщика и лишить себя возможности работать с другими машинными архитектурами, библиотеками, базами данных и т.д., если поставщик не поддерживает их. Одной из главных моих целей при проектировании C++ было дать пользователям свободу выбора, но среду программирования можно спроектировать так, что эта цель не будет достигнута [Stroustrup, 1987d]: «Следует приложить усилия к тому, чтобы исходный текст программы можно было без больших затрат переносить между такими средами».

Точно так же, как я не питаю надежды на создание единой большой стандартной библиотеки, не вижу и возможности для создания стандартной среды разработки для C++ [Stroustrup, 1987d]:

> «По крайней мере, для C++ всегда будет существовать несколько разных сред разработки и исполнения, радикально отличающихся друг от друга. Было бы абсурдом ожидать общей среды исполнения, скажем, для Intel 80286 или Cray XMP и столь же наивно надеяться на создание общей среды разработки, пригодной как для исследователя-одиночки, так и для коллектива из 200 программистов, работающих над крупным проектом. Однако ясно и то, что необходимо пользоваться единым стилем там, где это имеет смысл».

Наличие множества библиотек, сред программирования и исполнения важно для поддержки широкого диапазона приложений на C++. С таким приоритетом C++ разрабатывался с 1987 г. и даже раньше. Объяснение данному подходу – оценка C++ как языка программирования общего назначения (см. разделы 1.1 и 4.2).

7.2. Преподавание и изучение C++

На эволюцию языка оказали большое влияние методы его изучения. Поэтому разобраться в самом C++, а также в причинах его быстрого развития сложно без правильного понимания того, как следует преподавать и изучать данный язык.

Я долго размышлял, как неопытные пользователи будут изучать и применять C++, много времени преподавал сам, в основном ученым мужам. Все это существенно повлияло на проектирование языка с первых дней его создания. Также облик C++ во многом определили все успехи и неудачи, сопутствующие изложению собственных идей, и изучение программ, написанных теми, кого обучали я и мои коллеги.

По прошествии нескольких лет к указанному вопросу выработался особый подход: главный упор в изучении C++ делался на изложение основных понятий, а затем на взаимосвязь этих понятий друг с другом и с наиболее важными средствами языка. Подробное изучение отдельных возможностей предлагалось по мере углубления знаний и по мере необходимости овладения данным материалом. Там, где этот подход не срабатывал, язык приходилось модифицировать. В результате он развивался в направлении лучшей поддержки проектирования.

Люди, с которыми я работал и которых обучал, как правило, были профессиональными программистами и проектировщиками, вынужденными учиться без отрыва от работы. Отсюда и желание спроектировать C++ так, чтобы его можно было осваивать постепенно, чтобы возможности языка могли изучаться последовательно, чтобы получаемая от овладения языком польза была почти пропорциональна затраченным усилиям.

Полагаю, что практические соображения, лежащие в основе многих дискуссий о языках программирования, языковых средствах, стилях программирования и т.д., больше касаются вопросов обучения языку, нежели языка как такового. Для многих ключевая проблема может быть сформулирована так:

Если у меня мало времени для изучения новых методов и концепций, как мне лучше всего приступить к изучению C++?

В начале 1993 г. я ответил на этот вопрос в конференции comp.lang.c++:

«Ясно, что для использования C++ наилучшим образом в произвольно взятой ситуации необходимо глубокое понимание многих концепций и методов, но это достигается только годами изучения и экспериментирования. Новичку (в C++, а не в программировании вообще) вряд ли поможет, если ему посоветуют сначала досконально изучить C, Smalltalk, CLOS, Pascal, ML, Eiffel, ассемблер, объектно-ориентированные базы данных, методы верификации программ, а затем уже брать уроки по овладению C++. Все эти предметы достойны изучения, но программисты-практики (и студенты) не могут на несколько лет бросить работу ради систематического изучения языков и методов программирования.

С другой стороны, многие новички понимают: недостаток знаний бывает опасен, поэтому следует иметь хоть какие-то гарантии того, что небольшое время, которое можно уделить учебе перед началом нового проекта (или уже в ходе работы над ним), окупится, а не окажется потраченным напрасно. И еще хотелось бы быть уверенными, что те вопросы, которые удастся понять сразу же, будут частью пути к более полному пониманию, а не просто изолированным островком знания.

Разумеется, таким критериям удовлетворяет не один-единственный подход. Выбор способа обучения зависит от исходной подготовки человека, его неотложных потребностей и времени, которым он располагает. Полагаю, что многие преподаватели, авторы учебников и участники электронных конференций недооценивают важность данного факта, ведь кажется, что намного эффективнее и проще обучать большие группы, а не возиться с отдельным студентом.

Следует задуматься над следующими вопросами общего характера:

- Я не знаю ни C, ни C++. Нужно ли начать с изучения C?
- Я хочу заниматься объектно-ориентированным программированием. Нужно ли мне учить Smalltalk, прежде чем я возьмусь за C++?
- Надо ли приступать к изучению C++ как языка объектно-ориентированного программирования или как улучшенного C?
- Сколько времени нужно для изучения C++?

Я не претендую на то, что мои ответы на данные вопросы единственно правильны, но они базируются на многолетнем опыте обучения C++ и другим языкам, на чтении кратких курсов по проектированию и программированию с использованием C++, на консультировании по вопросам внедрения и применения C++ и т.д. Итак, по порядку:

Я не знаю ни C, ни C++. Нужно ли начать с изучения C? Нет. Начинайте прямо с изучения C++. То подмножество C++, которое составляет C, проще для изучения и использования, чем сам C. C++ предоставляет лучшие гарантии, чем C, за счет более строгой проверки типов. Кроме того, в C++ есть немало второстепенных возможностей, например оператор `new`, более удобных и менее подверженных ошибкам, чем их аналоги в C. Чтобы успешно применять C, нужно знать много приемов и методов, которые совсем или почти не нужны в C++. В хороших учебниках по C часто (и не без причины) упор делается именно на приемах, без владения которыми не справиться с большим проектом. Наоборот, в удачных учебниках по C++ основное внимание уделяется методам и средствам C++, которые ведут к абстрагированию данных и объектно-ориентированному программированию. Зная конструкции C++, совсем просто выучить их низкоуровневые аналоги в C, если это необходимо.

Мой совет таков: для изучения C возьмите работу Кернигана и Ричи [Kernighan, 1988], для изучения C++ – второе издание моей книги «Язык программирования C++» [2nd]. Достоинство этих книг в том, что они сочетают введение в средства и приемы работы с языком с полным справочным руководством. В двух указанных изданиях описывается язык как таковой, а не его конкретная реализация, и не делается попытка описать библиотеки, поставляемые с определенным компилятором.

Также всегда разумно получать информацию хотя бы из двух источников, чтобы избежать предвзятости и необъективности.

Я хочу заниматься объектно-ориентированным программированием. Нужно ли мне учить Smalltalk прежде, чем я возьмусь за C++? Нет. Если вы планируете изучить C++, то и изучайте C++. В каждом из таких языков, как C++, Smalltalk, Simula, CLOS, Eiffel, по-своему трактуются ключевые концепции абстрагирования и наследования, и в каждом из них методики проектирования поддержаны несколько по-разному. При изучении Smalltalk вы можете получить ценные уроки, но при этом не научитесь писать программы на C++. Если у вас нет времени на изучение и осмысление обоих языков, то освоение Smalltalk лишь помешает правильному проектированию программ на C++.

Естественно, идеально было бы выучить и C++, и Smalltalk и свободно пользоваться двумя источниками, но люди, у которых не хватает времени глубоко усвоить новые идеи, нередко приходят к «Smalltalk на C++», то есть применяют такие приемы проектирования на Smalltalk, которые не приспособлены для C++.

Аргументируя изучение Smalltalk, часто говорят, что это «чистый» язык, который заставляет думать и программировать «в объектно-ориентированном стиле». Не буду вступать в споры о «чистоте», отмечу лишь, что, на мой взгляд, язык программирования общего назначения должен поддерживать более одного стиля (парадигмы) программирования.

Дело в том, что стили, которые подходят для Smalltalk и хорошо в нем поддержаны, необязательно годятся и для C++. В частности, слепое подражание Smalltalk приводит к неэффективным,

неудачным и трудным для сопровождения программам на C++. Это происходит потому, что для проектирования хорошей программы на C++ необходимо пользоваться преимуществами статической системы контроля типов, а не сражаться с ней. Smalltalk же поддерживает динамические типы (и только их), а при переводе такого проектирования на C++ приходится постоянно прибегать к небезопасным и некрасивым приведениям типов.

Думается, что большая часть приведений типов в программах на C++ – признак неудачного проектирования. Опираясь на свой опыт, отмечу, что программисты, давно пишущие на языке C, а также те, кто пришли к объектно-ориентированному программированию через Smalltalk, чаще всего пользуются приведениями типов, которых при другом проектировании легко можно было бы избежать.

Кроме того, Smalltalk учит видеть в наследовании единственное или, по крайней мере, основное средство структурирования программ и организации классов в виде иерархий с одним общим корнем. В C++ классы – это типы, а наследование ни в коей мере не является единственным средством структурирования. В частности, основным средством для построения контейнерных классов являются шаблоны.

Я также с большим подозрением отношусь к призывам заставить писать в объектно-ориентированном стиле. Если человек не хочет чему-то учиться, то он и не научится. Если вы не можете продемонстрировать принципы, стоящие за абстрагированием данных и объектно-ориентированным программированием, то получите лишь нелепые и неправильные примеры использования тех языковых средств, которые призваны поддержать эти концепции. В равной степени это относится к C++, Smalltalk и любому другому языку.

Более подробное обсуждение взаимосвязей между разными средствами C++ и проектированием программ см. в книге «Язык программирования C++ (второе издание)» [2nd].

Надо ли приступать к изучению C++ как языка объектно-ориентированного программирования или как улучшенного C? Это зависит от обстоятельств, от того, зачем вы хотите овладеть C++. Именно ответ на данный вопрос и должен определить ваш подход к изучению языка. Судя по моему опыту, лучше всего изучать C++ «снизу вверх», то есть сначала освоить те средства, которые предоставлены для традиционного процедурного программирования (улучшенный C), затем переходить к средствам абстрагирования данных и, наконец, к использованию иерархий классов как способа организации взаимосвязанных классов.

На мой взгляд, проскочить первые этапы галопом опасно, поскольку есть вероятность упустить какую-то ключевую концепцию.

Например, опытный программист на C может решить, что подмножество, составляющее улучшенный C, ему хорошо знакомо и, читая учебник, пропустит страниц сто, где оно описывается. При этом он пройдет мимо возможности перегрузки функций, мимо разницы между присваиванием и инициализацией, мимо использования оператора `new` для выделения памяти, мимо ссылок и некоторых других не столь важных возможностей, а позже, когда придет время для более сложных вопросов, эти упущения не раз дадут о себе знать. Если вы действительно знакомы с концепциями, применяемыми в C-подмножестве, то на чтение сотни страниц уйдет всего пара часов, а некоторые встретившиеся при этом детали окажутся интересными и полезными. Если же нет, то потратить время на чтение просто необходимо.

Есть мнение, что при таком «постепенном подходе» программист так всю жизнь и будет писать в стиле C. Конечно, это не исключено, но вероятность такого результата гораздо меньше, чем пытаются представить поборники «чистых» языков и принуждения при обучении программированию. Главное – осознать, что успешное применение C++ как языка для абстрагирования данных и объектно-ориентированного программирования требует понимания нескольких новых концепций, которые не имеют прямых аналогов в C или Pascal.

C++ – не просто новый синтаксис для выражения старых идей. По крайней мере, для большинства программистов это не так. Отсюда следует, что требуется образование, а не только

обучение. Новые концепции надо изучать и осваивать на практике. Испытанные приемы работы придется пересмотреть. Не стоит бросаться делать все «старым добрым способом», нужно искать новые пути. Допустимо утверждение, что затраты времени и сил на изучение ключевых концепций абстрагирования данных и методов объектно-ориентированного программирования начинают давать отдачу не в отдаленной перспективе, а уже через 3–12 месяцев. Использование C++ может принести пользу и без этого, но она будет тем больше, чем лучше усвоены основные концепции, и я не понимаю, зачем вообще переходить на C++ человеку, который не готов на начальном этапе потрудиться как следует.

Приступая к изучению C++ впервые или после долгого перерыва, изыщите возможность прочитать по теме хорошую книжку или несколько удачно подобранных статей (они в избытке содержатся в журналах «The C++ Report» и «The C++ Journal»). Также стоит посмотреть на исходные тексты какой-нибудь большой библиотеки и поразмыслить над применяемыми там приемами. Это не повредит и тем, кто уже некоторое время работает с C++. С момента появления языка он и относящиеся к нему методы программирования и проектирования сильно изменились. Чтобы убедиться в этом, достаточно бегло сравнить первое и второе издания книги «Язык программирования C++».

Сколько времени нужно для изучения C++? Это зависит от разных факторов: от вашего опыта и от того, что вы понимаете под словами «изучение C++». Для освоения синтаксиса и основ программирования на C++ как на улучшенном C, а также для определения и использования простых классов большинству программистов достаточно одной-двух недель. Это легкая часть работы. Самое трудное, но и самое интересное, – овладение новыми методами проектирования и программирования. Большинство опытных программистов, с которыми я беседовал, говорят, что для глубокого освоения C++ и тех способов абстрагирования данных и объектно-ориентированного программирования, которые поддерживаются языком, им потребовалось от полугода до полутора лет. При этом учились они прямо на рабочем месте и продолжали писать программы, но в этот период использовали C++ «осторожно». Если у вас есть возможность посвятить некоторое время исключительно изучению C++, то вы освоитесь быстрее, но у вас не будет возможности применить новые идеи к реальным проектам, а без этого обретенная уверенность может оказаться обманчивой. Объектно-ориентированные проектирование и программирование – это практические, а не теоретические дисциплины. Если их не применять на практике или применять только к «игрушечным» примерам, то могут появиться негативные последствия.

Замечу, что овладение языком C++ – это прежде всего изучение методов проектирования и программирования, а не деталей. Рекомендую сначала проштудировать хороший учебник по C++, затем какую-нибудь книгу по проектированию, например [Booch, 1991], где есть прекрасные содержательные примеры на пяти языках (Ada, CLOS, C++, Smalltalk и Object Pascal).[1] Больше всего в этой книге мне нравится изложение концепций проектирования и главы, где подобраны примеры.

Взгляд на язык с точки зрения проектировщика резко контрастирует со скрупулезным изучением деталей определения языка, ведущимся обычно по книге ARM, в которой содержится много полезной информации, но ничего не сказано о том, как писать программы на C++.

При изучении C++ важно все время помнить о ключевых концепциях, чтобы не запутаться в технических деталях. При этом условии и изучение, и использование языка станут приятным и продуктивным делом. Даже поверхностное знание C++ может оказаться преимуществом по сравнению с владением C, а уж если приложить усилия к пониманию абстракций данных и объектно-ориентированных методов, то пользы будет намного больше».

Надо учитывать и современное состояние инструментальных средств и библиотек. Имея более защищенную среду, например включающую по умолчанию

[1] Во втором издании книги [Booch, 1993] используется только C++.

всесторонние проверки во время исполнения, и небольшую четко определенную библиотеку базовых классов, в рискованные эксперименты с C++ можно пуститься и несколько раньше. Это позволит сместить акценты от языковых средств C++ к тем методам программирования и проектирования, которые он поддерживает.

Не стоит с головой погружаться в синтаксис и несущественные технические детали языка, как это любят делать программисты с большим стажем. Нередко такой интерес – не что иное, как обычное нежелание изучать новые методы программирования.

В каждой учебной группе и в каждом проекте обязательно находится кто-то, кто не может поверить, что C++ может быть достаточно эффективен, и потому старается не выходить за рамки испытанного C-подмножества. Только зная о конкретных результатах измерения производительности отдельных средств C++ и написанных на нем системах (например, [Russo, 1988], [Russo, 1990], [Keffer, 1992]), можно преодолеть стойкое убеждение, что средства более удобные, чем в C, обязательно должны требовать больших затрат.

Пользователи, убежденные в том, что эффективность – аспект несущественный, также есть в каждой учебной или проектной группе. Они проектируют до такой степени обобщенные системы, что их медленная работа заметна даже на самом современном оборудовании. К сожалению, это обычно не видно в программах, которые пишутся во время обучения, поэтому проблемы из-за пренебрежительного отношения к ним проявляются лишь позже, уже в реальных проектах.

7.3. Пользователи и приложения

Один аспект применения C++ никак не давал мне покоя: слишком многие приложения, с ним связанные, были довольно-таки необычными. Конечно, это могло быть потому, что такие приложения интереснее обсуждать, но я склонен подозревать, что причина более серьезна. Сила C++ в гибкости, эффективности и переносимости, поэтому его можно применять в проектах, где используются необычное оборудование, нестандартная операционная среда или требуется интерфейс с другими языками. Пример такого проекта – система для Уолл Стрит, которая должна была работать на мейнфреймах совместно с программами, написанными на COBOL, на рабочих станциях вместе с Fortran-программами, на ПК вместе с модулями на C и в сети, которая связывала все это оборудование.

Данный пример убедительно свидетельствует о том, что с помощью C++ можно разрабатывать передовое промышленное материальное обеспечение. В этом отношении он заметно отличается от языков, использующихся в основном для экспериментов – промышленных, академических – или для обучения. Конечно, C++ находит широкое применение как в экспериментальных и исследовательских работах, так и в образовательных целях. Но его роль в промышленных программах обычно является определяющим фактором при выборе проектного решения.

7.3.1. Первые пользователи

Вначале мирок C with Classes и C++ был совсем небольшим. Его характерной особенностью было изобилие личных контактов, которые способствовали обмену

идеями и быстрому решению возникающих проблем. Тогда я мог лично разбираться в проблемах пользователей, исправлять ошибки в Cfront и библиотеках, а иногда даже вносить изменения в язык. Я уже упоминал в разделах 2.14 и 3.3.4, что в то время пользователями были в основном исследователи и разработчики из Bell Labs.

7.3.2. Сферы применения C++

К сожалению, многие пользователи не документируют свой опыт, а некоторые организации считают его чуть ли не государственной тайной. Поэтому до программистов и администраторов доходят не только правдивые данные о языках и методах программирования, но и много мифов, неточной, а иногда и откровенно ложной информации. Отсюда постоянное дублирование усилий и повторение известных ошибок. Цель этого раздела – показать несколько областей, в которых использовался C++, и побудить разработчиков документировать свой опыт, чтобы им могло воспользоваться все сообщество пользователей. В каждой из упоминаемых областей трудились, по крайней мере, два человека в течение не менее двух лет. Самый большой из известных мне документированных проектов состоит из 5 млн строк на C++ и разрабатывался коллективом из 200 человек свыше семи лет.

Итак, области использования C++: анимация, автономные погружаемые аппараты, биллинговые системы (в сфере телекоммуникаций), управление кегельбаном, трассировка монтажа печатных схем, CAD/CAM, моделирование химических процессов, управление агентством по продаже автомобилей, CASE, компиляторы, матобеспечение для пультов управления, моделирование циклотрона, системы баз данных, отладчики, системы поддержки принятия решений, обработка цифровых фотографий, электронная почта, управление вышивальной машиной, экспертные системы, автоматизация производства, финансовая отчетность, телеметрия полетов, обмен валют, перевод денежных сумм, генеалогические исследования, управление газоперекачивающими станциями и биллинговые системы для них, графика, описание аппаратуры, управление больничными архивами, управление промышленным роботом, моделирование набора машинных команд, интерактивные мультимедийные средства, магнитогидродинамика, обработка медицинских образов, медицинский мониторинг, управление ракетами, управление ипотечной компанией, сетевые протоколы, системы управления и администрирования сетей, мониторинг сетей, операционные системы (реального времени, распределенные, рабочих станций, мейнфреймов, объектно-ориентированные), среды программирования, страховой бизнес, моделирование ударных волн, управление бойнями, программное обеспечение коммутаторов, инструментальные средства для тестирования, трейдерские системы, обработка транзакций, системы передачи данных, управление транспортным парком, пользовательские интерфейсы, видеоигры, виртуальная реальность.

7.4. Коммерческая конкуренция

Среди программистов, в прессе, на конференциях и в электронных средствах всегда велись (да и сейчас ведутся) споры о том, какой язык лучший и кто победит

в борьбе за пользователей. Я считаю такие дебаты бессмысленными и свидетельствующими о недостатке информации, но от этого они не становятся менее значимыми для программиста, руководителя или профессора, которые должны выбрать язык программирования для следующих проектов. Люди спорят о языках программирования с поистине религиозным ожесточением и зачастую полагают, что выбор языка – самая важная часть работы. Поэтому принятие решения об использовании того или иного языка может стать серьезной проблемой как для отдельного программиста, так и для целой организации. Этот вопрос редко решается удовлетворительно, а еще реже сделанный выбор беспристрастно документируется, чтобы другие могли извлечь из этого пользу. Ко всему прочему, в организациях по разным причинам считают, что управление разработкой, сделанной на нескольких языках, – слишком трудоемкий процесс. Проблема усугубляется и тем, что проектировщики языков и разработчики компиляторов не считают важной задачу взаимодействия между выбранным языком и всеми остальными.

Еще хуже то, что программисты-практики вынуждены оценивать язык как инструмент, а не как интеллектуальное достижение. Поэтому они сравнивают компиляторы, инструментальные средства, различные оценки производительности, организацию технической поддержки, библиотеки, возможности обучения (книги, журналы, конференции, учебные курсы, помощь консультантов) и т.д. на текущем этапе и в краткосрочной перспективе. Заглядывать далеко в будущее обычно слишком опасно из-за огромного объема коммерческой рекламы и потенциальной возможности принять желаемое за действительное.

Поначалу многие считали язык Modula-2 конкурентом C++. Однако до выхода первой коммерческой версии C++ в 1985 г. его вообще вряд ли можно было рассматривать как конкурента, а к тому времени C пользовался в США куда большим успехом, чем Modula-2. Позже вошли в моду рассуждения о том, какой язык – C++ или Objective C [Cox, 1986] – следует считать настоящим объектно-ориентированным C. Для многих организаций, которые могли бы воспользоваться C++, возможной альтернативой был язык Ada. Кроме того, для приложений, где не требовалось ни прямого доступа к оборудованию и операционной системе, ни максимальной производительности, нередко предлагались Smalltalk [Goldberg, 1983] и объектно-ориентированный вариант Lisp [Kiczales, 1992]. Позже проводился сравнительный анализ пригодности для некоторых приложений C++ и Eiffel [Meyer, 1988] или C++ и Modula-3 [Nelson, 1991].

7.4.1. Традиционные языки

По-моему, основным конкурентом C++ всегда был C. Сегодня C++ является наиболее распространенным объектно-ориентированным языком только потому, что это единственный язык, способный соперничать с C на его собственном поле и одновременно предоставлять значительные усовершенствования. В C++ есть путь перехода от C к стилям проектирования и реализации систем, которые основаны на более прямом отображении понятий предметной области на концепции языка (это обычно и называется абстракцией данных или объектно-ориентированным программированием). С другой стороны, многие организации, рассматривающие вопрос о новом языке программирования, традиционно использовали

какой-нибудь язык собственной разработки (обычно вариант Pascal) или Fortran. Но ведь если не касаться серьезных научных расчетов, то при поверхностном рассмотрении эти языки можно считать эквивалентными C, сравнивая их с C++.

Конкуренция с языком Fortran более жесткая. У него есть преданные поклонники, которые – как и немалая часть программистов на C – не очень интересуются прочими языками программирования и тонкостями информатики. Им просто нужно делать свою работу. Многие компиляторы Fortran генерируют эффективный код для высокопроизводительных компьютеров, а для пользователей это имеет первостепенное значение. В некоторых задачах C++ удавалось успешно соперничать с Fortran. Но и здесь C++ обязательно будет конкурентоспособен: компиляторы проводят более эффективную оптимизацию, например, при встраивании вызовов функции, а к тщательно разработанным библиотекам, написанным на Fortran, можно напрямую обращаться из C++-программ.

C++ все чаще используется для расчетных и научных задач [Forslund, 1990], [Budge, 1992], [Barton, 1994]. Это породило ряд предложений о расширениях. Как правило, в основе лежал опыт Fortran, поэтому судьба этих предложений сложилась не очень удачно. Я надеюсь, что акцентирование внимания на высокоуровневых свойствах, с одной стороны, и методы оптимизации – с другой, в конечном итоге сослужат научному сообществу большую пользу, чем простое добавление низкоуровневых средств из Fortran. Убежден, что у C++ есть потенциал для применения в научных расчетах, и хотел бы обеспечить в этой области большую поддержку, чем имеющаяся сейчас.

7.4.2. Современные языки

С 1984 по 1989 гг. в соревновании с языками, поддерживающими механизмы абстракции, с маркетинговых позиций C++ явно проигрывал. В частности, маркетинговый бюджет AT&T в это время чаще всего был на нуле, а общие ее расходы на рекламу C++ составили около 3 тыс. долларов. Из них треть была потрачена на рассылку компаниям, приобретшим лицензию на UNIX, информационного письма о том, что C++ существует и продается. Разумеется, никакого эффекта это не возымело. Еще 2 тыс. долларов ушло на организацию приема для участников первой конференции по C++ 1987 г. в Санта-Фе. Это тоже слабо поспособствовало известности C++, но, по крайней мере, нам устроили вечеринку. На первой конференции OOPSLA сотрудники AT&T арендовали самый маленький стенд. Стенд обслуживался добровольцами, которые пользовались обычными досками вместо компьютеров и делали копии технической документации на бланках для регистрации посетителей. Мы хотели распространять хоть какие-то рекламные сувениры с символикой C++, но средств не было.

Вплоть до сегодняшнего дня причастность AT&T к C++ выражается в традиционной для Bell Labs политике подталкивания разработчиков и исследователей к написанию статей, научным спорам и участию в конференциях, а не в обдуманном продвижении C++ на рынок. Внутри компании C++ всегда был неким «движением снизу» без финансовой и административной поддержки. Естественно, тот факт, что C++ появился на свет в подразделении AT&T, пошел ему на пользу, но такая помощь была заработана тяжкой борьбой за выживание в большой компании.

По сравнению с другими языками, сильные стороны C++ – способность работать в традиционной компьютерной среде, эффективность по быстродействию и памяти, гибкость концепции классов, низкая цена, а также то, что язык не является собственностью какой-то одной компании. Слабости – некоторые неудачные конструкции, унаследованные от C, отсутствие эффектных нововведений, например встроенной поддержки баз данных, отсутствие изощренных сред разработки (только недавно появились среды, наличие которых программисты на Smalltalk или Lisp считают само собой разумеющимся, см. раздел 7.1.5), недостаточное количество стандартных библиотек (лишь в последнее время получили широкое распространение большие библиотеки для C++, но они не являются стандартными, см. раздел 8.4) и неспособность продавцов вкладывать в рекламу такие же средства, какие вкладывают более богатые конкуренты.

В отличие от традиционных языков, C++ имеет механизм наследования, что является большим преимуществом. По сравнению же с языками, поддерживающими наследование, в C++ предусмотрен статический контроль типов. Из всех языков только Eiffel и Modula-3 в какой-то мере похожи на C++. В языке Ada9X, идущем на смену Ada, тоже есть наследование.

C++ проектировался как язык для системного программирования и разработки приложений, в которых есть большой системный компонент. Эта область была хорошо знакома мне и моим коллегам. Решение не жертвовать сильными сторонами C++ в угоду большей популярности оказалось главной причиной успеха языка. Только время покажет, не помешало ли это завоеванию еще большей аудитории. И если вдруг окажется, что помешало, я бы не стал считать это трагедией, поскольку не разделяю точку зрения, что один язык должен быть всем и для всех. C++ и так хорошо служит интересам тех, на кого был первоначально рассчитан. При этом я полагаю, что с появлением библиотек привлекательность C++ может возрасти многократно.

7.4.3. Как выдержать конкуренцию

Люди часто удивляются, почему AT&T позволяет другим компаниям реализовывать C++. Это свидетельствует о незнании законов и непонимании целей AT&T. После опубликования справочного руководства по C++ [Stroustrup, 1984] любой желающий может создать компилятор для языка. Более того, AT&T не только не препятствовала приходу новых людей на растущий рынок компиляторов C++, инструментальных средств, образовательных услуг и т.д., но всячески приветствовала и поощряла данную тенденцию. Многие просто упускают из виду, что AT&T в гораздо большей степени является потребителем программных продуктов, нежели их производителем. Поэтому компании очень выгодны труды «конкурентов» на поле C++.

Язык, принадлежащий одной компании, никогда не стал бы пользоваться таким успехом, как C++. Создание компилятора, инструментария, библиотек, образовательной инфраструктуры – все это просто не под силу одной, сколь угодно крупной организации. Кроме того, запатентованный язык неизбежно отражал бы политику и курс компании, а это помешало бы ему выжить в открытом и свободном мире. Думаю, что язык, которому удалось выдержать испытания в стенах Bell

Labs и устоять против стихии рынка, не может быть совсем уж плохим, даже если он не подчиняется диктату академической моды.

Само собой, стратегию определяют не какие-то безликие компании. Ее формулируют люди, и согласие достигается именно между ними. Стратегия развития C++ идет от преобладавших в Bell Labs и других подразделениях AT&T идей. Язык никогда бы не стал популярным, если бы концепция общедоступного программного обеспечения не нашла поддержки.

Конечно, согласие достигалось не всегда просто, иногда не достигалось и вовсе. Мне рассказывали, как однажды некий менеджер высказал идею засекретить C++ как конкурентное преимущество AT&T. Его разубедил другой менеджер, добавив: «В любом случае идея абсурдна, поскольку Бьерн уже раздал 700 копий справочного руководства». Разумеется, все экземпляры были отправлены после получения надлежащего разрешения и одобрения моим начальством.

Важным фактором, свидетельствующим одновременно и за, и против C++, было согласие сообщества принимать язык со всеми его недостатками. Такая открытость заставляла нас быть честными, работать над улучшением языка и его инструментальных средств и не давала пользователям питать нереалистичные надежды.

Некоторые удивляются, почему, говоря о «коммерческой конкуренции», я не называю конкретных языковых средств, инструментов, дат выхода версий, не упоминаю маркетинговых стратегий, обзоров или коммерческих организаций. Отчасти потому, что я уже не раз обжигался в спорах с фанатичными поборниками разных языков и на маркетинговых кампаниях, где главенствуют циники. С другой стороны, мой подход к выбору языка основан на вере в то, что конкретные языковые средства и инструменты в широком смысле не так уж важны и являются лишь мишенью псевдонаучных перепалок. Тут действует некий вариант закона больших чисел.

На любом из упоминавшихся выше языков, в том числе на C, можно реализовать легкую часть проекта. На любом из них эту же часть проекта можно реализовать более изящно, чем на C. Однако гораздо важнее, можно ли на языке реализовать весь проект, или удастся ли все проекты, с которыми сталкивается организация – будь то компания или факультет университета, – удачно выполнить с помощью данного языка.

Настоящая конкуренция – это состязание между сообществами пользователей. Хорошо организованное сообщество, объединенное некоей общей идеей, имеет локальное преимущество, но в длительной перспективе и с более широкой точки зрения оказывается в проигрыше.

Широкий диапазон возможностей C++, разнообразие круга пользователей, способность справляться с практическими проблемами, высокая эффективность – вот что составляет истинную силу языка.

Глава 8. Библиотеки

Понять жизнь можно, лишь оглядываясь назад,
но жить-то приходится, смотря вперед.

Сёрен Кьеркегор

8.1. Введение

Проектирование библиотеки оказывается лучше включения нового средства в язык гораздо чаще, чем можно предположить. С помощью классов можно представить почти все необходимые концепции. Использование библиотек обычно не помогает при решении синтаксических проблем, но иногда выручают конструкторы и перегрузка операторов. При необходимости специальную семантику или особо высокую производительность можно реализовать на других языках. Примером служат библиотеки для высокопроизводительных векторных операций, которые реализуются с помощью встраиваемых операторных функций и компилируются в код, оптимизированный для специального оборудования.

Размышляя о библиотеках, сначала следует рассматривать возможность их применения. Проектирование библиотек – почти всегда самый конструктивный способ приложения энергии для энтузиастов, жаждущих нововведений. Только если не удается достичь результата на этом пути, следует думать о расширении языка.

8.2. Проектирование библиотеки C++

Библиотека Fortran – это набор подпрограмм, библиотека C – набор функций и ассоциированных с ними структур данных, библиотека Smalltalk – иерархия классов с корнем в определенном месте стандартной иерархии Smalltalk. А что такое библиотека C++? Ясно, что она может походить на библиотеку Fortran, C или Smalltalk. Но также она может быть набором абстрактных типов с несколькими реализациями (см. раздел 13.2.2), набором шаблонов (см. главу 15)... Допустимы и другие варианты. Проектировщик библиотеки C++ может выбирать из нескольких базовых структур или даже предложить несколько вариантов интерфейса к одной библиотеке. Например, библиотеку, организованную как набор абстрактных типов, удастся представить и как набор функций для программы на C. А библиотека, устроенная в виде иерархии классов, может внешне выглядеть как набор описателей (handles).

Вариативность подходов отражает разнообразие потребностей сообщества пользователей C++. К библиотеке, поддерживающей высокопроизводительные вычисления, предъявляются иные требования, чем к библиотеке интерактивной

графики. И обе они сильно отличаются от библиотеки, предоставляющей низкоуровневые структуры данных для создания других библиотек.

C++ развивался в направлении поддержки различных структур библиотек, а некоторые из последних нововведений специально спроектированы для того, чтобы разные библиотеки можно было использовать совместно.

8.2.1. Альтернативы при проектировании библиотеки

При создании первых библиотек C++ часто слепо копировались приемы из других языков. Например, первоначальный вариант моей библиотеки для поддержки многозадачности [Stroustrup, 1980b], [Stroustrup, 1987b] – самой первой библиотеки для C++ – предоставлял средства, аналогичные механизмам моделирования из Simula. Библиотека для работы с комплексными числами [Rose, 1984] содержала функции, похожие на функции для арифметики с плавающей точкой из стандартной математической библиотеки C. Библиотека NIH Кита Горлена [Gorlen, 1990] представляла собой аналог библиотеки Smalltalk. По мере того как пользователи, работавшие раньше на других языках, переключаются на C++, появляются и новые библиотеки в стиле «раннего C++». Программисты еще не до конца освоили методы проектирования на нем; создавая такие библиотеки, они одновременно знакомятся с возможностями, которые предоставляет C++.

О каких альтернативах идет речь? Отвечая на этот вопрос, часто акцентируют внимание на языковых средствах. Следует ли пользоваться встраиваемыми функциями, виртуальными функциями, множественным наследованием, иерархиями с одним корнем, абстрактными классами, перегруженными операторами? Все это второстепенно. Перечисленные и другие средства существуют для поддержки фундаментальных возможностей. Должно ли проектирование:

- ставить во главу угла эффективность во время исполнения?
- минимизировать время перекомпиляции после изменений?
- максимально облегчать перенос на другую платформу?
- разрешать пользователям расширять базовую библиотеку?
- допускать использование без исходных текстов?
- сочетаться с существующими нотациями и стилями?
- обеспечивать использование из программ, написанных не на C++?
- быть пригодным для неопытных пользователей?

Только после ответа на вопросы такого рода можно переходить к техническим деталям. Современные библиотеки часто предоставляют самые разнообразные классы, чтобы помочь пользователю сделать правильный выбор. Например, библиотека может располагать очень простым и эффективным классом для работы со строками. Кроме того, для той же цели в ней может быть размещен высокоуровневый класс с большим числом средств и возможностей настройки пользователем (см. раздел 8.3).

8.2.2. Языковые средства и построение библиотеки

Концепция классов и система типов C++ – основа проектирования любой библиотеки. Сильные и слабые стороны данных аспектов определяют структуру

библиотек C++. Главная рекомендация авторам библиотек – не пытайтесь сражаться с системой типов. Над базовыми механизмами языка пользователь может одержать лишь пиррову победу. Изящества, простоты использования и эффективности можно достичь, только оставаясь в рамках базовой структуры языка. Если эти рамки вас стесняют, стоит подумать о смене языка.

Базовая структура C++ поощряет программирование с сильной типизацией. В C++ класс – это тип. Сочетание правил наследования, механизма абстрактных классов и механизма шаблонов подталкивают пользователя к тому, чтобы манипулировать объектами в точном соответствии с предоставляемыми интерфейсами. Можно говорить и более определенно: не обходите систему контроля путем приведения типов. Приведения необходимы для многих низкоуровневых операций, иногда для отображения высокоуровневых интерфейсов на низкоуровневые, но библиотека, при работе с которой приходится постоянно прибегать к приведениям типов, возлагает на пользователя непомерный, а чаще всего и ненужный труд. Семейство функций `printf` из библиотеки C, указатели типа `void*`, объединения и другие низкоуровневые средства лучше всего изъять из интерфейсов библиотеки, поскольку из-за них возникают «дыры» в системе контроля библиотечных типов.

8.2.3. Как работать с разнообразными библиотеками

Нельзя ожидать, что две произвольно взятые библиотеки смогут работать вместе. Многие библиотеки оказываются совместимыми, но в обычно это является предметом особой заботы и программиста, и разработчика библиотеки, и проектировщика языка.

По мере развития C++ удалось обеспечить достаточные средства для решения фундаментальных проблем, с которыми сталкивается пользователь при попытке применить независимо разработанные библиотеки. Стоит добавить, что и разработчики теперь задумываются о том, что их библиотеку, возможно, будут применять вместе с другими.

Проблема использования одного и того же имени в разных библиотеках решается с помощью пространства имен (см. раздел 17.2). В основе общей модели обработки ошибок лежит обработка исключений (см. главу 16). Шаблоны (см. главу 15) дают механизм определения контейнеров и алгоритмов, не зависящих от типа данных; конкретные типы пользователь указывает при вызове библиотеки. Конструкторы и деструкторы обеспечивают единый способ инициализации и уничтожения объектов (см. раздел 2.11). Абстрактные классы – это механизм определения интерфейсов независимо от реализации (см. раздел 13.2.2). Идентификация типа во время исполнения позволяет восстановить информацию о типе, утраченную из-за того, что переданный библиотеке объект возвращен обратно в виде экземпляра менее конкретного типа, то есть экземпляра базового класса (см. раздел 14.2.1). Конечно, это лишь одно из возможных применений указанных языковых средств, но рассмотрение их в качестве способа поддержки построения программ из независимо разработанных библиотек может быть полезным.

Рассмотрим в этом ключе множественное наследование (см. раздел 12.1). Библиотеки в духе Smalltalk основаны на одном «универсальном» корневом классе.

Сложности появляются, когда таких класса два. Но если библиотека написана для конкретной предметной области, то иногда помогает множественное наследование:

```
class GDB_root :
   public GraphicsObject,
   public DataBaseObject {};
```

Неразрешимая проблема возникает, когда оба «универсальных» базовых класса предоставляют один и тот же сервис, например идентификацию типов во время исполнения и механизм ввода/вывода объектов. Некоторые из подобных проблем лучше всего решаются путем выноса общих возможностей в стандартную библиотеку или реализации их в самом языке. Можно также поместить общую функциональность в иерархию классов с новым общим корнем. Однако объединить подобным образом «универсальные» библиотеки так просто не получится, и это необходимо учитывать при проектировании библиотеки.

Управление памятью – еще одна область, в которой имеется целый ряд проблем как для разработчиков библиотек, так и для пользователей, вынужденных обращаться сразу к нескольким библиотекам (см. раздел 10.7).

8.3. Ранние библиотеки

Первым кодом, написанным на C With Classes, была библиотека для поддержки многозадачности (см. раздел 8.3.2.1), в которой реализовались Simula-подобные механизмы параллельности для моделирования [Stroustrup, 1980b]. Первыми реальными программами, в которых использовалась эта библиотека, стали приложения для моделирования сетевого трафика, проектирования топологии печатных плат и т.д. Эта библиотека интенсивно используется и по сей день. Стандартная библиотека C – без каких бы то ни было дополнительных затрат – поставлялась вместе с C++ с самого начала, равно как и все прочие написанные на C библиотеки. Классические типы данных – строки символов, массивы с контролем выхода за границы, динамические массивы и списки – использовались как примеры при проектировании и тестировании ранних реализаций C++ (см. раздел 2.14).

Сначала работы, связанные с такими контейнерными классами, как списки и массивы, сильно осложнялись из-за отсутствия в языке поддержки параметризованных типов (см. раздел 9.2.3). Приходилось обходиться макросами. Макросы, поддерживаемые препроцессором C, помогли нам приобрести опыт, пригодившийся затем при разработке параметризованных типов.

Сами классы проектировали мы с Джонатаном Шопиро. В 1983 г. он реализовал классы строк и списков, которые широко применялись в AT&T и легли в основу классов, включенных в нынешнюю библиотеку «Стандартных компонентов». Эта библиотека была разработана в Bell Labs и продается через USL. Создание ранних библиотек шло параллельно с проектированием языка, в частности с проектированием механизма перегрузки.

Основная цель первых библиотек для работы со строками и списками – предоставление относительно простых классов, которые можно было бы использовать как строительные блоки в приложениях и при создании более сложных библиотек.

Альтернативная возможность – написание кода непосредственно на C или C++, поэтому эффективности уделялось особое внимание. Предпочтение было отдано замкнутым классам, а не иерархиям и встраиванию критических по времени операций. Кроме того, классы проектировались так, чтобы их можно было применять в традиционно написанных программах без крупных переделок или дополнительного обучения программистов. В частности, не делалось никаких попыток предоставить пользователю возможность модифицировать поведение этих классов с помощью замещения виртуальных функций в производных классах. Если бы пользователю понадобился общий модифицируемый класс, его можно было бы написать, взяв «стандартный» в качестве основы. Например:

```
class String {   // простой и эффективный
   // ...
};

class My_string {   // общий и адаптируемый
   String rep;
   // ...
public:
   // ...
   virtual void append(const String&);
   virtual void append(const My_string&);
   // ...
};
```

8.3.1. Библиотека потокового ввода/вывода

Семейство функций `printf` из библиотеки C – эффективный и во многих случаях удобный механизм ввода/вывода. Однако эти функции не обеспечивают контроля типов и не допускают применения определенных пользователем типов (классов и перечислений). Поэтому я начал думать о безопасной, лаконичной, расширяемой и эффективной альтернативе `printf`. Отчасти вдохновение снизошло на меня, когда я прочел последние полторы страницы «Ada Rationale» («Обоснование языка Ada») [Ichbiah, 1979], где доказывается, что нельзя построить лаконичную и безопасную библиотеку ввода/вывода без специальной поддержки со стороны языка. Я воспринял этот категоричный тезис как вызов. В результате появилась библиотека потокового ввода/вывода, впервые реализованная в 1984 г. и описанная в работе [Stroustrup, 1985]. Вскоре после этого Дейв Пресотто (Dave Presotto) переписал ее с целью улучшить эффективность за счет отказа от использования стандартных функций ввода/вывода из библиотеки C. Вместо этого он напрямую обращался к средствам операционной системы. Работа была сделана без изменения интерфейсов потока.

Для знакомства с новой концепцией потокового ввода/вывода рассматривался такой пример:

```
fprintf(stderr, "x = %s\n",x);
```

Поскольку `fprintf()` не проверяет переданных аргументов, основываясь на предположении, что они соответствуют форматной строке, это небезопасно. Обратимся к работе [Stroustrup, 1985], где в связи с этим говорится:

«Если бы x принадлежало определенному пользователем типу, например complex, то нельзя было бы задать формат вывода x так же спокойно, как для типов, о которых printf «знает» (к примеру, %s и %d). Для вывода комплексного числа программисту пришлось бы писать последовательность вызовов функций, например:

```
fprintf(stderr,"x = ");
put_complex(stderr,x);
fprintf(stderr,"\n");
```

Это неизящно. А уж в C++-программе, где применяется много определенных пользователем типов, такая запись была бы совсем неудобной.

Безопасного использования типов и единообразия можно достичь с помощью одного перегруженного имени для семейства функций вывода. Например:

```
put(stderr,"x = ");
put(stderr,x);
put(stderr,"\n");
```

Тип аргумента определяет, какая именно функция put будет вызвана. Однако и это слишком многословно. В C++ для решения задачи используется поток вывода, для которого оператор << имеет смысл «вывести в»:

```
cerr << "x = " << x << "\n";
```

где cerr – стандартный поток вывода для ошибок (эквивалентный stderr из C). Если x имеет тип int и равно 123, то результат этого вывода будет таким:

```
x = 123
```

и в конце – символ новой строки.

Данный способ можно использовать, если только x принадлежит типу, для которого определен оператор <<, а определить такой оператор в новом типе просто. Так, если x имеет определенный пользователем тип complex и равно (1,2.4), то это предложение выведет в cerr строку

```
x = (1,2.4)
```

Такой потоковый ввод/вывод реализован с помощью средств языка, доступных любому программисту. Как в C, так и в C++ нет встроенных в язык средств ввода/вывода. Потоковый ввод/вывод включен в библиотеку».

Возможность использования вместо именованной функции оператор вывода первоначально предложил Дуг Макилрой по аналогии с операторами перенаправления ввода/вывода в командных интерпретаторах UNIX (>, >>, | и т.д.). Требуется только, чтобы оператор возвращал свой левый операнд для передачи последующим операторам:

«Функция operator<< возвращает ссылку на объект ostream, для которого она вызывалась, с тем чтобы можно было применить к ней следующий объект ostream. Например, предложение

```
cerr << "x = " << x;
```

где x имеет тип int, интерпретируется как

```
(cerr.operator<<("x = ")).operator<<(x);
```

В частности отсюда следует, что при печати в одном предложении нескольких элементов они будут выведены в ожидаемом порядке: слева направо».

Если бы я решил использовать обычную именованную функцию, то пользователю пришлось бы писать код, как в последнем примере. Для реализации ввода и вывода рассматривалось несколько операторов:

«Оператор присваивания подошел бы и для ввода, и для вывода, но нас не устраивает свойственная ему ассоциативность. То есть `cout=a=b` интерпретировалось бы как `cout=(a=b)`. Кроме того, большинство пользователей предпочло бы, чтобы оператор ввода отличался от оператора вывода.

Рассматривались и операторы < и >, но семантика «меньше чем» и «больше чем» так прочно укоренилась в сознании, что предложения ввода/вывода с их использованием были бы просто нечитаемы (видимо, для << и >> дело обстоит не так). Помимо этого, символ `'<'` расположен на той же клавише, что и `','`, поэтому многие писали бы выражения вроде

```
cout < x , y , z;
```

Правильно диагностировать такую ошибку было бы нелегко».

Надо сказать, что сегодня мы уже могли бы перегрузить нужной семантикой оператор «запятая» (см. раздел 11.5.5), но в C++ образца 1984 г. это было невозможно.

В именах стандартных потоков `cout`, `cin` и т.д. буква `c` означает `character` (символ). Эти потоки проектировались для ввода/вывода символов.

Для версии 2.0 Джерри Шварц переписал и частично перепроектировал потоковую библиотеку, чтобы она была полезна большему числу приложений и обеспечивала более высокую эффективность файлового ввода/вывода [Schwarz, 1989]. Значительного улучшения удалось добиться благодаря выдвинутой Эндрю Кенигом [Koenig, 1991] идее манипуляторов для управления такими деталями форматирования, как задание точности для вывода чисел с плавающей точкой и основания системы счисления для целых. Например:

```
int i = 1234;

cout  << i << ' '      // по умолчанию десятичная: 1234
   << hex << i << ' '  // шестнадцатеричная: 4d2
   << oct << i << '\n'; // восьмеричная: 2322
```

Опыт, приобретенный при работе с потоками, явился основной причиной для изменения базовой системы типов и правил перегрузки. Цель – позволить, чтобы значения типа `char` трактовались как символы, а не короткие целые числа, как это было в C (см. раздел 11.2.1). Например, в результате выполнения

```
char ch = 'b';
cout << 'a' << ch;
```

в версии 1.0 была бы выведена строка цифр, представляющих значения кодов символов `a` и `b`, тогда как в версии 2.0 выводится `ab`, как и ожидалось.

Библиотека iostreams, поставлявшаяся с версией Cfront 2.0, стала моделью для реализаций других поставщиков и того варианта iostream, который включат в будущий стандарт.

8.3.2. Поддержка параллельности

Множество библиотек и расширений было написано с целью поддержать параллельность. Ученые мужи были твердо убеждены в том, что многопроцессорные системы вскоре получат гораздо большее распространение, чем сейчас. Насколько я могу судить, такое убеждение поддерживается в течение вот уже 20 лет.

Многопроцессорные системы действительно становятся обычным делом, но то же можно сказать и о быстрых однопроцессорных машинах. Поэтому нужны, по крайней мере, две формы поддержки параллельности: многопоточность на одном процессоре и многопроцессность на нескольких. Кроме того, дополнительные требования предъявляются к программам, предназначенным для работы в сети. Из-за этого многообразия я рекомендовал реализовывать параллелизм в библиотеках, а не в самом языке. Такие языковые средства, как, скажем, задачи в Ada, были бы неудобны почти для всех пользователей.

Возможно спроектировать библиотеки для поддержки параллельности так, что по удобству и эффективности они почти не будут уступать встроенным средствам. Но, опираясь на библиотеки, можно поддержать самые разные модели параллельности, что обеспечит пользователям гораздо большее удобство, чем единая встроенная модель. Проблемы же переносимости, возникающие при использовании нескольких таких библиотек, можно разрешить с помощью тонкого слоя интерфейсных классов.

Примеры библиотек для поддержки параллельности можно найти в [Stroustrup, 1980b], [Shopiro, 1987], [Faust, 1990] и [Parrington, 1990]. Примеры расширений, в которых поддержаны некоторые виды параллельности, – это Concurrent C++ [Gehani, 1988], Compositional C++ [Chandy, 1993] и Micro C++ [Buhr, 1992]. Кроме того, для поддержки потоков и облегченных процессов доступны различные коммерческие пакеты.

8.3.2.1. Пример задачи

В качестве примера программы, в которой механизмы параллельности вынесены в библиотеку, приведу алгоритм нахождения простых чисел (решето Эратосфена), где каждое простое число обрабатывается отдельной задачей. В примере используются очереди из библиотеки поддержки многозадачности [Stroustrup, 1980b], куда помещаются целые числа, служащие для задач фильтрами:

```
#include <task.h>
#include <iostream.h>
class Int_message : public object {
   int i;
public:
   Int_message(int n) : i(n) {}
   int val() { return i; }
};
```

В очередях хранятся сообщения от классов, производных от `object`. Использование имени `object` доказывает, что библиотека была разработана довольно давно. В современной программе для реализации очереди я бы использовал шаблоны, чтобы обеспечить безопасность типов, но сейчас сохраню стиль, присущий

ранним библиотекам. Использование очереди для передачи одного целого числа, пожалуй, слишком расточительный, но простой прием, а очередь гарантирует необходимую синхронизацию примитивов `put()` и `get()` из разных задач. Применение очередей иллюстрирует то, как можно передавать информацию в процессе моделирования или в системах, где нет разделяемой памяти.

```
class sieve : public task {
   qtail* dest;
public:
   sieve(int prime, qhead* source);
};
```

Класс, производный от `task`, способен работать параллельно с другими задачами. Вся полезная работа выполняется в конструкторах задачи или в вызываемом из них коде. В нашем примере каждый `sieve` является задачей. Объект `sieve` получает число из входной очереди и проверяет, делится ли оно на простое число, связанное с этим `sieve`. Если нет – `sieve` передает число дальше, следующему `sieve`. Если его нет, то мы нашли новое простое число и можем создать новый `sieve` для его представления.

```
sieve::sieve(int prime, qhead* source) : dest(0)
{
   cout << "простое\t" << prime << '\n';
   for(;;) {
       Int_message* p = (Int_message*) source->get();
       int n = p->val();
       if (n%prime) {
           if (dest) {
               dest->put(p);
               continue;
           }

           // найдено простое число: создать новый объект sieve
           dest = new qtail;
           new sieve(n,dest->head());
       }
       delete p;
   }
}
```

Сообщение создается в свободной памяти и удаляется тем объектом `sieve`, который это сообщение получил. Задачи работают под управлением планировщика, то есть система обеспечения многозадачности – это не просто система сопрограмм, где управление передается явно.

Чтобы завершить программу, нам понадобится создать первый `sieve` в функции `main()`:

```
int main()
{
   int n = 2;
```

```
    qtail* q = new qtail;
    new sieve(n,q->head()); // создать первый объект sieve
    for (;;) {
        q->put(new Int_message(++n));
        thistask->delay(1);  // дадим возможность поработать
    }
}
```

Программа будет работать, пока полностью не исчерпает системные ресурсы. Я не позаботился о ее корректном завершении. Это далеко не самый эффективный способ вычисления простых чисел. На каждое простое число требуются одна задача и множество контекстных переключений. Программа могла бы работать как симулятор на одном процессоре с адресным пространством, разделяемым всеми задачами, или как настоящая параллельная программа на нескольких процессорах. Я тестировал ее в режиме симулятора на DEC VAX для 10 тыс. простых чисел (они же задачи). Еще более удивительный вариант решета Эратосфена на C++ см. в работе [Sethi, 1989].

8.3.2.2. Блокировка

При параллельности блокировка часто более важна, чем задача. Если программист может сказать, что требуется исключительный доступ к некоторым данным, то не всегда нужны глубокие знания о процессе, задаче, потоке и т.д. В некоторых библиотеках благодаря этому наблюдению появился стандартный интерфейс к механизмам блокировки. При переносе такой библиотеки на новую архитектуру требуется корректно реализовать этот интерфейс с учетом конкретных существующих на ней форм параллельности. Например:

```
class Lock {
    // ...
public:
    Lock(Real_lock&);     // поставить блокировку
    ~Lock();              // снять блокировку
};

void my_fct()
{
    Lock lck(q2lock);     // поставить блокировку на q2
    // использовать q2
}
```

Снятие блокировки в деструкторе упрощает код и повышает его надежность. В частности, этот стиль хорошо сочетается с механизмом исключений (см. раздел 16.5). При таком подходе блокировку, ключевые структуры данных и стратегии можно реализовать независимо от деталей механизмов параллельности.

8.4. Другие библиотеки

В этом параграфе представлен небольшой список других библиотек на C++, чтобы проиллюстрировать их разнообразие. Кроме упомянутых, существует

множество других библиотек, и каждый месяц появляется несколько новых. Похоже, что та форма индустрии программных компонентов, появление которой специалисты предсказывали уже на протяжении многих лет – и горевали по поводу ее отсутствия, – наконец-то родилась.

Раздел назван «Другие библиотеки», поскольку упоминаемые в нем библиотеки не оказали существенного влияния на развитие C++. Не надо понимать это как оценку их технических достоинств или важности для пользователей.

8.4.1. Базовые библиотеки

Существует два взгляда на то, что представляют собой базовые библиотеки – горизонтальная и вертикальная. Первая предлагает набор базовых классов, которые предположительно должны быть полезны любому программисту при работе над любым приложением. Обычно сюда включаются основные структуры данных: динамические и контролируемые массивы, списки, ассоциативные массивы, AVL-деревья и т.д., а также распространенные вспомогательные классы, такие как строки, регулярные выражения, дата и время. Как правило, горизонтальную базовую библиотеку стараются сделать максимально переносимой.

Целью вертикальной базовой библиотеки является предоставление полного набора сервисов для данного окружения, например X Windows System, MS Windows, MacApp, или некоторого множества таких окружений. Обычно в состав вертикальных библиотек входят и те базовые классы, которые включаются в горизонтальные, но акцент делается на классах, в полной мере использующих особенности именно выбранного окружения. Порой преобладают классы для поддержки пользовательских интерфейсов и графики. Частью такой библиотеки могут быть также интерфейсы к конкретным базам данных. Нередко классы из вертикальной библиотеки ориентированы на применение в некотором общем каркасе, поэтому использование отдельной части библиотеки затруднено.

Лично я предпочитаю разделять горизонтальные и вертикальные аспекты базовой библиотеки, что упрощает применение и выбор наиболее подходящей из них. С другой стороны, по ряду технических и коммерческих причин разумна интеграция.

Самые значительные из ранних базовых библиотек: созданная Китом Горленом NIH [Gorlen, 1990], предлагавшая набор классов в духе Smalltalk, и Interviews Марка Линтона (Mark Linton) [Linton, 1987], упрощавшая работу с X Windows из программ на C++. Компилятор GNU C++ (G++) поставляется с библиотекой, которую спроектировал Дуг Леа. Ее отличительной особенностью является эффективное использование абстрактных базовых классов [Lea, 1993]. Библиотека USL Standard Components [Carroll, 1993] содержит набор эффективно реализованных конкретных типов для различных структур данных, а также поддержку основных промышленных версий системы UNIX. Компания Rogue Wave продает библиотеку Tools++, включающую набор базовых классов, которые с 1987 г. писали Томас Кеффер (Thomas Keffer) и Брюс Эккель [Keffer, 1993]. Компания Glockenspiel в течение многих лет поставляла коммерческие библиотеки для различных применений [Dearle, 1990]. Компания Rational поставляет C++-версию библиотеки The Booch Components, которую Грейди Буч (Grady Booch) первоначально

реализовал на языке Ada. На C++ ее переписали сам Буч и Майк Вило (Mike Vilot). Версия на Ada занимает 125 тыс. строк исходного текста без учета комментариев, тогда как версия на C++ – всего 10 тыс. строк. Как видно, наследование в сочетании с шаблонами может стать мощным механизмом для организации библиотек без потери эффективности или ясности [Booch, 1993].

8.4.2. Устойчивость и базы данных

Под устойчивостью (persistence) понимаются разные аспекты. Некоторым пользователям просто нужен пакет для ввода/вывода объектов, который входит в состав многих библиотек. Другие хотят обеспечить прозрачную миграцию объекта из файла в память и обратно. Третьим необходим контроль версий и протоколирование транзакций. Четвертые не согласны ни на что, кроме распределенной системы с адекватным управлением параллельностью и полной поддержкой перемещения схем. Поэтому я думаю, что устойчивость следует обеспечивать за счет специальных библиотек, нестандартных расширений, а возможно, и продуктов третьих фирм. Имеющийся в C++ механизм идентификации типов во время исполнения содержит кое-какие «зацепки», которые оказались полезны для специалистов по устойчивости (см. раздел 14.2.5).

И NIH, и библиотека GNU содержат базовые механизмы для ввода/вывода объектов. POET – пример коммерческой C++-библиотеки для поддержки устойчивости. Существует около дюжины объектно-ориентированных баз данных, к которым можно обращаться из C++ и на C++ же и написанных. В качестве примера назову ObjectStore, ONTOS [Cattell, 1991] и Versant.

8.4.3. Библиотеки для численных расчетов

Компании Rogue Wave [Keffer, 1992] и Dyad предлагают много классов, ориентированных прежде всего на научные приложения. Основное назначение таких библиотек – представить нетривиальные математические методы в форме, удобной для специалистов, работающих в разных областях науки или техники. Вот пример использования поставляемой компанией Sandia National Labs библиотеки RHALE++ для математической физики:

```
void Decompose(const double delt, SymTensor& V,
          Tensor& R, const Tensor& L)
{
   SymTensor D = Sym(L);
   AntiTensor W = Anti(L);
   Vector z = Dual(V*D);
   Vector omega = Dual(W) - 2.0*Inverse(V-Tr(V)*One)*z;
   AntiTensor Omega = 0.5*Dual(omega);

   R = Inverse(One-0.5*delt*Omega) * (One+0.5*delt*Omega)*R;
   V += delt*Sym(L*V-V*Omega);
}
```

Согласно [Budge,1992], «этот код прозрачен, а лежащие в его основе библиотеки классов весьма многоплановы и удобны для сопровождения. Физику, знакомому

с алгоритмом полярной декомпозиции, смысл этого кода ясен с первого взгляда, так что не нужна даже дополнительная документация».

8.4.4. Специализированные библиотеки

Вышеупомянутые библиотеки предназначены главным образом для поддержки программирования как такового. Но не менее важны и узкоспециализированные библиотеки. В частности, можно найти общедоступные, коммерческие и разработанные для внутреннего применения библиотеки для таких областей, как гидродинамика, молекулярная биология, анализ сетей связи, пульты телефонных операторов и т.д. Для многих программистов именно подобные библиотеки являются свидетельством достоинств C++ в плане простоты программирования, уменьшения числа ошибок, сокращения затрат на сопровождение и т.д.

Вот пример из области моделирования сетей с коммутацией каналов [Eick, 1991]:

```
#include <simlib.h>

int trunks[] = { /* ... */ };
double load[] = { /* ... */ };
class LBA : public Policy { /* ... */ };

main()
{
   Sim sim;                                  // планировщик событий

   sim.network(new Network(trunks));         // создать сеть
   sim.traffic(new Traffic(load,3.0));       // матрица трафика
   sim.policy(new LBA);    // стратегия маршрутизации LBA

   sim.run(180); // моделировать в течение 180 минут

   cout<<sim;    // вывести результаты
}
```

Встречающиеся в этом фрагменте классы – это либо классы из библиотеки SIMLIB, либо произведенные от них пользователем. Они определяют сеть, нагрузку и стратегию маршрутизации для одного прогона модели.

Многие специализированные библиотеки, например для поддержки графики и визуализации, являются общими, но в этой книге я не ставил своей целью перечислять или классифицировать их.

8.5. Стандартная библиотека

Коль скоро библиотеки для C++ так разнообразны, возникает вопрос о том, какие из них должны быть стандартными, то есть специфицированными в стандарте и включенными в любую реализацию.

Прежде всего необходимо стандартизировать основные библиотеки, используемые повсеместно. Это значит, что нужно специфицировать точный интерфейс между C++ и стандартными библиотеками C, а также библиотеку iostreams.

Кроме того, нужно специфицировать базовую языковую поддержку, например, функции `::operator new(size_t)` и `set_new_handler()`, которые поддерживают оператор `new` (см. раздел 10.6), функции `terminate()` и `unexpected()` для поддержки обработки исключений (см. раздел 16.9) и классы `type_info`, `bad_cast` и `bad_typeid`, необходимые для поддержки идентификации типов во время исполнения (см. раздел 14.2).

Далее комитет должен разобраться, может ли удовлетворить желание иметь «более полезные и стандартные классы», например `string`, не занимаясь самостоятельно проектированием и не вступая в конкуренцию с индустрией производства библиотек для C++. Любые библиотеки, кроме библиотек C и iostreams, которые одобряет комитет, должны включать лишь строительные блоки. Основная роль стандартной библиотеки – облегчить взаимодействие между специализированными, независимо разработанными библиотеками.

Принимая во внимание все вышесказанное, комитет одобрил классы `string` и `wstring` и пытается объединить их в некий шаблон «строки, включающей абсолютно все». Одобрен также класс динамического массива `dynarray` [Stal, 1993], шаблон класса `bits<N>` для битовых множеств фиксированного размера, класс `bitstring` для битовых множеств с изменяемым размером. Кроме того, комитет принял классы комплексных чисел (предшественник – первоначальный класс `complex`, см. раздел 3.3), обсуждается вопрос, следует ли принять класс вектора для поддержки численных расчетов и научных приложений. О наборе стандартных классов, их спецификациях и даже названиях все еще ведутся оживленные споры.

На мой взгляд, в стандартную библиотеку должны были включены шаблоны списка и ассоциативного массива (отображения) – см. раздел 9.2.3. Однако, как и в случае с версией 1.0, они могут быть принесены в жертву желанию во что бы то ни стало завершить стандартизацию ядра языка вовремя.[1]

---

[1] Я очень рад, что могу взять эти слова обратно! Комитет проникся важностью вопроса и одобрил великолепную библиотеку контейнеров, итераторов и фундаментальных алгоритмов, спроектированную Алексом Степановым. Эта библиотека, которую часто называют STL (Standard Template Library), представляет собой изящный, эффективный, формально полный и хорошо протестированный каркас для построения и использования контейнеров (Alexander Stepanov и Meng Lee: *The Standard Template Library*, HP Labs Technical Report HPL-94-34 (R.1), 1994. Mike Vilot: *The C++ Report*, 1994). Разумеется, STL содержит классы для отображений и списков и включает в качестве частных случаев вышеупомянутые классы `dynarray`, `bits` и `bitstring`. Кроме того, комитет одобрил классы векторов для поддержки численных и научных расчетов, приняв за основу предложение Кена Баджа из Sandia Labs.

Глава 9. Перспективы развития языка C++

Нельзя дважды войти в одну и ту же реку.

Гераклит

9.1. Введение

Данная глава в большей степени основана на личных оценках и обобщениях. Здесь приведены ответы на некоторые общие вопросы и рассмотрены проблемы, которые всегда возникают при обсуждении проектирования C++. Глава состоит из трех взаимосвязанных частей:

- попытка оценить текущее состояние C++ с учетом имеющегося опыта, а также разобраться, каким мог бы стать язык (см. раздел 9.2);
- оценка проблем, которые станут актуальными для разработчиков программ в будущем, с целью понять, как с помощью C++ можно было бы способствовать их решению (см. раздел 9.3);
- взгляд на те области, в которых сам язык и его применение стоило бы значительно усовершенствовать (см. раздел 9.4).

9.2. Оценка пройденного пути

Часто говорят, что «задним умом всяк крепок». Это неверно. Данное утверждение основано на ложных допущениях о том, будто нам известны все существенные факты прошлого, что мы знаем все о текущем состоянии дел и достаточно беспристрастны. Обычно ни одно из этих условий не выполняется. Поэтому ретроспективный взгляд на явление, столь сложное и динамичное, как язык программирования и его широкомасштабное применение, не может быть простой констатацией факта. Так или иначе, я попытаюсь дать ответы на некоторые трудные вопросы:

- достигнуты ли основные цели C++?
- является ли C++ логически последовательным языком?
- какова основная недоработка языка?

Разумеется, ответы на эти вопросы взаимосвязаны: «да», «да» и «отказ от включения более полной библиотеки в версию 1.0».

9.2.1. Достигнуты ли основные цели C++?

«C++ – это язык программирования общего назначения, цель которого – сделать работу серьезных программистов более приятным занятием» [Stroustrup, 1986b]. Данная задача успешно решена. Точнее, достаточно образованным и опытным программистам язык дал возможность писать программы на более высоком уровне абстракции («точно так же, как в Simula»), не утратив присущей C эффективности. Это удалось сделать для приложений (внутренне сложных, ограниченных условиями среды выполнения), которым требуется быстродействие и большое потребление памяти.

В более широком смысле объектно-ориентированное программирование и абстракции данных стали доступны разработчикам, которые раньше относились к таким методам и поддерживающим их языкам (Smalltalk, Clu, Simula, Ada, OO Lisp и т.д.) пренебрежительно и даже презрительно («дорогие игрушки, непригодные для решения реальных задач»). Чтобы преодолеть такой барьер, было сделано следующее:

- C++ генерирует код, который по быстродействию и расходу памяти может на равных конкурировать к признанным лидером в этой области – языком C. Любой язык, задачей которого является не уступить или даже превзойти C, просто «обязан быть быстрым». Необходимую производительность C++ обеспечивает как для традиционно структурированного кода, так и для основанного на абстракциях данных и объектно-ориентированных методах;
- такой код может интегрироваться с существующими программами и получаться на широко распространенных платформах. Высокая степень переносимости считалась чрезвычайно важным критерием, как и способность сосуществовать с уже написанным кодом и с традиционными инструментами, например отладчиками и редакторами связей;
- в C++ допускается постепенный переход на новые приемы программирования. Благодаря C++ доступны каждому стали объектно-ориентированное программирование и абстракции данных.

C++ также явился мощным стимулом для языков, поддерживающих в том или ином виде объектно-ориентированное программирование и абстрагирование данных. Появление C++ помогло и пользователям других языков, побудив разработчиков этих языков увеличивать производительность и гибкость.

9.2.2. Является ли C++ логически последовательным языком?

В целом и я, и большинство пользователей довольны языком C++. Есть, конечно, много деталей, которые я улучшил бы, если бы мог. Однако фундаментальная концепция сильно типизированного языка, основанного на классах с виртуальными функциями и предоставляющего средства для низкоуровневого программирования, кажется мне здравой. Кроме того, все основные средства тесно связаны и поддерживают друг друга.

9.2.2.1. Что можно и нужно было сделать по-другому?

Какой язык справился бы с решением тех задач, для которых проектировался C++, лучше? Рассмотрим решения первого порядка (см. разделы 1.1, 2.3 и 2.7):

- использование статического контроля типов и классов в духе Simula;
- четкое разделение между языком и средой;
- совместимость с C на уровне исходных текстов («так близко к C, насколько это возможно»);
- совместимость с C на уровне редактирования связей и размещения объектов в памяти («истинные локальные переменные»);
- независимость от наличия сборщика мусора.

Я по-прежнему считаю, что статический контроль типов является необходимым условием хорошего проектирования и эффективности выполнения. Если бы мне пришлось создавать новый язык для тех задач, которые сегодня решаются на C++, я снова взял бы за основу модель контроля типов из Simula, а не из Smalltalk или Lisp. Как я много раз повторял: «Если бы я хотел имитировать Smalltalk, то построил бы гораздо лучшую имитацию. Лучше Smalltalk, чем Smalltalk, нет. Если вам нужен Smalltalk, им и пользуйтесь» [Stroustrup, 1990]. Одновременное наличие статического контроля типов и их динамической идентификации (например, в форме вызовов виртуальных функций) требует некоторых непростых компромиссов по сравнению с языками, где есть только статический или только динамический контроль типов. Модели статических и динамических типов не могут быть идентичны, поэтому всегда будет существовать некоторая сложность и отсутствие изящества, которых можно было бы избежать, если придерживаться какой-то одной модели. Однако мне не хотелось бы писать удовлетворяющие этому условию программы.

Я также считаю существенно важным разделение языка и среды и не желаю пользоваться только одним языком, единственным набором инструментальных средств и одной операционной системой. Чтобы иметь выбор, необходимо разделение. Однако если оно есть, то можно предоставить разные среды под разные требования к степени поддержки, потреблению ресурсов и переносимости.

Однако недостаточно сделать что-то новое, надо еще, чтобы люди смогли перейти к этому новому от старых инструментов и представлений. Если бы не было C, то я бы сделал C++ совместимым с каким-то другим языком. Будучи основан на C, C++ унаследовал некоторые синтаксические несуразицы, путаные правила преобразования для встроенных типов и т.д. Эти несовершенства были источником постоянных трудностей, но альтернатива – серьезная несовместимость между C и построенным на его базе языком или попытка внедрить язык, спроектированный с нуля, – принесла бы проблемы гораздо более серьезные, чем имеющиеся сейчас. В частности, совместимость с C на уровне компоновки и библиотек считалась абсолютно необходимой. Совместимость на уровне компоновки означала также и возможность связывания с программами на других языках, поскольку они могут быть связаны с кодом на C.

Должен ли язык предоставлять ссылочную семантику для переменных (это означает, что имя является указателем на объект, размещенный в другом месте), как в Smalltalk и Modula-3, или лучше иметь истинные переменные, как в языках C и Pascal? Вопрос решающий и тесно связанный еще с несколькими: сосуществование с другими языками, эффективность во время исполнения, управление памятью, использование полиморфных типов. В языке Simula есть ссылки на объекты классов и истинные переменные встроенных типов (и только). Я считаю открытым вопрос о том, можно ли спроектировать язык, сочетающий положительные стороны ссылок и истинных локальных переменных, сохранив определенную стройность, непротиворечивость. Между изяществом языка и преимуществами, которые дают ссылки и истинные переменные я предпочту иметь последние.

Должен ли новый язык напрямую поддерживать сборку мусора, как, скажем, Modula-3? Если да, то удалось бы достичь основных целей C++, если бы в нем был сборщик мусора? В качестве необязательной возможности сборщик мусора желателен. Но он может негативно повлиять на быстродействие программы, время реакции и простоту переноса. Поэтому необходимость платить за сборку мусора всегда и везде – вряд ли правильный ход. В C++ допускается наличие необязательного сборщика мусора [2nd, с. 466–468]. Ведутся экспериментальные работы по реализации таких компиляторов C++.

9.2.2.2. От чего стоило бы отказаться?

Еще в работе [Stroustrup, 1980] высказывалось опасение, что язык C with Classes может оказаться слишком громоздким. Среди всех высказанных претензий пожелание иметь «язык поменьше» стоит на первом месте, и все же пользователи забрасывают меня и комитет по стандартизации предложениями о расширениях. Я не вижу, от какой составляющей C++ можно было бы отказаться, не жертвуя при этом поддержкой той или иной важной сферы применения. Даже если полностью игнорировать вопросы совместимости, удастся упростить лишь немногие из фундаментальных механизмов C++. В основном это коснулось бы C-подмножества, но мы как-то забываем, что сам C – довольно большой и сложный язык.

C++ он поддерживает несколько способов написания программ, несколько парадигм программирования. Поэтому-то он так велик. В каком-то смысле C++ – это три языка в одном:

- C-подобный язык (для поддержки низкоуровневого программирования);
- Ada-подобный язык (для поддержки абстрактных типов данных);
- Simula-подобный язык (для поддержки объектно-ориентированного программирования);
- прослойка, необходимая для интеграции всех этих средств в один логически последовательный язык.

Писать программы в любом из этих стилей можно и на C, но он не предоставляет прямой поддержки ни для абстрагирования данных, ни для объектно-ориентированного программирования, тогда как C++ непосредственно поддерживает несколько альтернативных подходов.

При создании программы всегда есть некоторое количество вариантов, но в большинстве языков выбор сделал за вас проектировщик языка. В случае C++ это не так – выбор за вами. Такая гибкость, естественно, непереносима для тех, кто считает, что существует лишь один правильный способ действий. Она может также отпугнуть начинающих пользователей и преподавателей, полагающих, что язык хорош, если его можно полностью освоить за неделю. C++ к таким языкам не относится. Он был спроектирован как набор инструментов для профессионалов, и жаловаться на то, что в нем слишком много возможностей, – значит уподобляться дилетанту, который, заглянув в чемоданчик обойщика, восклицает, что столько разных молоточков никому не потребуется.

Каждый язык, используемый для решения нетривиальных задач, развивается, чтобы полнее удовлетворить потребности пользователей. Это неизбежно ведет к усложнению. C++ здесь не исключение, сложность языка увеличивается по мере усложнения задач, которые он призван решать. Если она не проявляется в самом языке, значит, присутствует в библиотеках и инструментальных средствах. Примеры языков, которые значительно «выросли» по сравнению с оригинальным проектом, – Ada, Eiffel, Lisp (CLOS) и Smalltalk. Из-за акцента C++ на статическом контроле типов усложнение в основном приняло форму расширений языка.

Я сделал возможным поэтапное изучение и использование языка (см. раздел 7.2), чтобы он не воспринимался таким громоздким. Типичное для больших языков снижение производительности также минимизировано путем исключения излишних затрат (см. раздел 4.5).

9.2.2.3. Что стоило бы добавить?

В целом добавлять хотелось бы как можно меньше. В письме рабочей группы по расширениям в составе комитета по стандартизации C++ эта позиция изложена так:

«Во-первых, мы попытаемся убедить вас не вносить новые предложения о расширении языка. C++, по нашему мнению, и так уже достаточно велик и сложен. Кроме того, на нем написаны миллионы строк кода, которые должны остаться работоспособными. Любые изменения будут проходить очень сложную процедуру согласования. Каждое добавление вызывает тревогу. Мы предпочитаем языковым расширениям специальные приемы программирования и библиотечные функции, где только это возможно.

Многие группы программистов хотели бы, чтобы любимая ими конструкция или библиотечный класс стали частью языка. Однако включение средств, полезных той или иной части сообщества пользователей, превратило бы C++ в конгломерат не связанных между собой возможностей».

И все же, что можно было бы добавить в C++ с учетом вышесказанного? Лично мне тех средств, которые описаны на страницах данной книги (включая и часть II, где говорится о шаблонах, исключениях, пространствах имен и идентификации типов во время исполн,ения), достаточно. Я хотел бы включить необязательную сборку мусора, но отношу это, скорее, к качеству компилятора, а не к средствам языка.

9.2.3. Основная недоработка языка

По моему мнению, только одну недоработку можно было бы назвать величайшей ошибкой C++. Выпуск версии 1.0 и первого издания моей книги [Stroustrup,

1986] следовало отложить до окончания разработки более полной библиотеки, включающей такие фундаментальные классы, как односвязные и двусвязные списки, ассоциативные массивы, массивы с контролем выхода за границы и простые строки. Их отсутствие заставило всех «изобретать велосипед» и породило ненужное разнообразие базовых классов. Пытаясь самостоятельно построить базовые классы, программисты вынуждены были сразу начать с «продвинутых» возможностей, еще не освоив основы C++. Кроме того, много усилий было потрачено на разработку методов и инструментов по ходу работы с библиотеками, которые имели серьезный внутренний дефект – отсутствие шаблонов.

В некотором смысле я бы мог избежать вышеназванной ошибки. В первоначальном плане книги значилось три раздела, посвященных библиотеке: о потоковом вводе/выводе, о контейнерных классах и о поддержке многозадачности. Приблизительно я знал, чего хочу, но, к сожалению, слишком устал и не мог реализовать контейнерные классы, не имея в каком-то виде шаблонов. Идея «подделать» шаблоны с помощью препроцессора или некоторой временной уловки в компиляторе не пришла мне тогда в голову.

9.3. Всего лишь мост?

Я создал C++ как мост, по которому люди могли бы перейти от традиционного программирования к абстракциям данных и объектно-ориентированному программированию. Есть ли у C++ будущее за этими рамками? Разве C++ – это всего лишь мост и ничего более? Будет ли этот язык еще представлять интерес и иметь ценность, когда объектно-ориентированное программирование будет применяться повсеместно? И если ответ положителен, то можно ли что-нибудь сделать для тех пользователей C++, которым не важна совместимость с C, не причинив вред тем, для кого она еще, по крайней мере, десять лет будет необходима?

Язык существует для того, чтобы помочь при решении задач. Если язык с самого начала оказался удачным, то будет существовать, пока есть задачи, при решении которых он оказался полезным. И станет пользоваться успехом, если только в каком-то другом языке не будут предложены гораздо лучшие решения для тех же задач. Поэтому возникают следующие вопросы:

- останутся ли актуальными задачи, которые помогает решать C++?
- появятся ли существенно лучшие решения?
- сможет ли C++ предложить хорошие решения новых задач?

Я отвечаю на вопросы так: «многие останутся», «медленно» и «да».

9.3.1. Мост нужен надолго

Пройдет еще достаточно времени, пока большинство пользователей преуспеет в использовании объектно-ориентированного программирования, проектирования и т.д. Роль C++ как моста и средства для гибридного проектирования и разработки не исчерпает себя в двадцатом столетии, а его значение как средства сопровождения и обновления старого кода будет актуально еще дольше.

Не стоит забывать, что даже переход от ассемблера к C осуществился не повсеместно. Точно так же и переход от C к C++ может занять длительное время.

Однако в этом и сильная сторона C++. Тем, кому действительно нужен чистый C, C++ предоставляет все его возможности без потери эффективности. Поддержка обоих стилей – как на переходный период, так и там, где стиль C действительно наиболее подходит, – одна из фундаментальных целей языка C++.

9.3.2. Если C++ – это ответ, то на какой вопрос?

C++ – язык общего назначения или, по крайней мере, многоцелевой. Отсюда следует, что для каждой конкретной ситуации можно построить язык или систему, которая бы отвечала требованиям данной ситуации лучше, чем C++. Однако сила C++ в том, что он дает достаточно хорошие ответы на многие вопросы, а не наилучший ответ на какой-то один вопрос. Например, C++, как и C, отлично подходит для низкоуровневого системного программирования, и его производительность в этой области обычно выше, чем у других языков высокого уровня. Однако для большинства машинных архитектур хороший программист на ассемблере смог бы написать код, несколько меньший по размерам и более быстрый, чем тот, который способен сгенерировать хороший компилятор.

С трудом представляю себе то приложение, для которого нельзя было бы построить специализированный язык, лучший, чем C++ или любой другой язык общего назначения. Поэтому самое большее, на что может рассчитывать язык общего назначения, – «быть вторым».

Отметив данный факт, я все же остановлюсь на сильных сторонах C++:

- низкоуровневое системное программирование;
- высокоуровневое системное программирование;
- встроенные системы;
- численные и научные расчеты;
- общее прикладное программирование.

Некоторые из этих категорий пересекаются. Ни у одной нет общепринятого определения. Но C++ остается хорошим выбором в каждой из этих областей. Более того, любой язык, подходящий для всех этих категорий, очень напоминал бы C++ в плане предоставляемых базовых сервисов, хотя в синтаксическом или семантическом отношении мог бы быть другим. Упомянутые области не исчерпывают всего разнообразия приложений, в которых успешно применялся C++, но именно они являются ключевыми.

9.3.2.1. Низкоуровневое системное программирование

Для низкоуровневого программирования C++ подходит лучше всех существующих языков. В нем сильные стороны C сочетаются с возможностью простого абстрагирования данных с нулевыми издержками по быстродействию и памяти, он удобен для создания больших программ. Никакой новый язык в этой сфере не сможет оказаться настолько лучше C++, чтобы заменить его. Системное программирование, включая разработку низкоуровневых компонентов, всегда будет сильной стороной C++. В этой области он выступает как улучшенный C. Еще много лет единственным конкурентом C++ здесь будет оставаться C, и все же C++ – более удачный выбор, поскольку он и есть C, только лучше. Я ожидаю, что низкоуровневое системное программирование будет постепенно утрачивать свою

важность, но останется одной из основных областей применения C++. Поэтому надо быть очень осторожным, чтобы не «улучшить» язык или компиляторы C++ настолько, что он станет языком только высокого уровня.

9.3.2.2. Высокоуровневое системное программирование

Размер и сложность традиционных систем растут очень быстро. В качестве примера можно привести ядра ОС, диспетчеры сетей, компиляторы, системы электронной почты, системы для фотонабора, для манипуляций с изображениями и звуком, системы связи, пользовательские интерфейсы и системы баз данных. Следовательно, традиционный акцент на низкоуровневой эффективности смещается в сторону общей структуры системы. Эффективность по-прежнему остается важным аспектом, но становится вторичной, поскольку никому не нужна, если большую систему не удастся экономно построить и сопровождать.

Имеющиеся в C++ средства абстрагирования данных и объектно-ориентированного программирования относятся именно к этой стороне вопроса. Шаблоны, пространства имен и исключения будут значить все больше и больше для программистов, работающих над такими приложениями. Изолирование необходимых нарушений системы типов в низкоуровневых функциях, подсистемах и библиотеках будет становиться все более критичным. При использовании этой техники основной код приложения станет безопасным с точки зрения типов, а значит, более удобным для сопровождения. Я ожидаю, что значимость высокоуровневого системного программирования с годами возрастет, но тем не менее останется областью, где C++ имеет большие преимущества.

Для высокоуровневого системного программирования хорошо подходят и многие другие языки, например Ada9X, Eiffel и Modula-3. Если не считать поддержки сборки мусора и параллельности, то они приблизительно эквивалентны C++ с точки зрения фундаментальных механизмов. Естественно, о качестве отдельных средств и глубине их интеграции в язык можно спорить бесконечно. Однако при наличии высококачественных реализаций каждый из этих языков может поддержать широкий круг приложений. Определяющими при разработке станут проблемы, не относящиеся к техническим деталям языка: управляемость, приемы проектирования и обучение программистов. У C++ есть определенные конкурентные преимущества: эффективность, гибкость, доступность и наличие большого числа пользователей.

Для некоторых крупных приложений автоматическая сборка мусора – это большое преимущество, для других – дополнительная сложность. Если в компиляторы C++ не будет включен необязательный сборщик мусора, то в определенных областях C++ займет невыгодные позиции, но я уверен, что такие компиляторы скоро появятся повсеместно.

9.3.2.3. Встроенные системы

В ряду областей системного программирования встроенные системы должны быть рассмотрены особо. Под встроенными системами понимаются программы, управляющие работой компьютеризованных устройств: видеокамер, автомобилей, ракет, телефонных коммутаторов и т.д. Думается, что важность таких систем со временем возрастет и применяться в них будет смесь низко- и высокоуровневого

системного программирования. В таком случае, C++ должен будет удовлетворять самым разнообразным требованиям, которые вряд ли сможет выполнить узкоспециализированный язык. В одних проектах начнут интенсивно использоваться исключения, в других они будут запрещены как слишком непредсказуемые. Аналогично требования к управлению памятью будут варьироваться от «полного отсутствия динамической памяти» до «необходимости использовать автоматическую сборку мусора». Ко всему прочему, начнется интенсивное применение самых разных моделей параллельности. Важно, что C++ – это язык, а не законченная система. Это позволяет использовать его при разработке специализированных систем и генерировать код для исполнения на специальном оборудовании. Возможность запускать C++ в отдельных средах разработки и под управлением симуляторов на стандартной аппаратуре может оказаться для проекта существенной. Тот факт, что написанные на C++ программы допустимо помещать в постоянную память, в свое время тоже был важен. С использованием C++ для программирования разнообразных компьютеризованных устройств у меня связаны большие надежды. В этой области могут быть востребованы сильные возможности C, поддерживаемые и в C++.

9.3.2.4. Численные и научные расчеты

Численные и научные расчеты – сравнительно небольшая область программирования, в ней не занято много специалистов, но она очень ценна и интересна. Я наблюдаю смещение акцента в сторону развитых алгоритмов, для которых важна способность языка описывать и эффективно использовать разнообразные структуры данных. Fortran такой гибкостью не обладает, но это компенсируется эффективностью при выполнении базовых операций над векторами. Важно, что C++-программа может вызывать подпрограммы, написанные на Fortran или ассемблере, там, где это необходимо или просто удобно. Интеграция численных программ в более крупные приложения предъявляет требования, которым удовлетворяет C++. Например, преимущества Fortran в низкоуровневых вычислениях не так существенны, когда основной упор делается на нечисленные аспекты: визуализацию, моделирование, доступ к базе данных и сбор данных в реальном времени.

9.3.2.5. Общее прикладное программирование

C++ не является идеальным инструментом для приложений, где не нужны развитые системные компоненты, а требования к быстродействию и потреблению памяти не слишком жесткие. Однако при поддержке со стороны библиотек и, возможно, сборщика мусора, он может найти применение и здесь.

Я полагаю, что во многих подобных предметных областях преобладающими станут специализированные языки, генераторы программ и инструментальные средства для прямого манипулирования. Например, зачем набирать текст программы для создания пользовательского интерфейса, когда это может сделать автоматический генератор при подаче ему на вход примера размещения элементов на экране? Точно так же, зачем программировать сложные математические расчеты на Fortran или C++, если можно использовать специализированный язык более высокого уровня? Однако и высокоуровневый язык, и генератор интерфейсов нужно реализовывать на подходящем языке. Самим генератором также зачастую

создается код на другом языке, который уже и выполняет нужные действия. Требованиям к языку реализации и к целевому языку C++ удовлетворяет очень хорошо, так что я предвижу возрастание его роли в таких приложениях. Эту роль C++ наследует от C. Но такие детали, как возможность объявлять переменные почти повсеместно в сочетании со средствами организации программ (пространства имен), делают C++ даже более удобным в качестве целевого языка, чем C.

Высокоуровневые инструменты и языки тяготеют к специализации. Поэтому хорошие средства такого уровня должны предоставлять пользователям возможность расширения и модификации стандартного поведения путем добавления кода, написанного на языке более низкого уровня. Механизмы абстракции C++ позволяют легко интегрировать его в каркас, предоставляемый инструментальным средством высокого уровня.

9.3.2.6. Смешанные системы

Самая существенная из сильных сторон C++ проистекает из его способности работать в системах, в которых сочетаются особенности приложений всех вышеупомянутых видов. Для пользовательских интерфейсов нужна графика; специфические приложения требуют наличия специализированных языков и генераторов программ; симуляторам и аналитическим подсистемам необходимо выполнять сложные вычисления; в коммуникационных подсистемах широко применяется системное программирование; большие системы, как правило, нуждаются в базе данных; для работы со специальной аппаратурой требуется низкоуровневое программирование. Во всех этих (и многих других) областях предпочтение будет отдано C++, если не в первую, то, по крайней мере, во вторую очередь. Но самым широкоиспользуемым он будет достаточно часто, для того чтобы считаться основным языком.

Все языки либо перестают существовать сами собой, либо изменяются в соответствии с новыми требованиями. Язык с многочисленным и энергичным сообществом пользователей, скорее, изменится, чем исчезнет. Это случилось с C, который уступил дорогу C++, и то же самое когда-нибудь произойдет с C++. C++ – относительно молодой язык, но все же стоит присмотреться к его сильным и слабым сторонам, чтобы задействовать первые и компенсировать последние.

C++ не совершенен. Однако он достаточно хорош, так что замена аналогичным языком ему не грозит. Только принципиально иной язык мог бы предоставить по-настоящему серьезные преимущества, чтобы его стали считать безусловно лучшим. Быть просто «улучшенным C++» недостаточно, чтобы сменить его. Вот почему C++ – это не просто улучшенный C. Если бы C++ не предлагал совершенно новые способы написания программ, то программисты не отказывались бы от C. Именно поэтому Pascal и Modula-2 потерпели неудачу как альтернативы C, хотя многие представители академических кругов рекламировали их в течение многих лет. Они просто не настолько сильно отличались от C, чтобы намного превзойти его.

Я не вижу принципиально отличного языка, который в ближайшем будущем мог бы заменить C++ в тех областях, где он применяется. Есть только языки, обладающие, по сути, теми же, хоть и иначе представленными возможностями, а также узкоспециализированные и экспериментальные языки.

9.4. Что может сделать C++ более эффективным

На протяжении многих лет ожидания опережали самые фантастические усовершенствования аппаратного и программного обеспечения, и в этом отношении вряд ли что-то изменится. Многое можно сделать, чтобы компилятор C++ стал еще полезнее пользователям, а программистам и проектировщикам есть чему поучиться, чтобы работать эффективнее. Выскажу несколько замечаний о том, что, по моему мнению, следует предпринять, чтобы программирование на C++ стало более продуктивным.

9.4.1. Стабильность и стандарты

Стабильность определения, основных библиотек и интерфейсов занимает высокое место в списке требований к дальнейшему развитию языка. Первое должен обеспечить комитет по стандартизации ANSI/ISO, второе – дело организаций и компаний, работающих над интерфейсами операционных систем, баз данных, динамически подключаемыми библиотеками и т.д.

Разумеется, пользователи, вправе требовать новых возможностей, но лично мне достаточно видеть C++ и таким, каким он представлен в этой книге. Думаю, что большинство программистов, пишущих промышленные коды, согласятся со мной. Не стоит забывать, что никакое средство, взятое в отдельности, не является достаточным для создания «удачной программы» (что бы ни имелось в виду под этим словосочетанием).

9.4.2. Обучение и приемы

Наибольший потенциал для улучшений языка я вижу в простом изучении новых приемов проектирования и программирования. Самых простых и экономичных результатов можно было бы добиться просто за счет более эффективного использования C++. И не нужно никаких дорогостоящих инструментов. С другой стороны, изменить образ мышления нелегко. Большинству программистов недостаточно просто выучить новый синтаксис, надо еще глубоко освоить новые концепции. Посмотрите, что написано в разделе 7.2 и в учебнике, где затрагиваются вопросы проектирования, например, в [2nd] или [Booch, 1993]. Я предчувствую, что в ближайшие несколько лет мы станем свидетелями значительного улучшения методов проектирования и программирования, но это все равно не дает нам права сидеть сложа руки и ждать.

9.4.3. Системные вопросы

C++ – язык, а не полная система. В большинстве случаев это достоинство, а для организации среды полного цикла разработки и тестирования существуют инструментальные средства. Но между языком и средой должен быть интерфейс. Его отсутствие привело к удручающе медленному развитию в области динамической загрузки. Пользователи либо ничего не предпринимали, полагаясь на механизмы, спроектированные для C++, либо работали над механизмами, достаточно общими для поддержки «любых объектно-ориентированных языков». С точки зрения программиста на C++ результаты оказались довольно плачевными.

Ранние эксперименты по интегрированию C++ и динамического связывания были настолько многообещающими, что я ожидал повсеместного распространения динамического связывания классов еще несколько лет назад. Например, еще в 1990 г. существовала работающая методика эффективного и безопасного инкрементного связывания, основанная на абстрактных типах [Stroustrup, 1987d], [Dorward, 1990]. В реальных системах она использовалась не очень широко, но абстрактные классы оказались важны для уменьшения числа повторных компиляций после изменения исходного кода и вообще для упрощения использования компонентов, полученных из разных источников (см. раздел 13.2.2).

Еще один важный вопрос, остающийся нерешенным, – поддержка эволюции программного обеспечения. Коль скоро библиотека выпущена в обращение, ее реализацию можно изменить только в том случае, если пользователи не зависят от таких деталей, как размер объекта, или могут и готовы перекомпилировать свой код с новой версией библиотеки. Объектные модели – OLE2 от Microsoft, SOM от IBM и CORBA от Object Management Group – решают эту проблему путем предоставления интерфейса, скрывающего детали реализации и задуманного независимым от конкретного языка. Последнее вызывает у программиста на C++ некоторые неудобства и обычно сопровождается потерями программы скорости или памяти. Кроме того, у каждого из гигантов индустрии, похоже, есть свой собственный «стандарт» для решения описанной проблемы. Только время покажет, насколько эти методы помогают или мешают C++-программистам. Механизм пространств имен – подход к эволюции интерфейсов внутри самого C++ (см. раздел 17.4.4).

С большой неохотой я вынужден признать, что некоторые системные проблемы стоило бы решать в самом C++. Вообще-то такие вопросы, как динамическое связывание классов и эволюция интерфейсов, логически не относятся к языку, и встраивание в язык средств для их решения нежелательно. Но выработать по-настоящему стандартное решение можно только, работая с самим языком. Например, соглашения о вызове подпрограмм, написанных на Fortran и C, стали стандартом де-факто. Так случилось потому, что C и Fortran популярны, а интерфейсы вызова просты и эффективны. Для принятия стандарта это минимальный набор условий. Мне не нравится такой вывод: из него следует, что на пути использования разных языков в системе стоит барьер, если только один из языков не предоставит механизма, который другими будет принят в качестве стандарта.

9.4.4. За пределами файлов и синтаксиса

Как я вижу среду для разработки программ на C++? Прежде всего – инкрементная компиляция. Если вносится небольшое изменение, то система «понимает», что оно небольшое, и генерирует новую версию программы мгновенно. Моментальные ответы хотелось бы получать также на простые вопросы и указания типа: «Показать объявление `f`», «Какие еще `f` есть в области действия?», «Как разрешен этот вызов оператора `+`?», «Какие классы произведены от `Shape`?» и «Какие деструкторы вызываются в конце этого блока?»

В программе на C++ есть много информации, которая в типичной среде доступна только компилятору. Уверен, что она должна быть предоставлена программисту.

Однако люди в большинстве своем смотрят на C++-программу, как на набор исходных файлов или строк символов. Программа – это набор типов, функций, предложений и т.д. Данные понятия представляются в виде символов в файлах только для удобства изображения в традиционных средах программирования.

То, что в основе реализаций C++ лежат символьно-ориентированные инструменты, всегда было главным препятствием на пути развития языка. Если нужно препроцессировать и перекомпилировать каждый заголовочный файл, прямо или косвенно включенный в файл, где находится слегка измененная функция, то для этого требовалось определенное, пусть и небольшое время. Существует несколько методов, позволяющих избежать ненужных перекомпиляций, но, по-моему, наиболее перспективный и интересный подход – отказаться от традиционного исходного текста и положить в основу инструментов абстрактное внутреннее представление. Ранний вариант такого представления можно найти в работах [Murray, 1992], [Koenig, 1992]. Естественно, текст все равно необходим – его вводят и читают пользователи, – но он легко преобразуется системой во внутреннюю форму и реконструируется по запросу. Форматирование текста с соблюдением некоторых правил отступа – лишь один из многих возможных взглядов на программу. Простейшее применение этого замечания: текст программы, который каждый из пользователей видит отформатированным в своем любимом стиле, а вы – в вашем.

Нетекстовое представление могло бы быть создано языками более высокого уровня, генераторами программ, инструментами визуального программирования и т.д. Это позволило бы таким инструментам работать в обход обычного синтаксиса C++ и даже помогло бы избавить язык от некоторых неудачных особенностей его синтаксиса. Я утверждаю, что система типов и семантика C++ чище, чем его синтаксис.

Из представления о синтаксисе как об интерфейсе между языком и пользователем следует, что возможны и другие интерфейсы. Единственная фундаментальная константа – это базовая семантика языка. Она не должна меняться ни при каких обстоятельствах, и, учитывая это, вполне можно выдать C++-код в обычной текстовой форме по запросу.

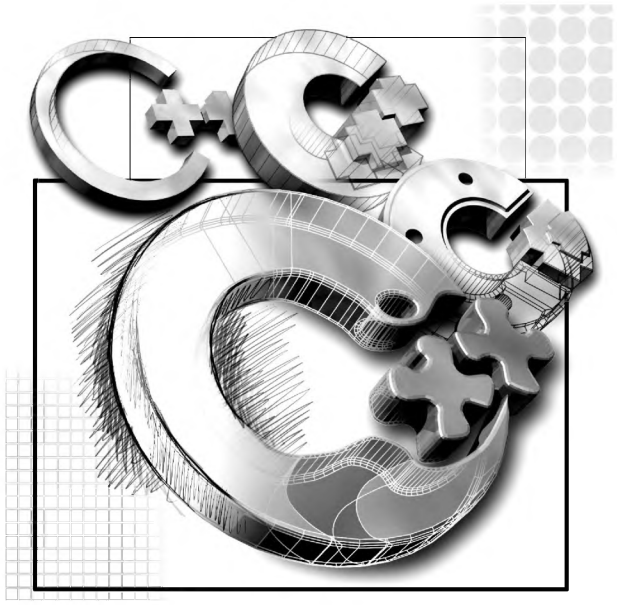
В среде, основанной на абстрактном представлении C++, можно было бы предоставлять альтернативные способы создания и просмотра программ, а равно и другие пути компоновки, компиляции и исполнения кода. Например, компоновка выполнялась бы до генерации кода, поскольку наличие объектного кода не являлось бы обязательным для доступа к информации о связях. Различие между интерпретатором и компилятором в значительной мере сгладилось бы, поскольку их работа была бы основана на одной и той же информации примерно в одном и том же формате.

9.4.5. Подведение итогов и перспективы

C++ представляет собой множество составляющих, и в этом его наиболее сильная сторона. Так же и развитие языка является следствием не одного какого-то улучшения, а многих усовершенствований в самых разных областях (улучшенные библиотеки, более совершенные методы проектирования, наличие стандарта

языка, необязательной сборки мусора, стандартов на передачу объектов, баз данных, нетекстовых сред разработки, более богатый набор инструментов, более быстрые компиляторы и т.д.).

Думается, что пока мы работали лишь с наименьшей частью возможностей языка C++. Я предвижу, что в будущем центр деятельности и развития сместится от собственно языка к инструментам, средам, библиотекам, приложениям и т.д., которые зависят от него и возводятся на его фундаменте.



Часть II

Глава 10. Управление памятью

Глава 11. Перегрузка

Глава 12. Множественное наследование

Глава 13. Уточнения понятия класса

Глава 14. Приведение типов

Глава 15. Шаблоны

Глава 16. Обработка исключений

Глава 17. Пространства имен

Глава 18. Препроцессор C

В части II описываются средства C++, появившиеся после выхода версии 1.0. Разбиение на главы основано на логических взаимосвязях между различными возможностями. Хронология их включения в C++ для языка в целом не существенна и здесь не отражена. Порядок глав тоже не играет большой роли.

Глава 10. Управление памятью

Никакому гению не преодолеть проклятия деталей.

Анонимный автор

10.1. Введение

В C++ есть оператор `new` для выделения памяти из кучи и оператор `delete` для освобождения выделенной памяти (см. раздел 2.11.2). Но иногда пользователю необходим более тщательный контроль над распределением и освобождением памяти.

Важный частный случай – выделение памяти на уровне часто используемого класса [2nd, стр. 177]. Многие программы создают и уничтожают большое число мелких объектов, принадлежащих нескольким важным классам, например узлы дерева или связанного списка, точки, линии, сообщения и т.д. На выделение и освобождение памяти для таких объектов с помощью распределителя общего назначения может уходить почти все время работы программы, причем память будет использоваться неэффективно. Здесь играют роль два фактора: затраты по времени и памяти, присущие универсальному распределителю, и фрагментация свободной памяти из-за того, что в ней создаются объекты разных размеров. Я обнаружил, что применение специализированного распределителя, определенного на уровне класса, обычно позволяет вдвое увеличить скорость действия симулятора, компилятора или аналогичной программы по сравнению с применением стандартного алгоритма управления памятью. Приходилось видеть и десятикратное ускорение в случаях, когда проблемы фрагментации были особенно острыми. Средства, включенные в версию 2.0, позволяют добавить в класс распределитель (написанный самостоятельно или взятый из стандартной библиотеки) всего за несколько минут.

Более тщательное управление необходимо также при работе с программами, которые должны функционировать без остановки в течение длительного времени и в условиях крайней ограниченности ресурсов. Аналогичные требования предъявляются к системам реального времени, в которых требуется гарантировать предсказуемое потребление памяти с минимальными затратами. Традиционно в таких программах вообще старались избегать динамической памяти. Но для управления ограниченными ресурсами можно применить специализированный распределитель.

И, наконец, я сталкивался со случаями, когда из-за требований аппаратуры или системы объект следовало разместить по конкретному адресу.

Вследствие всего этого механизм управления памятью в C++ (см. раздел 2.11.2) в версии 2.0 был пересмотрен. В основном усовершенствования коснулись

механизмов контроля над распределением памяти, при этом предполагалось, что программист понимает, как это происходит. Эти механизмы должны были работать совместно с другими языковыми средствами, чтобы инкапсулировать те фрагменты кода, где осуществляется нестандартное управление памятью. Работа над ними завершилась в 1992 г. введением операторов `operator new[]` и `operator delete[]` управления памятью для массивов.

В нескольких случаях предложения исходили от моих друзей из компании Mentor Graphics, где на C++ была создана большая и сложная система CAD/CAM. Арчи Лахнер из Mentor подал мне несколько блестящих идей об управлении памятью, которые были учтены в версии 2.0.

10.2. Отделение распределения памяти и инициализации

Применявшийся до выхода 2.0 способ управлять распределением и освобождением памяти на уровне класса с помощью присваивания указателю `this` (см. раздел 3.9.) провоцировал ошибки и был объявлен устаревшим. В версии 2.0 в качестве альтернативы допускались раздельные выделение и инициализация памяти. Это значит, что инициализация выполняется конструктором после того, как память уже выделена каким-то независимым механизмом. Такой прием позволяет использовать разные механизмы распределения, в том числе написанные пользователем. Память для статических объектов выделяется на стадии редактирования связей, память под локальные объекты распределяется из стека, а для динамических объектов – с помощью подходящего `operator new()`. Освобождение памяти производится аналогично. Например:

```
class X {
   // ...
public:
   void* operator new(size_t sz);    // выделить sz байт
   void operator delete(void* p);    // освободить p

   X();       // инициализировать
   X(int i); // инициализировать

   ~X();      // очистка
   // ...
};
```

Тип `size_t` – это зависящий от реализации интегральный тип, используемый для представления размеров объектов; он позаимствован из стандарта ANSI C.

Именно с помощью оператора `new` правильно выделяется и инициализируется память, если эти операции выполняются раздельно. Компилятор генерирует вызов распределителя памяти `X::operator new()` и вызов конструктора `X` в месте вызова `new` для `X`. Логически `X::operator new()` вызывается раньше конструктора. Поэтому он должен вернуть `void*`, а не `X*`. Конструктор создает объект уже в выделенной памяти.

Деструктор, наоборот, уничтожает объект, но не освобождает его память, полагая, что это сделает `operator delete()`. Поэтому `X::operator delete()` принимает аргумент типа `void*`, а не `X*`.

Применимы обычные правила наследования, поэтому память для объектов производного класса выделяется с помощью `operator new()` из базового класса:

```
class Y : public X {    // объекты класса Y также распределяются
                        // с помощью X::operator new()
   // ...
};
```

Чтобы этот фрагмент кода работал, `X::operator new()` нужен аргумент, указывающий, сколько памяти выделить: `sizeof(Y)` чаще всего не совпадает с `sizeof(X)`. К сожалению, неопытные пользователи обычно удивляются, когда узнают, что нужно объявить этот аргумент, но не передавать его явно при вызове. Идея об определенной пользователем функции, фактический аргумент которой передает «система», некоторыми усваивается с трудом. Но в качестве компенсации за излишнюю сложность базовый класс теперь может предоставлять услуги по выделению и освобождению памяти всем производным от него классам. Кроме того, не возникает ненужных исключений из правил наследования.

10.3. Выделение памяти для массива

Определенный в классе оператор `X::operator new()` применяется только к индивидуальным объектам класса `X` (включая и объекты производных от `X` классов, в которых нет собственного `operator new()`). Отсюда следует, что предложение

```
X* p = new X[10];
```

не порождает вызова `X::operator new()`, поскольку `X[10]` – массив, а не объект типа `X`.

Это стало причиной претензий, поскольку пользователи не могли управлять распределением памяти для массивов. Однако я настаивал на том, что массив объектов `X` – не то же самое, что `X`, а стало быть, использовать распределитель для класса `X` нельзя. Если бы его можно было применять к массивам, то автору `X::operator new()` пришлось бы специально рассматривать выделение памяти для массива «на всякий случай» и тем самым усложнять алгоритм для стандартной ситуации. А если алгоритм не является настолько нужным, то зачем вообще писать специализированный распределитель? Кроме того, я подчеркнул, что управления выделением памяти для одномерных массивов типа `X[d]` все равно недостаточно: как тогда быть с многомерными массивами `X[d1][d2]`?

Однако отсутствие механизма управления выделением памяти для массивов создавало неудобства в реальных ситуациях, так что комитет по стандартизации, в конце концов, предложил решение проблемы. Самым неприятным было то, что, с одной стороны, пользователю нельзя было запретить распределять массив в свободной памяти, с другой стороны, не было способа контролировать такое распределение. В системах, опирающихся на логически различные схемы управления

памятью, это могло стать причиной серьезных трудностей, поскольку пользователь по наивности размещал большие динамические массивы в области, предназначенной для размещения объектов.

Решение заключалось в том, чтобы просто определить пару функций специально для выделения и освобождения памяти под массивы:

```
class X {
   // ...
   void* operator new(size_t sz);    // выделение памяти для объектов
   void operator delete(void* p);

   void* operator new[](size_t sz);  // выделение памяти для массивов
   void operator delete[](void* p);
};
```

Распределитель памяти для массивов применим к массивам любой размерности. Как и любой другой распределитель, `operator new[]` должен предоставить запрошенное число байт; и не имеет значения, как будет использоваться эта память. В частности, распределителю не нужно задавать ни размерность массива, ни число элементов в нем. На введении операторов для выделения и освобождения памяти под массивы больше других настаивала Лаура Йекер (Layra Yaker) из компании Mentor Graphics.

10.4. Размещение объекта в памяти

С помощью единого механизма были решены две смежные проблемы:

- нужен механизм размещения объекта по фиксированному адресу. Например, объект, описывающий процесс, следует разместить по адресу, который диктует специализированная аппаратура;
- требуется механизм распределения памяти для объекта из конкретной арены. Например, нужно разместить объект в разделяемой памяти многопроцессорной системы или в области, управляемой диспетчером устойчивых объектов.

Решение – разрешить перегрузку `operator new()` и предоставить синтаксис для передачи оператору `new` дополнительных аргументов. Например, `operator new()` для размещения объекта по фиксированному адресу можно было определить так:

```
void* operator new(size_t, void* p)
{
   return p; // разместить объект по адресу 'p'
}
```

а вызывать следующим образом:

```
void* buf = (void*)0xF00F; // специальный адрес

X* p2 = new(buf)X;  // конструируем X в области 'buf'
                    // вызывается operator new(sizeof(X),buf)
```

Из-за такого использования синтаксис `new(buf)X` для передачи дополнительных аргументов оператору `operator new()` получил название «синтаксис размещения». Отметим, что первым аргументом любого `operator new()` является объем выделяемой памяти, и в данном случае размер объекта подставляется системой неявно.

В то время я, мягко говоря, недооценил важность оператора размещения. При наличии размещения оператор `new` перестает быть просто механизмом распределения памяти. Поскольку с фиксированным адресом можно ассоциировать произвольные логические свойства, то `new` приобретает черты универсального менеджера ресурсов.

Для конкретной арены `operator new()` можно определить так:

```
void* operator new(size_t s, fast_arena& a)
{
   return a.alloc(s);
}
```

а использовать следующим образом:

```
void f(fast_arena& arena)
{
   X* p = new(arena)X;  // распределить X на арене
   // ...
}
```

Здесь предполагается, что `fast_arena` – это класс, который имеет функцию-член `alloc()`, применяющуюся для выделения памяти. Например:

```
class fast_arena {
   // ...
   char* maxp;
   char* freep;
   char* expand(size_t s); // получить дополнительную память от
                           // распределителя общего назначения
public:
   void* alloc(size_t s) {
       char* p = freep;
       return ((freep+=s)<maxp) ? p : expand(s);
   }
   void free(void*) {}  // ничего не делает
   clear();             // освободить всю выделенную память
};
```

Такая арена специально предназначена для быстрого выделения памяти и почти мгновенного ее освобождения. Важной областью применения арен является предоставление специальной семантики управления памятью.

10.5. Проблемы освобождения памяти

Между операторами `operator new()` и `operator delete()` имеется очевидная и намеренная асимметрия: первый можно перегружать, второй – нет. Это

сравнимо с аналогичной асимметрией между конструкторами и деструкторами. Таким образом, при создании объекта вы можете выбрать, скажем, один из четырех распределителей памяти и один из пяти конструкторов, но во время уничтожения объекта в вашем распоряжении будет лишь один способ:

```
delete p;
```

Причина в том, что при создании объекта вы знаете о нем все, а при уничтожении имеется лишь указатель, который может принадлежать или не принадлежать точному типу объекта.

В случаях, когда пользователь уничтожает объект производного класса, пользуясь указателем на объект базового класса, не обойтись без виртуальных деструкторов:

```
class X {
   // ...
   virtual ~X();
};

class Y : public X {
   // ...
   ~Y();
};

void f(X* p1)
{
   X* p2 = new Y;
   delete p2;// вызывается Y::~Y
   delete p1;// вызывается правильный деструктор,
             // каким бы он ни оказался
}
```

Гарантируется также, что если в иерархии классов определены локальные операторы `operator delete()`, то будет вызван нужный. Если бы виртуальный деструктор не использовался, то действия по очистке, заданные в деструкторе класса `Y`, никогда не выполнялись бы.

Однако для выбора функции освобождения памяти в языке нет средств, парных механизму выбора функции распределения:

```
class X {
   // ...
   void* operator new(size_t);// обычное выделение памяти
   void* operator new(size_t, Arena&);    // из арены Arena

   void operator delete(void*);
   // нельзя определить void operator delete(void*, Arena&);
};
```

Ведь нельзя предполагать, что в точке уничтожения объекта известно, как он был создан. Идеальный вариант – тот, при котором пользователю вообще не нужно освобождать память, занятую объектом. Для этого, в частности, применяются

специальные арены. Арену можно определить так, что в некоторой известной точке программы будет освобождаться вся память, занятая этой программой. Можно также написать для нее сборщик мусора. Первый подход очень распространен, второй – нет. Специальный сборщик мусора должен быть написан удачно, иначе проще встроить в программу стандартный [Boehm, 1993].

Чаще при программировании функций `operator new()` оставляют некоторый признак, который доступен при вызове `operator delete()` и показывает, как следует освобождать память. Заметим, что все это относится к управлению памятью, следовательно, находится на более низком уровне, чем создание объектов с помощью конструкторов и их уничтожение деструкторами. Поэтому область памяти, где хранится такой признак, не является частью объекта, но как-то связана с ним. Например, `operator new()` мог бы поместить информацию, относящуюся к управлению памятью, в слово, предшествующее тому, на которое указывает возвращенное им значение. Также `operator new()` мог бы оставить эту информацию там, где конструкторы или другие функции имеют возможность найти ее и выяснить, распределена ли память для объекта из кучи.

Был ли запрет на перегрузку `delete()` ошибкой? Не уверен, но думаю, что этот случай из разряда тех, где любое решение порождает проблемы.

Возможность вызывать деструктор явно была введена в версии 2.0 для тех моментов, когда выделение и освобождение памяти абсолютно разъединены. Примером может служить контейнер, который предназначен для полного управления памятью хранящихся в нем объектов.

10.5.1. Освобождение памяти для массивов

В C++ единичный объект, обозначенный указателем, вполне может оказаться первым элементом массива. Так же было и в C. Если указатель указывает на первый элемент массива, то обычно говорят, что он указывает на массив. Обычно компилятор не может отличить указатели на единичный объект и на массив. Выделение и освобождение памяти для массивов производится именно через такие указатели. Например:

```
void f(X* p1)    // p1 может указывать на единичный
                 // объект или на массив
{
   X* p2 = new X[10];   // p2 указывает на массив
   // ...
}
```

Как можно гарантировать правильное удаление массива? В частности, как гарантировать, что для каждого элемента массива вызван деструктор? В версии 1.0 не было удовлетворительного ответа на эти вопросы. В версии 2.0 появился оператор удаления массива `delete[]`:

```
void f(X* p1)// p1 может указывать на единичный
             // объект или на массив
{
   X* p2 = new X[10];   // p2 указывает на массив
   // ...
   delete p2;           // ошибка: p2 указывает на массив
```

```
    delete[] p2;          // правильно
    delete p1;            // может, и правильно
    delete[] p1;          // может, и правильно
}
```

С помощью `delete` не всегда возможно освобождать память и для единичных объектов, и для массивов. Это помогает избежать лишних сложностей при обработке наиболее частого случая выделения и освобождения памяти для одного объекта, а также позволяет не загромождать объекты информацией, необходимой только для правильного удаления массивов.

В промежуточном варианте `delete[]` программист должен был указывать число элементов в массиве, например:

```
delete[10] p2;
```

Это слишком часто приводило к ошибкам, поэтому за хранение информации о числе элементов в массиве отвечал компилятор.

10.6. Нехватка памяти

Невозможность получить запрошенный ресурс – распространенная проблема. Еще до появления версии 2.0 стало понятно, что ее решение следует искать в области обработки исключений (см. раздел 3.15). Однако до обработки исключений в общем виде (см. главу 16) было еще далеко, а конкретную проблему нехватки памяти следовало решать немедленно. Для переходного периода, растянувшегося на несколько лет, нужно было предложить хоть какое-то решение.

Самыми животрепещущими являлись две проблемы:

- пользователю следует передать управление во всех случаях, когда вызов библиотечной функции завершается неудачей из-за нехватки памяти (или по любой другой причине). Это непререкаемое требование исходило от пользователей, работающих в AT&T;
- нельзя требовать, чтобы средний пользователь проверял ошибку после выполнения каждой операции выделения памяти. Практика работы с C показывает, что пользователи не проводят такую проверку систематически, даже если она необходима.

Первое требование было удовлетворено тем, что при нехватке памяти конструктор не вызывался, а `operator new()` возвращал `0`. В этом случае выражение `new` также принимало значение `0`, а важные приложения оказывались защищенными от нехватки памяти. Например:

```
void f()
{
    X* p = new X;
    if (p == 0) {
        // обработать ошибку выделения памяти
        // конструктор не вызывается
    }
    // использовать p
}
```

Для выполнения второго требования был введен обработчик new_handler – определенная пользователем функция, которая гарантированно вызывалась при невозможности выделить память оператором new. Например:

```
void my_handler() { /* ... */ }

void f()
{
   set_new_handler(&my_handler);     // начиная с этого момента для
                                     // обработки нехватки памяти
                                     // используется my_handler
   // ...
}
```

Этот прием описан в [Stroustrup, 1986] и является общим способом обработки ошибок при выделении ресурсов. С помощью обработчика new_handler можно:

- найти дополнительные ресурсы, например свободную память, которую можно выделить;
- вывести сообщение об ошибке и каким-то способом закончить выполнение программы.

Применение обработки исключений позволяет программе отреагировать на ошибку не столь радикальным способом, как завершение работы (см. раздел 16.5).

10.7. Автоматическая сборка мусора

При проектировании C++ я сознательно решил не полагаться на автоматическую сборку мусора, опасаясь весьма значительных издержек по времени и памяти, которые удалось наблюдать в различных сборщиках мусора. Также могли возникнуть дополнительные сложности при написании компилятора и переносе его на другие платформы. Кроме того, сборка мусора сделала бы C++ непригодным для многих низкоуровневых задач, ради которых язык создавался.

По-моему, если вам нужна сборка мусора, то вы можете либо сами реализовать некоторую автоматизированную схему управления памятью, либо воспользоваться языком, который поддерживает сборку мусора напрямую, например старым добрым Simula. Сегодня и для реализации, и для переноса имеется больше ресурсов. Существует много программ на C++, которые просто нельзя переписать на других языках. Технология сборки мусора усовершенствовалась, а многие из тех доморощенных схем, с которыми я работал, не перешли из отдельных проектов в библиотеки общего назначения. Но более важно то, что с помощью C++ реализуются все более важные проекты. В некоторых из них сборка мусора определенно была бы полезна, а связанные с ней затраты вполне терпимы.

10.7.1. Необязательный сборщик мусора

В C++ надо включить необязательный сборщик мусора. Некоторые реализации уже существуют, и переход из исследовательской стадии в промышленную эксплуатацию – лишь вопрос времени.

Главные причины, по которым наличие сборщика мусора желательно, таковы:

- сборка мусора упрощает построение и применение некоторых библиотек;
- в некоторых приложениях это средство более надежно, чем написанная пользователем схема управления памятью.

Аргументов «против» несколько больше, но они менее принципиальны, поскольку связаны с реализацией и эффективностью:

- сборка мусора сопряжена с такими издержками времени и памяти, которые неприемлемы для многих приложений C++, работающих на существующем оборудовании;
- многие методы сборки мусора вызывают временную приостановку обслуживания, что нетерпимо в таких приложениях, как системы реального времени, драйверы устройств, системы контроля, пользовательский интерфейс на медленно работающем оборудовании и ядра операционных систем;
- ряд приложений не имеет аппаратных ресурсов, типичных для обычного компьютера;
- некоторые схемы сборки мусора несовместимы с несколькими базовыми механизмами C, например с арифметическими действиями над указателями, массивами без контроля выхода за границу и неконтролируемыми аргументами функций (типа `printf`);
- определенные схемы сборки мусора налагают ограничения на способ размещения объекта в памяти или на создание объектов, которые существенно усложняющих интерфейс с другими языками.

Есть и другие аргументы «за» и «против» наличия сборщика мусора. Сравнивая системы, имеющие и не имеющие сборку мусора, следует помнить: не каждая программа должна работать без остановки; не любой код является частью базовой библиотеки; многие программы могут управлять памятью самостоятельно, без сборки мусора и схожих методов типа подсчета ссылок. C++ не нужна сборка мусора, так же как и языку без истинных локальных переменных (см. раздел 2.3). Если для управления памятью достаточно более конкретных методов (скажем, специализированных распределителей, см. разделы 10.2, 10.4, 15.3.1, [2nd, §5.5.6, §13.10.3]), а также автоматического и статического хранения (см. раздел 2.4), то можно получить существенный выигрыш во времени и памяти по сравнению с автоматической сборкой мусора и оператором освобождения памяти.

Общий вывод таков: сборка мусора желательна и возможна, но мы не можем позволить, чтобы от нее была зависима семантика C++ и большинства стандартных базовых библиотек.

Значит, реальная проблема заключается в том, стоит ли включать необязательный сборщик мусора для C++. Когда (не «если»!) сборщик мусора станет доступен, мы сможем писать программы на C++ двумя способами. В целом это не сложнее, чем управляться с несколькими библиотеками, поддерживать приложение на разных платформах и т.д. Необходимость делать такой выбор – обычное следствие самой природы языка общего назначения, который к тому же широко применяется. Бессмысленно требовать одинаковой среды выполнения для головки самонаводящейся ракеты, персонального компьютера, телефонного коммутатора,

клона UNIX, мейнфрейма IBM, компьютера MAC и т.д. и т.п. Правильно реализованная сборка мусора станет еще одной возможностью при выборе среды выполнения приложения.

Может ли данное средство стать законным и полезным, если его не специфицировать в стандарте C++? Видимо, да. Но мы не можем включить его в стандарт, поскольку не существует схемы, хоть в какой-то мере пригодной для стандартизации. Экспериментальные данные должны показать, что схема достаточно удобна для широкого круга реальных приложений. Кроме того, у нее не должно быть неустранимых недостатков, которые сделали бы C++ непригодным для применения в каком-либо важном классе приложений. При наличии хотя бы одного успешного эксперимента разработчики компиляторов рьяно возьмутся за поиск лучших реализаций. Нам остается только надеяться, что они не выберут несовместимые между собой схемы.

10.7.2. Как должен выглядеть необязательный сборщик мусора?

В идеале число программ, которые могут работать как со сборщиком мусора, так и без него, должно быть как можно большим. Это важная и труднодостижимая цель для разработчиков компиляторов, проектировщиков библиотек и программистов, решающих прикладные задачи.

Лучший вариант, если сборка мусора не используется, – компилятор со сборщиком мусора должен создавать такую же эффективную по времени и памяти программу, как и компилятор без данного средства. Это легко, если заставить программиста указать, что «никакая часть этой программы не использует сборку мусора», но очень трудно, если компилятор должен обеспечить максимальную эффективность за счет адаптирующегося алгоритма сборки мусора.

Наоборот, разработчику сборщика мусора могут потребоваться «подсказки» со стороны пользователя, чтобы обеспечить приемлемую производительность средства. Например, может потребоваться указание, для каких объектов нужна сборка мусора, а для каких – нет (например, когда они созданы в написанной на C или Fortran библиотеке, не поддерживающей сборку мусора). Если такие подсказки вообще возможны, то компилятор без сборщика мусора должен их игнорировать. Другое решение – предоставить способ быстро убрать подсказки из исходного текста.

Некоторые операции C++, например устаревшие приведения типов, объединения, состоящие из указателей и не-указателей, арифметические действия над указателями и т.д. плохо сочетаются со сборкой мусора. В удачно написанной C++-программе такие операции встречаются нечасто, поэтому возникает желание вообще запретить их. Таким образом, вступают в конфликт две возможности:

- запретить небезопасные операции, что делает программу надежней, а сборку мусора более эффективной;
- не запрещать ни одну C++-программу, правильную с точки зрения текущего определения языка;

Думается, что можно найти компромисс и изобрести такую схему сборки мусора, которая подойдет почти для любой корректной C++-программы, а при отсутствии небезопасных операций станет только эффективнее.

При реализации схемы сборки мусора следует решить, нужно ли вызывать деструктор для объекта, попавшего в мусор. В [2nd] я писал:

«Сборку мусора можно рассматривать как моделирование бесконечной памяти в ограниченном объеме. Имея это в виду, мы можем ответить на вопрос: «Должен ли сборщик мусора вызывать деструктор для каждого попавшего в мусор объекта?» Ответ – нет, так как объект просто помещен в свободную память, а не удален и не уничтожен. Если оценивать ситуацию под этим углом, то `delete` – это не более чем требование вызвать деструктор (вместе с извещением системы о том, что память объекта можно использовать повторно). Но что, если мы действительно хотим выполнить некоторое действие над объектом, распределенным в свободной памяти, но так и не удаленным? Данная проблема не касается объектов в статической и автоматической памяти: для них деструкторы всегда вызываются неявно. Еще отметим, что действия, выполняемые «во время сборки мусора», могут осуществляться в любое время, начиная с момента последнего использования объекта и до завершения работы программы. Значит, на момент выполнения этих действий неизвестно состояние программы.

Если же такого рода действия все же необходимы, то проблему их выполнения в неопределенный момент уничтожения можно решить с помощью сервера регистрации. Когда объекту нужно произвести какие-то действия в конце программы, он помещает свой адрес и указатель на функцию очистки в некоторый глобальный ассоциативный массив».

Описанная модель, конечно, работоспособна, но, быть может, вызов деструкторов сборщиком мусора все же удастся сделать достаточно простым. Это зависит от того, какие объекты попадают в мусор и какие действия выполняют деструкторы. К сожалению, данная проблема из числа тех, для которых трудно поставить реальный эксперимент, да и в других языках похожего опыта, кажется, нет.

Итак, я не думаю, что создать приемлемый для C++ механизм сборки мусора просто, но это не невозможно. А если учесть, сколько людей думают над данной проблемой, то наверняка скоро появится несколько решений.

Глава 11. Перегрузка

Дьявол прячется в деталях.

Анонимный автор

11.1. Введение

Операторы призваны обеспечить удобство нотации. Рассмотрим формулу `F=M*A` (сила = масса * ускорение). В учебниках по элементарной физике эта формула не записывается в виде `assign(F,multiply(M,A))`[1]. Если переменные могут иметь разные типы, то нужно определить, разрешать ли смешанную арифметику или требовать явного приведения операндов к общему типу. Например, если `M` имеет тип `int`, а `A` – тип `double`, то мы можем либо принять запись `M*A` и считать, что `M` нужно перед умножением привести к типу `double`, либо потребовать от программиста писать нечто вроде `double(M)*A`.

Для C++, как для C, Fortran и почти всех языков, применяемых для вычислений, выбран первый подход. И это чревато определенными осложнениями. С одной стороны, пользователи хотят естественных преобразований «без протестов» компилятора, с другой – некорректные преобразования могут привести к сбою в работе программы. Если добавить сюда требование совместимости с довольно хаотичной системой встроенных типов и преобразований C, то становится понятно, что проблема действительно трудна.

Стремление к гибкости и свободе выражения противоречит стремлению к безопасности, предсказуемости и простоте. В этой главе рассматривается, к каким усовершенствованиям механизмов перегрузки привел данный конфликт.

11.2. Разрешение перегрузки

Перегрузка имен функций и операторов в том виде, в каком она первоначально появилась в C++ (см. раздел 3.6), [Stroustrup, 1984b], завоевала популярность, но выявились и присущие механизму проблемы. В [Stroustrup, 1989b] так резюмированы улучшения, введенные в версию 2.0:

«Механизм перегрузки в C++ был пересмотрен. Цель – обеспечить разрешение типов, которые раньше считались «слишком похожими», и добиться независимости от порядка объявлений. Получившаяся в результате схема более выразительна и позволяет выявить больше ошибок неоднозначности».

---

[1] Некоторые предпочли бы F=MA, но объяснение того, как такая нотация могла бы работать (перегрузка отсутствующего пробела), выходит за рамки этой книги.

Благодаря более детальному механизму разрешения появилась возможность перегружать имена, принимая во внимание различия между `int` и `char`, `float` и `double`, `const` и не-`const`, а также различия между базовым и производным классами. Независимость от порядка позволила избавиться от ряда ошибок. Ниже все аспекты перегрузки рассматриваются по очереди, также объясняется, почему ключевое слово `overload` больше не употребляется.

11.2.1. Детальное разрешение

В первоначальном варианте правила перегрузки в C++ учитывали ограничения, присущие встроенным типам C [Kernighan, 1978]: не было значений типа `float` (точнее, `rvalue` такого типа), поскольку при вычислениях тип `float` сразу же расширялся до `double`. Аналогично не было и значений типа `char`, так как при каждом использовании `char` происходило расширение до `int`. Отсюда недовольство пользователей, вызванное невозможностью естественным образом создать библиотеку для вычислений с плавающей точкой одинарной точности, а также жалобы на то, что при работе с функциями, манипулирующими символами, легко сделать ошибки, которых можно было бы избежать.

Рассмотрим функцию вывода. Если мы не можем перегрузить ее на основе различия между `int` и `char`, приходится заводить два имени. В первоначальном варианте потоковой библиотеки (см. раздел 8.3.1.) были такие функции:

```
ostream& operator<<(int);  // вывод int (включая преобразованные
                           // char) в виде последовательности цифр
ostream& put(char c);      // вывод char в виде символов
```

Однако многие писали так:

```
cout<<'X';
```

и, естественно, удивлялись, почему выводится `88` (числовое значение ASCII `'X'`), а не обычный символ `X`.

Для решения проблемы правила типов в C++ были изменены таким образом, что типы `char` и `float` не подвергались расширению в механизме перегрузки. Помимо этого тип символьного литерала, например, `'X'` был определен как `char`. В то же время была принята появившаяся незадолго до этого в ANSI C нотация для записи литералов типа `unsigned` и `float`. Поэтому стала возможной следующая запись:

```
float abs(float);
double abs(double);
int abs(int);
unsigned abs(unsigned);
char abs(char);

void f()
{
   abs(1);       // abs(int)
   abs(1U);      // abs(unsigned)
   abs(1.0);     // abs(double)
```

```
    abs(1.0F);    // abs(float)
    abs('a');     // abs(char)
}
```

В C принято, что символьный литерал ('a') имеет тип `int`. Как ни странно, приписывание 'a' типа `char` в C++ не приводит к проблемам совместимости. Если не считать примера `sizeof('a')`, то любая конструкция, которая может быть записана в C и C++, приводит к одному и тому же результату.

При назначении символьному литералу типа `char` я отчасти опирался на сообщение Майка Тимана об опыте использования в GNU C++ флага компилятора, обеспечивающего такую интерпретацию.

С другой стороны, выяснилось, что разницу между `const` и не-`const` можно успешно использовать. Наглядным примером такой перегрузки является пара функций

```
char* strtok(char*, const char*);
const char* strtok(const char*, const char*);
```

в качестве альтернативы стандартной функции из библиотеки ANSI C

```
char* strtok(const char*, const char*);
```

Функция `strtok()` из библиотеки C возвращает подстроку константной строки, переданной в качестве первого аргумента. Описать данную подстроку без квалификатора `const` в C++ невозможно, поскольку это расценивается как неявное нарушение системы типов. С другой стороны, несовместимость с C должна быть сведена к минимуму, поэтому предоставление двух вариантов функции `strtok` представляется наиболее разумным вариантом.

Допущение перегрузки на основе наличия или отсутствия `const` – часть общей стратегии ужесточения правил употребления `const` (см. раздел 13.3).

Опыт показал, что при сопоставлении функций следует принимать во внимание иерархии, образованные в результате открытого наследования. При наличии выбора следует отдавать предпочтение преобразованию в самый низший в иерархии класс. Аргумент `void*` выбирается лишь в том случае, если никакой другой указатель не подходит. Например:

```
class B { /* ... */ };
class BB : public B { /* ... */ };
class BBB : public BB { /* ... */ };

void f(B*);
void f(BB*);
void f(void*);

void g(BBB* pbbb, BB* pbb, B* pb, int* pi)
{
    f(pbbb);      // f(BB*)
    f(pbb);       // f(BB*)
    f(pb);        // f(B*)
    f(pi);        // f(void*)
}
```

Данное правило разрешения неоднозначности соответствует правилу для вызова виртуальных функций, в соответствии с которым выбирается член самого нижнего в иерархии класса. Изменение было очевидным. В результате исчезли ошибки.

Правда, у этого правила есть одно интересное свойство. Оно устанавливает `void*` в качестве корня дерева преобразований классов. Это согласуется с представлением о том, что конструктор создает объект в неформатированной памяти, а деструктор превращает объект в нее (см. разделы 2.11.1 и 10.2). Преобразование вида «`B*` в `void*`» позволяет рассматривать объект как неформатированную память, если никакие другие его свойства не представляют интереса.

11.2.2. Управление неоднозначностью

В первоначальном механизме перегрузки разрешение неоднозначностей зависело от порядка объявлений. Объявления рассматривались поочередно, и предпочтение отдавалось тому, которое встретилось в тексте раньше. Чтобы избежать недоразумений, при сопоставлении принимались во внимание только преобразования, не сужающие тип. Например:

```
overload void print(int);  // первоначальные (до 2.0) правила:
void print(double);

void g()
{
   print(2.0);             // print(double): print(2.0)
                           // преобразование double->int не
                           // рассматривается
   print(2.0F);            // print(double): print(double(2.0F))
                           // преобразование float->int не
                           // рассматривается
                           // преобразование float->double
                           // рассматривается
   print(2);               // print(int): print(2)
}
```

Это правило легко формулируется, эффективно компилируется, доступно для понимания, тривиально реализуется в компиляторе, но является постоянным источником ошибок. Изменение порядка объявлений на обратный полностью меняет смысл кода:

```
overload void print(double);  // первоначальные правила:
void print(int);

void g()
{
   print(2.0);   // print(double): print(2.0)
   print(2.0F);  // print(double): print(double(2.0F))
                 // преобразование float->double
                 // рассматривается
```

```
    print(2);     // print(double): print(double(2))
                  // преобразование int->double
                  // рассматривается
}
```

Таким образом, зависимость от порядка чревата ошибками. Это стало серьезным препятствием на пути эволюции C++ в направлении более интенсивного использования библиотек. Моей цели – распространить взгляд на программирование как на составление программ из независимо разработанных кусков (см. также раздел 11.3) – наряду со многими другими факторами препятствовала и зависимость от порядка.

Независимые от порядка правила перегрузки весьма усложняют определение и реализацию C++, поскольку следует поддерживать совместимость как с C, так и с предыдущим вариантом C++. В частности, не годилось простое правило: «Выражение, которое может быть корректно интерпретировано хотя бы двумя разными способами, неоднозначно и, стало быть, незаконно». Если принять данный тезис, то незаконными будут все вызовы `print()`, приведенные в примере выше.

Я решил, что требуется некое правило «лучшего соответствия», которое позволило бы предпочесть точное соответствие типов соответствию, требующему преобразований, а безопасные преобразования (типа `float` в `double`) – небезопасным (сужающим тип, искажающим значение и т.д., например, `float` в `int`). В результате обсуждения, уточнения и ревизии продолжались несколько лет. Некоторые детали все еще обсуждаются комитетом по стандартизации. Вместе со мной наиболее активное участие в процессе принимали Дуг Макилрой, Энди Кениг, Джонатан Шопиро. Еще на ранней стадии Дуг заметил, что мы чересчур близко подошли к попытке спроектировать «естественную» систему для неявных преобразований. Он считал правила PL/I (которые помогал проектировать сам) доказательством того, что такую «естественную» систему нельзя создать для богатого набора типов данных – а в C++ есть весьма богатый набор встроенных типов с нерегулируемыми правилами преобразований, а также возможность определять преобразования между произвольными типами, определенными пользователем. Желание сохранить совместимость с C, ожидания пользователей, намерение разрешить им определять типы, с которыми можно было бы работать точно так же, как с встроенными, – все это не позволяло запретить неявные преобразования. Полагаю, что решение оставить неявные преобразования было правильным. Я также согласен с наблюдением Дуга, что задача свести к минимуму количество неприятностей, которые могут преподнести неявные преобразования, сложна по существу. Согласен я и с тем, что полностью их избежать не удастся (по крайней мере, при соблюдении требования о совместимости с C). Просто у программистов разные ожидания, поэтому, какое правило ни выбери, для кого-то оно обернется неприятным сюрпризом.

Фундаментальная проблема заключается в том, что граф неявных преобразований встроенных типов содержит циклы. Так, существует неявное преобразование не только из `char` в `int`, но и из `int` в `char`. Это потенциальный источник бесконечного числа мелких ошибок. Поэтому было решено отказаться от схемы, основанной на графе преобразований. Вместо нее мы изобрели систему соответствий

между переданными при вызове типами формальных и фактических аргументов. Предпочтительными считались такие соответствия, при которых выполнялись наименее опасные преобразования. Это позволило избежать противоречий со стандартными правилами расширения типов и стандартными преобразованиями в C. В [Stroustrup, 1989b] для версии 2.0 данная схема описана так:

«...за исключением немногих случаев, когда старые правила могли зависеть от порядка, новые совместимы, и ранее написанные программы дадут те же результаты, что и до введения новых правил. Компиляторы C++, выпущенные за последние два года, выдавали предупреждения по поводу тех зависящих от порядка разрешений неоднозначности, которые теперь объявлены «вне закона».

В C++ различаются пять видов соответствий:

- не требующее никаких или требующее только неизбежных преобразований (например, имени массива в указатель, имени функции в указатель на функцию и `T` в `const T`);
- нуждающееся в расширении интегральных типов (в соответствии со стандартом ANSI C, то есть: `char` в `int`, `short` в `int` и их `unsigned` аналогов), а также `float` в `double`;
- требующее стандартных преобразований (`int` в `double`, `unsigned int` в `int`, `derived*` в `base*` и т.п.);
- работающее с определенными пользователем преобразованиями (конструкторами и конверторами);
- учитывающее многоточия (`...`) в объявлении функции.

Рассмотрим сначала функции с одним аргументом. Всегда нужно выбирать лучшее соответствие, а именно то, которое расположено выше в приведенном списке. Если есть два лучших соответствия, то вызов признается неоднозначным и компилятор выдает ошибку».

Приведенные выше примеры иллюстрируют правило. Более точное его изложение можно найти в ARM.

Следующее правило необходимо соблюдать, когда функция имеет более одного аргумента [ARM]:

«Если при вызове передается более одного аргумента, то выбирается функция, у которой хотя бы для одного из них соответствие лучше, а для остальных не хуже. Например:

```
class complex {
   // ...
   complex(double);
};

void f(int,double);
void f(double,int);
void f(complex,int);
void f(int ...);
void f(complex ...);

void g(complex z)
{
   f(1,2.0); // f(int,double)
   f(1.0,2);     // f(double,int)
   f(z,1.2);     // f(complex,int)
```

```
    f(z,1,3);      // f(complex ...)
    f(2.0,z);      // f(int ...)
    f(1,1);        // ошибка: неоднозначность
                   // f(int,double) или f(double,int)?
}
```

Нежелательное сужение типа `double` до `int` в третьем и предпоследнем вызовах влечет за собой предупреждение компилятора. Такое сужение разрешено в целях сохранения совместимости с C. В данном случае сужение безвредно, но во многих других может привести к искажению значения, поэтому необоснованно применять его не стоит».

Уточнение формулировки этого правила для нескольких аргументов привело к появлению «правила пересечения» [ARM, стр. 312–314]. Впервые его сформулировал Эндрю Кениг в беседе с Дугом Макилроем, Джонатаном Шопиро и мной. Кажется, именно Шопиро обнаружил странные примеры, доказавшие необходимость этого правила [ARM, стр. 313].

Обратите внимание на то, как серьезно мы подходили к вопросам совместимости. Если бы, принимая проектные решения, мы систематически отдавали предпочтение простоте и изяществу, а не совместимости, то сегодня C++ был бы куда компактнее и чище. Но в то же время он был бы языком, интересным лишь малой горстке ценителей.

11.2.3. Нулевой указатель

Кажется, ничто не вызвало более жарких споров, чем вопрос о том, как правильно обозначить нулевой указатель (не указывающий ни на какой объект). C++ унаследовал определение нулевого указателя от классического C [Kernighan, 1978]:

«Константное выражение со значением `0` преобразуется в указатель, обычно называемый нулевым. Гарантируется, что такой указатель можно отличить от указателя на любой объект или функцию».

ARM дополнительно предупреждает:

«Отметим, что нулевой указатель не обязательно представляется той же комбинацией битов, что и целый `0`».

Это предостережение против распространенного заблуждения, что если выражение `p=0` «делает» указатель `p` нулевым, то представление нулевого указателя должно быть в точности таким, как у целого нуля, то есть состоять из одних нулевых битов. Но C++ достаточно сильно типизирован, для того чтобы позволить разработчику компилятора выбрать для нулевого указателя любое представление, независимо от того, как он выглядит в исходном тексте программы. Одно из исключений – использование многоточия для подавления контроля аргументов функции:

```
int printf(const char* ...);  // неконтролируемый вызов в стиле C

printf(fmt, 0, (char)0, (void*)0, (int*)0, (int(*)())0);
```

Здесь необходимы приведения типов для явного указания, какой `0` требуется. В этом примере могло бы быть передано пять разных значений.

В K&R C аргументы функции вообще не проверялись, и даже в ANSI C нельзя полагаться на контроль аргументов, поскольку он необязателен. Из-за этого обнаружить все нули в программе на C или C++ нелегко. А поскольку в других языках принято использование символической константы для представления нулевого указателя, то программисты на C традиционно применяли для этой цели макрос `NULL`. К сожалению, в K&R C нет корректного переносимого определения `NULL`. В ANSI C разумным и все чаще употребляющимся определением `NULL` является `(void*)0`.

Однако в C++ и `(void*)0` нельзя счесть удачным обозначением нулевого указателя:

```
char* p = (void*)0; // корректно в C, но не в C++
```

Указатель `void*` нельзя присвоить ничему без явного приведения типа. Если разрешить неявное преобразование `void*` к другому типу указателя, то появится серьезная прореха в системе типов. Можно было бы считать `(void*)0` специальным случаем, но вводить его стоит только тогда, когда ничего другого не остается. Кроме того, порядок применения C++ был определен задолго до принятия стандарта ANSI C, и я не хочу, чтобы особо важная часть C++ зависела от макросов (см. главу 18). Поэтому я использовал просто `0`. Те, кто не может жить без символической константы, обычно определяют нечто похожее на

```
const int NULL = 0; // или
#define NULL 0
```

Для компилятора определенный таким образом `NULL` и `0` – синонимы. К сожалению, многие включили в свой код определения `NULL`, `NIL`, `Null`, `null` и т.д., так что вводить еще одно просто опасно.

Есть одна ошибка, которую невозможно обнаружить, если `0` (как бы он ни был назван) используется в качестве нулевого указателя. Рассмотрим пример:

```
void f(char*);

void g() { f(0); }  // вызывает f(char*)
```

Если добавить еще одну функцию `f()`, то семантика `g()` изменится без предупреждения:

```
void f(char*);
void f(int);

void g() { f(0); }  // вызывает f(int)
```

Этот неприятный побочный эффект возникает из-за того, что `0` – это значение типа `int`, которое может быть расширено до нулевого указателя, а не прямое обозначение последнего. Думаю, что хороший компилятор мог бы выдать предупреждение, но во время работы над Cfront я об этом не подумал. Можно было бы считать вызов `f(0)` неоднозначным, а не разрешать его в пользу `f(int)`, но это могло бы не понравиться тем, кто хочет видеть особый смысл в `NULL` или `nil`.

После очередной перепалки в comp.lang.c++ и comp.lang.c один из моих друзей заметил: «Если проблема нулевого указателя для них самая страшная, то им просто повезло». Опыт показывает, что использование `0` для обозначения нулевого указателя на практике не вызывает сложностей. Но меня по-прежнему удивляет правило, согласно которому результат вычисления любого константного выражения, равный `0`, принимается в качестве нулевого указателя. Согласно этому правилу, `2-2` и `~-1` – нулевые указатели. Но ведь ясно, что присвоение `2+2` или `-1` указателю – ошибка типизации. Безусловно, разработчикам компиляторов такое правило тоже не нравится.

11.2.4. Ключевое слово overload

В первоначальном варианте в C++ допускалось использование одного имени для обозначения двух функций только после объявления `overload`. Например:

```
overload max;       // "overload" - устарело в 2.0
int max(int,int);
double max(double,double);
```

Я счел слишком опасным такое использование без явного декларирования намерения. Например:

```
int abs(int);       // "overload abs" отсутствует
double abs(double); // раньше считалось ошибкой
```

Тому было две причины:

- опасение необнаруженных неоднозначностей;
- потенциальная возможность того, что программа может быть некорректно скомпонована, если программист явно не объявит, какие функции следует перегружать.

Первое соображение оказалось беспочвенным. Те немногие проблемы, с которыми все же пришлось столкнуться, устранялись за счет применения независимых от порядка правил разрешения перегрузки. Второе опасение действительно связано с общей сложностью, возникающей из-за правил раздельной компиляции в C, но к перегрузке это отношения не имеет (см. раздел 11.3).

С другой стороны, сами объявления `overload` стали источником серьезной проблемы: нельзя объединить два независимо разработанных фрагмента программы, в которых одно и то же имя используется для разных функций, если в обоих случаях оно не объявлено как перегруженное. Но так обычно не происходит; чаще всего нужно перегрузить имя библиотечной функции C, объявленное в заголовочном файле, например:

```
/* Заголовочный файл стандартной математической библиотеки math.h */
   double sqrt(double);
   /* ... */
// Заголовочный файл библиотеки для работы с комплексными числами
// complex.h
   overload sqrt;
   complex sqrt(complex);
   // ...
```

Теперь можно написать

```
#include <complex.h>
#include <math.h>
```

но не

```
#include <math.h>
#include <complex.h>
```

поскольку использование слова `overload` только во втором объявлении – ошибка. Эту проблему удастся смягчить, изменив порядок объявлений, введя ограничения на использование заголовочных файлов или вставки `overload` повсюду. Но все-таки гораздо лучшим решением представлялся отказ от объявлений `overload` и самого ключевого слова.

11.3. Типобезопасная компоновка

Компоновка в C проста и абсолютно небезопасна. Вы объявляете функцию

```
extern void f(char);
```

и компоновщик связывает эту `f` со всем, что найдет. А обнаружить он может функцию с совершенно другим числом аргументов или даже вовсе не функцию. Обычно это приводит к ошибке во время выполнения (нарушению защиты и т.д.). Особенно неприятными проблемы компоновки становятся при увеличении программы и интенсивном использовании библиотек. Программисты на C научились с этим справляться. Однако с появлением механизма перегрузки задачу надо было срочно решать, но так, чтобы оставалась возможность вызывать написанные на C функции без дополнительных сложностей и лишних затрат.

11.3.1. Перегрузка и компоновка

До версии 2.0 проблема компоновки программ, написанных на смеси C и C++, решалась так: имя, сгенерированное для C++-функции, должно по возможности быть таким же, как было бы сгенерировано для одноименной C-функции. Поэтому функция `open()` получила бы имя `open` в системе, где C не модифицирует имена, и `_open` – в системе, где в начало добавляется подчеркивание, и т.д.

Для перегруженных функций такой простой схемы недостаточно. Частично ключевое слово `overload` и было введено для того, чтобы отличить трудный случай от легкого (см. также раздел 3.6).

В основе первоначального решения, как и последующих, лежала идея о том, чтобы закодировать информацию о типе в имени, с которым работает компоновщик (см. раздел 3.3.3). Чтобы оставить возможность компоновки с C-функцией, кодировались имена только второго и последующих вариантов перегруженной функции. Стало быть, программисту приходилось писать:

```
overload sqrt;
double sqrt(double);    // компоновщику доступно: sqrt
complex sqrt(complex); // компоновщику доступно: sqrt__F7complex
```

Компилятор C++ генерировал код. Он ссылался на `sqrt` и `sqrt__F7complex`. К счастью, это было задокументировано только в разделе «Ошибки» руководства по C++.

Использовавшаяся до 2.0 схема перегрузки взаимодействовала с традиционной схемой компоновки C-программ таким способом, при котором становились очевидными худшие стороны того и другого. Нам пришлось решать три проблемы:

- отсутствие у компоновщика возможности контролировать типы;
- использование ключевого слова `overload`;
- связывание фрагментов программы, написанных на C и C++.

Первая проблема решается путем добавления к имени каждой функции информации о ее типе. Вторая – отказом от слова `overload`. Для решения третьей программисту на C++ необходимо явно указать, когда функция должна связываться в стиле C. Поэтому в [Stroustrup, 1988a] я писал:

«Вопрос в том, можно ли реализовать решение, основанное на этих трех посылках, без заметных затрат и с минимальными неудобствами для программистов на C++. Идеальное решение должно:

- не требовать никаких изменений в C++;
- обеспечить типобезопасную компоновку;
- допускать простое и удобное связывание с C-функциями;
- не «ломать» существующий код на C++;
- разрешать использование заголовочных файлов C (в стиле ANSI);
- обеспечивать хорошее обнаружение и диагностику ошибок;
- быть удобным инструментом для построения библиотек;
- не приводить к издержкам во время выполнения, компиляции и компоновки.

Нам не удалось найти решение, которое бы строго удовлетворяло всем этим условиям, но принятая схема является неплохим вариантом».

Ясно, что во время компоновки потребовалось бы проверять все типы. Но тогда возникал вопрос, как это сделать без написания нового компоновщика для каждой системы.

11.3.2. Реализация компоновки в C++

Имя каждой C++-функции кодируется путем добавления к нему типов аргументов. Этим гарантируется, что программа будет скомпонована только в том случае, если у каждой функции есть определение и типы аргументов в объявлении и определении совпадают. Рассмотрим, например, объявления

```
f(int i) { /* ... */ }              // определяет f__Fi
f(int i, char* j) { /* ... */ }     // определяет f__FiPc
```

которые можно правильно обработать:

```
extern f(int);              // относится к f__Fi
extern f(int,char*);        // относится к f__FiPc
extern f(double,double);    // относится к f__Fdd
```

```
void g()
{
   f(1);          // связывается с f__Fi
   f(1,"asdf");   // связывается с f__FiPc
   f(1,1);        // попытка связать с f__Fdd
                  // ошибка во время компоновки:
                  // f_Fdd не определена
}
```

Это оставляет открытым вопрос о том, как вызвать C- или C++-функции. Для этого программист должен указать, что функция имеет C-компоновку. В противном случае предполагается, что это C++-функция, и ее имя кодируется. С этой целью в C++ была введена спецификация компоновки:

```
extern "C" {
   double sqrt(double); // sqrt(double) имеет C-компоновку
}
```

Спецификация компоновки не затрагивает семантику программы, в которой используется `sqrt()`, а лишь дает компилятору информацию, что при генерировании объектного кода он должен использовать для `sqrt()` принятые в C соглашения об именах. Это означает, что имя данного экземпляра `sqrt()` будет равно `sqrt` или `_sqrt`, или иному варианту, в зависимости от конкретной системы. Можно представить себе систему, в которой для C приняты правила типобезопасной компоновки из C++, так что видимое компоновщику имя функции `sqrt()` будет `sqrt__Fd`.

Естественно, кодирование типа с помощью добавления суффикса – лишь один из возможных способов реализации, но он был успешно применен в Cfront, а затем широко растиражирован. У этого способа есть важные свойства: простота и совместимость с существующими компоновщиками. Такая реализация идеи типобезопасной компоновки не является абсолютно безопасной, но ведь в любом случае лишь очень немногие из полезных систем безопасны на 100%. Более полное описание схемы кодирования имен, примененной в Cfront, приводится в [ARM, §7.2c].

11.3.3. Анализ пройденного пути

Видимо, мы правильно определили приоритеты: типобезопасная компоновка, наличие разумной реализации и возможность явной компоновки с программами, написанными на других языках. Как и ожидалось, с помощью новой системы компоновки решался целый ряд проблем. Помимо всего прочего, в процессе перехода к новому стилю обнаружилось на удивление большое число ошибок при компоновке старых программ, написанных на C и C++. Я тогда отметил: «При переходе на типобезопасную компоновку чувствуешь себя, как после первой проверки C-программы с помощью `lint`, – несколько обескураженно». `Lint` – это популярный инструмент проверки раздельно компилируемых частей C-программы на предмет непротиворечивого использования типов [Kernighan, 1984]. Во время опытной эксплуатации типобезопасной компоновки я пытался отслеживать результаты ее

работы. С помощью данного средства выявлялись необнаруженные ошибки в каждой из больших программ на C и C++, которые мы пытались откомпилировать и связать.

К нашему удивлению, некоторые программисты сознательно вносили ошибки в объявления функций, просто чтобы подавить сообщения об ошибках. Например, вызов `f(1,a)` приводит к ошибке, если `f()` не объявлена. Я наивно ожидал, что в таком случае программист либо добавит правильное объявление функции, либо включит его в заголовочный файл. Оказалось, что была даже третья возможность – просто поместить любое объявление, не противоречащее вызову:

```
void g()
{
   void f(int ...); // чтобы подавить сообщение об ошибке
   // ...
   f(1,a);
}
```

Типобезопасная компоновка выдает сообщение об ошибке, если объявление не соответствует определению.

Также была обнаружена проблема переносимости. Многие объявляли библиотечные функции прямо в коде, вместо того чтобы включить нужный заголовочный файл. Полагаю, что это делалось для уменьшения времени компиляции, но в результате при переносе на другую систему объявление оказывалось неверным. Типобезопасная компоновка позволила нам выявить целый ряд таких проблем переносимости (в основном между UNIX System V и BSD UNIX).

Прежде чем остановиться на той схеме, которая включена в язык, был рассмотрен ряд других возможностей [Stroustrup, 1988]:

- не вводить явных директив компоновки, а положиться на инструментальные средства при связывании с C-функциями;
- выполнять типобезопасную компоновку и перегрузку только для функций, явно помеченных ключевым словом `overload`;
- осуществлять типобезопасную компоновку только для функций, которые никак не могли быть C-функциями, поскольку их типы нельзя выразить на C.

Опыт использования принятой схемы убедил меня в том, что проблемы, которых я опасался в случае выбора альтернативного решения, были вполне реальными. Например, распространение контроля на все функции стало благом, а программирование на смеси C и C++ оказалось настолько популярным, что любое усложнение совместной компоновки было бы воспринято болезненно.

Две особенности вызывали со стороны пользователей нарекания, которые не утихли до сих пор. В первом случае я считаю, что мы были правы, а относительно второго не уверен.

Функция, объявленная как имеющая C-компоновку, по-прежнему обладает семантикой вызова, принятой в C++. Это значит, что формальные аргументы должны быть объявлены, а фактические – соответствовать им с учетом правил сопоставления и разрешения неоднозначности, действующих в C++. Некоторые пользователи хотели бы, чтобы функции с C-компоновкой подчинялись правилам

вызова C. В таком случае можно было упростить использование заголовочных файлов C. Но это же позволило бы небрежным программистам вернуться к ослабленному контролю типов, характерному для C. Еще один аргумент против специальных правил для C связан с тем, что другие программисты высказывали такие же просьбы для компоновки с Pascal, Fortran и PL/I с поддержкой соответствующих правил вызова. Например, для функций с Pascal-компоновкой предлагалось неявно преобразовывать C-строки в Pascal-строки; для функций с Fortran-компоновкой – реализовать вызов по ссылке и добавлять информацию о типе массива и т.д. Если бы мы предоставили специальные возможности для C, то были бы обязаны «наделить» компилятор C++ знанием соглашений о вызове, принятых в огромном количестве языков. Было правильным воспротивиться такому давлению, хотя включение подобных дополнительных услуг и помогло бы отдельным программистам, работающим на смеси языков. Располагая только лишь семантикой C++, многие обнаружили, что для построения интерфейсов с такими языками, как Pascal и Fortran, где поддерживается передача аргументов по ссылке, полезны ссылки C++ (см. раздел 3.7).

С другой стороны, акцентирование внимания только на компоновке породило определенную сложность. В нашем решении прямо не рассматривались проблемы среды, поддерживающей программирование на смеси языков, и указатели на функции с разными соглашениями о вызове. Пользуясь правилами компоновки C++, мы можем непосредственно выразить, каким соглашениям о вызове подчиняется написанная на C или C++ функция. Но нельзя просто сказать, что сама функция подчиняется соглашениям C++, а ее аргументы – соглашениям C. Можно выразить это косвенно [ARM, стр. 118], например:

```
typedef void (*PV)(void*,void*);

void* sort1(void*, unsigned, PV);
extern "C" void* sort2(void*, unsigned, PV);
```

Здесь `sort1()` имеет C++-компоновку, принимает указатель на функцию с C++-компоновкой; `sort2()` имеет C-компоновку, принимает указатель на функцию с C++-компоновкой. Это простые случаи. Другой пример:

```
extern "C" typedef void (*CPV)(void*,void*);

void* sort3(void*, unsigned, CPV);
extern "C" void* sort4(void*, unsigned, CPV);
```

Здесь `sort3()` имеет C++-компоновку и принимает указатель на функцию с C-компоновкой; `sort4()` имеет C-компоновку и принимает указатель на функцию с C-компоновкой. Это почти предел того, что можно выразить в языке. Альтернативы тоже не очень удачны: можно либо ввести соглашения о вызове в систему типов, либо использовать при вызове переходники для преобразования одних соглашений в другие.

Компоновка, межъязыковые вызовы и передача объектов из одного языка в другой – это непростые проблемы, у них много аспектов, зависящих от реализации. В данной области правила меняются по мере возникновения новых языков, аппаратных архитектур и методов реализации.

11.4. Создание и копирование объектов

Меня часто просили запретить некоторые операции в языке. Некоторые хотели оптимизировать реализацию классов, для чего необходимо снять разрешение на проведение над объектами классов таких операций, как копирование, наследование и распределение в стеке. В других случаях, когда с помощью объектов представлялись сущности реального мира, для обеспечения требуемой семантики не было нужды во всех операциях, которые поддерживает C++.

Ответ на все подобные просьбы был найден во время работы над версией 2.0. Если вы хотите что-то запретить, сделайте соответствующую операцию закрытой функцией-членом (см. раздел 2.10).

11.4.1. Контроль допустимости копирования

Чтобы запретить копирование объектов класса `X`, достаточно сделать закрытыми оператор присваивания и копирующий конструктор.

```
class X {
   X& operator=(const X&); // присваивание
   X(const X&);            // копирующий конструктор
   // ...
public:
   X(int);
   // ...
};

void f()
{
   X a(1);       // правильно: можно создавать объекты класса X
   X b = a;      // ошибка: X::X(const X&) закрыт
   b = a;        // ошибка: X::operator=(const X&) закрыт
}
```

Конечно, внутри функций-членов класса `X` может копировать объекты данного класса, но в реальных ситуациях это допустимо, а иногда и необходимо. Не помню, кто первый додумался до этого решения; скорее всего, не я [Stroustrup, 1986, стр. 172].

Считаю неправильным, что операции копирования определены по умолчанию и во многих своих классах запрещаю копирование объектов. Однако присваивание по умолчанию и копирующие конструкторы перешли в C++ от C и эти возможности часто применяются.

11.4.2. Управление распределением памяти

С помощью объявления ряда операций закрытыми можно добиться и других эффектов. Например, если объявить закрытым деструктор, то будет запрещено размещение объектов в стеке и в глобальной памяти, а также случайное употребление `delete`:

```
class On_free_store {
   ~On_free_store();    // закрытый деструктор
   // ...
```

```
public:
   static void free(On_free_store* p) { delete p; }
   // ...
};

On_free_store glob1;    // ошибка: деструктор закрыт

void f()
{
   On_free_store loc;   // ошибка: деструктор закрыт
   On_free_store* p = new On_free_store;  // правильно
   // ...
   delete p; // ошибка: деструктор закрыт
   On_free_store::free(p); // правильно
}
```

Разумеется, подобный класс, как правило, будет использоваться вместе с хорошо оптимизированным распределителем свободной памяти или при наличии некоторой семантики, выигрывающей оттого, что объект находится в свободной памяти.

Противоположного эффекта – разрешения глобальных и локальных переменных при запрете на размещение объектов в свободной памяти – можно достичь с помощью необычного использования `operator new()`:

```
class No_free_store {
   class Dummy { };
   void* operator new(size_t,Dummy);
   // ...
};

No_free_store glob2;    // правильно

void g()
{
   No_free_store loc;   // правильно
   No_free_store* p = new No_free_store;  // ошибка:
       // No_free_store::operator new(size_t) отсутствует
}
```

11.4.3. Управление наследованием

Закрытый деструктор предотвращает также и наследование. Например:

```
class D : public On_free_store {
   // ...
};

D d;  // ошибка: нельзя вызвать закрытый деструктор базового класса
```

Это позволяет классу с закрытым деструктором стать логическим дополнением абстрактного класса. Наследовать классу `On_free_store` невозможно, поэтому

при вызовах виртуальных функций данного класса необязательно использовать механизм виртуализации. Однако я не думаю, что какой-нибудь из современных компиляторов выполняет такую оптимизацию.

Впоследствии Эндрю Кениг обнаружил, что можно запретить наследование даже вне зависимости от того, в какой памяти может размещаться объект:

```
class Usable_lock {
   friend Usable;
private:
   Usable_lock() {}
};

class Usable : public virtual Usable_lock {
   // ...
public:
   Usable();
   Usable(char*);
   // ...
};

Usable a;

class DD : public Usable { };

DD dd; // ошибка: DD::DD() недоступен
       // Usable_lock::Usable_lock() - закрытый член
```

Этот пример основан на правиле, что производный класс может вызывать конструктор виртуального базового класса (явно или неявно).

Однако такие примеры – скорее, предмет для интеллектуальных дебатов, чем для применения на практике.

11.4.4. Почленное копирование

Первоначально присваивание и инициализация были по умолчанию определены как побитовое копирование. Это приводило к неприятностям, когда объект класса, в котором определен оператор присваивания, использовался в качестве члена класса, в котором такой оператор не был определен:

```
class X { /* ... */ X& operator=(const X&); };

struct Y { X a; };

void f(Y y1, Y y2)
{
   y1 = y2;
}
```

Здесь `y2.a` копируется в `y1.a` побитово. Очевидно, что это неправильно и является результатом недосмотра при проектировании оператора присваивания и копирующего конструктора. После ряда споров и упреков со стороны Эндрю

Кенига было принято простое решение: копирование объектов определено как почленное копирование не-статических членов и объектов базовых классов.

Согласно этому определению, `x=y` означает то же, что `x.operator=(y)`. Но здесь есть одно интересное (хотя и не всегда желательное) следствие. Рассмотрим пример:

```
class X { /* ... */ };
class Y : public X { /* ... */ };

void g(X x, Y y)
{
   x = y; // x.operator=(y): правильно
   y = x; // y.operator=(x): ошибка x - не объект класса Y
}
```

По умолчанию присваивание объекту класса `X` – это `X::operator=(const X&)`, поэтому `x=y` – законная операция, ибо `Y` открыто наследует `X`. Обычно такую операцию называют вырезанием (slicing), так как `x` присваивается «срез» `y`. Копирующие конструкторы трактуются аналогично.

С практической точки зрения вырезание представляется мне сомнительным трюком, но я не вижу способа запретить его, не вводя слишком уж специальных правил. Кроме того, в то время Рави Сетхи как раз просил меня ввести именно такую «семантику вырезания», она была нужна ему для теоретических и педагогических целей: если не разрешить присваивание объекта производного класса объекту его открытого базового класса, то это будет единственным местом в C++, где производный объект нельзя использовать вместо базового.

Но при этом остается открытой проблема, связанная с операциями копирования по умолчанию: указатели-члены копируются, а объекты указания – нет. Это почти всегда неправильно, но запретить операцию нельзя из-за совместимости с C. Однако компилятор легко может выдать предупреждение, если класс с указателем-членом инициализируется при помощи копирующего конструктора или оператора присваивания по умолчанию. Например:

```
class String {
   char *p;
   int sz;
public:
   // здесь не определены операции копирования (небрежность)
};

void f(const String& s)
{
   String s2 = s;    // предупреждение: копируется указатель
   s2 = s;           // предупреждение: копируется указатель
}
```

Семантику присваивания и копирования по умолчанию можно назвать поверхностным копированием. Иными словами, копируются члены класса, но не объекты, на которые эти члены указывают. Альтернативу – рекурсивное копирование

указываемых объектов (так называемое глубокое копирование) – следует определять явно. Поскольку объекты могут указывать сами на себя, то вряд ли возможно другое решение. В обычном случае не стоит пытаться определять присваивание как глубокое копирование; гораздо лучше определить виртуальную (или невиртуальную) функцию копирования (см. [2nd, стр. 217–220] и раздел 13.7).

11.5. Удобство нотации

Я хотел позволить пользователю самому определять смысл каждого оператора, если только это разумно и не вступает в серьезное противоречие с предопределенной семантикой. Было бы проще разрешить перегрузку всех без исключения операторов либо запретить перегрузку любого оператора, имеющего предопределенную семантику для объектов класса. Принятое в результате компромиссное решение устраивает не всех.

Почти все споры и подавляющее большинство сложностей, с которыми мы столкнулись, относятся к операторам, которые не укладываются в привычную схему бинарных или префиксных арифметических операторов.

11.5.1. «Умные» указатели

До версии 2.0 пользователи не могли переопределять оператор разыменования `->`. Это осложняло создание классов объектов, ведущих себя как «умные» указатели. При определении перегрузки операторов `->` виделся мне как бинарный оператор со специальными правилами для правого операнда (имени члена). В этой связи вспоминается встреча в компании Mentor Graphics в Орегоне, когда Джим Ховард (Jim Howard) доказал мне, что я заблуждался. Оператор `->`, объяснил он, можно рассматривать как унарный постфиксный оператор, результат которого применяется к имени члена. Пересматривая механизм перегрузки, я воспользовался этой идеей.

Если тип значения, возвращаемого функцией `operator->()`, используется, то он должен быть указателем на класс или объект класса, в котором определен `operator->()`. Например:

```
struct Y { int m; };

class Ptr {
   Y* p;
   // ...
public:
   Ptr(Symbolic_ref);
   ~Ptr;

   Y* operator->()
   {
       // проверить p
       return p;
   }
};
```

Здесь класс Ptr определен так, что его объекты ведут себя как указатели на объекты класса Y с тем отличием, что при каждом обращении производятся некоторые вычисления:

```
void f(Ptr x, Ptr& xr, Ptr* xp)
{
   x->m;  // x.operator->()->m; то есть x.p->m
   xr->m; // xr.operator->()->m; то есть xr.p->m
   xp->m; // ошибка: у Ptr нет члена m
}
```

Такие классы особенно полезны в виде шаблонов (см. раздел 15.9.1), [2nd]:

```
template<class Y> class Ptr { /* ... */ };

void f(Ptr<complex> pc, Ptr<Shape> ps) { /* ... */ }
```

Это было понятно уже после реализации перегрузки -> в 1986 г. К сожалению, написать такой код удалось лишь после реализации шаблонов.

Для обычных указателей -> – это синоним некоторых применений * и []. Например, для объявления Y* p имеют место тождества:

```
p->m == (*p).m == p[0].m
```

Как обычно, для определенных пользователем операторов таких гарантий не дается. При желании эквивалентность можно обеспечить:

```
class Ptr {
   Y* p;
public:
   Y* operator->() { return p; }
   Y& operator*() { return *p; }
   Y& operator[](int i) { return p[i]; }
   // ...
};
```

Перегрузка оператора -> важна для целого класса программ. Причина в том, что косвенное обращение – ключевая концепция, а перегрузка -> дает красивый, прямой и эффективный способ реализации ее в программах. Еще одно применение оператора -> – это ограниченная, но очень полезная форма реализации делегирования в C++ (см. раздел 12.7).

11.5.2. «Умные» ссылки

Решив разрешить перегрузку оператора ->, я задумался над тем, можно ли аналогичным образом перегрузить оператор . (точка).

В то время мне казались убедительными следующие рассуждения: если obj – это объект класса, то obj.m имеет смысл для каждого члена m этого объекта. Переопределяя встроенные операции, мы не хотим получить мутирующий язык (хотя по очень веским причинам это правило нарушено для =, а также для унарного &).

Если разрешить перегрузку . для класса X, то не удастся обычным способом обратиться к его членам; придется использовать указатель и оператор ->, хотя -> и & тоже могут быть переопределены.

Эти аргументы являются весомыми, но не решающими. В частности, в 1990 г. Джим Эдкок (Jim Adcock) предложил разрешить перегрузку оператора . точно так же, как оператора ->.

Зачем может понадобиться перегрузка operator.()? Чтобы создать класс, работающий как «описатель» (handle) или «заместитель» (proxy) другого класса, в котором и выполняются основные действия. В качестве примера, приводившегося во время первых обсуждений перегрузки operator.(), рассмотрим класс целых чисел повышенной разрядности:

```
class Num {
   // ...
public:
   Num& operator=(const Num&);
   int operator[](int);        // получить одну цифру
   Num operator+(const Num&);
   void truncateNdigits(int); // отбросить
   // ...
};
```

Я надеялся определить класс RefNum, который ведет себя как Num&, но выполняет некоторые дополнительные действия. Например, если я могу написать

```
void f(Num a, Num b, Num c, int i)
{
   // ...
   c = a+b;
   int digit = c[i];
   c.truncateNdigits(i);
   // ...
}
```

то хотелось иметь возможность написать и такое:

```
void g(RefNum a, RefNum b, RefNum c, int i)
{
   // ...
   c =  a+b;
   int digits = c[i];
   c.truncateNdigits(i);
   // ...
}
```

Допустим, что operator.() определен полностью аналогично operator->(). Сначала попробуем очевидное определение RefNum:

```
class RefNum {
   Num *p;
public:
   RefNum(Num& a) { p = &a; }
```

```
    Num& operator.() { do_something(p); return *p; }
    void bind(Num& q) { p = &q; }
};
```

К сожалению, это не дает нужного результата, так как . не указана явно во всех случаях:

```
c = a+b;                // точки нет
int digits = c[i];      // точки нет
c.truncateNdigits(i);   // вызов operator.()
```

Придется написать функции-переходники, гарантирующие, что при применении операторов к RefNum выполняются правильные действия:

```
class RefNum {
    Num *p;
public:
    RefNum(Num& a) { p = &a; }
    Num& operator.() { do_something(p); return *p; }
    void bind(Num& q) { p = &q; }

    // функции-переходники
    RefNum& operator=(const RefNum& a)
        { do_something(p); *p=*a.p; return *this; }
    int operator[](int i)
        { do_something(p); return (*p)[i]; }
    RefNum operator+(const RefNum& a)
        { do_something(p); return RefNum(*p+*a.p); }
};
```

Это чересчур утомительно. Поэтому мы с Эндрю Кенигом и многие другие подумывали о том, чтобы применять operator.() к каждой операции над RefNum. В этом случае наш пример работал бы правильно и при исходном определении RefNum.

Однако если применять operator.() таким образом, то для доступа к любому члену RefNum пришлось бы использовать указатель:

```
void h(RefNum r, Num& x)
{
    r.bind(x);        // ошибка: Num::bind отсутствует
    (&r)->bind(x);    // правильно: вызывается RefNum::bind
}
```

Пользователи C++ разошлись во мнениях относительно того, какая интерпретация operator.() лучше. Я склоняюсь к тому, что если уж разрешать перегрузку operator.(), то он должен вызываться как для явных, так и для неявных операций. В конце концов, определяется он для того, чтобы не писать функции-переходники. Если неявные использования . не интерпретируются с помощью operator.(), то по-прежнему приходится писать множество переходников либо не перегружать этот оператор вообще.

Если operator.() перегружен, то эквивалентность a.m и (&a)->m более не гарантируется. Но ее можно поддержать, нужным образом определив operator->()

и operator&(), так что я не считаю это серьезной проблемой. Однако если поступить так, то получить доступ к членам класса, реализующего «умную» ссылку, не удастся вообще. Например, RefNum::bind() станет недоступной функцией.

Важно ли все вышесказанное? Некоторые отвечают: «Нет, поскольку «умную» ссылку, как и обычную, должно быть невозможно привязать к новому объекту». Однако мой опыт показывает, что для «умных» ссылок операция повторной привязки или какая-то другая часто необходимы.

Итак, мы оказались в затруднении: можно либо поддержать эквивалентность a.m и (&a)->m, либо сохранить доступ к членам класса, реализующего «умную» ссылку, но не то и другое одновременно.

Один из способов разрешить дилемму – использовать operator.() для a.m только в том случае, если класс ссылки сам не содержит члена с именем m. Мне такое решение нравится больше всего.

Однако по поводу важности перегрузки operator.() нет единого мнения, она не включена в C++, и ожесточенные споры вокруг этого вопроса продолжаются.

11.5.3. Перегрузка операторов инкремента и декремента

Перегружать операторы инкремента ++ и декремента -- пользователи могли всегда. Но в версии 1.0 не было механизма для различения префиксной и постфиксной нотации. Так, в классе

```
class Ptr {
   // ...
   void operator++();
};
```

один и тот же Ptr::operator++() использовался в следующих предложениях:

```
void f(Ptr& p)
{
   p++;   //p.operator++()
   ++p;   //p.operator++()
}
```

Некоторые, в особенности Брайан Керниган, отмечали, что такое ограничение неестественно для C и не дает программистам определить класс, который можно было использовать вместо обычного указателя.

Конечно, при проектировании механизма перегрузки операторов в C++ я думал о раздельной перегрузке префиксного и постфиксного инкрементов, но решил, что вводить для этой цели дополнительный синтаксис не стоит. Многочисленные предложения по этому поводу, полученные за несколько лет, доказали мою неправоту. Надо было лишь найти способ выразить различие между префиксной и постфиксной нотацией с помощью минимальных изменений.

Я раздумывал об очевидном решении – добавить в C++ ключевые слова prefix и postfix:

```
class Ptr_to_X {
   // ...
```

```
    X& operator prefix++(); // префиксный ++
    X operator postfix++(); // постфиксный ++
};
```

или

```
class Ptr_to_X {
    // ...
    X& prefix operator++(); // префиксный ++
    X postfix operator++(); // постфиксный ++
};
```

но получил обычную порцию гневных отповедей от людей, которым новые ключевые слова не по душе. Было предложено несколько вариантов без новых ключевых слов, например:

```
class Ptr_to_X {
    // ...
    X& ++operator(); // префиксный ++
    X operator++();  // постфиксный ++
};
```

или

```
class Ptr_to_X {
    // ...
    X& operator++(); // префиксный, поскольку возвращает ссылку
    X operator++();  // постфиксный, поскольку не возвращает ссылку
};
```

Первое решение показалось мне слишком заумным, второе – чересчур утонченным. В конечном итоге я остановился на таком варианте:

```
class Ptr_to_X {
    // ...
    X& operator++();     // префиксный: нет аргументов
    X operator++(int);   // постфиксный, так как есть аргумент
};
```

Это работает, не требует нового синтаксиса и имеет некоторую внутреннюю логику. Все остальные унарные операторы префиксные и не принимают аргументов, если определены как функции-члены. Достаточно необычный и посему неиспользуемый аргумент типа `int` употребляется для обозначения не менее необычного постфиксного оператора. Другими словами, для постфиксных операторов `++` как бы стоит между первым (реальным) и вторым (воображаемым) операндом.

Объяснения необходимы, так как описанный механизм не имеет аналогов и потому выглядит чужеродным. Будь у меня выбор, я бы предпочел ключевые слова `prefix` и `postfix`, но в то время это было невозможным. Имеет значение, однако, лишь то, что прием работает, а те немногие программисты, которым он действительно нужен, в состоянии его оценить и использовать.

11.5.4. Перегрузка ->*

Перегружать оператор `->*` было разрешено в основном потому, что для этого не было противопоказаний. Он оказался полезным для задания операций привязки, семантика которых по какой-то причине похожа на семантику встроенного оператора `->*` (см. раздел 13.11). Никаких специальных правил не потребовалось; `->*` ведет себя, как любой бинарный оператор.

Оператор `.*` не был включен в состав перегружаемых по той же причине, что и оператор `.` (см. раздел 11.5.2).

11.5.5. Перегрузка оператора «запятая»

По настоянию Маргарет Эллис я разрешил перегрузку оператора `,` (запятая). Главным образом потому, что не смог найти причин для отказа. На самом деле причина есть: выражение `a,b` уже определено для любых `a` и `b`, поэтому перегрузка позволяет программисту изменить семантику встроенного оператора. Правда, это возможно лишь тогда, когда или `a`, или `b` являются объектами класса. На практике для использования `operator,()` причин мало, так что введен он, скорее, для поддержания единого стиля языка.

11.6. Добавление в C++ операторов

Сколько бы операторов ни было, их всегда не хватает. Создается впечатление, что, если не считать тех программистов, кто в принципе против всяких операторов, все остальные постоянно мечтают о каких-нибудь дополнительных.

11.6.1. Оператор возведения в степень

Почему в C++ нет оператора возведения в степень? Прежде всего потому, что его нет в C. Семантика операторов C выбрана максимально простой: все они соответствуют машинным командам обычного компьютера. Оператор возведения в степень этому критерию не удовлетворяет.

Оператор возведения в степень не был добавлен в C++ сразу же. Моей целью было предоставление механизмов абстрагирования, а не новых примитивных операций. Оператору возведения в степень пришлось бы приписать семантику для встроенных арифметических типов. Но это та сторона C, которую я предпочел оставить без изменений. Далее, пользователи и так критиковали C и C++ за то, что в них слишком много операторов с неочевидными приоритетами. Несмотря на эти сдерживающие факторы, я все-таки думал об операторе возведения в степень и, возможно, добавил бы его в язык, если бы не некоторые технические проблемы. Сомневаясь в реальной необходимости такого оператора в языке, поддерживающем перегрузку и встраивание вызовов функций, я все же считал лучшим включить его, просто чтобы опровергнуть общепринятое мнение о том, что без него никак не обойтись.

Пользователи хотели иметь оператор возведения в степень в виде `**`. Но это привело бы к осложнениям, поскольку выражение `a**b` допускается синтаксисом C и обозначает умножение на разыменованный указатель `b`.

```
double f(double a, double* b)
{
   return a**b;  // означает a*(*b)
}
```

Кроме того, среди сторонников включения оператора возведения в степень не было согласия относительно его приоритета и ассоциативности:

```
c = b**c**d;     // (b**c)**d или b**(c**d)?
a = -b**c;       // (-b)**c или -(b**c)?
```

В конце концов у меня не было большого желания описывать математические свойства операции возведения в степень.

В результате я решил, что для пользователей будет лучше, если я сосредоточусь на других вопросах. Оглядываясь назад, отмечу: все эти проблемы были разрешимы. Но нужно ли было решать их? Вопрос о нужности оператора встал ребром, когда в 1992 г. Мэтт Остерн (Matt Austern) представил комитету по стандартизации C++ законченное предложение. Попутно оно обросло массой комментариев и стало предметом оживленной дискуссии в Internet.

Зачем пользователям нужен оператор возведения в степень:

- потому что они привыкли к нему в Fortran;
- пользователи полагают, будто оператор скорее окажется оптимизированным, чем функция возведения в степень;
- вызов функции выглядит некрасиво в тех выражениях, которые обычно пишут физики и другие основные пользователи этого оператора.

Достаточно ли указанных причин, чтобы уравновесить технические сложности и возражения другой стороны? Можно ли преодолеть технические трудности? Рабочая группа по расширениям обсудила данные вопросы и решила не включать оператор возведения в степень. Итог подвел Дэг Брюк:

- оператор дает некоторое удобство нотации, но не предоставляет никакой новой функциональности. Члены рабочей группы, представляющие интересы тех, кто постоянно выполняет научные или инженерные расчеты, отметили, что и удобство нотации минимально;
- новое средство предстоит изучать каждому пользователю C++;
- пользователи подчеркивали, что обязательно должна быть возможность подставить собственные функции возведения в степень вместо применяемых по умолчанию, а при наличии встроенного оператора это было бы невозможно;
- предложение недостаточно мотивировано. В частности, глядя на программу из 30 тыс. строк на Fortran, нельзя было с уверенностью сказать, что этот оператор начнет широко использоваться в C++;
- предложение требует добавления нового оператора и еще одного уровня приоритета, что усложнит язык.

Приведенное краткое заключение далеко не полностью отражает глубину обсуждения вопроса. Так, некоторые члены комитета просмотрели множество написанных в разных компаниях программ и обнаружили, что возведение в степень не имеет такого принципиального значения, как подчас утверждается. Также было

отмечено, что в большинстве случаев в Fortran-программах возведение в степень имеет вид `a**n`, где `n` – небольшое целое число; вместо этого вполне можно написать `a*a` или `a*a*a`.

И все же изложу некоторые технические соображения. Какой оператор лучше всего подошел бы в C++ на роль оператора возведения в степень? В C использованы все печатные символы из набора ASCII за исключением `@` и `$`, которые по разным причинам были сочтены непригодными. Рассматривались операторы `!`, `~`, `*~`, `^^` и даже одиночная `^` (если хотя бы один операнд имеет не-интегральный тип). Однако `@`, `$`, `~` и `!` присутствуют не на всех национальных клавиатурах (см. раздел 6.5.3.1); кроме того многие считали, что `@` и `$` плохо смотрятся в этой роли. Лексемы `^` и `^^` воспринимаются C-программистами как «исключающее или». Вдобавок следовало обеспечить возможность комбинировать оператор возведения в степень с оператором присваивания так же, как для других арифметических операторов: например, + и = дает +=. Это сразу исключает `!`, поскольку `!=` уже имеет семантику. Мэтт Остерн предложил `*^` и, наверное, данное решение – лучшее из возможных.

Все остальные технические вопросы можно было уладить, ориентируясь на Fortran, который является стандартом в этой области.

Я снова вернулся к `**` для обозначения оператора возведения в степень в C++. Да, я продемонстрировал, что с помощью традиционных методов невозможно использовать `**` для указанной операции, но, обдумывая этот вопрос еще раз, понял, что проблему совместимости с C можно обойти, применив определенный прием при проектировании компилятора. Предположим, мы ввели оператор `**`. Несовместимость можно устранить, если определить его семантику как «разыменовать и умножить» в случае, когда второй операнд является указателем:

```
void f(double a, double b, int* p)
{
   a**b;  // означает pow(a,b)
   a**p;  // означает a*(*p)
   **a;   // ошибка: a - не указатель
   **p;   // ошибка: означает *(*p), но *p - не указатель
}
```

Разумеется, `**` должна быть лексемой языка. Но в таком случае при использовании в объявлении ее следует интерпретировать как двойную косвенность:

```
char** p; // означает char * * p
```

Основная проблема здесь в том, что приоритет `**` должен быть выше, чем у `*`, если мы хотим, чтобы выражение `a/b**c` интерпретировалось, как принято в математике, т.е. `a/(b**c)`. С одной стороны, `a/b**p` в C означает `(a/b)*(*p)`, а при новых правилах семантика изменилась бы на `a/(b*(*p))`. Очевидно, что такой код в C и C++ встречается редко. Можно было бы пойти на то, чтобы он перестал работать, если бы мы все же решили ввести оператор возведения в степень, особенно принимая во внимание, что компилятору совсем несложно выдать предупреждение в случае возможного изменения смысла. Меня позабавило, какую реакцию вызвало мое полусерьезное предложение об использовании `**`. И до сих пор меня не перестает удивлять тот нешуточный пыл, с которым обсуждаются

несущественные синтаксические вопросы, например следует ли записывать возведение в степень в виде `pow(a,b)` или `a**b`, или `a*^b`.

11.6.2. Операторы, определяемые пользователем

Удалось бы избежать споров об операторе возведения в степень, предоставив механизм, который позволил бы пользователям определять собственные операторы? Тогда проблема отсутствия нужного оператора в общем виде была бы решена.

Когда речь заходит об операторах, неизменно обнаруживается, что того набора, который предлагают C и C++, недостаточно для выражения всех необходимых операций. Обычное решение – определить функцию. Однако когда для класса есть возможность написать

```
a*b
```

то запись вроде

```
pow(a,b)
abs(a)
```

выглядит неудачно. Поэтому есть просьбы придать смысл таким выражениям, как

```
a pow b
abs a
```

Это можно сделать. Как именно – показано в Algol68. Но дальше пользователи хотят, чтобы осмысленными были и такие строки:

```
a ** b
a // b
|a
```

Можно сделать и это. Вопрос только в том, действительно ли стоит разрешать пользователям определять операторы самим. В [ARM] отмечалось:

> Такое расширение, однако, резко усложнило бы синтаксический анализ и могло непредсказуемым образом сказаться на читаемости программы. Пришлось бы или разрешить пользователям задавать приоритет и ассоциативность новых операторов, или считать, что для всех новых операторов эти атрибуты фиксированы. В любом случае порядок вычисления выражения вида
>
> ```
> a = b**c**d; // (b**c)**d или b**(c**d)?
> ```
>
> был бы неочевиден. Понадобилось бы также разрешать синтаксические конфликты с обычными операторами. В предположении, что ** и // определены как бинарные операторы, рассмотрим такой пример:
>
> ```
> a = a**p; // a**p ИЛИ a*(*p)
> a = a//p;
> *p = 7;   // a = a*p = 7; может быть
> ```

Следовательно, определенные пользователем операторы должны либо состоять только из обычных символов, либо включать в себя какой-то легко отличимый префикс, например . (точка):

```
a pow b;  // альтернатива 1
a .pow b; // альтернатива 2
a .** b;  // альтернатива 3
```

Определенным пользователем операторам следует назначить приоритет. Самый простой способ – считать, что у всех подобных операторов приоритет одинаков и совпадает с приоритетом какого-либо встроенного оператора. Однако этого недостаточно для «корректного» определения оператора возведения в степень. Например:

```
operator pow: binary, precedence between * and unary
```

Также я опасаюсь, что программы, содержащие определенные пользователем операторы, для которых он же задал приоритеты, будет невозможно читать. Например, в языках программирования приоритет операции возведения в степень определяется по-разному. Поэтому пользователи станут неодинаково определять приоритет оператора `pow`. Значит, синтаксический анализатор будет по-разному разбирать выражение

```
a = - b pow c * d;
```

в разных программах.

Проще присвоить всем определенным пользователем операторам один и тот же приоритет. Это решение казалось заманчивым, пока не выяснилось, что даже со своими ближайшими сотрудниками – Эндрю Кенигом и Джонатаном Шопиро – мы не можем прийти к общему мнению относительно того, каким должен быть этот приоритет. Очевидные варианты – «очень высокий» (скажем, выше, чем у умножения) и «очень низкий» (например, ниже, чем у присваивания). Увы, число случаев, когда один выбор представляется идеальным, другой – абсурдным, не поддается исчислению. Так, даже простейшие примеры не удается написать «правильно», если есть только один уровень приоритета. Убедитесь сами:

```
a = b * c pow d;
a = b product c pow d;
a put b + c;
```

Поэтому в C++ и нет определяемых пользователем операторов.

11.6.3. Составные операторы

C++ поддерживает перегрузку унарных и бинарных операторов. Видимо, стоило бы поддержать и перегрузку составных операторов. В ARM данная идея объяснялась следующим образом:

Например, два умножения в примере

```
Matrix a, b, c, d;
// ...
a = b * c * d;
```

можно было бы реализовать специальным оператором «двойного умножения», определенным так:

```
Matrix operator * * (Matrix&, Matrix&, Matrix&);
```

и тогда предложение интерпретировалось бы:

```
a = operator * * (b,c,d);
```

Иначе говоря, обнаружив один раз объявление

```
Matrix operator * * (Matrix&, Matrix&, Matrix&);
```

компилятор начнет искать повторяющиеся умножения `Matrix` и вызывать для их интерпретации функцию. Слишком сложные для интерпретации последовательности будут обрабатываться обычными (унарными и бинарными) операторами.

Такое расширение предлагалось несколько раз как эффективный способ обработки повторяющихся вычислений в научных расчетах, где используются определенные пользователем типы. Например, оператор

```
Matrix operator = * + (
   Matrix&,
   const Matrix&,
   double,
   const Matrix&
);
```

можно было бы использовать для обработки предложений типа:

```
a=b*1.7+d;
```

Естественно, пробел в таких объявлениях очень важен. Впрочем, вместо него для обозначения позиций операндов можно было использовать и любую другую лексему:

```
Matrix operator.=.*.+.(
   Matrix&,
   const Matrix&,
   double,
   const Matrix&
);
```

До выхода ARM я никогда не видел изложения этого решения в печати, но такая техника часто применяется в кодогенераторах. Данная идея представляется многообещающей для поддержки оптимизированных операций с векторами и матрицами, но у меня так и не нашлось времени продумать ее до конца. В существующей нотации это было бы неплохой поддержкой старого приема, заключающегося в определении функций для выполнения составных операций над переданными аргументами.

11.7. Перечисления

В C концепция перечислений выглядит незаконченной. В первоначальном варианте языка их не было. Перечисления ввели без всякой охоты в качестве уступки тем, кто настойчиво требовал более основательных символических констант, чем препроцессорные макросы без параметров. Поэтому в C значение перечислителя имеет тип `int`, равно как и значение переменной, объявленной как имеющая тип перечисления. Значение типа `int` можно присваивать переменной типа перечисления. Например:

```
enum Color { red, green, blue };

void f()  /* функция C */
```

```
{
   enum Color c = 2;    /* правильно */
   int i = c;           /* правильно */
}
```

Для тех стилей программирования, которые должен был поддерживать C++, не было нужды в перечислениях, поэтому правила C перешли в C++ без изменения.

Правда, в комитете ANSI C изменили или, если хотите, уточнили определение перечислений таким образом, что указатели на разные перечисления оказались разными типами:

```
enum Vehicle { car, horse_buggy, rocket };

void g(pc,pv) enum Color* pc; enum Vehicle* pv;
{
   pc = pv;      /* возможно, некорректно в ANSI C */
}
```

Я долго обсуждал возникшую проблему с такими экспертами по C, как Дэвид Хансон (David Hanson), Брайан Керниган, Эндрю Кениг, Дуг Макилрой, Дэвид Проссер и Деннис Ричи. Мы так и не смогли прийти к окончательному выводу – что само по себе было недобрым знаком, – но решили, что в комитете собирались объявить этот пример незаконным.

Меня такая неопределенность не устраивала из-за перегрузки функций. Так, я должен знать, что объявлено в следующем примере:

```
void f(Color*);
void f(Vehicle*);
```

Одна и та же функция, объявленная дважды, или две перегруженные функции? Уверен, что формулировки всегда должны быть четкими, а оставлять решение на усмотрение разработчика компилятора не стоит. Аналогичный пример

```
void f(Color);
void f(Vehicle);
```

также должен интерпретироваться однозначно. В C и C++ в том виде, в каком последний существовал до выхода ARM, это считалось одной функцией, объявленной дважды. Однако правильнее было бы рассматривать каждое перечисление как отдельный тип. Например:

```
void h()  // C++
{
   Color c = 2;  // ошибка
   c = Color(2); // правильно: 2 явно преобразуется в Color
   int i = c;    // правильно: c неявно преобразуется в int
}
```

При обсуждении перечислений с программистами на C++ кто-нибудь всегда громко требовал именно такого решения проблемы. Быть может, я и действовал чересчур стремительно, но принятый вариант является лучшим.

11.7.1 Перегрузка на базе перечислений

Я упустил из виду очевидный факт: раз каждое перечисление – это отдельный тип, определенный пользователем, следовательно, он имеет все те же права, что и класс. В таком случае, на базе перечислений можно перегружать операторы. Указал на это Мартин О'Риордан (Martin O'Riordan) на заседании комитета ANSI/ISO. Вместе с Дэгом Брюком они продумали все детали, и перегрузка на базе перечислений была включена в C++. Например:

```
enum Season { winter, spring, summer, fall };

Season operator++(Season s)
{
   switch (s) {
   case winter: return spring;
   case spring: return summer;
   case summer: return fall;
   case fall:   return winter;
   }
}
```

Я применял такие переключатели, чтобы избежать приведений типов и операций над целыми числами.

11.7.2. Тип Boolean

Одно из самых распространенных перечислений – это

```
enum bool { false, true };
```

В любой сколько-нибудь сложной программе можно встретить нечто похожее, например:

```
#define bool char
#define Bool int
typedef unsigned int BOOL;
typedef enum { F, T } Boolean;
const true = 1;
#define TRUE 1
#define False (!True)
```

Примеры можно множить до бесконечности. Однако семантика большинства вариантов слегка различается и при совместном использовании в программе нескольких вариантов возникает конфликт.

Конечно, данная проблема была известна давно. Естественно, прежде всего на ум приходит мысль определить перечисление. Однако Дэг Брюк и Шон Корфилд изучили сотни тысяч строк на C++, и выяснилось, что в большинстве случаев при использовании типа `Boolean` необходимо свободно преобразовывать его в `int` и обратно. Поэтому определение `Boolean` в виде перечисления испортило бы слишком много программ. Так стоит ли вообще о нем беспокоиться, если:

- тип `Boolean` существует вне зависимости от того, есть он в стандарте C++ или нет;

- наличие множества конфликтующих друг с другом определений делает неудобным и небезопасным любой тип `Boolean`;
- многие пользователи хотят иметь возможность перегружать функции на базе типа `Boolean`.

К моему удивлению, комитет ANSI/ISO принял эти аргументы, поэтому `bool` теперь входит в C++ как отдельный интегральный тип и имеет литералы `true` и `false`. Ненулевые значения можно неявно преобразовать в `true`, а `true` неявно преобразуется в `1`. Нуль можно неявно преобразовать в `false`, а `false` неявно преобразуется в `0`. За счет этого обеспечивается высокая степень совместимости.

Глава 12
Множественное наследование

Потому что у тебя есть отец и мать.

comp.lang.c++

12.1. Введение

Множественное наследование – возможность иметь более одного прямого базового класса – зачастую считалось самой важной особенностью версии 2.0. В то время я не соглашался с этим, так как полагал, что совокупность улучшений системы типов гораздо важнее.

Кроме того, не стоило включать в версию 2.0 множественное наследование. Оно действительно необходимо в C++, но гораздо менее важно, чем параметризованные типы, а для некоторых даже параметризованные типы – нестоящая мелочь по сравнению с обработкой исключений. Но получилось так, что параметризованные типы появились только в версии 3.0, а исключения – еще позже. Лично мне параметризованных типов не хватало гораздо больше, чем множественного наследования.

Работа над множественным наследованием проводилась именно в то время по ряду причин: оно хорошо согласуется с системой типов C++ без крупных расширений, а реализацию можно было осуществить в рамках Cfront. Следует учесть еще один фактор, абсолютно иррациональный: никто, по-видимому, не сомневался, что я смогу эффективно реализовать шаблоны. А вот множественное наследование, по общему мнению, с трудом поддавалось эффективной реализации. Например, рассуждая о C++, Брэд Кокс (Brad Cox) в своей книге, посвященной Objective C, заявил, что добавить множественное наследование в C++ невозможно [Cox, 1986]. Таким образом, мне был брошен вызов. Я не смог его не принять, потому что думал о множественном наследовании еще в 1982 г. (см. раздел 2.13), а в 1984 г. нашел для него простую и эффективную реализацию. Полагаю, что это единственный случай, когда на последовательность событий оказала влияние мода.

В сентябре 1984 г. я представил механизм перегрузки операторов в C++ на конференции IFIP WG2.4 в Кентербери [Stroustrup, 1984b]. Там я встретил Стейна Крогдала (Stein Krogdahl) из университета Осло; Стейн как раз заканчивал работу над предложением о включении множественного наследования в Simula [Krogdahl, 1984]. Его идеи легли в основу реализации обыкновенных множественных базовых классов в C++. Уже позже стало известно, что почти идентичное

предложение для Simula уже рассматривалось ранее. Оле-Йохан Дал (Ole-Johan Dahl) рассматривал множественное наследование еще в 1966 г. и отверг его из-за неизбежного усложнения сборщика мусора в Simula [Dahl, 1988].

12.2. Базовые классы

Первая и самая важная причина для рассмотрения множественного наследования заключалась в следующем: необходимо было обеспечить такое объединение двух классов в один, при котором объекты результирующего класса ведут себя, как объекты любого из базовых классов [Stroustrup, 1986]:

«Довольно стандартный пример использования множественного наследования – создание двух библиотечных классов `displayed` и `task` для представления объектов, одновременно находящихся под управлением диспетчера вывода на экран и обладающих свойствами сопрограммы, работающей под управлением планировщика. Тогда программист мог бы создавать такие классы:

```
class my_displayed_task : public displayed, public task {
   // ...
};

class my_task : public task { // не наследует displayed
   // ...
};

class my_displayed : public displayed { // не наследует task
   // ...
};
```

Если использовать только одиночное наследование, то программисту доступны лишь два последних варианта».

В то время меня беспокоило, что библиотечные классы становятся слишком громоздкими, поскольку должны удовлетворять как можно большему числу требований. Множественное наследование я рассматривал как потенциально важное средство организации библиотек с меньшим числом классов и зависимостей между ними. Пример с классами `task` и `displayed` показывает, как можно представить свойства параллельности и вывода на экран с помощью отдельных классов, не перекладывая работу на программиста.

«Неоднозначности разрешаются во время компиляции:

```
class A { public: void f(); /* ... */ };
class B { public: void f(); /* ... */ };
class C : public A, public B {/* f() нет ... */ };

void g()
{
   C* p;
   p->f();    // ошибка: неоднозначность
}
```

В этом смысле C++ отличается от объектно-ориентированных диалектов Lisp, поддерживающих множественное наследование» [Stroustrup, 1987].

Разрешение неоднозначности с помощью зависимости от порядка, когда предпочтение отдается `A::f` просто потому, что `A` встречается раньше `B`, я отверг, так как имел негативный опыт использования подобных зависимостей в других местах (см. разделы 11.2.2. и 6.3.1). Я отказался также от всех форм динамического разрешения, помимо виртуальных функций, поскольку они не годятся для статически типизированного языка с жесткими требованиями к эффективности.

12.3. Виртуальные базовые классы

Вот цитата из работы [Stroustrup, 1987]:

«Класс может встречаться в направленном ациклическом графе наследования более одного раза:

```
class task : public link { /* ... */ };
class displayed : public link { /* ... */ };
class displayed_task
   : public displayed, public task { /* ... */ };
```

В данном случае объект класса `displayed_task` имеет два подобъекта класса `link`: `task::link` и `displayed::link`. Часто это бывает полезно, например в случае реализации списков, когда требуется, чтобы каждый элемент списка содержал поле связи. В результате объект класса `displayed_task` находится одновременно в списках объектов `displayed` и `task`".

Это можно представить и графически, показав, из каких подобъектов состоит `displayed_task` (см. рис. 12.1).

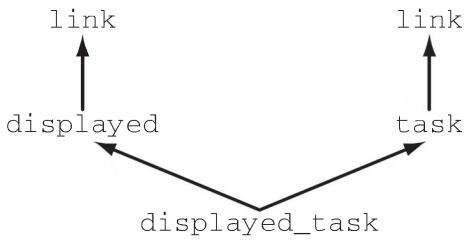


Рис. 12.1

Я не считаю такой стиль представления списка идеальным во всех ситуациях, но иногда он оптимален. Поэтому C++ поддерживает приведенный выше пример. По умолчанию базовый класс, появляющийся в иерархии наследования дважды, будет представлен двумя подобъектами. Однако возможно и другое разрешение ситуации [Stroustrup, 1987]:

«Я называю это независимым множественным наследованием. Но во многих случаях применения множественного наследования предполагается наличие зависимости между базовыми классами (например, предоставление каких-то возможностей для окна). Такие зависимости можно выразить с помощью объекта, разделяемого различными производными классами. Другими словами, требуется указать, что базовый класс должен привносить лишь один объект в конечный

производный класс, даже если он встречается в иерархии наследования несколько раз. Чтобы отличить этот случай от независимого множественного наследования, такие базовые классы описываются как виртуальные:

```
class AW: public virtual W { /* ... */ };
class BW: public virtual W { /* ... */ };
class CW: public AW, public BW { /* ... */ };
```

Единственный объект класса W будет разделяться на AW и BW; иными словами, в CW должен быть включен только один объект W, хотя CW наследует и AW, и BW. Если не считать того, что в производном классе возникает всего один объект, виртуальный базовый класс ведет себя точно так же, как обыкновенный.

Виртуальность W – это свойство наследования, которое задается для AW и BW, а не для самого W. Все виртуальные базовые классы в графе наследования относятся к одному и тому же объекту».

Графически это представлено на рис. 12.2.

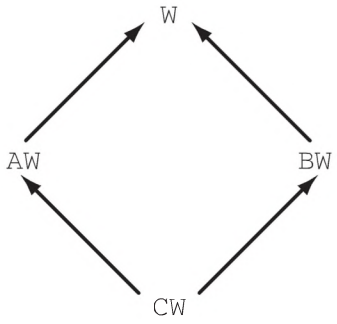


Рис. 12.2

Какие классы могут выступать в роли W, AW, BW и CW в реальных программах? Первоначальным примером была простая оконная система, основанная на идеях, почерпнутых из литературы по Lisp, см. рис. 12.3.

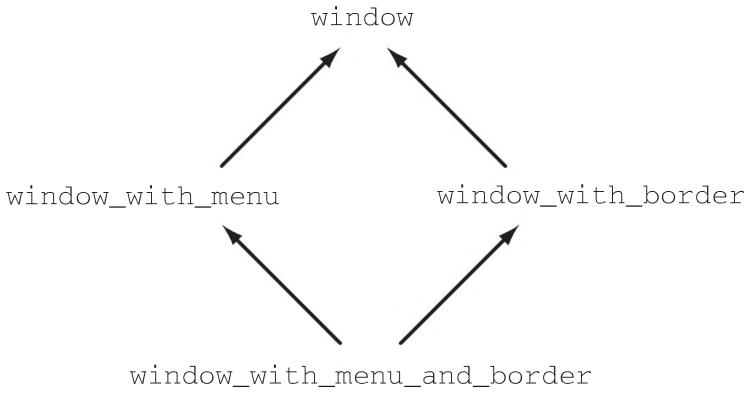


Рис. 12.3

На мой взгляд, этот пример искусственный, но он взят из реальной программы и, что особенно важно для представления, интуитивно понятен. Несколько примеров можно найти и в стандартной библиотеке потокового ввода/вывода [Shopiro, 1989] (см. рис. 12.4).

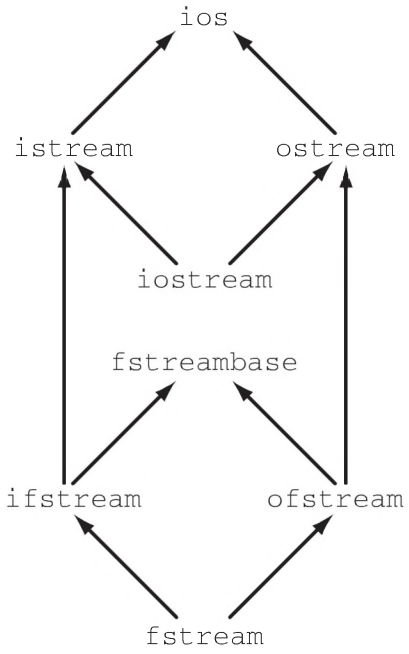


Рис. 12.4

Мне неизвестно, почему виртуальные базовые классы следует считать более полезными или фундаментальными, чем обыкновенные, и наоборот. Поэтому C++ поддерживает те и другие. По умолчанию решено было принять обыкновенный базовый класс, так как на его реализацию, в отличие от виртуального, тратится меньше времени и памяти; к тому же «программирование с помощью виртуальных базовых классов несколько сложнее, чем с помощью обыкновенных. Проблема в том, чтобы избежать множественных вызовов функции из виртуального класса, когда это нежелательно» [2nd], см. также раздел 12.5.

В связи с трудностями реализации возникло искушение не включать в язык понятие виртуального базового класса. Однако я счел решающим довод о необходимости какого-то способа представления зависимостей между потомками одних родителей («братьев»). Классы-«братья» могут взаимодействовать между собой только через общий корневой класс, через глобальные данные или через явные указатели. Если бы не было виртуальных базовых классов, то необходимость существования общего корня привела бы к чрезмерно широкому использованию «универсальных» базовых классов. Смешанный стиль, описанный в разделе 12.3.1, дает пример такого взаимодействия «братьев».

Если уж поддерживать множественное наследование, то подобный механизм необходим. С другой стороны, наиболее полезным я считаю простое применение множественного наследования вроде определения класса, который обладает суммой атрибутов двух других независимых классов.

12.3.1. Виртуальные базовые классы и виртуальные функции

Сочетание абстрактных и виртуальных базовых классов предназначалось для поддержки стиля программирования, который был бы приблизительно эквивалентен смешанному стилю (mixin style) в некоторых системах на Lisp. Это означает, что абстрактный класс определяет интерфейс, а реализацию предоставляет ряд производных классов. Каждый производный класс (компонент – mixin) делает вклад в полный класс (смесь – mix).

Чтобы поддержать смешанный стиль, необходимы два условия:

- возможность замещать виртуальные функции базового класса из двух различных производных классов; в противном случае существенные части реализации должны наследоваться только по одной цепочке, как в примере `slist_set` из раздела 13.2.2;
- возможность определить, какая функция замещает виртуальную, и перехватить некорректные замещения; иначе пришлось бы полагаться на зависимость от порядка или разрешение во время выполнения.

Рассмотрим приведенный выше пример. Предположим, что в классе `W` есть виртуальные функции `f()` и `g()`:

```
class W {
   // ...
   virtual void f();
   virtual void g();
};
```

а в каждом из `AW` и `BW` замещающие одну из них:

```
class AW : public virtual W {
   // ...
   void g();
};

class BW : public virtual W {
   // ...
   void f();
};

class CW : public SW, public BW, public virtual W {
   // ...
};
```

Тогда `CW` можно использовать так:

```
CW* pcw = new CW;
AW* paw = pcw;
BW* pbw = pcw;

void fff()
{
   pcw->f(); // вызывается BW::f()
   pcw->g(); // вызывается AW::g()

   paw->f(); // вызывается BW::f()!
   pbw->f(); // вызывается AW::g()!
}
```

Как обычно, одна и та же виртуальная функция вызывается независимо от типа указателя на объект. Именно это и позволяет различным классам добавлять

функциональность к общему базовому классу и пользоваться тем, что добавил другой класс. Естественно, при проектировании производных классов это следует иметь в виду, а иногда необходимо кое-что знать о классах, наследующих из тех же базовых.

Чтобы замещение виртуальных функций вдоль различных ветвей было возможным, необходимо выработать правило о том, какие сочетания замещающих функций допустимы, а какие должны считаться ошибкой. При обращении к виртуальной функции нужно, чтобы вызывалась одна и та же реальная функция независимо от того, как задан объект класса. Мы с Эндрю Кенигом сформулировали единственное, на наш взгляд, правило, которое гарантирует такое поведение:

«Имя `B::f` доминирует над именем `A::f`, если его класс `A` является для класса `B` базовым. Если одно имя имеет более высокий приоритет по сравнению с другим, то при выборе между ними не возникает неоднозначности; используется доминирующее имя.

Например:

```
class V { public: int f(); int x; };
class B : public virtual V { public: int f(); int x; };
class C : public virtual V { };

class D : public B, public C { void g(); };

void D::g()
{
   x++;   // правильно: B::x доминирует над V::x
   f();   // правильно: B::f() доминирует над V::f()
}
```

Графически это выглядит так, как показано на рис. 12.5.

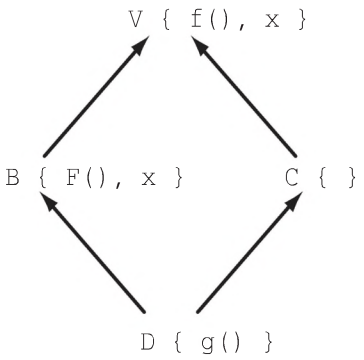


Рис. 12.5

Отметим, что доминирование определяется по отношению к любым именам, а не только к функциям.

Правило доминирования необходимо для применения виртуальных функций, ибо именно оно определяет, какая функция будет исполняться при виртуальном вызове. Опыт показывает, что оно столь же хорошо подходит и для невиртуальных функций. При использовании ранних версий компилятора, где правило доминирования к невиртуальным функциям не применялось, возникали ошибки, и приходилось писать плохо структурированные программы».

Для разработчика компилятора правило доминирования – обычное правило поиска имени, применяемое для того, чтобы понять, существует ли уникальная

функция, которую можно поместить в таблицу виртуальных функций. Более слабое правило не гарантировало бы этого, а более сильное привело бы к запрету осмысленных вызовов.

Правила для абстрактных классов и правило доминирования гарантируют, что объекты можно создавать только для таких классов, которые предоставляют полный и непротиворечивый набор сервисов. Без этих правил при работе над нетривиальными проектами у программиста было бы мало надежд избежать серьезных ошибок во время выполнения.

12.4. Модель размещения объекта в памяти

Множественное наследование усложняет модель объекта в двух отношениях:

- у объекта может быть несколько таблиц виртуальных функций;
- при просмотре таблицы виртуальных функций должен существовать способ найти подобъект, соответствующий классу, из которого взята замещающая функция.

Рассмотрим пример:

```
class A {
public:
   virtual void f(int);
};

class B {
public:
   virtual void f(int);
   virtual void g();
};

class C : public A, public B {
public:
   void f(int);
};
```

Объект класса C мог бы выглядеть в памяти, как показано на рис. 12.6.

Две таблицы виртуальных функций `vtbl` необходимы потому, что в программе могут быть объекты классов `A` и `B` наряду с объектами класса `C`, в которых есть части, соответствующие классам `A` и `B`. Получая указатель на `B`, надо уметь вызывать виртуальную функцию, не зная, что такое `B` – «настоящий объект класса `B`», «часть `B`, принадлежащая объекту класса `C`» или еще какой-то объект, включающий `B`. Поэтому у каждого `B` должна быть своя `vtbl`, доступ к которой осуществляется одинаково во всех случаях.

Часть `delta` необходима потому, что, коль скоро `vtbl` найдена, должна быть вызвана функция именно из того подобъекта, где она определена. Например, при вызове `g()` для объекта `C` нужно, чтобы указатель `this` был направлен на подобъект `B` объекта `C`, тогда как при вызове `f()` для `C` нужно, чтобы `this` указывал на весь объект `C`.

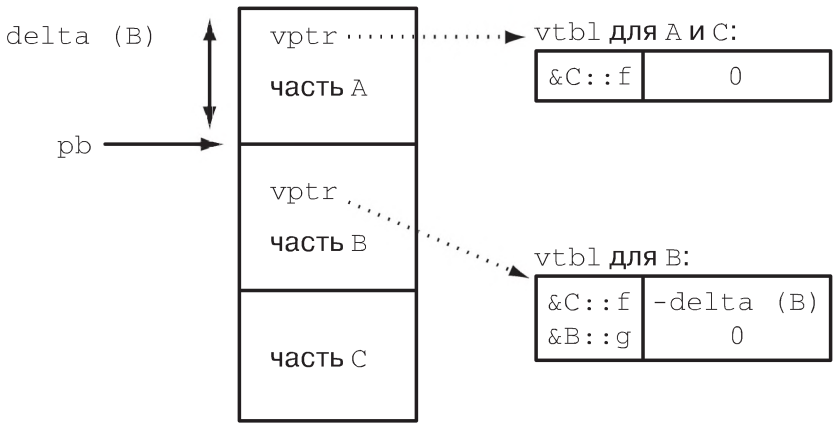


Рис. 12.6

Если объект размещен в памяти, как предложено выше, то вызов

```
void ff(B* pb)
{
   pb->f(2);
}
```

можно реализовать, например, таким кодом

```
/* сгенерированный код */
vtbl_entry* vt = &pb->vtbl[index(f)];
(*vt->fct)((B*)((char*)pb+vt->delta),2);
```

Такую стратегию я применял в первой реализации множественного наследования в Cfront. Ее достоинство в том, что она легко выражается на C и, следовательно, переносима. К тому же в сгенерированном коде нет ветвлений, а значит, он эффективно выполняется на машинах с конвейерной архитектурой.

Возможен и другой подход, при котором в таблице виртуальных функций не надо хранить дельту для указателя `this`; вместо этого хранится указатель на исполняемый код. Если корректировка `this` не требуется, то указатель на `vtbl` в действительности указывает на экземпляр выполняемой виртуальной функции; если же `this` нужно подправить, то хранящийся в `vtbl` указатель содержит адрес кода, который его корректирует, а затем выполняет нужный экземпляр виртуальной функции. При такой схеме объявленный выше класс C будет представлен, как показано на рис. 12.7.

Данная схема обеспечивает более компактное хранение таблиц виртуальных функций. Кроме того, если дельта равна `0`, обращения к виртуальным функциям происходят быстрее. Заметим, что при одиночном наследовании дельта всегда равна `0`. Передача управления после модификации в случае ненулевой дельты может привести к сбою в работе программы, если она осуществляется на машинах с конвейерной архитектурой, но такие затраты зависят от конкретной аппаратуры, так что общего решения не существует. Недостаток этой схемы в меньшей переносимости. Например, не на всех машинах возможен переход внутрь тела другой функции.

Код, корректирующий указатель `this`, обычно называют шлюзом (thunk). Это название восходит к первым реализациям Algol60, где такие небольшие фрагменты кода применялись для реализации вызова по имени.

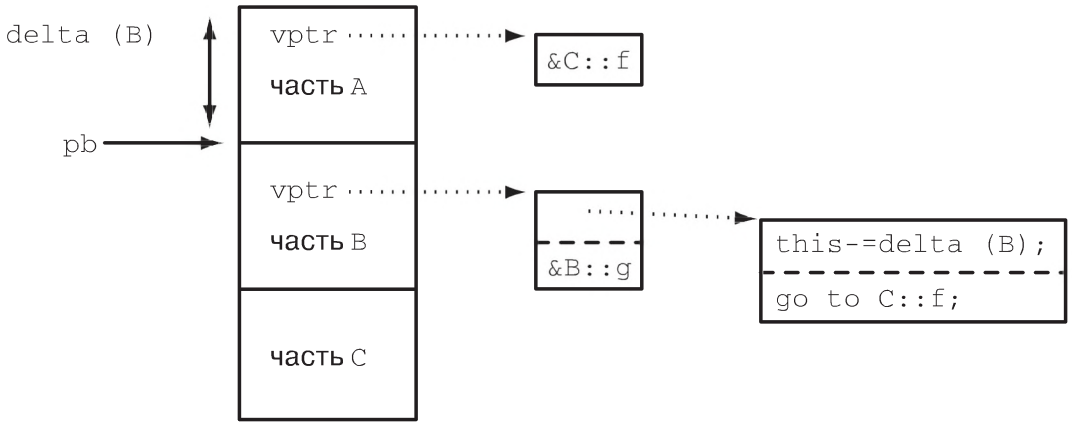


Рис. 12.7

Когда я проектировал и реализовывал множественное наследование, мне были известны только две описанные стратегии. Для проектирования языка они почти эквивалентны, но у реализации с помощью шлюзов есть одно полезное свойство: она не влечет никаких расходов времени и памяти в случае одиночного наследования, а это в точности согласуется с правилом нулевых издержек (см. раздел 4.5). Обе стратегии реализации обеспечивают приемлемую производительность во всех изученных мною случаях.

12.4.1. Размещение в памяти объекта виртуального базового класса

Почему виртуальный базовый класс был назван именно так? Часто я ухожу от ответа, отшучиваюсь, что «виртуальный» – значит «волшебный», и приступаю к более важным вопросам. Однако есть объяснение и получше. Оно родилось в спорах с Дугом Макилроем задолго до первой публичной презентации множественного наследования в C++. Виртуальная функция – это функция, которая отыскивается с помощью косвенного вызова через объект. Аналогично объект, представляющий виртуальный базовый класс, не имеет фиксированного места в производных от него классах и, стало быть, тоже должен находиться с помощью некоторой формы косвенности. Кроме того, базовый класс определяется как неименованный член. Поэтому, если бы были разрешены виртуальные члены-данные, виртуальный базовый класс являлся бы примером такого члена. Я хотел реализовать виртуальные базовые классы именно в таком стиле. Например, если есть класс `X` с виртуальным базовым классом `V` и виртуальной функцией `f`, в результате получается то, что изображено на рис. 12.8, вместо «оптимизированной» реализации в Cfront (см. рис. 12.9).

На этих рисунках `&X::Vobj` – смещение объекта, представляющего в `X` часть, относящуюся к виртуальному базовому классу `V`. Первая модель является более чистой и общей. Она требует при выполнении чуть больше времени, чем «оптимизированная» модель, зато экономит память.

Виртуальные члены-данные – одно из тех расширений C++, которые пользователи часто просят ввести. Обычно автор предложения хочет иметь «статические виртуальные данные», «константные виртуальные данные» или даже «константные статические виртуальные данные». Однако, как правило, при этом имеется в виду

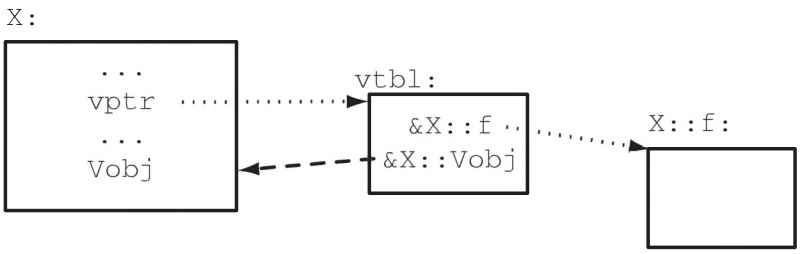


Рис. 12.8

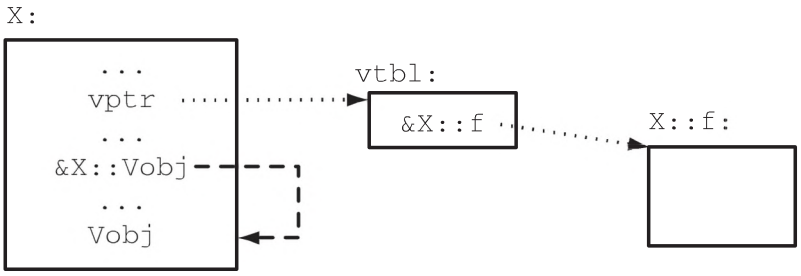


Рис. 12.9

одно конкретное применение: идентификация типа объекта во время исполнения. Есть и другие способы добиться такого результата (см. раздел 14.2).

12.4.2. Виртуальные базовые классы и приведение типов

Иногда пользователи удивляются, почему свойство «быть виртуальным базовым классом» является атрибутом наследования, а не самого базового класса. Дело в том, что в описанной выше модели размещения объекта в памяти не хватает информации для нахождения производного класса, когда имеется указатель на один из его виртуальных базовых классов; нет «обратного указателя» на объемлющие объекты. Например:

```
class A : public virtual complex { /* ... */ };
class B : public virtual complex { /* ... */ };
class C : public A, public B { /* ... */ };

void f(complex* p1, complex* p2, complex* p3)
{
   (A*)p1;    // ошибка: нельзя приводить из типа виртуальной базы
   (A*)p2;    // ошибка: нельзя приводить из типа виртуальной базы
   (A*)p3;    // ошибка: нельзя приводить из типа виртуальной базы
}
```

Если имеется вызов

```
void g()
{
   f(new A, new B, new C);
}
```

то объект `complex`, на который указывает `p1`, вряд ли окажется в той же позиции относительно `A`, как и `complex`, на который указывает `p2`. Следовательно, приведение от

типа виртуального базового класса к типу производного класса требует некоторого действия во время выполнения, основанного на той информации, которая хранится в объекте базового класса. Но в объектах простых классов такой информации не может быть при заданных ограничениях на размещение в памяти.

Если бы я относился к приведениям типов с меньшей осторожностью, то счел бы невозможность приведения от типа виртуального базового класса более серьезным недостатком. Так или иначе, классы, доступ к которым производится через виртуальные функции, и классы, где просто хранится несколько элементов данных, чаще всего являются наилучшими кандидатами на роль виртуальных базовых классов. Если в базовом классе есть только члены-данные, не следует передавать указатель на него, как на полный класс. Если же, с другой стороны, в базовом классе есть виртуальные функции, то их можно вызывать. В любом случае в приведении типа не должно возникать необходимости. Ну, а если без приведения типа от базового к производному классу все-таки не обойтись, оператор `dynamic_cast` (см. раздел 14.2.2) решает проблему для базовых классов с виртуальными функциями.

Обыкновенные базовые классы занимают одну и ту же позицию во всех объектах данного производного класса, и эта позиция известна компилятору. Поэтому указатель на обыкновенный базовый класс можно приводить к типу указателя на производный класс без каких-либо дополнительных проблем или затрат.

Если бы было введено требование о том, что всякий «потенциальный виртуальный базовый класс» должен объявляться явно, то ко всем им разрешалось бы применять специальные правила, например добавить информацию, необходимую для приведения типа от виртуального базового класса к производному от него. Если бы я решил сделать виртуальный базовый класс специальным видом класса, то программистам пришлось бы держать в голове две концепции: обыкновенного и виртуального классов, что неудобно.

Вместо этого можно было пойти на дополнительные затраты, добавив информацию, необходимую только настоящему виртуальному базовому классу, во все объекты классов. Но такой прием привел бы к заметным издержкам даже в тех приложениях, где виртуальные базовые классы не используются, а также возникли бы проблемы совместимости, относящиеся к модели размещения в памяти.

Поэтому я разрешил использовать любой класс в качестве виртуального базового и согласился с запретом на приведение типов, сочтя его ограничением, присущим использованию виртуальных базовых классов.

12.5. Комбинирование методов

Часто функция в производном классе использует функцию с тем же именем, определенную в одном из базовых классов. Этот прием называется комбинированием методов и поддерживается в некоторых объектно-ориентированных языках, но не в C++ (за исключением конструкторов, деструкторов и операций копирования). Может, стоило реанимировать понятие о функциях `call` и `return` (см. раздел 2.11.3) в подражание методам `:before` и `:after`, имеющимся в языке CLOS, но пользователи и так жалуются на сложности механизмов множественного наследования.

Вместе с тем методы легко комбинировать вручную. Главное – избежать нескольких вызовов функции из виртуального класса, если это нежелательно. Вот один из возможных подходов:

```
class W {
   // ...
protected:
   void _f() { /* выполняет то, что необходимо самому W */ }
   // ...
public:
   void f() { _f(); }
   // ...
};
```

В каждом классе имеется защищенная функция `_f()`, выполняющая операции, необходимые самому классу, и используемая производными классами. Есть также открытая функция `f()`, которая является интерфейсом, предлагаемым сторонним классам, функциям и т.д. В производном классе `f()` вызывает собственную `_f()` и позволяет базовым классам вызвать их `_f()`:

```
class A : public virtual W {
   // ...
protected:
   void _f() { /* выполняет то, что необходимо A */ }
   // ...
public:
   void f() { _f(); W::_f(); }
   // ...
};

class B : public virtual W {
   // ...
protected:
   void _f() { /* выполняет то, что необходимо B */ }
   // ...
public:
   void f() { _f(); W::_f(); }
   // ...
};
```

В частности, такой стиль позволяет в классе, который косвенно произведен от класса `W` дважды, вызывать `W::f()` только один раз:

```
class C : public A, public B, public virtual W {
   // ...
protected:
   void _f() { /* выполняет то, что необходимо C */ }
   // ...
public:
   void f() { _f(); A::_f(); B::_f(); W::_f(); }
   // ...
};
```

Это менее удобно, чем автоматически генерируемые составные функции, но в некоторых отношениях обладает большей гибкостью.

12.6. Полемика о множественном наследовании

Множественное наследование в C++ вызвало споры ([Cargill, 1991], [Carroll, 1991], [Waldo, 1991], [Sakkinen, 1992], [Waldo, 1993]) по нескольким причинам. Аргументы против него в основном касались реальной и воображаемой сложности концепции, ее полезности и влияния, которое она оказывает на другие средства языка и инструментальные средства:

- множественное наследование виделось как первое существенное расширение C++. Некоторые опытные пользователи C++ считали, что это ненужное усложнение, которое повлечет за собой поток новых средств. Например, на самой первой конференции по C++ в Санта-Фе (см. раздел 7.1.2) Том Каргилл сорвал аплодисменты, высказав забавное, но не очень реалистичное предложение: каждый, кто предлагает включить в C++ новое средство, должен сказать, какое из старых средств, равных по сложности новому, следует исключить. Мне такой подход нравится, но я все же не могу утверждать, что C++ стал бы лучше, не будь в нем множественного наследования, или что C++ образца 1985 г. лучше, чем C++ 1993 г. Веселье продолжил Джим Уолдо (Jim Waldo). Он продолжил мысль Тома, высказав следующую идею: обязать предлагающих новые средства пожертвовать... свою почку. Это, по мнению Джима, заставит каждого не один раз подумать, прежде чем вносить предложение, причем даже лишенные инстинкта самосохранения не смогут внести более двух предложений. Впрочем, не все были так настроены против новых возможностей, как можно было бы подумать, читая журналы, сетевые новости и выслушивая вопросы, задаваемые после доклада;
- я реализовал множественное наследование так, что издержки неизбежны даже для тех, кто пользуется лишь одиночным. Это нарушает правило «чем не пользуетесь, за то не платите» (см. раздел 4.5) и создает ложное впечатление, что множественное наследование якобы неэффективно вовсе. Такие издержки казались мне вполне приемлемыми, потому что они совсем невелики (один доступ к массиву плюс одно сложение на каждый вызов виртуальной функции) и потому что я знал простой способ реализации множественного наследования без всяких изменений реализации вызовов виртуальных функций в иерархии одиночного наследования (см. раздел 12.4). Я выбрал такую «субоптимальную» реализацию, поскольку она в наибольшей степени переносима;
- Smalltalk не поддерживает множественное наследование, а для многих пользователей слова «объектно-ориентированное программирование» и «Smalltalk» – синонимы. Такие люди часто подозревают, что если Smalltalk не поддерживает какую-то возможность, то она либо плоха, либо не нужна. Разумеется, я не разделяю этих взглядов. Очевидно, Smalltalk и выиграл бы от наличия множественного наследования, возможно, – нет; это не имеет отношения к делу. Однако мне было ясно, что некоторые приемы, которые

ревнители Smalltalk предлагали в качестве альтернатив множественному наследованию, не годились для статически типизированного языка, каким является C++. Языковые войны почти всегда бессмысленны; те же из них, которые развертываются вокруг отдельного средства языка без учета всего окружения, бессмысленны вдвойне. Нападки на множественное наследование, которые на самом деле направлены на статическую систему типов или являются простым отражением воображаемых атак на Smalltalk, лучше просто не принимать во внимание;

- мое изложение множественного наследования [Stroustrup, 1987] было перенасыщено техническими деталями, акцентировало внимание на вопросах реализации и плохо объясняло то, как им можно воспользоваться в программировании. В результате многие решили, что у множественного наследования мало применений, а реализовать его безмерно трудно. Подозреваю, что, если бы я в той же манере изложил принципы одиночного наследования, вывод был бы аналогичным;
- некоторые пользователи убеждены, что множественное наследование неудачно в принципе, поскольку «его слишком трудно использовать, а значит, результатом будет плохое проектирование и изобилующие ошибками программы». Конечно, множественное наследование можно употребить во вред, как, впрочем, и любое другое неординарное средство языка. Но мне известны реальные программы, которые с помощью множественного наследования удалось структурировать лучше, чем при использовании одиночного. При этом не просматривались очевидные альтернативы, которые позволили бы упростить структуру или сопровождение программы. Видимо, некоторые обвинения в адрес множественного наследования (дескать, оно является источником ошибок) основаны исключительно на опыте работы с другими языками, которые не могут диагностировать ошибки на стадии компиляции так же хорошо, как C++;
- ряд пользователей считает множественное наследование слишком слабым механизмом и в качестве альтернативы указывает на делегирование. Делегирование – это механизм, позволяющий перепоручить операцию другому объекту во время выполнения [Stroustrup, 1987]. Один из вариантов делегирования был реализован и испытан в C++. Результаты обескуражили: практически все пользователи столкнулись с серьезными трудностями вследствие разных изъянов в разработке своей программы, основанной на делегировании (см. раздел 12.7);
- утверждалось, что само по себе множественное наследование приемлемо, но его включение в C++ приводит к трудностям с другими потенциальными возможностями языка, например со сборкой мусора, и без необходимости усложняет построение инструментальных средств, в частности, баз данных для C++. Безусловно, множественное наследование усложняет создание инструментария. Но только время покажет, перевешивает ли усложнение преимущества при проектировании и реализации приложений, которые дает такое наследование;

- после включения множественного наследования в C++ говорилось, что идея хороша, но ее воплощение неудачно. Такие аргументы могут представлять интерес для проектировщиков «C++++», но мне они не очень помогают в работе над языком, его инструментальными средствами и приемами программирования на нем. Пользователи нечасто приводят практические свидетельства в пользу желаемых улучшений, сами предложения редко доводятся до деталей, варианты радикально отличаются друг от друга, а проблемы, связанные с изменением существующих правил, почти никогда не рассматриваются.

Сейчас, как и тогда, я думаю, что у всех вышеупомянутых доводов есть один общий недостаток – множественное наследование в них трактуется чересчур серьезно. А ведь это средство не решает – да и не должно решать – всех проблем, поскольку обходится слишком дешево. Иметь в своем распоряжении множественное наследование иногда удобно. Грейди Буч [Booch, 1991] выразил эту мысль чуть более экспрессивно: «Множественное наследование – как парашют; необязательно пользоваться им часто. Зато, когда возникает необходимость, без него не обойтись». Такое мнение частично основано на опыте, приобретенном при переписывании компонент Буча с Ada на C++ (см. раздел 8.4.1). Библиотека контейнерных классов и ассоциированных с ними операций, которую реализовали Грейди Буч и Майк Вило, – это один из лучших примеров применения множественного наследования [Booch, 1990], [Booch, 1993b].

Я прекратил участие в дебатах по поводу множественного наследования: оно есть в C++ и исключить или радикально изменить его невозможно. Как уже было сказано, считаю, что множественное наследование временами полезно. Некоторые настаивают на том, что для их подходов к проектированию и реализации данное средство абсолютно необходимо. Однако в широком смысле о пользе множественного наследования в C++ судить рано – не хватает данных и опыта. Наконец, я не люблю тратить время на бесплодные дискуссии.

По всей вероятности, наиболее удачно множественное наследование применяется в следующих ситуациях:

- объединение независимых или почти независимых иерархий; примером могут служить классы `task` и `displayed` (см. раздел 12.2);
- композиция интерфейсов; примером является библиотека потокового ввода/вывода (см. раздел 12.3);
- составление класса из интерфейса и реализации; пример – класс `slist_set` (см. раздел 13.2.2).

Дальнейшие примеры множественного наследования можно найти в разделах 13.2.2, 14.2.7 и 16.4.

В основном неудачи связаны с попытками насильно внедрить стиль, чуждый C++. В частности, прямое копирование применяемого в CLOS стиля проектирования, который основан на линеаризации разрешения неоднозначности, сопоставлении имен, разделяемых в иерархии, и применении методов `:before` и `:after` для создания составных операций, значительно усложняет большие программы.

12.7. Делегирование

В первоначальном проекте множественного наследования, который был представлен на Европейской конференции группы пользователей UNIX, состоявшейся в Хельсинки в мае 1987 г. [Stroustrup, 1987], фигурировало понятие делегирования [Agha, 1986].

Пользователю разрешалось задать указатель на некоторый класс вместе с именами базовых классов в объявлении класса. Описанный таким образом объект использовался точно так же, как если бы это был объект, представляющий базовый класс. Например:

```
class B { int b; void f(); };
class C : *p { B* p; int c; };
```

Нотация `:*p` означает, что объект, на который указывает `p`, будет использоваться так, будто он представляет базовый класс для `C`.

```
void f(C* q)
{
   q->f();    // означает q->p->f()
}
```

После инициализации `C::p` объект класса `C` выглядел бы примерно так, как это показано на рис. 12.10.

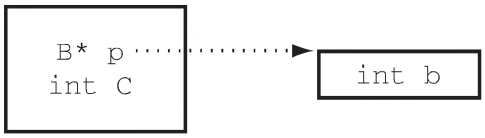


Рис. 12.10

Концепция выглядела многообещающей для представления структур, требующих большей гибкости, чем может дать обычное наследование. В частности, присваивание делегирующему указателю могло бы использоваться для изменения конфигурации объекта во время выполнения. Реализация была тривиальной, затраты – минимальными. Поэтому данную идею испытали несколько пользователей. Много времени и сил здесь положил Билл Хопкинс (Bill Hopkins). К сожалению, все пользователи, применившие механизм делегирования, пострадали от серьезных ошибок и путаницы. Из-за этого возможность была исключена как из проекта, так и из Cfront версии 2.0.

Причины ошибок:

- функции в делегирующем классе не замещают функции в классе, которому операция делегируется;
- функция, которой передается управление, не может воспользоваться функциями из делегирующего класса или каким-то иным способом «вернуться» в делегирующий объект.

Ясно, что две эти проблемы взаимосвязаны. Разумеется, пользователи были предупреждены. Предостережения не помогли. Более того, я сам забыл собственные

правила и попался в ловушку. Таким образом, проблему нельзя было назвать мелким огрехом, который исправляется с помощью обучения и предупреждений компилятора. В то время она казалась непреодолимой.

Сегодня мне кажется, что указанные проблемы имеют фундаментальный характер. Для решения первой потребовалось бы изменять таблицу виртуальных функций объекта, которому делегируется управление, если он связан с делегирующим объектом. Это плохо согласуется с языком в целом и с большим трудом поддается разумному определению. Кроме того, обнаружились примеры, когда мы хотели, чтобы два разных объекта делегировали управление одному и тому же «разделяемому» объекту. Нашлись и такие задачи, в которых нужно было делегировать управление через `B*` объекту производного класса `D`.

Поскольку делегирование не поддержано в C++ напрямую, нужно искать обходные пути, если оно все же необходимо. Часто решить проблему, требующую делегирования, можно с помощью «умного» указателя (см. раздел 11.5.1). Вместо этого делегирующий класс может предоставить полный интерфейс, а затем запросы «вручную» переадресовываются какому-то другому объекту (см. раздел 11.5.2).

12.8. Переименование

В конце 1989 – начале 1990 гг. была обнаружена проблема, проистекающая из конфликта имен в иерархиях с множественным наследованием [ARM]:

«Объединение двух иерархий классов путем использования их в качестве базовых классов для общего производного может стать источником трудностей, когда одно и то же имя используется в обеих иерархиях, но обозначает разные операции. Например:

```
class Lottery {            // лотерея
   // ...
   virtual int draw();     // тянуть жребий
};

class GraphicalObject {    // графический объект
   // ...
   virtual void draw();    // рисовать
};

class LotterySimulation    // моделирование лотереи
   : public Lottery, public GraphicalObject {
   // ...
};
```

В классе `LotterySimulation` нам хотелось бы заместить как `Lottery::draw()`, так и `GraphicalObject::draw()`, но двумя разными функциями, поскольку экземпляры `draw()` в базовых классах имеют разную семантику. Желательно, чтобы в классе `LotterySimulation` были различные, не допускающие двусмысленного толкования имена для унаследованных функций `Lottery::draw()` и `GraphicalObject::draw()`.

Семантика концепции очень проста, а реализация тривиальна; трудно лишь найти подходящий синтаксис. Предлагался такой вариант:

```
class LotterySimulation
   : public Lottery , public GraphicalObject {
```

```
    // ...
    virtual int l_draw() = Lottery::draw;
    virtual int go_draw() = GraphicalObject::draw;
};
```

Это естественное расширение синтаксиса виртуальных функций».

После продолжительной полемики в рабочей группе по расширениям и подготовки необходимых документов Мартином О'Риорданом и мной, предложение было вынесено на заседание комитета по стандартизации в Сиэтле в июле 1990 г. Подавляющее большинство участников обсуждения высказалось за то, чтобы сделать это первым «сторонним» расширением C++. Но в этот момент Бет Крокетт (Beth Crockett) из компании Apple приостановила работу комитета, воспользовавшись «правилом двух недель»: любой представитель комитета может перенести голосование по предложению на следующее заседание, если тема обсуждения не была передана ему, по крайней мере, за две недели до начала текущего заседания. Это правило защищает от поспешного принятия решения по вопросу, в котором человек разобрался не до конца, и гарантирует достаточное время для консультаций с коллегами.

Вы, наверное, догадались, что этим вето Бет не снискала популярности. Однако ее осторожность оказалась не напрасной и уберегла нас от серьезной ошибки. Спасибо! Когда мы вернулись к рассмотрению вопроса, Дуг Макилрой заметил, что проблема и без того имела решение в C++ [ARM]:

«Переименования можно достичь, введя дополнительный класс для каждого класса, содержащего виртуальную функцию, которая нуждается в замещении функцией с другим именем. Также для каждой такой функции необходимо написать специальную переадресующую функцию. Например:

```
class LLottery : public Lottery {
    virtual int l_draw() = 0;
    int draw() { return l_draw(); } // замещает
};

class GGraphicalObject : public GraphicalObject {
    virtual void go_draw() = 0;
    void draw() { go_draw(); } // замещает
};

class LotterySimulation
    : public LLottery , public GGraphicalObject {
    // ...
    int l_draw(); // замещает
    int go_draw();   // замещает
};
```

Следовательно, расширение языка для выражения переименования не является необходимым и рассматривать его стоит лишь в случае, если окажется, что потребность в разрешении таких конфликтов имен возникает часто».

На следующем заседании я описал прием. Во время обсуждения мы согласились, что вряд ли конфликты встречаются настолько часто, чтобы вводить в язык

специальное средство. Также было отмечено, что едва ли объединением больших иерархий классов каждый день будут заниматься новички. А опытные программисты без труда воспользуются обходным путем.

Еще одно, более общее возражение против переименования: я не люблю отслеживать цепочки псевдонимов при сопровождении кода. Если я вижу в тексте имя `f`, которое в заголовке определено как `g`, в документации названо `h`, а в коде вызывается как `k`, то проблема налицо. Разумеется, приведенный пример – крайность, но в таком стиле могут работать приверженцы макросов.

Синонимы бывают полезны, а иногда и необходимы. Однако их употребление следует свести к минимуму, если мы хотим сохранить ясность и общность кода в различных контекстах. Благодаря новым средствам, прямо поддерживающим переименование, началось бы простое потворство использованию (в том числе, и необоснованному) синонимов. Это соображение стало и доводом против общего механизма переименования при обсуждении пространств имен (см. главу 17).

12.9. Инициализаторы членов и базовых классов

При включении множественного наследования пришлось пересмотреть синтаксис инициализации членов и базовых классов. Например:

```
class X : public A, public B {
   int xx;
   X(int a, int b)
       : A(a),    // инициализации базового класса A
         B(b),    // инициализации базового класса B
        xx(1)     // инициализация члена xx
   { }
};
```

Этот синтаксис инициализации параллелен синтаксису инициализации объектов класса:

```
A x(1);
B y(2);
```

В то же время порядок инициализации по определению совпадал с порядком объявлений. Оставить порядок инициализации неспецифицированным, как было в первоначальном определении C++, значило предоставить неоправданную свободу действий разработчикам компиляторов в ущерб пользователям.

В большинстве случаев порядок инициализации членов не имеет значения. Обычно зависимость от порядка является признаком плохого проектирования. Но есть несколько случаев, когда программисту абсолютно необходим контроль над порядком инициализации. Рассмотрим передачу объектов между машинами. Объект должен реконструироваться приемником в порядке, обратном тому, который передатчик использовал при его разложении. Это нельзя гарантировать для объектов, которыми обмениваются программы, скомпилированные разными компиляторами, если в языке нефиксированный порядок конструирования. На данную особенность мне указал Кит Горлен, прославившийся своей библиотекой NIH (см. раздел 7.1.2).

В первоначальном определении C++ не требовалось, да и не разрешалось именовать базовый класс в инициализаторе. Например, если есть класс `vector`:

```
class vector {
   // ...
   vector(int);
};
```

то можно произвести от него другой класс `vec`:

```
class vec : public vector {
   // ...
   vec(int,int);
};
```

Конструктор класса `vec` должен вызывать конструктор `vector`:

```
vec::vec(int low, int high)
   : (high-low-1)   // аргумент для конструктора базового класса
{
   // ...
}
```

Данная нотация была причиной путаницы.

Явное использование имени базового класса, как требует версия 2.0, сделала запись достаточно прозрачной даже для неискушенных пользователей:

```
vec::vec(int low, int high) : vector(high-low-1)
{
   // ...
}
```

Я считаю, что первоначальный вариант является классическим примером синтаксиса логичного, минимального и слишком лаконичного. Проблемы, которые возникали при изложении студентам темы инициализации базовых классов, полностью исчезли с появлением нового синтаксиса.

Старый стиль инициализатора базового класса был сохранен на переходный период. Его можно было использовать только при одиночном наследовании, поскольку в противном случае он неоднозначен.

Глава 13. Уточнения понятия класса

Говори, что хочешь сказать, просто и прямо.

Брайан Керниган

13.1 Введение

Поскольку концепция классов является центральной в C++, не прекращаются запросы о ее модификации и расширении. Почти все просьбы о модификации приходится отклонять ради сохранения существующего кода, а большинство предложений о расширениях отвергались как излишние, непрактичные, не согласующиеся с другими частями языка или просто как «слишком сложные для немедленной реализации». В этой главе представлено несколько уточнений, которые я счел достаточно важными, чтобы подробно рассмотреть и в большинстве случаев принять. Центральный вопрос – сделать концепцию классов достаточно гибкой, чтобы выразить все, что необходимо, оставаясь в рамках системы типов и не прибегая к приведениям типов и низкоуровневым конструкциям.

13.2. Абстрактные классы

Последнее средство, добавленное в версию 2.0 перед фиксацией, – абстрактные классы. Запоздалые модификации версий всегда непопулярны, а внесение в последний момент изменений в определение языка еще того хуже. Мне казалось, что несколько руководителей заподозрили меня в утрате представлений о реальности, когда я стал настаивать на включении этой возможности. К счастью, Барбара Му поддержала меня в том, что абстрактные классы необходимы уже сегодня, а не через год-два.

Абстрактный класс предоставляет интерфейс. Прямая поддержка абстрактных классов:

- помогает находить ошибки, возникающие из-за неправильного понимания роли классов как интерфейсов и их значения в представлении объектов;
- способствует применению стиля проектирования, основанного на отделении спецификации интерфейсов от их реализации.

13.2.1. Абстрактные классы и обработка ошибок

С помощью абстрактных классов можно устранить некоторые источники ошибок [Stroustrup, 1989b]:

«Одна из целей статического контроля типов – обнаружить ошибки и противоречия еще до запуска программы. Замечено, что большой класс поддающихся обнаружению ошибок пропускается системой контроля типов C++. Мало того, чтобы выявить такую ошибку, программисты вынуждены писать дополнительный код и генерировать более объемные программы.

Рассмотрим классический пример класса `shape` (геометрическая фигура). Сначала мы объявляем класс `shape`, который представляет общее понятие фигуры. В нем должны быть две виртуальных функции: `rotate()` и `draw()`. Естественно, никаких объектов класса `shape` нет, есть только конкретные геометрические фигуры. К сожалению, в C++ нельзя выразить эту простую концепцию напрямую.

Правила C++ говорят, что виртуальные функции, такие как `rotate()` и `draw()`, должны быть определены в том классе, в котором они впервые объявлены. Причина требования – нужно скомпоновать C++-программу, пользуясь традиционными редакторами связей. Кроме того, необходимо гарантировать, что не будет вызываться неопределенная виртуальная функция. Поэтому программисту предстоит написать примерно такой код:

```
class shape {
   point center;
   color col;
   // ...
public:
   point where() { return center; }
   void move(point p) { center=p; draw(); }
   virtual void rotate(int)
      { error("нельзя повернуть"); abort(); }
   virtual void draw()
      { error("нельзя нарисовать"); abort(); }
   // ...
};
```

Если при этом пользователь забудет определить функцию `draw()` в классе, производном от `shape` (безобидная ошибка), или создаст «просто» фигуру и попытается ей воспользоваться (глупая ошибка), то произойдет серьезный сбой во время выполнения. А если программист и не допустит таких ошибок, то память компьютера засоряется ненужными таблицами виртуальных функций для классов типа `shape` и невызываемыми функциями типа `draw()` и `rotate()`. Связанные с этим затраты могут оказаться заметными.

Решение – дать пользователю возможность сказать, что у некоторой виртуальной функции нет определения, то есть она является «исключительно виртуальной». Это делается с помощью инициализатора `=0`.

```
class shape {
   point center;
   color col;
   // ...
public:
   point where() { return center; }
   void move(point p) { center=p; draw(); }
   virtual void rotate(int) = 0;     // чисто виртуальная функция
   virtual void draw() = 0;          // чисто виртуальная функция
   // ...
};
```

Класс, где есть хотя бы одна исключительно виртуальная функция, называется абстрактным. Его можно использовать только в качестве базового для какого-то другого класса. В частности, нельзя создавать объекты абстрактного класса. Класс, производный от абстрактного, должен либо содержать определения исключительно виртуальных функций, либо снова объявить их такими.

Концепции исключительно виртуальных функций было отдано предпочтение по сравнению с явным объявлением класса абстрактным, поскольку выборочное определение функций оказывается намного более гибким».

Как показано выше, в C++ и раньше была возможность выразить понятие абстрактного класса, но для этого требовалось проделать больше работы, чем хотелось бы. Некоторые пользователи считали это важным вопросом (см. например, [Johnson, 1989]). Но лишь за несколько недель до выпуска версии 2.0 я осознал, что очень немногие вообще понимают эту концепцию и что непонимание сути абстрактных классов было источником многих ошибок проектирования.

13.2.2. Абстрактные типы

Типичной претензией к C++ было (и остается) то, что закрытые данные включаются в объявление класса. Поэтому при изменении закрытых данных приходится перекомпилировать весь код, в котором данный класс используется. Часто недовольство по этому поводу формулировали так: «абстрактные типы в языке C++ не являются по-настоящему абстрактными» или «данные на самом деле не являются скрытыми». Многие пользователи думали, что уж если они могут поместить представление объекта в закрытую секцию класса, то обязательно должны так поступить. Разумеется, это неправильно (поэтому-то данная проблема многие годы и ускользала от моего внимания). Если вы не хотите помещать представление в класс, так и не надо! Просто перенесите спецификацию представления в какой-то производный класс. Нотация абстрактных классов позволяет сделать это явно. Например, можно так определить множество указателей типа `T`:

```
class set {
public:
   virtual void insert(T*) =0;
   virtual void remove(T*) =0;

   virtual int is_member(T*) =0;

   virtual T* first() =0;
   virtual T* next() =0;

   virtual ~set() { }
};
```

Здесь есть вся информация, необходимая для использования класса `set`, а также отсутствуют детали представления или реализации. Только тот, кто действительно создает объекты классов `set`, должен знать, как представлено множество. Например, если есть объявление

```
class slist_set : public set, private slist {
   slink* current_elem;
```

```
public:
   void insert(T*);
   void remove(T*);

   int is_member(T*);

   T* first();
   T* next();

   slist_set() : slist(), current_elem(0) { }
};
```

то можно создать объекты `slist_set`, с которыми смогут работать пользователи класса `set`, даже не подозревающие ни о каком `slist_set`. Например:

```
void user1(set& s)
{
   for (T* p = s.first(); p; p=s.next()) {
      // использовать p
   }
}

void user2()
{
   slist_set ss;
   // ...
   user1(ss);
}
```

Важно, что функцию, в которой используется абстрактный класс `set`, например `user1()`, можно откомпилировать, не включая заголовочные файлы, содержащие определение `slist_set` и классы типа `slist`, от которых последнее зависит.

Как уже отмечено выше, компилятор не позволяет создать объект абстрактного класса. Например:

```
void f(set& s1)  // правильно
{
   set s2;       // ошибка: объявление объекта
                 // абстрактного класса set
   set* p = 0;   // правильно
   set& s3 = s1; // правильно
}
```

Концепция абстрактных классов важна потому, что позволяет провести более четкую границу между пользователем и разработчиком реализации. Абстрактный класс – интерфейс к реализациям, содержащимся в производных от него классах. Это уменьшает число перекомпиляций после изменений, равно как и объем информации, необходимой для компиляции обычного фрагмента кода. С помощью абстрактных классов уменьшается зависимость пользователя от разработчика, снимаются претензии тех, кто жалуется на долгую компиляцию. Данное средство

заодно служит и интересам поставщиков библиотек, которым больше не нужно беспокоиться, как изменения реализации отразятся на работе пользователей. Я видел системы, время компиляции которых уменьшилось на порядок после того, как основные интерфейсы были реализованы в виде абстрактных классов. Попытка объяснить все это в [Stroustrup, 1986b] оказалась безуспешной, и лишь после появления в языке явной поддержки для абстрактных классов моя мысль стала понятной пользователям [2nd].

13.2.3. Синтаксис

Странный синтаксис `=0` был выбран вместо очевидной альтернативы ввести ключевое слово `pure` или `abstract` только потому, что тогда я не видел возможности добавить новое ключевое слово. Если бы я предложил `pure`, то версия 2.0 вышла бы без абстрактных классов. Чтобы не рисковать сроками выпуска версии и не навлекать на себя упреков по поводу нового ключевого слова, я воспользовался традиционным для C и C++ соглашением о том, что `0` означает «отсутствует». Синтаксис `=0` не противоречит моим взглядам на тело функции как на инициализатор функции, а также представлениям (упрощенным, но обычно соответствующим реальности) о том, что множество виртуальных функций реализовано в виде вектора указателей на функции (см. раздел 3.5.1). На самом деле, реализация синтаксиса `=0` путем помещения `0` в `vtbl` – не лучший способ. Я помещал в `vtbl` указатель на функцию `__pure_virtual_called`; затем эту функцию можно было определить так, чтобы она выводила подходящее сообщение об ошибке во время выполнения.

Лучше объявлять отдельные функции исключительно виртуальными, а не весь класс – абстрактным, поскольку в этом случае удается добиться большей гибкости. Я высоко ценю возможность определять класс поэтапно, то есть часть виртуальных функций определить в базовом классе, а остальные – в производных.

13.2.4. Виртуальные функции и конструкторы

Конструирование объекта из базовых классов и членов (см. раздел 2.11.1) влияет на работу механизма виртуальных функций. Подчас по этому поводу возникала путаница. Поэтому здесь объясняется, почему избранный для C++ путь является почти безальтернативным.

13.2.4.1. Вызов исключительно виртуальной функции

Как вообще можно вызвать чисто виртуальную функцию, а не замещающую ее в одном из производных классов? Объекты абстрактных классов могут существовать только в качестве базовых для каких-то других классов. После того как объект производного класса сконструирован, наша виртуальная функция уже замещена какой-то другой, определенной в производном классе. Однако в процессе конструирования собственный конструктор абстрактного класса может по ошибке вызвать исключительно виртуальную функцию:

```
class A {
public:
   virtual void f() =0;
```

```
    void g();
    A();
};

A::A()
{
    f();   // ошибка: вызвана исключительно виртуальная функция
    g();   // выглядит вполне безобидно
}
```

Незаконный вызов `A::f()` легко обнаруживается компилятором. Однако `A::g()` может быть объявлена, скажем, как

```
void A::g() { f(); }
```

в какой-то другой единице трансляции. Такую ошибку сможет обнаружить лишь компилятор, выполняющий перекрестный анализ нескольких единиц трансляции. Альтернативой же является ошибка во время выполнения.

13.2.4.2. Конструирование в порядке «сначала базовый»

Я всегда предпочитаю проект, не оставляющий места для ошибок во время выполнения. Но нет способа сделать программирование абсолютно безопасным. В частности, конструкторы создают среду, в которой работают функции-члены (см. раздел 2.11.1). Однако пока создание такой среды не завершено, программисту даются частичные гарантии. Рассмотрим пример:

```
class B {
public:
    int b;
    virtual void f();
    void g();
    // ...
    B();
};

class D : public B {
public:
    X x;
    void f();
    // ...
    D();
};

B::B()
{
    b++;   // неопределенное поведение: B::b еще не инициализирована
    f();   // вызывается: B::f(), а не D::f()
}
```

Компилятор может выдать предупреждения об обеих потенциальных проблемах. Если вы действительно собирались вызвать функцию `f()` из самого класса `B`, напишите это явно – `B::f()`.

Поведение данного конструктора контрастирует с тем, как пишется обыкновенная функция (зависящая от правильной работы конструктора):

```
void B::g()
{
   b++;   // правильно, так как B::b - это член
          //  B::B должен инициализировать его
   f();   // вызывается: D::f(), если для D вызывается B::g
}
```

Разница между тем, какую функцию вызывает `f()` при обращении из `B::B()` и из `B::g()` в случае, когда `B` есть часть `D`, может удивить неискушенного пользователя.

13.2.4.3. Что, если?..

Рассмотрим, что произошло бы при выборе альтернативного решения, когда каждый вызов виртуальной функции `f()` приводит к вызову замещающей:

```
void D::f()
{
   // операция, написанная из расчета на то, что D::X правильно
   // инициализирован в D::D
}
```

Если бы замещающую функцию допускалось вызывать в процессе конструирования, ни в одной виртуальной функции не было бы гарантии, что объект будет правильно инициализирован конструкторами. Поэтому в любой замещающей функции пришлось бы так же аккуратно контролировать все ошибки, как это обычно делают в конструкторах. На самом деле написание замещающей функции оказалось бы сложнее, чем написание конструкторов, так как в последнем сравнительно просто определить, что было проинициализировано, а что нет. Автор замещающей функции, не имеющий гарантий того, что конструктор уже отработал, мог бы:

- продолжать неоправданно надеяться на завершение инициализации;
- пытаться защититься от неинициализированных базовых классов и членов.

Первый вариант лишает конструкторы привлекательности. Второй – практически нереализуем, поскольку у производного класса может быть много непосредственных и косвенных базовых классов, а во время выполнения нет возможности проверить, была ли инициализирована произвольная переменная.

```
void D::f()  // нехарактерно для C++
{
   if (base_initialized) {
            // действие, опирающееся на то, что D::X
            // инициализирована D:D
   }
   else {
       // сделать все возможное, не полагаясь на то,
       // что D::X инициализирована
   }
}
```

Следовательно, если бы конструкторы вызывали замещающие функции, то для разумного написания последних пришлось бы значительно ограничить использование конструкторов.

При проектировании языка я принял такую точку зрения: пока конструктор объекта полностью не отработал, объект надо рассматривать, как недостроенное здание. В обмен на это разрешается считать, что после завершения конструирования объект пригоден к использованию.

13.3. Константные функции-члены

В Cfront 1.0 константность поддерживалась не вполне последовательно. Когда реализация стала более строгой, в определении языка обнаружилось несколько прорех. Программисту нужно было предоставить возможность определять, какие функции-члены могут изменять состояние объекта, а какие – нет:

```
class X {
   int aa;
public:
   void update() { aa++; }
   int value() const { return aa; }
   void cheat() const { aa++; }      // ошибка: *this константный
};
```

Функция-член, объявленная с модификатором `const`, например `X::value()`, называется константной. Гарантируется, что такая функция не изменяет значения объекта. Константную функцию-член можно применять как к константным, так и к неконстантным объектам. Но неконстантная функция-член, такая как `X::update()`, может применяться только к таким же объектам:

```
int g(X o1, const X& o2)
{
   o1.update();  // правильно
   o2.update();  // ошибка: o2 - константный объект
   return o1.value() + o2.value();   // правильно
}
```

Технически такая модель обеспечивается за счет того, что указатель `this` для неконстантной функции-члена класса `X` указывает на `X`, а для константной – на `const X`.

Различие между константными и неконстантными функциями позволяет выразить в C++ разделение между теми функциями, которые модифицируют состояние объекта, и теми, которые этого не делают. Константные функции-члены оказались среди тех свойств языка, которые получили большую поддержку во время дискуссий на семинаре разработчиков компиляторов в Эстес Парк (см. раздел 7.1.2).

13.3.1. Игнорирование const при приведении типов

C++ спроектирован для того, чтобы обнаруживать случайные ошибки, а не предотвращать намеренное «мошенничество». Для меня это означало, что должна существовать возможность принудительного игнорирования `const`. Я думал, что

компилятор не должен запрещать программисту явно обходить систему типов. Так, в [Stroustrup, 1992b] читаем:

«Иногда бывает полезно, чтобы объекты представлялись пользователям константными, но на самом деле меняли свое состояние. Такие классы можно создавать с помощью явного приведения типов:

```
class XX {
   int a;
   int calls_of_f;
   int f() const { ((XX*)this)->calls_of_f++; return a; }
   // ...
};
```

Если есть явное преобразование типа, значит, программа спроектирована недостаточно удачно. Изменение состояния константного объекта может вводить в заблуждение, в определенном контексте приводить к ошибкам. Если же объект находится в постоянной памяти, то этот прием и вовсе не работает. Лучше вынести переменную часть объекта в отдельный объект:

```
class XXX {
   int a;
   int& calls_of_f;
   int f() const { calls_of_f++; return a; }
   // ...
   XXX() : calls_of_f(*new int) { /* ... */ }
   ~XXX() { delete &calls_of_f; /* ... */ }
   // ...
};
```

Не стоит забывать, что основная цель модификатора `const` – специфицировать интерфейс, а не помочь оптимизатору. Следует также отметить, что свобода действий и гибкость иногда бывают полезны, но могут применяться и неправильно».

Включение оператора `const_cast` (см. раздел 14.3.4) позволяет программисту отличить приведения типов с целью отбрасывания `const` от тех, которые применяются для других манипуляций с типами.

13.3.2. Уточнение определения const

Чтобы гарантировать, что некоторые, но не все константные объекты можно разместить в постоянной памяти (ПЗУ), я принял следующее правило: любой объект, имеющий конструктор (то есть нуждающийся в инициализации во время выполнения), не может размещаться в ПЗУ, а все остальные константные – могут. Это напрямую связано с вопросом о том, что, когда и как может быть инициализировано (см. раздел 3.11.4). Данное правило позволяет инициализировать константные объекты во время выполнения, но допускает также и размещение в ПЗУ тех объектов, которые в такой инициализации не нуждаются. Типичным примером служит большой массив простых объектов, скажем, таблица, сгенерированная синтаксическим анализатором YACC.

Привязка концепции `const` к конструкторам была компромиссом между самой идеей `const`, соображением о том, что программисту нужно доверять, и существующей ситуацией в области аппаратного обеспечения. По предложению Джерри

Шварца это правило было заменено другим, более точно отражающим мое первоначальное намерение относительно const. Объект, объявленный как const, считается неизменным с момента завершения конструирования до начала работы деструктора. Результат операции записи в объект между этими двумя моментами не определен.

Когда я первоначально проектировал концепцию const, то, помнится, говорил, что в идеале константный объект был бы записываемым, пока не отработал до конца конструктор, затем становился доступным только для чтения, а после входа в деструктор он снова мог изменяться. Можно вообразить себе тэгированную архитектуру, благодаря которой обеспечивается подобное поведение. При такой реализации возникала бы ошибка во время выполнения при попытке изменить объект, объявленный как const. С другой стороны, можно было бы попробовать изменить объект, не объявленный с модификатором const, но переданный по константной ссылке или указателю. В обоих случаях пользователь сначала должен «снять константность». В результате отмена const для объекта, который первоначально был объявлен константным, и последующее его изменение приводят к непредсказуемому результату. Но те же действия, примененные к объекту, который не был изначально объявлен константным, вполне законны и четко определены.

Отметим, что после такого уточнения правил семантика const не зависит от того, имеет ли тип конструктор или нет; в принципе конструктор есть у всех типов. Любой объект, объявленный как const, теперь можно размещать в ПЗУ, в сегменте кода, защищать с помощью контроля доступа и т.д. – все это с целью гарантировать неизменность после первоначальной инициализации. Такая защита, однако, не является обязательной, так как современные системы в основном не в состоянии предохранить каждый константный объект от всех видов искажения.

Реализация все еще оставляет большую свободу выбора в том, как следует управлять константными объектами. Не возникает логической проблемы в случае, если сборщик мусора или система баз данных изменит значение константного объекта (например, перенесет его на диск или обратно), при условии, что для пользователя объект будет выглядеть неизменившимся.

13.3.3. Ключевое слово mutable и приведение типов

Снятие модификатора const все-таки вызывает некоторые возражения: во-первых, это разновидность приведения типов, во-вторых, не гарантируется правильная работа во всех случаях. Как можно написать класс типа XX (см. раздел 13.3.1), который не нуждался бы ни в приведении типов, ни в косвенном обращении, как в классе XXX? Томас Нго (Thomas Ngo) предложил, что должна быть возможность объявить член таким образом, чтобы он никогда не считался константным, даже если является членом константного объекта. Данное предложение рассматривалось в комитете в течение нескольких лет, пока Джерри Шварц не нашел вариант, который всех устроил. Первоначально предлагалась нотация ~const для выражения семантики «никогда не может быть const». Даже некоторые сторонники самой концепции считали такую нотацию слишком неудачной, поэтому в одобренном комитетом ANSI/ISO варианте фигурировало ключевое слово mutable:

```
class XXX {
   int a;
   mutable int cnt; // cnt никогда не может быть const
public:
   int f() const { cnt++; return a; }

   // ...
};

XXX var;         // var.cnt изменяема (разумеется)

const XXX cnst;  // cnst.cnt изменяема, так как
                 // XXX::cnt объявлен со словом mutable
```

Концепция действительно снижает потребность в приведениях типов в реальных системах, но не до такой степени, как ожидалось. Даг Брюк и другие изучили много реальных программ для уточнения того, какие приведения использовались для снятия константности и от каких можно было бы отказаться за счет `mutable`. Исследование подтвердило вывод о том, что в общей ситуации от «снятия `const`» нельзя отказаться (см. раздел 14.3.4) и что слово `mutable` позволяет избежать этого менее чем в половине случаев. Преимущества, которые дает `mutable`, похоже, сильно зависят от стиля программирования. В каких-то программах все приведения можно было бы заменить использованием `mutable`, а в других – ни одного.

Некоторые пользователи надеялись, что пересмотр концепции `const` и введение `mutable` открывают дорогу серьезным оптимизациям кода. Вряд ли это так. Основное достоинство – повышение ясности кода и увеличение числа объектов с предвычисленными значениями, которые можно поместить в ПЗУ, сегменты кода и т.д.

13.4. Статические функции-члены

Статический (`static`) член-данные класса – это такой член, копия которого существует в единственном экземпляре, а не в каждом объекте. Значит, к статическому члену можно получить доступ, не ссылаясь ни на какой объект. Статические члены уменьшают число глобальных имен, делают очевидным, к какому классу логически принадлежат статические объекты; также обращение к ним осуществляется в соответствии с правилами контроля доступа. Это, безусловно, удобно для поставщиков библиотек, поскольку позволяет избежать загрязнения глобального пространства имен и, следовательно, упрощает написание кода библиотеки и делает более безопасной работу с несколькими библиотеками.

Данные соображения применимы к функциям в той же мере, что и к объектам. Поставщик библиотеки обычно хочет сделать неглобальными имена функций. Я заметил, что для имитации статических функций-членов применялись непереносимые конструкции вроде `((X*)0)->f()`. Этот прием – бомба замедленного действия, поскольку рано или поздно кто-то объявит функцию, вызываемую таким образом, виртуальной. Тогда последствия вызова будут непредсказуемы, ибо по нулевому адресу нет объекта класса `X`. Но даже если `f()` невиртуальная функция,

такие вызовы завершатся неудачно при работе с некоторыми компиляторами, поддерживающими динамическое связывание.

Во время курса, который я читал в 1987 г. для EUUG (Европейская группа пользователей UNIX) в Хельсинки, Мартин О'Риордан указал мне, что статические функции-члены – очевидное и полезное обобщение. Возможно, именно тогда эта идея и прозвучала впервые. Мартин в то время работал в ирландской компании Glockenspiel, а позже стал главным архитектором компилятора Microsoft C++. Позже Джонатан Шопиро поддержал эту идею и не дал ей затеряться во время работы над версией 2.0.

Статическая функция-член остается членом класса, значит, ее имя находится в области действия класса и к нему применимы обычные правила контроля доступа. Например:

```
class task {
   // ...
   static task* chain;
public:
   static void schedule(int);
   // ...
};
```

Объявление статического члена – это всего лишь объявление, так что соответствующий объект или функция должны иметь где-то в программе единственное определение. Например:

```
task* task::chain = 0;
void task::schedule(int p) { /* ... */ }
```

Статическая функция-член не ассоциирована ни с каким конкретным объектом, и для ее вызова не нужен специальный синтаксис для функций-членов. Например:

```
void f(int priority)
{
   // ...
   task::schedule(priority);
   // ...
}
```

В некоторых случаях класс используется просто как область действия, в которую глобальные имена помещаются под видом статических членов, чтобы не засорять глобальное пространство имен. Это является одним из источников концепции пространств имен (см. главу 17).

Статические функции-члены оказались среди тех свойств языка, которые были единодушно поддержаны во время дискуссий на семинаре разработчиков компиляторов в Эстес Парк (см. раздел 7.1.2).

13.5. Вложенные классы

Как уже отмечалось в разделе 3.12, вложенные классы повторно введены в C++ в ARM. Это сделало более регулярными правила областей действия и улучшило средства локализации информации. Теперь стало возможно написать:

```
class String {
   class Rep {
      // ...
   };
   Rep* p;    // String - это описатель для Rep
   static int count;
   // ...
public:
   char& operator[](int i);
   // ...
};
```

чтобы оставить класс `Rep` локальным. К сожалению, это приводило к увеличению объема информации, необходимой для объявления классов, следовательно, к увеличению времени компиляции и частоты перекомпиляций. Слишком много интересной и изменяющейся информации помещалось во вложенные классы. Во многих случаях эта информация была неважна для пользователей такого класса, как `String` и ее следовало расположить в другом месте, наряду с прочими деталями реализации. Тони Хансен предложил разрешить опережающие объявления вложенных классов точно так же, как это делалось для функций-членов и статических членов:

```
// файл String.h (интерфейс):

class String {
   class Rep;
   Rep* p;    // String - описатель для Rep
   static int count;
   // ...
public:
   char& operator[](int i);
   // ...
};

// файл String.c (реализация):

class String::Rep {
   // ...
};

static int String::count = 1;

char& String::operator[](int i)
{
   // ...
}
```

Это расширение было принято просто как исправление недочета. Однако технику, которую оно поддерживает, не следует недооценивать. Ведь многие продолжают

перегружать заголовочные файлы всякой чепухой и страдают оттого, что перекомпиляция занимает много времени. Поэтому так важны любой прием, средство, которые позволяют уменьшить зависимость пользователя от разработчика компилятора.

13.6. Ключевое слово inherited

На одном из первых заседаний комитета по стандартизации Дэг Брюк предложил расширение, которым заинтересовались несколько человек [Stroustrup, 1992b]:

«Многие иерархии классов строятся «инкрементно», путем расширения функциональности базового класса за счет производного. Как правило, функция из производного класса вызывает функцию из базового, а затем выполняет некоторые дополнительные действия:

```
struct A { virtual void handle(int); };
struct D : A { void handle(int); };
void D::handle(int i);
{
   A::handle(i);
   // прочие действия
}
```

Вызов `handle()` необходимо квалифицировать как имя класса, чтобы избежать рекурсивного зацикливания. С помощью предлагаемого расширения этот пример можно было бы переписать так:

```
void D::handle(int i);
{
   inherited::handle(i);
   // прочие действия
}
```

Квалификацию с помощью ключевого слова `inherited` допустимо рассматривать как обобщение квалификации именем класса. Тогда решается целый ряд связанных с этим потенциальных проблем, что особенно важно при сопровождении библиотек классов».

Я обдумывал это предложение на ранних стадиях проектирования C++, но предпочел квалификацию именем базового класса, поскольку такой способ мог работать множественным наследованием, а `inherited::` – нет. Однако Дэг отметил, что описанные проблемы можно решить, сочетая обе схемы:

«По большей части при разработке иерархий имеется в виду одиночное наследование. Если изменить дерево наследования так, что класс `D` наследует одновременно `A` и `B`, то получим:

```
struct A { virtual void handle(int); };
struct B { virtual void handle(int); };
struct D : A, B { void handle(int); };

void D::handle(int i);
{
   A::handle(i);           // однозначно
   inherited::handle(i);   // неоднозначно
}
```

В таком случае `A::handle()` – корректная конструкция C++, хотя, возможно, и неправильная. Использование `inherited::handle()` здесь неоднозначно и вызывает ошибку во время компиляции. Я думаю, что такое поведение желательно, ибо заставляет пользователя, объединяющего иерархии, явно разрешать неоднозначность. С другой стороны, данный пример показывает, что при использовании множественного наследования полезность ключевого слова `inherited::` ограничена».

Меня убедили эти доводы, к тому же сопроводительная документация была тщательно продумана. Таким образом, предложение полезно, нетрудно для понимания и реализации. Имелся и опыт применения, поскольку некий подобный вариант был реализован компанией Apple на основе экспериментов с Object Pascal, а кроме того, это не что иное, как вариация на тему слова `super` в Smalltalk.

Но вслед за последним обсуждением предложения в комитете Дэг сам посоветовал включить это в книгу как хрестоматийный пример хорошей, но непринятой идеи [Stroustrup, 1992b]:

«Предложение хорошо аргументировано... В данном случае его реализовывал представитель от компании Apple. При обсуждении мы быстро пришли к выводу, что серьезных изъянов нет. В отличие от более ранних предложений на ту же тему (некоторые из которых восходили еще к временам споров по поводу множественного наследования в 1986 г.) оно корректно разрешало неоднозначности при использовании множественного наследования. Мы согласились, что реализовать расширение будет просто, а программистам оно будет весьма полезно.

Заметим, что всего этого еще недостаточно, чтобы одобрить предложение. Нам известно несколько десятков мелких усовершенствований вроде этого и, по крайней мере, дюжина крупных. Если бы мы приняли их все, то язык утонул бы из-за собственной тяжести (вспомним корабль «Vasa»!). Однако в какой-то момент обсуждения вошел Майкл Тиман, бормоча себе под нос что-то вроде «но нам такое расширение ни к чему; мы и без него умеем это делать». Когда утихли возгласы «да нет же!», Майкл продемонстрировал нам следующее:

```
class foreman : public employee {
   typedef employee inherited;
   // ...
   void print();
};

class manager : public foreman {
   typedef foreman inherited;
   // ...
   void print();
};

void manager::print()
{
   inherited::print();
   // ...
}
```

Дальнейшее обсуждение данного примера можно найти в [2nd, стр. 205]. Мы не заметили, что восстановление в C++ вложенных классов открыло возможность управления областью действия и разрешением имен типов точно так же, как и любых других имен.

Увидев, что такой прием есть, мы решили направить усилия на работу над каким-нибудь другим аспектом стандарта. Преимущества `inherited::` как встроенного средства не перевешивали того факта, что программист и так мог использовать определенное ключевое слово для обозначения базового класса, применяя уже имеющиеся возможности. Поэтому было решено не принимать расширение `inherited::`».

13.7. Ослабление правил замещения

Рассмотрим, как можно написать функцию, возвращающую копию объекта. Если имеется копирующий конструктор, то это тривиально:

```
class B {
public:
   virtual B* clone() { return new B(*this); }
   // ...
};
```

Теперь можно клонировать объект любого класса, производного от `B`, если в нем замещена функция `B::clone`. Например:

```
class D : public B {
public:
      // старое правило:
      // clone() должна возвращать B*, если хочет заместить B::clone()
   B* clone() { return new D(*this); }

   void h();
   // ...
};

void f(B* pb, D* pd)
{
   B* pb1 = pb->clone();
   B* pb2 = pd->clone();   // pb2 указывает на D
   // ...
}
```

Увы, знание о том, что `pd` указывает на `D` (или на объект какого-то класса, производного от `D`), утрачено:

```
void g(D* pd)
{
   B* pb1 = pd->clone();   // правильно
   D* pd1 = pd->clone();   // ошибка: clone() возвращает B*
   pd->clone()->h();       // ошибка: clone() возвращает B*

   // неудачный обходной путь

   D* pd2 = (D*)pd->clone();
   ((D*)pd->clone())->h();
}
```

Оказалось, в реальных программах это мешает. Также было отмечено, что правило, согласно которому замещающая функция должна иметь в точности тот же тип, что и замещаемая, можно ослабить, не нарушая систему типов и не усложняя реализацию. Например, можно было бы разрешить такое:

```
class D : public B {
public:
      // заметим, clone() возвращает D*
   D* clone() { return new D(*this); }

   void h();
   // ...
};

void gg(B* pb, D* pd)
{
   B* pb1 = pd->clone();   // правильно
   D* pd1 = pd->clone();   // правильно
   pd->clone()->h();       // правильно

   D* pd2 = pb->clone();   // ошибка (как и раньше)
   pb->clone()->h();       // ошибка (как и раньше)
}
```

Это расширение – предложил Алан Снайдер (Alan Snyder) – стало первым официально представленным на рассмотрение комитета и было принято в 1992 г.

Но прежде чем одобрить его, следовало ответить на два вопроса:

- не вызовет ли расширение каких-либо серьезных проблем в реализации (скажем, связанных с множественным наследованием или указателями на члены)?
- какие преобразования, которые можно было бы применить к типу значения, возвращаемого замещающей функцией, стоит рассматривать?

Первый вопрос меня не очень волновал: я думал, что знаю, как реализовать ослабление правил в стандартном случае. Однако Мартина О'Риордана это беспокоило, поэтому он представил комитету детальное письменное обоснование возможности реализации.

Трудней всего было решить, стоит ли вообще заниматься ослаблением и для каких именно преобразований. Часто ли возникает необходимость вызывать виртуальные функции для объекта производного типа и выполнять операции над возвращаемым значением, имеющим этот тип? Несколько человек, в особенности Джон Бранс (John Bruns) и Билл Гиббонс, уверенно доказывали, что такая необходимость возникает постоянно, а не только в примерах из теории компиляторов, вроде `clone`. В конце концов меня убедило сделанное Тедом Голдстейном (Ted Goldstein) наблюдение, что почти две трети всех приведений типов в программе из 100 тыс. строк, над которой он работал в Sun, были нужны только для того, чтобы обойти трудности, не возникшие бы при ослаблении правила замещения. Другим словами, ослабление позволяет продолжать важную работу, оставаясь в рамках системы типов и не прибегая к приведениям. Итак, ослабление правила

о типах значений, возвращаемых замещающими функциями, оказывалось в русле моих усилий сделать программирование на C++ более безопасным, простым и декларативным. Данное средство не только устраняло много обычных приведений типов, но также препятствовало искушению злоупотребить новыми динамическими приведениями, которые обсуждались в то же самое время (см. раздел 14.2.3).

Рассмотрев несколько вариантов, мы разрешили замещение `B*` на `D*` и `B&` на `D&`, где `B` – достижимый базовый класс для `D`. Кроме того, можно добавлять или убирать `const`, если это безопасно. Не допускались такие технически возможные преобразования, как `D` в достижимый базовый класс `B`, `D` – в класс `X`, для которого есть преобразование из `D`, `int*` в `void*`, `double` в `int` и т.д. Нам казалось, что преимущества подобных преобразований не перевешивают стоимости реализации и потенциального запутывания пользователей.

13.7.1. Ослабление правил аргументов

По моему опыту, ослаблению правила замещения для типа возвращаемого значения неизменно сопутствует неприемлемое «эквивалентное» предложение ослабить правила для типов аргументов. Например:

```
class Fig {
public:
   virtual int operator==(const Fig&);
   // ...
};

class ColFig : public Fig {
public:
   // Предполагается, что ColFig::operator==()
   // замещает Fig::operator==()
   // (не разрешено в C++)

   int operator==(const ColFig& x);
   // ...
private:
   Color col;
};

int ColFig::operator==(const ColFig& x)
{
   return col == x.col && Fig::operator==(x);
}
```

Выглядит привлекательно и позволяет писать неплохой код, например:

```
void f(Fig& fig, ColFig& cf1, ColFig& cf2)
{
   if (fig==cf1) {  // сравнить объекты Fig
      // ...
   } else if (cf1==cf2) {  // сравнить объекты ColFig
      // ...
   }
}
```

К сожалению, такое написание ведет и к неявному нарушению системы типов:

```
void g(Fig& fig, ColFig& cf)
{
   if (cf==fig) {   // что сравнивается?
      // ...
   }
}
```

Если `ColFig::operator==()` замещает `Fig::operator==()`, то при выполнении сравнения `cf==fig` будет вызван оператор `ColFig::operator==()` с аргументом типа `Fig`. Это неудачно, так как `ColFig::operator==()` обращается к члену `col`, а в классе `Fig` нет такого члена. Если бы для `ColFig::operator==()` было бы решено изменить его аргумент, то испортилась бы память. Я рассматривал такой сценарий, когда только начинал проектировать правила для виртуальных функций, и счел его неприемлемым.

Если бы данное разрешение было позволено, пришлось бы проверять каждый аргумент виртуальной функции во время выполнения. Устранить такие проверки во время оптимизации кода было бы нелегко. Без проведения глобального анализа мы никогда не узнаем, не объявлен ли объект в каком-то другом файле, причем с типом, для которого замещение потенциально опасно. Затраты, связанные с такой проверкой, довольно существенны. Кроме того, если каждый вызов виртуальной функции может в принципе возбуждать исключение, то пользователю надо быть к этому готовым. Я счел, что не смогу пойти на такой шаг.

В качестве альтернативного решения можно поручить программисту явно проверять, когда для аргумента типа производного класса необходимо использовать другую ветвь обработки. Например:

```
class Figure {
public:
   virtual int operator==(const Figure&);
   // ...
};

class ColFig : public Figure {
public:
   int operator==(const Figure& x);
   // ...
private:
   Color col;
};

int ColFig::operator==(const Figure& x);
{
   if (Figure::operator==(x)) {
      const ColFig* pc = dynamic_cast<const ColFig*>(&x);
      if (pc) return col == pc->col;
   }
   return 0;
}
```

При таком подходе проверяемое во время выполнения приведение с помощью dynamic_cast (см. раздел 14.2.2) служит дополнением к ослабленным правилам замещения. Ослабление позволяет безопасно и декларативно работать с типами возвращаемых значений; dynamic_cast не препятствует явной и относительно безопасной работе с типами аргументов.

13.8. Мультиметоды

Я неоднократно возвращался к механизму вызова виртуальных функций на базе более чем одного объекта. Часто его называют механизмом мультиметодов. Отказался я от мультиметодов с сожалением, поскольку идея мне нравилась, но найти для нее приемлемую форму не получалось. Рассмотрим пример:

```
class Shape {
   // ...
};

class Rectangle : public Shape {
   // ...
};

class Circle : public Shape {
   // ...
};
```

Как можно было бы спроектировать функцию пересечения двух фигур intersect(), которая бы корректно вызывалась для обоих своих аргументов? Например:

```
void f(Circle& c, Shape& s1, Rectangle& r, Shape& s2)
{
   intersect(r,c);
   intersect(c,r);
   intersect(c,s2);
   intersect(s1,r);
   intersect(r,s2);
   intersect(s1,c);
   intersect(s1,s2);
}
```

Если r и s относятся соответственно к Circle и Shape, то хотелось бы реализовать intersect четырьмя функциями:

```
bool intersect(const Circle&, const Circle&);
bool intersect(const Circle&, const Rectangle&);
bool intersect(const Rectangle&, const Circle&);
bool intersect(const Rectangle&, const Rectangle&);
```

Каждый вызов должен был бы обращаться к нужной функции точно так же, как в случае с виртуальными функциями. Но правильную функцию следует выбирать на основе вычисляемых во время выполнения типов обоих аргументов. Фундаментальная проблема, как мне виделось, заключалась в том, чтобы найти:

- механизм вызова, который был бы таким же простым и эффективным, как поиск в таблице виртуальных функций;
- набор правил, которые позволяли бы разрешать все неоднозначности на этапе компиляции.

Видимо, эта проблема разрешима, но мое внимание никогда не было сосредоточено на ней столь длительное время, чтобы успеть продумать все детали.

Если я хотел, чтобы программа работала быстро, то для получения эквивалента таблицы виртуальных функций требовалось, судя по всему, очень много памяти. Любое же решение, не расходующее «впустую» много памяти на дублирование элементов таблиц, либо медленно осуществлялось, либо имело непредсказуемые характеристики эффективности, либо то и другое вместе. Например, каждая реализация примера с `Circle` и `Rectangle`, не требующая поиска подходящей функции во время выполнения, нуждалась в четырех указателях на функции. Добавим еще класс `Triangle` – и требуется уже девять указателей. Произведем от `Circle` класс `Smiley` – и необходимо целых шестнадцать указателей, хотя семь из них следовало бы сэкономить, используя элементы таблицы, где вместо `Smiley` стоит `Circle`.

Более того, массивы указателей на функции, эквивалентные таблицам виртуальных функций, нельзя было построить, пока не станет известна вся программа, то есть на стадии компоновки. Причина в том, что не существует одного класса, которому принадлежат все замещающие функции. Да и не может его быть, так как любая замещающая функция зависит от двух или более аргументов. В то время проблема была неразрешима, поскольку я не хотел вводить в язык средств, требующих нетривиальной поддержки со стороны компоновщика. Однако, как оказалось впоследствии, такой поддержки можно дожидаться годами.

Меня беспокоил – хотя и не казался неразрешимым – вопрос о том, как разработать схему, синтаксис которой был бы однозначным. Очевидный ответ таков: все вызовы мультиметодов должны подчиняться таким же правилам, как и любые другие вызовы. Однако такое решение ускользало от меня, поскольку я искал какой-то специальный синтаксис и специальные правила для вызова мультиметодов. Например:

```
(r@s)->intersect(); // вместо intersect(r,s)
```

гораздо лучшее решение было предложено Дугом Леа [Lea, 1991]. Позволим явно объявлять аргументы со словом `virtual`. Например:

```
bool intersect(virtual const Shape&, virtual const Shape&);
```

Замещающей может быть любая функция, имя и типы аргументов которой сопоставляются с учетом ослабленных правил соответствия в стиле тех, что приняты для типа возвращаемого значения. Например:

```
bool intersect(const Circle&, const Rectangle&) // замещает
{
   // ...
}
```

И наконец, мультиметоды можно вызывать с помощью обычного синтаксиса вызова, как показано выше.

Мультиметоды – один из интересных вопросов вроде «что, если...» в C++. Мог бы я в то время спроектировать и реализовать их достаточно хорошо? Было бы их применение настолько важным, чтобы оправдать потраченные усилия? Какую работу пришлось бы оставить незавершенной, чтобы хватило времени на проектирование и реализацию мультиметодов? Примерно с 1985 г. я сожалею, что не включил это средство в язык. Единственный раз я официально упомянул о мультиметодах на конференции OOPSLA, когда выступил против языковой нетерпимости и бессмысленности ожесточенных баталий по поводу языков [Stroustrup, 1990]. Я привел в качестве примера некоторые концепции из языка CLOS, которые мне чрезвычайно нравились, и особо выделил мультиметоды.

13.8.1. Когда нет мультиметодов

Так как же написать функцию, вроде `intersect()`, без мультиметодов?

До появления идентификации типа во время исполнения (см. раздел 14.2) единственным средством динамического разрешения на основе типа были виртуальные функции. Поскольку нам нужно разрешение на основе двух аргументов, то виртуальную функцию придется вызывать дважды. В приведенном выше примере с `Circle` и `Rectangle` есть три возможных статических типа аргумента, поэтому можно предоставить три виртуальные функции:

```
class Shape {
   // ...
   virtual bool intersect(const Shape&) const =0;
   virtual bool intersect(const Rectangle&) const =0;
   virtual bool intersect(const Circle&) const =0;
};
```

В производных классах эти функции замещаются:

```
class Rectangle {
   // ...
   bool intersect(const Shape&) const;
   bool intersect(const Rectangle&) const;
   bool intersect(const Circle&) const;
};
```

Любой вызов `intersect()` разрешается вызовом подходящей функции из классов `Circle` или `Rectangle`. Затем надо позаботиться о том, чтобы функция, принимающая неконкретизированный аргумент типа `Shape`, вызывала другую виртуальную функцию для замещения его более конкретным:

```
bool Rectangle::intersect(const Shape& s) const
{
   return s.intersect(*this); // *this - это Rectangle:
          // разрешаем в зависимости от s
}
bool Circle::intersect(const Shape& s) const
```

```
{
   return s.intersect(*this); // *this - это Circle:
          // разрешаем в зависимости от s
}
```

Остальные функции `intersect()` просто выполняют то, что от них требуется, «зная» типы обоих аргументов. Заметим, что для применения такой техники необходима лишь первая функция `Shape::intersect()`. Оставшиеся две функции в классе `Shape` – просто оптимизация, возможная, если о производном классе все известно во время проектирования базового.

Этот прием получил название двойной диспетчеризации и впервые был описан в работе [Ingalls, 1986]. Для C++ данный метод плох тем, что при добавлении класса в иерархию необходимо изменять уже существующие классы. Производный класс, например `Rectangle`, должен «знать» обо всех своих «братьях», иначе не удастся составить правильный набор виртуальных функций. Так, при добавлении класса `Triangle` придется модифицировать и `Rectangle`, и `Circle`, а также – если желательна показанная выше оптимизация – и `Shape`.

```
class Rectangle : public Shape {
   // ...
   bool intersect(const Shape&);
   bool intersect(const Rectangle&);
   bool intersect(const Circle&);
   bool intersect(const Triangle&);
};
```

В целом двойная диспетчеризация в C++ – это эффективный метод «иерархического продвижения», если есть возможность модифицировать объявления при добавлении новых классов и если набор производных классов меняется не слишком часто.

Иной способ – хранить в объектах определенный идентификатор типа и после его проверки выбирать, какую функцию вызывать. Использование функции `typeid()` для идентификации типа во время выполнения (см. раздел 14.2.5) – одна из возможностей. Можно завести структуру, где хранятся указатели на функции, и использовать для доступа к ней идентификатор типа. Достоинство данного метода в том, что базовый класс ничего не должен «знать» о существовании производных. Например, при наличии подходящих определений функция

```
bool intersect(const Shape* s1, const Shape* s2)
{
   int i = find_index(s1.type_id(),s2.type_id());
   if (i < 0) error("плохой индекс");
   extern Fct_table* tbl;
   Fct f = tbl[i];
   return f(s1,s2);
}
```

будет вызывать правильную функцию при любых допустимых типах обоих своих аргументов. По сути дела, это реализация вручную вышеупомянутой таблицы виртуальных функций для мультиметодов.

Относительная легкость, с которой можно имитировать мультиметоды в любом конкретном примере, и была той причиной, по которой решение данной проблемы достаточно долго не представлялось срочным. Как ни суди, это не что иное, как техника, всегда применявшаяся для имитации виртуальных функций в C. Такие обходные пути приемлемы, если нужда в них возникает не слишком часто.

13.9. Защищенные члены

Простая модель сокрытия данных на основе закрытых и открытых секций класса прекрасно работала в C++, пока он использовался в основном как язык для абстрагирования данных. Данная модель была вполне пригодна и для большого класса задач, где наследование применялось для объектно-ориентированного программирования. Однако, когда речь заходит о классах, выясняется, что к любому из них обращаются либо производные от него классы, либо прочие классы и функции. Функции-члены и дружественные функции, реализующие операции с классом, воздействуют на его объекты от имени одного из этих «пользователей». Механизм открытых и закрытых членов позволяет четко различать автора класса и т.п., но не дает возможности приспособить поведение к нуждам производных классов.

Вскоре после выхода версии 1.0 ко мне в кабинет заглянул Марк Линтон и умолял добавить третий уровень контроля доступа, который напрямую поддержал бы стиль, применяемый в библиотеке Interviews (см. раздел 8.4.1). Она разрабатывалась в Стэнфорде. Мы выбрали слово `protected` для обозначения членов класса, которые вели себя как «открытые» для прочих членов самого этого класса и производных от него, но как «закрытые» для всех остальных «пользователей».

Марк был главным создателем библиотеки Interviews. Основываясь на опыте и примерах из настоящих программ, он убедительно доказывал, что защищенные данные абсолютно необходимы при проектировании эффективной и расширяемой инструментальной библиотеки для X Windows. Альтернативой защищенным данным, по его словам, были неприемлемо низкая эффективность, неуправляемое распространение встраиваемых интерфейсных функций или глобальные данные. Защищенные данные и, в более широком смысле, защищенные члены казались меньшим злом. Кроме того, так называемые чистые языки вроде Smalltalk поддерживали именно такую – довольно слабую – защиту вместо более сильной защиты C++, основанной на концепции `private`. Мне самому приходилось писать код, в котором данные были объявлены как `public` просто для того, чтобы ими можно было пользоваться в производных классах. Видел я также и программы, в которых концепция `friend` использовалась неудачно и лишь ради того, чтобы дать доступ явно поименованным производным классам.

Это были веские аргументы, убедившие меня в том, что защищенные члены надо разрешить. Однако «веские аргументы» могут найтись для каждого мыслимого языкового средства и для любого возможного его использования. Нам же нужны факты. Не имея фактов и должным образом осознанного опыта, мы уподобляемся греческим философам, которые на протяжении нескольких веков вели блистательные дискуссии, но так и не смогли договориться, из каких же четырех (а может, пяти) основных субстанций состоит Вселенная.

Примерно через пять лет Марк запретил использование защищенных членов-данных в Interviews, поскольку они стали источником ошибок и серьезно усложнили сопровождение. В пленарном докладе Барбары Лисков (Barbara Liskov) на конференции OOPSLA [Liskov, 1987] достаточно подробно описываются теоретические и практические проблемы, связанные с контролем доступа на основе концепции `protected`. Мой опыт показывает, что всегда существовали альтернативы размещению информации в общем базовом классе, где она была бы доступной производным классам. Я сомневался по поводу `protected` еще и потому, что при небрежном программировании общий базовый класс слишком уж легко использовать как разновидность глобальных данных.

К счастью, вас никто не принуждает пользоваться защищенными данными в C++. По умолчанию члены класса имеют атрибут `private`, и обычно такой выбор лучше. Замечу, впрочем, что ни одно из этих возражений не относится к защищенным членам-функциям. Убежден, что `protected` прекрасно подходит для специфицирования операций, которыми можно пользоваться в производных классах.

Защищенные члены появились в версии 1.2. Защищенные базовые классы были впервые описаны в ARM и включены в версию 2.1. Оглядываясь назад, я думаю, что `protected` – это тот случай, когда «веские аргументы» и мода сбили меня с пути истинного и заставили отказаться от своих убеждений ради новых возможностей.

13.10. Улучшенная генерация кода

Для некоторых пользователей самой важной особенностью версии 2.0 была не какая-то новая возможность, а простая оптимизация использования памяти. С самого начала Cfront генерировал неплохой код. Даже в 1992 г. код, произведенный Cfront, был лучшим в сравнительных тестах компиляторов C++ на платформе SPARC. Если не считать оптимизации возврата значений, предложенной в [ARM, §12.1c] и реализованной в версии 3.0, то никаких существенных улучшений в области генерации кода с версии 1.0 в Cfront не было. Но первая версия впустую расходовала память, так как для каждой единицы трансляции генерировался собственный набор таблиц виртуальных функций всех классов, использовавшихся в этой единице. Объем напрасно потраченной памяти мог достигать нескольких мегабайтов. В то время (примерно 1984 г.) я считал этот расход необходимым из-за отсутствия поддержки со стороны компоновщика. К 1987 г. поддержки так и не появилось. Поэтому я вернулся к проблеме и решил ее с помощью простого приема, разместив таблицу виртуальных функций класса сразу после определения первой невиртуальной и невстраиваемой функции в нем. Например:

```
class X {
public:
   virtual void f1() { /* ... */ }
   void f2();
   virtual void f3() = 0;
   virtual void f4();   // первая невстраиваемая
                        // невиртуальная функция
   // ...
};
```

```
// в каком-то файле

void X::f4() { /* ... */ }

// Cfront поместит таблицу виртуальных функций здесь
```

Я выбрал такой вариант, поскольку он осуществляется вне зависимости от компоновщика. Решение несовершенно, так как память все равно тратится зря для классов, в которых нет невстраиваемых невиртуальных функций, но подобные расходы перестали быть серьезной проблемой на практике. В обсуждении деталей такой оптимизации принимали участие Эндрю Кениг и Стэн Липпман. Конечно, в других компиляторах может быть реализовано иное решение, более соответствующее среде, в которой они работают.

В качестве альтернативы мы рассматривали такую возможность: оставить генерацию таблиц виртуальных функций в каждой единице трансляции, а затем запустить препроцессор, который удалит все таблицы, кроме одной. Однако это трудно было сделать переносимым способом, кроме того, способ неэффективен. Зачем генерировать таблицы, а потом только тратить время на их удаление? Но возможности остаются открытыми для тех, кто готов создавать собственный компоновщик.

13.11. Указатели на функции-члены

Первоначально в языке не было способа представить указатель на функцию-член. Приходилось использовать обходные пути и преодолевать ограничение в таких случаях, как обработка ошибок, где традиционно использовались указатели на функции. Если есть объявление

```
struct S {
   int mf(char*);
}
```

то по привычке писался такой код:

```
typedef void (*PSmem)(S*,char*);

PSmem m = (PSmem)&S::mf;

void g(S* ps)
{
   m(ps, "hello");
}
```

Для того чтобы данный прием работал, приходилось повсюду включать в код явные приведения типов, которых в принципе не должно было быть. Этот код к тому же базировался на предположении, что функции-члену передается указатель на объект ее класса («указатель `this`») в качестве первого аргумента, то есть так, как реализовано в Cfront (см. раздел 2.5.2).

Еще в 1983 г. я пришел к выводу, что такой вариант никуда не годится, но считал, что данную проблему легко решить и тем самым закрыть «дыру» в системе

типов C++. Закончив работу над версией 1.0, я нашел выход и включил указатель на функцию-член в версию 1.2. Оказалось, что для сред, в которых функции обратного вызова использовались как основной механизм коммуникации, решение имеет первостепенное значение.

Термин «указатель на член» несколько неточен, поскольку это, скорее, смещение, значение, идентифицирующее член объекта. Однако, назови я его «смещением», пользователи сделали бы ошибочный вывод, будто указатель на член – просто индекс, и могли бы решить, что к нему применимы какие-то арифметические операции. Это привело бы к еще большей путанице. Данный термин был выбран потому, что проектировал механизм в форме синтаксической параллели указателям в C.

Давайте рассмотрим синтаксис функции C/C++ на примере:

```
int f(char* p) { /* ... */ }  // определяем функцию
int (*pf)(char*) = &f;        // объявляем и инициализируем
                              // указатель на функцию
int i = (*pf)("hello");       // вызываем через указатель
```

Включая в подходящие места `S::` и `p->`, я построил прямую параллель с функциями-членами:

```
class S {
   // ...
   int mf(char*);
};

int S::mf(char* p); { /* ... */ }    // определяем функцию
int (S::*pmf)(char*) = &S::mf;       // объявляем и инициализируем
                                     // указатель на функцию-член
S* p;
int i = (p->*pmf)("hello");          // вызываем через указатель
                                     // и объект
```

Семантически и синтаксически понятие указателя на функцию-член осмысленно. Мне оставалось обобщить его, включив понятие данных-членов, и найти стратегию реализации. В разделе благодарностей работы [Lippman, 1988] есть такие строки:

«Проектирование указателей на члены было выполнено совместно Бьерном Страуструпом и Джонатаном Шопиро. Много полезных замечаний высказал также Дуг Макилрой. Стив Дьюхерст немало сделал для переработки механизма указателей на члены с учетом множественного наследования».

Тогда я многократно повторял, что указатели на члены оказались более полезными, чем планировалось изначально. То же можно сказать о версии 2.0 в целом.

Указатели на данные-члены оказались полезны и для описания размещения класса C++ в памяти таким способом, который не зависит от реализации [Hubel, 1992].

Глава 14. Приведение типов

Умный человек не станет изменять мир.

Дж. Б. Шоу

14.1. Крупные расширения

Шаблоны (см. главу 15), исключения (см. главу 16), идентификацию типов во время исполнения (см. раздел 14.2) и пространства имен (см. главу 17) часто называют крупными расширениями. Они оказывают влияние на способ организации программ, так что считать ли их расширениями или неотъемлемыми особенностями C++ – это дело вкуса.

Мелкие же расширения не влияют на структуру программы в целом, не сказываются на проектировании. Они названы так вовсе не потому, что для их описания требуется всего несколько строк в руководстве, а для реализации – несколько строк кода в компиляторе. На самом деле некоторые крупные расширения описать и реализовать проще, чем кое-какие мелкие.

Конечно, не каждую возможность, имеющуюся в языке, легко отнести к одной из этих двух категорий. Например, вложенные функции можно считать мелким или крупным расширением в зависимости от того, насколько важным вам кажется их применение для выражения итераций. Любопытно, что проявленный общественностью интерес к тому или иному средству обратно пропорционален его важности. С другой стороны, именно такое реагирование понятно: гораздо проще определить свое отношение к мелкому усовершенствованию, нежели к крупному; не очень существенные средства, в отличие от принципиально важных, нетрудно соотнести с текущим состоянием языка.

Поскольку поддержка построения библиотек и создания программного обеспечения из относительно независимых частей – главная цель C++, именно с ней и связаны крупные расширения: шаблоны, обработка исключений, идентификация типа во время исполнения и пространства имен. Первое и второе средства я с самого начала считал необходимой частью того, чем должен стать C++ (см. разделы 2.9.2 и 3.15). Идентификация типа во время исполнения рассматривалась еще в первой редакции предварительного стандарта C++ (см. раздел 3.5), но была отложена в надежде, что окажется ненужной. Пространства имен – единственное крупное расширение, которого не было в первоначальной концепции C++, оно решает проблему, к которой я безуспешно подступался в первой версии (см. раздел 3.12).

14.2. Идентификация типа во время исполнения

Во многих отношениях дискуссия об идентификации типа объекта во время исполнения напоминает обсуждение множественного наследования (см. раздел 12.6). Множественное наследование считалось первым крупным расширением изначального определения C++. Идентификация типа во время исполнения (часто ее называют RTTI – Run-Time Type Identification) – первое крупное расширение, которое заранее не предполагалось стандартизировать и которое не описано в ARM.

В C++ вводилась прямая поддержка нового стиля программирования, и некоторые пользователи начали:

- объявлять, что такая поддержка не нужна;
- утверждать, что новый стиль несовершенен («не в стиле C++»);
- говорить, что это чересчур расточительно;
- думать, что это слишком сложно и запутанно;
- предполагать, что новый стиль повлечет за собой массу новых расширений.

Кроме того, RTTI раскритиковали, поскольку она вообще была связана с механизмом приведения типов в C++. Так, многим пользователям не нравится, что старые приведения типов можно применять, чтобы обойти контроль доступа для закрыто наследуемых базовых классов и отменить действие модификатора `const`. Для такой критики есть важные и серьезные основания; они обсуждаются в разделе 14.3.

И снова, как и раньше, я защищал новое средство, доказывая, что для кого-то оно важно, а остальным не мешает, что реализовать его несложно и что если не поддержать RTTI напрямую, то пользователям придется ее эмулировать. Пришлось даже сделать экспериментальную реализацию за два утра, продемонстрировав, что RTTI, по крайней мере, на порядок проще исключений и шаблонов и на два порядка проще множественного наследования.

Добавить средства идентификации типа объекта во время исполнения предложил Дмитрий Ленков [Lenkov, 1991]. Он основывался на опыте использования больших библиотек на C++ типа Interviews [Linton, 1987], NIH [Gorlen, 1990] и ET++ [Weinand, 1988]. Исследовался также механизм досье (dossier) [Interrante, 1990].

Механизмы RTTI, которые предоставлялись разными библиотеками, были несовместимы между собой, и это стало препятствием для использования сразу нескольких библиотек. Кроме того, в каждом случае от проектировщика базовых классов требовалось многое предвидеть. Следовательно, был необходим какой-то механизм, который поддерживался бы самим языком.

Я занимался проектированием такого механизма в соавторстве с Ленковым [Stroustrup, 1992]. В июле 1991 г. на заседании в Лондоне предложение было представлено комитету ANSI/ISO, а в марте 1993 г. на заседании в Портленде его одобрили.

Механизм идентификации типа во время исполнения состоит из трех частей:

- оператор `dynamic_cast` для получения указателя на объект производного класса при наличии указателя на базовый класс этого объекта. Оператор

dynamic_cast возвращает такой указатель лишь тогда, когда объект, на который направлен исходный указатель, действительно принадлежит специфицированному производному классу. В противном случае он возвращает 0;

- оператор typeid для идентификации точного типа объекта при наличии указателя на базовый класс;
- структура type_info, позволяющая получить дополнительную информацию, ассоциированную с типом.

При обсуждении RTTI внимание будет уделено преимущественно указателям.

14.2.1. Зачем нужен механизм RTTI

Допустим, в библиотеке имеется класс dialog_box, интерфейсы которого выражены в терминах объектов этого класса. Я же пользуюсь как dialog_box, так и собственным классом dbox_w_str.

```
class dialog_box : public window {   // библиотечный класс
   // ...
public:
   virtual int ask();
   // ...
};

class dbox_w_str : public dialog_box {     // мой класс
   // ...
public:
   int ask();
   virtual char* get_string();
   // ...
};
```

Когда библиотека или система возвращает мне указатель на dialog_box, как я могу узнать, принадлежит ли на самом деле моему классу dbox_w_str?

Замечу, что я не могу модифицировать библиотеку, чтобы ввести в dbox_w_str идентифицирующие признаки. А даже если бы мог, то не стал бы, чтобы не думать об этом классе в последующих версиях и об ошибках, которые я тем самым мог внести в «стандартную» библиотеку.

14.2.2. Оператор dynamic_cast

Наивное решение – найти тип объекта, на который имеется указатель, и сравнить его с типом класса dbox_w_str:

```
void my_fct(dialog_box* bp)
{
   if (typeid(*bp) == typeid(dbox_w_str)) {     // наивное решение
      dbox_w_str* dbp = (dbox_w_str*)bp;

      // воспользоваться dbp
   }
   else {
```

```
        // работать с *bp, как с обычным диалоговым окном
    }
}
```

Оператор `typeid()`, которому в качестве операнда передано имя типа, возвращает объект, идентифицирующий этот тип. Если операндом является выражение, `typeid()` возвращает объект, который идентифицирует тип объекта, обозначаемого этим выражением. В частности, `typeid(*bp)` возвращает объект, позволяющий программисту задавать вопросы о типе объекта, на который указывает `bp`. В данном случае нас интересует только, совпадает ли тип этого объекта с типом `dbox_w_str`.

Это простейший из вопросов, которые можно задать, но, как правило, он некорректен. Спрашиваем-то мы, чтобы узнать, можно ли безопасно воспользоваться той или иной возможностью производного класса. А чтобы применить ее, нам нужен указатель на объект этого производного класса. В примере выше мы использовали приведение типа в строке, следующей за проверкой. Но ведь обычно нас интересует, удастся ли безопасно выполнить приведение. Этот запрос можно дать непосредственно, воспользовавшись оператором `dynamic_cast`:

```
void my_fct(dialog_box* bp)
{
    if (dbox_w_str* dbp = dynamic_cast<dbox_w_str*>(bp)) {

        // воспользоваться dbp
    }
    else {

        // работать с *bp, как с обычным диалоговым окном
    }
}
```

Оператор `dynamic_cast<T*>(p)` преобразует свой операнд в нужный тип `T*`, если `*p` действительно принадлежит типу `T` или типу класса, производного от `T`. В противном случае значение `dynamic_cast<T*>(p)` равно `0`.

Есть несколько причин для объединения проверки и приведения типа в одну операцию:

- динамическое приведение не допускает несоответствия между приведением и проверкой;
- пользуясь данными, которые содержатся в объектах с информацией о типе, можно выполнять приведение к типам, которые не полностью определены в области действия проверки;
- при таком же условии допустимо выполнять приведение от виртуального базового класса к производному (см. раздел 14.2.2.3);
- статическое приведение типов не дает правильных результатов во всех случаях (см. раздел 14.3.2.1).

В большинстве ситуаций можно обойтись оператором `dynamic_cast`. Думается, что это важнейшая часть механизма RTTI и именно на нее пользователи должны обратить особое внимание.

Оператор dynamic_cast можно использовать и для приведения к ссылочным типам. Если приведение к типу ссылки заканчивается неудачно, возникает исключение bad_cast. Например:

```
void my_fct(dialog_box& b)
{
   dbox_w_str& db = dynamic_cast<dbox_w_str&>(b);

   // воспользоваться db
}
```

Я пользуюсь приведением к ссылке, когда хочу проверить предположение о ее типе и считаю, что ошибочное предположение – это сбой в программе. Если же нужно выбрать один из нескольких возможных вариантов, то использую приведение к указателю и проверяю результат.

Не помню, когда я пришел к выводу, что приведение с проверкой типа – лучший способ идентификации типа во время исполнения, коль скоро в языке есть явная поддержка этого. Данную идею мне подал кто-то из специалистов компании Xerox PARC. Предложение заключалось в том, чтобы обычные приведения выполняли проверку. Как отмечается в разделе 14.2.2.1, этот вариант сопровождался большими затратами и проблемами с совместимостью, но я осознал, что слегка отличный синтаксис приведения может свести к минимуму неправильное использование, к которому располагает механизм переключения по типу вроде предложения INSPECT в Simula.

14.2.2.1. Синтаксис

Спор о том, как должен выглядеть оператор dynamic_cast, имел синтаксические аспекты.

Приведения типов в наибольшей степени провоцируют ошибки при программировании на языке C++. И синтаксически они принадлежат к числу наиболее неудачных особенностей. Естественно, по мере возможности я хотел:

- устранить приведения вообще;
- сделать их безопасными;
- предложить такой синтаксис приведений, который с очевидностью показывал бы, что используется небезопасная информация;
- предложить альтернативы приведению типов, чтобы не возникало желания прибегать к ним.

Вывод о том, что 3-е и 4-е сочетания реализуемы, а 1-е и 2-е – нет, отражает dynamic_cast.

Рассматривая первое сочетание, мы отметили, что ни в каком языке, поддерживающем системное программирование, нельзя полностью отказаться от приведения типов. Определенные его формы нужны даже для эффективной поддержки численных расчетов. Поэтому следовало лишь постараться минимизировать употребление приведений и по возможности безопасно определить их поведение. Исходя из этой предпосылки, мы с Ленковым разработали предложение, в котором унифицировались динамические и статические приведения с использованием

старого синтаксиса. Поначалу это казалось хорошей мыслью, но при ближайшем изучении вскрылись некоторые проблемы:

- динамические и обыкновенные неконтролируемые приведения типов – это принципиально различные операции. Динамическое приведение для получения результата исследует объект и может вернуть ошибку во время выполнения, если что-то не получается. Обыкновенное приведение выполняет операцию, которая зависит исключительно от участвующих типов и на которую не влияет значение объекта (если не считать проверку на нулевой указатель). Обыкновенное приведение типов не может завершиться с ошибкой, оно просто возвращает новое значение. Использование одного и того же синтаксиса для динамических и обыкновенных приведений затрудняет истинное понимание происходящего;
- если динамические преобразования синтаксически не выделены, то их довольно трудно найти в тексте программы (например, с помощью утилиты `grep`, стандартной для системы UNIX);
- если динамические преобразования синтаксически не выделены, то невозможно поручить компилятору отвергнуть их некорректное использование; он просто должен выполнять то приведение, которое возможно для участвующих в операции типов. Если же различие есть, то компилятор может выдать ошибку при попытке динамического приведения для объектов, которые не поддерживают проверок во время исполнения;
- семантика программы, в которой есть обыкновенные приведения, может измениться, если всюду, где это возможно, будут применяться динамические приведения. Примерами служат приведения к неопределенным классам и приведения внутри иерархий с множественным наследованием (см. раздел 14.3.2). Такое изменение вряд ли оставит семантику всех разумных программ неизменной;
- затраты на проверку типов будут ненулевыми даже для старых программ, где корректность приведения и так тщательно проверялась другими средствами;
- предлагавшийся способ «отключения проверки» путем приведения к `void*` и обратно не был бы стопроцентно надежным, поскольку в некоторых случаях могла измениться семантика. Возможно, все такие случаи – это отклонения, но из-за необходимости разбираться в коде процесс отключения проверки приходилось бы осуществлять вручную, что вело бы к ошибкам. Мы были также против любой техники, которая могла увеличивать число неконтролируемых приведений типов в программах;
- если часть приведений сделать безопасными, это повысило бы доверие к приведению вообще; но нашей конечной целью было уменьшить число всех и всяческих приведений (включая и динамические).

После долгих споров была найдена формулировка: «Должно ли в идеальном языке быть более одной нотации для преобразования типов?» Да, если операции, принципиально различные семантически, различаются в языке и синтаксически. Поэтому мы оставили попытки приспособить старый синтаксис приведений.

Мы подумывали о том, стоит ли объявить прежний синтаксис устаревшим, заменив его чем-то аналогичным

```
Checked<T*>(p);      // преобразование p к типу T* с проверкой во время
                     // исполнения
Unchecked<T*>(p);    // неконтролируемое преобразование p к типу T*
```

В результате все преобразования бросались бы в глаза, так что исчезли бы проблемы, связанные с поиском традиционных приведений в программах на языках C и C++. Кроме того, синтаксически все преобразования следовали бы тому же образцу `<T*>`, что и шаблоны типов (см. главу 15). Рассуждения в этом направлении привели нас к альтернативному синтаксису, изложенному в разделе 14.3.

Какое-то время популярна была нотация `(?T*)p`, поскольку она напоминает традиционный синтаксис приведения `(T*)p`. Но некоторым пользователям данная нотация и не нравилась по той же причине, а другие считали ее «слишком непонятной». К тому же я обнаружил в этой нотации существенный изъян. При использовании конструкции `(?T*)p` многие забывали бы вставить знак `?`. Но тогда приведение, которое должно было быть относительно безопасным и контролируемым, превратилось бы в свою противоположность. Например:

```
if (dbox_w_string* p = (dbox_w_string*)q) // динамическое
                                          // преобразование
{
   // *q есть dbox_w_string
}
```

Слишком уж просто позабыть о `?` и тем самым сделать комментарий ложным. Заметим, что глобальный поиск старых приведений не защитит от такой ошибки, а тем из нас, кто привык работать на C, особенно легко допустить ее, а потом не найти при чтении кода.

В качестве самого многообещающего рассматривался такой вариант:

```
(virtual T*)p
```

Он сравнительно легко распознается как человеком, так и инструментальными средствами, слово `virtual` указывает на логическую связь с классами, имеющими виртуальные функции (полиморфными типами), а общий вид конструкции напоминает традиционные приведения. Однако многие пользователи сочли его «слишком непонятным», а те, кому не нравились старые приведения типов, отнеслись к такому синтаксису враждебно. Я легко согласился с такой критикой: интуиция подсказывала, что синтаксис `dynamic_cast` лучше подходит к C++ (в этом со мной были согласны многие из тех, кто имел солидный опыт работы с шаблонами). Мне также казалось, что синтаксис `dynamic_cast` более удачный и будет способствовать вытеснению прежнего (см. раздел 14.3).

14.2.2.2. Примеры использования динамических приведений

Введение в язык идентификации типа во время исполнения разделяет объекты на две категории:

- те, с которыми ассоциирована информация о типе, так что их тип можно определить во время исполнения почти независимо от контекста;
- объекты, для которых это утверждение неверно.

Почему? Мы не можем идентифицировать во время исполнения тип объектов встроенных типов (например, `int` или `double`), поскольку это привело бы к неприемлемому расходу времени и памяти, а также породило проблемы совместимости с моделью размещения объектов в памяти. То же самое относится и к объектам простых классов, и к структурам в стиле C. Следовательно, первая граница проходит между объектами классов с виртуальными функциями и классов без таковых. Для первых можно легко предоставить идентификацию типа во время исполнения, для вторых – нет.

Кроме того, классы с виртуальными функциями часто называют полиморфными, и только их объектами можно безопасно манипулировать через базовый класс. Под словом «безопасно» я здесь понимаю то, что язык гарантирует использование объекта лишь в соответствии с его типом. Разумеется, некоторые программисты могут продемонстрировать, что в особых случаях такого рода манипуляции с неполиморфными классами не нарушают систему типов.

Поэтому с учетом задач программирования представляется естественным обеспечить идентификацию типа во время исполнения только для полиморфных типов. Поддержка RTTI для других типов сводится просто к ветвлению по полю типа. Конечно, язык не должен препятствовать и такому стилю программирования, но нет необходимости усложнять язык только для того, чтобы поддержать это явно.

Практика показывает, что предоставление RTTI исключительно для полиморфных типов вполне приемлемо. Однако пользователи часто не понимают, какие объекты полиморфны и можно ли применять к ним динамическое приведение типов. К счастью, компилятор выдаст сообщение об ошибке, если догадка программиста окажется неверной. Я долго и настойчиво искал приемлемый способ явно сказать: «Этот класс поддерживает RTTI (вне зависимости от того, есть в нем виртуальные функции или нет)», – но не нашел такого, на который стоило бы тратить силы.

14.2.2.3. Приведение из виртуальных базовых классов

Появление оператора `dynamic_cast` позволило решить одну проблему. С помощью обычного приведения нельзя осуществить преобразование из виртуального базового класса в производный: объект не содержит для этого достаточно информации – см. раздел 12.4.1.

Однако информации, необходимой для идентификации типа во время исполнения, вполне достаточно и для того, чтобы реализовать динамическое приведение из полиморфного виртуального базового класса. Поэтому старое ограничение снимается:

```
class B { /* ... */ virtual void f(); };
class V { /* ... */ virtual void g(); };
class X { /* нет виртуальных функций */ };

class D : public B, public virtual V, public virtual X {
   // ...
};
```

```
void g(D& d)
{
   B* pb = &d;
   D* p1 = (D*)pb;                 // правильно: не проверяется
   D* p2 = dynamic_cast<D*>(pb);   // правильно: проверяется во время
                                   // исполнения
   V* pv = &d;
   D* p3 = (D*)pv;                 // ошибка: приведение из виртуального
                                   // базового класса невозможно
   D* p4 = dynamic_cast<D*>(pv);   // правильно: проверяется во время
                                   // исполнения
   X* px = &d;
   D* p5 = (D*)px;                 // ошибка: приведение из виртуального
                                   // базового класса невозможно
   D* p6 = dynamic_cast<D*>(px);   // ошибка: приведение из
                                   // неполиморфного типа невозможно
}
```

Разумеется, такое приведение можно выполнить лишь в том случае, если тип производного класса определяется однозначно.

14.2.3. Правильное и неправильное использование RTTI

Пользоваться механизмом явной идентификации типа во время исполнения стоит только при необходимости. Статическая проверка (во время компиляции) безопаснее, связана с меньшими затратами и в тех случаях, когда ее хватает, позволяет получать лучше структурированные программы. Например, RTTI можно использовать для написания явно видимых операторов переключения:

```
// неправильное применение информации о типе:

void rotate(const Shape& r)
{
   if (typeid(r) == typeid(Circle)) {
      // ничего не делать
   }
   else if (typeid(r) == typeid(Triangle)) {
      // повернуть треугольник
   }
   else if (typeid(r) == typeid(Square)) {
      // повернуть квадрат
   }
   // ...
}
```

Я слышал, будто о таком стиле говорили, что «он соединяет синтаксическое изящество C++ с эффективностью Smalltalk», но это излишняя любезность. К сожалению, с помощью этого кода нельзя корректно обработать классы, производные от встречающихся в нем, следовательно, при добавлении в программу нового класса его придется модифицировать.

В таком случае, лучше пользоваться виртуальными функциями. Мой опыт работы с подобным кодом в Simula и послужил той причиной, по которой в C++ первоначально не были включены средства для идентификации типа во время исполнения (см. раздел 3.5).

У многих пользователей, имеющих опыт работы с такими языками, как C, Pascal, Modula-2, Ada и т.д., возникает почти непреодолимое желание организовать код в виде набора предложений `switch`. Как правило, этого делать не следует. Заметьте, пожалуйста, что, хоть комитет по стандартизации и одобрил механизм RTTI в C++, он реализован не в виде переключения по типу (как предложение `INSPECT` в Simula). Видимо, не стоит напрямую поддерживать такую модель организации программы. Примеров, где это оправдано, гораздо меньше, чем поначалу кажется многим программистам. Потом же оказывается, что для реорганизации нужно приложить слишком много усилий.

Зачастую RTTI используется правильно, когда некий сервисный код выражен в терминах одного класса, а пользователь намерен расширить его функциональность с помощью наследования. Примером может служить класс `dialog_box` из раздела 14.2.1. Если пользователь хочет и может модифицировать определения библиотечных классов, скажем, того же `dialog_box`, то к RTTI можно и не прибегать, в ином случае это средство потребуется обязательно. Но даже если пользователь готов модифицировать базовые классы, на этом пути могут возникнуть определенные трудности. Например, может оказаться необходимым ввести заглушки вместо некоторых виртуальных функций, например `get_string()` в классах, для которых такие функции не нужны или бессмысленны. Данная проблема обсуждается в книге [2nd, §13.13.6] под заголовком «Fat Interfaces» («толстые» интерфейсы). О применении RTTI для реализации простой системы ввода/вывода объектов говорится в разделе 14.2.7.

У людей, ранее работавших с языками, где интенсивно используется динамическая проверка типов (например, Smalltalk), возникает желание применить RTTI в сочетании с излишне обобщенными типами. Например:

```
// неправильное применение информации о типе:

class Object { /* ... */ };

class Container : public Object {
public:
   void put(Object*);
   Object* get();
   // ...
};

class Ship : public Object { /* ... */ };

Ship* f(Ship* ps, Container* c)
{
   c->put(ps);
   // ...
```

```
    Object* p = c->get();
    if (Ship* q = dynamic_cast<Ship*>(p))  // проверка во время
                                           // выполнения
        return q;

    // сделать что-то еще (обычно - обработать ошибку)
}
```

Здесь класс `Object` совершенно не нужен. Он слишком общий, так как не соответствует ни одному из понятий предметной области и заставляет программиста оперировать на более низком уровне абстракции, чем необходимо. Такого рода задачи лучше решать с помощью шаблонов контейнеров, содержащих указатели только одного типа:

```
template<class T> class Container {
public:
    void put(T*);
    T* get();
    // ...
};

Ship* fp(Ship* ps, Container<Ship>* c)
{
    c->put(ps);
    // ...
    return c->get();
}
```

В сочетании с виртуальными функциями этот прием подходит в большинстве случаев.

14.2.4. Зачем давать «опасные средства»

Итак, если я точно предвидел, что RTTI будут использовать ненадлежащим образом, то зачем вообще спроектировал этот механизм и боролся за его одобрение?

Хорошие программы – это продукт хорошего образования, удачного проектирования, адекватного тестирования и т.д., а не наличия в языке средств, которые предположительно должны использоваться только «правильно». Употребить во вред можно любое средство, поэтому вопрос не в том, можно ли чем-то воспользоваться неправильно (можно!) или будут ли этим неправильно пользоваться (будут!). Вопрос в том, необходимо ли данное средство при условии его правильного употребления настолько, чтобы оправдать затраты на его реализацию, удастся ли смоделировать его с помощью других средств языка и реально ли с помощью обучения обратить во благо неправильное использование.

Размышляя некоторое время над RTTI, я пришел к выводу, что перед нами обычная проблема стандартизации:

- в большинстве крупных библиотек RTTI поддерживается;
- как правило, средство предоставляется в таком виде, который требует от пользователя написания громоздкого дополнительного кода, в результате чего весьма вероятны ошибки;

- реализации в разных библиотеках несовместимы между собой;
- обычно реализация является недостаточно общей;
- в большинстве случаев RTTI выглядит как «привлекательная штучка», которой так и хочется воспользоваться, а не как опасное устройство, к которому надо прибегать в крайнем случае;
- в любой крупной библиотеке, похоже, существует немного случаев, где применение RTTI необходимо настолько, что без нее библиотека либо вообще не могла бы предоставить какую-нибудь возможность, либо предоставляла ее в очень сложном (как для пользователей, так и для разработчиков) виде.

Предоставив стандартный механизм RTTI, мы сможем преодолеть барьер на пути использования библиотек из разных источников (см. раздел 8.2.2). Удастся реализовать механизм RTTI логически последовательно, попытаемся сделать его безопасным, предупреждая о возможных неправильных способах использования.

Наконец, при проектировании C++ был заложен принцип: когда все сказано и сделано, надо доверять программисту. Возможная польза важнее, чем предполагаемые ошибки.

Однако некоторые пользователи, в частности Джим Уолдо, энергично доказывают, что в RTTI возникает необходимость очень редко, а неправильное понимание, лежащее в основе всех ошибок применения данного средства, распространено широко, значит, общее воздействие этого механизма на C++ негативно. Только время покажет, кто прав.

14.2.5. Оператор typeid()

Я надеялся, что оператор `dynamic_cast` сможет удовлетворить все потребности, так что никаких других механизмов RTTI пользователям не понадобится. Однако многие пользователи, с которыми я обсуждал этот вопрос, со мной не согласились и указали еще на два момента:

- необходимость знать точный тип объекта, то есть быть уверенным, что объект принадлежит именно классу `X`, а не классу `X` или производному от него (только последнее и можно получить от оператора `dynamic_cast`);
- использование точного типа объекта как источника дополнительной информации об этом типе.

Нахождение точного типа объекта иногда называют идентификацией типа, поэтому соответствующий оператор я назвал `typeid`.

Точный тип объекта нужно знать для того, чтобы применить ко всему объекту некоторый стандартный сервис. В идеале такие сервисы должны предоставляться в виде виртуальных функций, так что точный тип знать не обязательно, но, когда этой функции нет, знание точного типа и выполнение над ним операции необходимы. Системы объектного ввода/вывода и системы баз данных работают именно так. В подобных случаях нельзя предполагать наличие общего интерфейса у всех управляемых объектов, поэтому приходится прибегать к точному типу. Другое, гораздо более простое применение – получить имя класса для выдачи диагностического сообщения:

```
cout << typeid(*p).name();
```

Встроенный оператор `typeid()` используется для получения доступа к информации о типах во время выполнения. Если бы он являлся функцией, ее объявление могло выглядеть так:

```
class type_info;
const type_info& typeid(имя_типа);   // псевдообъявление
const type_info& typeid(выражение); // псевдообъявление
```

Таким образом, `typeid()` возвращает ссылку на неизвестный тип, который называется `type_info`.[1] Если операндом является имя типа, то `typeid()` возвращает ссылку на объект `type_info`, представляющий тип именно с этим именем. Если же операнд – это выражение, то `typeid()` возвращает ссылку на объект `type_info`, представляющий тип объекта, обозначенного этим выражением.

Причина, по которой `typeid()` возвращает именно ссылку, а не указатель на `type_info`: мы хотели запретить обычные операции с указателями, например `==` или `++`, над объектами, возвращенными `type_id()`. Например, вовсе не очевидно, что каждая реализация сможет гарантировать уникальность объектов для идентификации типа. А это значит, что сравнение результатов `typeid()` нельзя определить просто как сравнение указателей на объекты типа `type_info`. Если же `typeid()` возвращает ссылку на `type_info`, то нетрудно определить оператор `==`, так чтобы он правильно обрабатывал возможное дублирование объектов `type_info` для одного и того же типа.

14.2.5.1. Класс type_info

Класс `type_info` определен в заголовочном файле `<type_info.h>`, который нужно включать, если используется результат, выданный `typeid()`. Точное определение класса `type_info` зависит от реализации, но это полиморфный тип, в котором имеются операторы сравнения и операция, возвращающая имя типа:

```
class type_info {
   // представление зависит от реализации

private:
   type_info(const type_info&);           // пользователи не могут
   type_info& operator=(const type_info&);// копировать type_info

public:
   virtual ~type_info();                  // полиморфный

   int operator==(const type_info&) const;// можно сравнивать
   int operator!=(const type_info&) const;
   int before(const type_info&) const;    // упорядоченный

   const char* name() const;              // имя типа
};
```

[1] Комитет по стандартизации все еще обсуждает соглашения об именовании стандартных библиотечных классов. Я выбрал те имена, которые, скорее всего, будут утверждены.

Может предоставляться и более детальная информация. Однако пользователи понимают «детальную информацию» по-разному. Тем же, кому она вообще не нужна, хотелось бы уменьшить расход памяти. Поэтому предоставляемые типом `type_info` сведения намеренно сведены к минимуму.

Функция `before()` предназначена для сортировки объектов типа `type_info`, чтобы их можно было хранить в хэш-таблицах и т.п. Отношение, описываемое этой функцией, никак не связано с отношением наследования (см. раздел 14.2.8.3). Более того, нет никакой гарантии, что `before()` будет давать одинаковые результаты в разных программах или при разных запусках одной и той же программы. В этом смысле `before()` напоминает оператор взятия адреса.

14.2.5.2. Расширенная информация о типе

Иногда знание точного типа объекта – это лишь первый шаг к получению более детальной информации об этом типе.

Рассмотрим, как компилятор или инструментальное средство могли бы предоставить пользователю информацию о типах во время исполнения. Предположим, что имеется инструмент, генерирующий таблицу объектов типа `My_type_info`. Наилучший способ представления информации пользователю – применение ассоциативного массива (отображение, словарь), который сопоставляет такие таблицы с именами типов. Чтобы получить таблицу для конкретного типа, можно было бы написать:

```
#include <type_info.h>

extern Map<My_type_info,const char*> my_type_table;

void f(B* p)
{
   My_type_info& mi = my_type_table[typeid(*p).name()];
   // воспользоваться mi
}
```

Кто-то предпочел бы индексировать таблицы непосредственно значениями `typeid()`, а не требовать от пользователя обращения к строке `name()`:

```
extern Map<Your_type_info,type_info*> your_type_table;

void g(B* p)
{
   Your_type_info& yi = your_type_table[&typeid(*p)];
   // воспользоваться yi
}
```

С помощью такого способа ассоциирования результатов `typeid` с информацией пользователи могут привязывать к типам различную информацию (см. рис. 14.1). Аналогичные действия производятся и инструментальными средствами. При этом добавленные сведения не мешают друг другу.

Это очень важно, поскольку вероятность, что есть информация, устраивающая всех пользователей, равна нулю. Любой набор данных о типе, устраивающий

даже большинство пользователей, повлек бы за собой огромные издержки для тех, кому нужны лишь минимальные сведения.

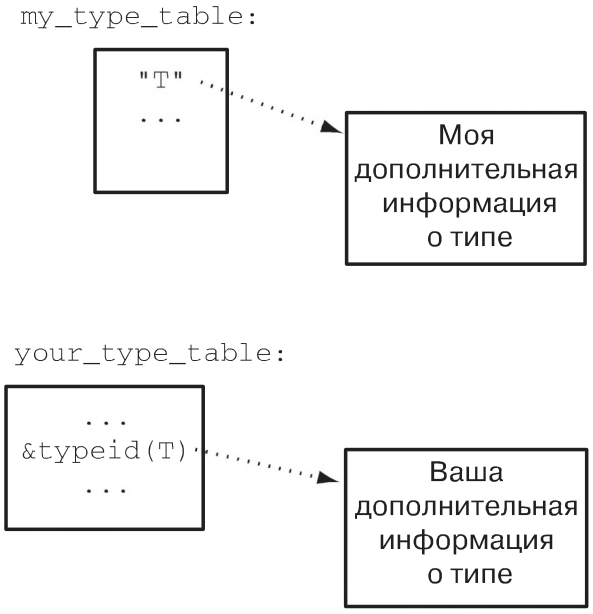


Рис. 14.1

Разработчик компилятора может по своему выбору предоставлять информацию о типе, специфичную для данной реализации. Доступ к такой расширенной системной информации можно получать с помощью ассоциативного массива точно так же, как и к дополнительной информации, предоставляемой пользователем. Вместо этого допустимо поместить расширенную информацию в класс `Extended_type_info`, производный от `type_info` (см. рис. 14.2).

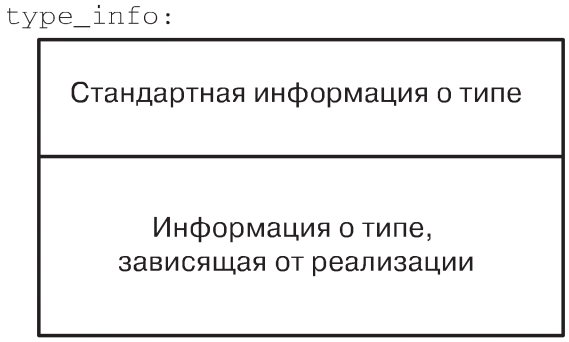


Рис. 14.2

Чтобы понять, имеется ли нужная расширенная информация о типе, можно было бы воспользоваться оператором `dynamic_cast`:

```
#include <type_info.h>

typedef Extended_type_info Eti;

void f(Sometype* p)
```

```
{
   if (Eti* p = dynamic_cast<Eti*>(&typeid(*p))) {
       // ...
   }
}
```

Какую расширенную информацию мог бы получить пользователь с помощью компилятора или инструментального средства? Практически любую имеющуюся в компиляторе, если во время выполнения она понадобится некоторой программе. Например:

- сведения о размещении объекта для ввода/вывода объектов или отладки;
- указатели на функции создания и копирования объектов;
- таблицы функций вместе с символическими именами для вызова из интерпретатора;
- списки объектов данного типа;
- ссылки на исходный код функций-членов;
- встроенную документацию по классу.

Все это вынесено в библиотеки, возможно даже стандартные, поскольку есть очень много разных потребностей, зависящих от реализации деталей и слишком много информации, чтобы поддерживать ее в самом языке. С другой стороны, некоторые возможные применения позволяют обойти статический контроль типов, а другие требуют неоправданно больших затрат.

14.2.6. Модель размещения объекта в памяти

Вот как мог бы размещаться в памяти объект с таблицей виртуальных функций и информацией о типе (см. рис. 14.3).

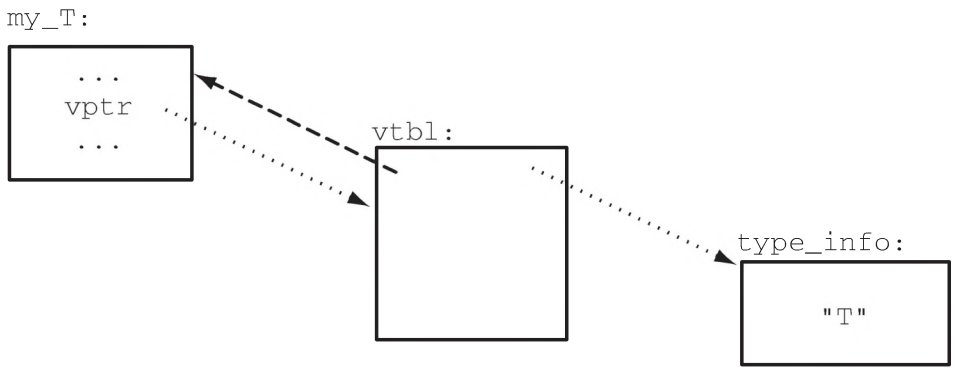


Рис. 14.3

Пунктирная стрелка с длинными штрихами представляет смещение, по которому можно найти начало полного объекта, имея лишь указатель на полиморфный подобъект. Это абсолютный аналог смещения (дельты), используемого при реализации виртуальных функций (см. раздел 12.4).

Для каждого типа с виртуальными функциями генерируется объект типа `type_info`. Эти объекты не обязательно должны быть уникальными. Однако хороший компилятор будет везде, где возможно, генерировать уникальные объекты

type_info и лишь для тех типов, для которых действительно используется в каком-то виде информация о типе. В простой реализации объект type_info для класса разрешается поместить прямо рядом с vtbl.

Реализации, основанные на Cfront или позаимствовавшие из него схему размещения таблицы виртуальных функций, можно модернизировать для поддержки RTTI так, что существующий код даже не придется перекомпилировать. Просто о поддержке RTTI я размышлял в то время, когда реализовывал версию 2.0, и оставил два пустых слова в начале vtbl, имея в виду именно это расширение. Но в то время RTTI не была добавлена, так как я сомневался в том, что она потребуется, и не знал, в какой форме ее предоставить пользователям. В качестве эксперимента был реализован простой вариант, когда у каждого объекта класса с виртуальными функциями можно было бы запросить его имя. Тогда я убедился, что смогу добавить поддержку RTTI, если потребуется, и удалил эту возможность.

14.2.7. Простой ввод/вывод объектов

Ниже представлены идея, как можно было бы использовать RTTI в простой системе ввода/вывода объектов, и возможная схема реализации такой системы. Пользователь хочет прочитать объекты из потока, убедиться, что они имеют ожидаемые типы, а затем как-то их применить. Например:

```
void user()
{
    // открыть файл, предполагая, что он содержит объекты типа
    // Shape, и присоединить к этому файлу ss в качестве istream
    // ...

    io_obj* p = get_obj(ss);    // читать объект из потока

    if (Shape* sp = dynamic_cast<Shape*>(p)) {
        sp->draw();   // использовать Shape
        // ...
    }
    else {
        // ошибка: в файле есть объект,
        // не принадлежащий классу Shape
    }
}
```

Функция user() работает с фигурами только через абстрактный класс Shape и потому готова принять любую фигуру. Использование dynamic_cast необходимо, поскольку система объектного ввода/вывода способна работать и со многими другими видами объектов, так что пользователь может случайно открыть файл, содержащий объекты, но только из классов, о которых он ранее не слышал.

Такая система объектного ввода/вывода предполагает, что каждый объект, который можно записывать или считывать, принадлежит классу, производному от io_obj. Класс io_obj должен быть полиморфным, чтобы к нему можно было применять оператор dynamic_cast. Например:

```
class io_obj {   // полиморфный
   virtual io_obj* clone();
};
```

Функция `get_obj()` является самой важной в системе объектного ввода/вывода. Она читает данные из `istream` и создает из них объекты классов. Допустим, что данным, представляющим в потоке объект, предшествует строка, идентифицирующая его класс. Тогда `get_obj()` должна считать эту строку и вызвать функцию, которая может создать и инициализировать объект нужного класса. Например:

```
typedef io_obj* (*PF)(istream&);

Map<String,PF> io_map; // отображение строк на функции создания

io_obj* get_obj(istream& s)
{
   String str;
   if (get_word(s,str) == 0)   // прочитать в str начальное слово
      throw no_class;

   PF f = io_map[str]; // найти функцию по "str"
   if (f == 0) throw unknown_class; // функции не нашлось
   io_obj* p = f(s);    // сконструировать объект из потока
   if (debug) cout << typeid(*p).name() << '\n';
}
```

Отображение `io_map` класса `Map` – это ассоциативный массив, который содержит пары, состоящие из строки имени и функции, способной сконструировать объект класса с этим именем. Тип `Map` в любом языке относится к числу наиболее полезных и эффективных структур данных. Одна из первых реализаций этой идеи принадлежит Эндрю Кенигу [Koenig, 1988], см. также [2nd, §8.8].

Обратите внимание на использование `typeid()` для отладки. При данном проектировании это единственное место, где используется RTTI.

Мы могли бы, конечно, определить класс `Shape` обычным образом, наследуя классу `io_obj`, как того требует функция `user()`:

```
class Shape : public io_obj {
   // ...
};
```

Но было бы лучше (а во многих случаях и более реалистично) использовать уже имеющийся класс `Shape`, не изменяя иерархию:

```
class iocircle : public Circle, public io_obj {
public:
   iocircle* clone()    // замещает io_obj::clone
      { return new iocircle(*this); }

   iocircle(istream&); // инициализует из входного потока
```

```
    static iocircle* new_circle(istream& s)
    {
        return new iocircle(s);
    }
    // ...
};
```

Конструктор `iocircle(istream&)` инициализирует объект на основе данных, считываемых из потока `istream`. Функция `new_circle` помещается в `io_map`, чтобы сделать класс известным системе объектного ввода/вывода. Например:

```
io_map["iocircle"]=&iocircle::new_circle;
```

Другие фигуры конструируются аналогично:

```
class iotriangle : public Triangle, public io_obj {
    // ...
};
```

Поскольку постоянно адаптировать новые типы к объектному вводу/выводу утомительно, можно воспользоваться шаблонами:

```
template<class T>
class io : public T, public io_obj {
public:
    io* clone()   // замещает io_obj::clone
        { return new io(*this); }

    io(istream&); // инициализирует из входного потока

    static io* new_io(istream& s)
    {
        return new io(s);
    }
    // ...
};
```

Тогда можно было бы определить `iocircle` следующим образом:

```
typedef io<Circle> iocircle;
```

Но определить функцию `io<Circle>::io(istream&)` все же пришлось бы явно, поскольку ей нужно знать о деталях `Circle`.

Столь простая система объектного ввода/вывода не может удовлетворить всем пожеланиям пользователя, но весь ее код умещается на одной странице, а основные механизмы могут найти различные применения. В обычной ситуации такую технику можно использовать для вызова функции на основе строки, указанной пользователем.

14.2.8. Другие варианты

Механизм RTTI можно сравнить с луковицей. Снимая слой за слоем, вы все равно обнаруживаете полезные средства. Но если злоупотребить ими, то можно горько пожалеть.

Главный принцип описанных выше механизмов RTTI заключается в том, чтобы максимально упростить программирование и минимизировать зависимость от реализации. Но RTTI следует применять как можно реже:

- лучше всего вовсе не пользоваться информацией о типе во время исполнения, а всецело полагаться на статическую проверку (на этапе компиляции);
- если это невозможно, следует использовать только динамические приведения типов. В таком случае нам даже не нужно знать точное имя типа объекта и включать относящиеся к RTTI заголовочные файлы;
- когда недостаточно и этого, можно сравнивать значения, возвращаемые `typeid()`, для чего необходимо знать точное имя, по крайней мере, одного из типов. Предполагается, что рядовому пользователю нет нужды знать что-нибудь еще о типах во время исполнения;
- наконец, если абсолютно необходимо иметь о типе более детальную информацию – скажем, для реализации отладчика, системы баз данных или какой-либо иной формы системы объектного ввода/вывода, – можно воспользоваться операциями над `typeid`.

Такое предоставление ряда средств, тесно связанных с динамическими свойствами класса, противоположно другому подходу: создать класс, описывающий стандартный взгляд на эти свойства, – метакласс. В C++ поощряется использование статической системы типов (более безопасной и эффективной), которая понятней пользователям и дает возможность оценить разные аспекты класса для получения детальной информации.

Рассматривалось несколько альтернатив такому «луковичному подходу».

14.2.8.1. Метаобъекты

«Луковичный» объект отличается от принятых в языках Smalltalk и CLOS, хотя нечто подобное несколько раз предлагалось и для C++. В таких системах вместо `type_info` применяются метаобъекты, способные во время исполнения принимать запросы на выполнение любой операции, допустимой над объектом в языке. По сути дела, механизм метаобъектов подразумевает наличие встроенного интерпретатора всего языка в исполняющей среде. Представлялось, что подобный потенциальный способ обхода механизмов защиты опасен для эффективности языка и не стыкуется с базовыми принципами проектирования и документирования, основанными на статическом контроле типов.

Возражения против того, чтобы понятие метаобъектов легло в основу C++, отнюдь не означают, что метаобъекты нельзя с пользой применить. Безусловно, можно, и концепция расширенной информации о типе поможет тем, кто действительно хотел бы предоставить данные средства на уровне библиотек. Но я не могу рекомендовать такое проектирование и реализацию для программирования типичных задач на C++ [2nd, §12].

14.2.8.2. Оператор опроса типа

Для многих пользователей оператор, отвечающий на вопрос «принадлежит ли `*pb` классу `D` или производному от него?», кажется гораздо более понятным, чем

dynamic_cast, который выполняет приведения только тогда, когда ответ положительный. Иными словами, хотелось бы иметь возможность писать такой код:

```
void my_fct(dialog_box* bbp)
{
   if (dbp->isKindOf(dbox_w_str)) {

      dbox_w_str* dbsp = (dbox_w_str*)dbp;

      // воспользоваться dbsp
   }
   else {
      // считать, что *dbp - это обычное диалоговое окно
   }
}
```

К сожалению, в ряде случаев приведение не дает правильного результата (см. раздел 14.3.2.1). Это пример того, как трудно бывает импортировать концепции из других языков. В Smalltalk есть оператор isKindOf для опроса типа, но данный язык не нуждается в дальнейшем приведении. Однако прямой перенос isKindOf в C++ привел бы как к техническим, так и к стилистическим трудностям.

В действительности именно аргументы стилистического характера заставили меня отдать предпочтение какой-то форме условного приведения типа, и только позже я обнаружил некоторые технические доводы против оператора опроса типа наподобие isKindOf. Таким образом, разделение проверки и преобразования типа излишне многословно и оставляет возможность рассогласования между результатами того и другого.

14.2.8.3. Упорядоченность типов

Было предложение определить для объектов type_info операторы <, <= и т.д., чтобы выразить отношение порядка в иерархии классов. Это несложно, но слишком мудрено. Кроме того, здесь есть те же проблемы с явными операциями сравнения типов, что и описанные в 14.2.8.2. Нам в любом случае нужно приведение, так лучше уж воспользоваться dynamic_cast.

14.2.8.4. Мультиметоды

Более привлекательно выглядит применение RTTI для поддержки так называемых мультиметодов, то есть возможности вызывать виртуальную функцию на базе более чем одного объекта. Такое средство стало бы подарком для программистов, пишущих код, с помощью которого осуществляются бинарные операции над различными объектами (см. раздел 13.8). Очевидно, объекты type_info могли бы хранить необходимую для этого информацию. К тому же мультиметоды становятся, скорей всего, последующим расширением, а не альтернативой принятому подходу.

Однако предложение не было внесено, поскольку я не мог отчетливо представить все последствия такого изменения и не хотел предлагать новое крупное расширение, не имея опыта его применения в контексте C++.

14.2.8.5. Методы без ограничений

Имея RTTI, удастся поддержать методы без ограничений (unconstrained methods), то есть в объекте `type_info` для некоторого класса можно хранить достаточно информации, чтобы во время исполнения проверить, поддерживается в нем данная функция или нет. Так можно было реализовать проверяемые во время исполнения вызовы функций в духе Smalltalk. Однако такое расширение противоречило стремлению способствовать эффективному и безопасному с точки зрения типов программированию. Динамическое приведение позволяет сначала проверить, а потом вызвать функцию

```
if (D* pd = dynamic_cast<D*>(pb)) {  // *pb принадлежит типу D?
   pd->dfct();   // вызвать функцию из D
   // ...
}
```

вместо того чтобы сначала вызвать, а потом проверить, что получилось (как в языке Smalltalk):

```
pb->dfct();      // в надежде, что pb указывает на что-то,
                 // имеющее dfct; если это не так,
                 // обработать неудачный вызов где-то в
                 // другом месте
```

Первая стратегия в большей степени опирается на статическую проверку типов (уже на этапе компиляции известно, что для класса `D` определена функция `dfct`), позволяет обойтись без лишних затрат при подавляющем большинстве вызовов, не нуждающихся в проверке, и содержит визуальное указание на то, что происходит нечто неординарное.

14.2.8.6. Контролируемая инициализация

Мы также думали, что стоит контролировать присваивание и инициализацию подобно тому, как это делается в языках Beta и Eiffel, например:

```
void f(B* pb)
{
   D* pd1 = pb;  // ошибка: несоответствие типов
   D* pd2 ?= pb; // правильно, проверка того, что *pb принадлежит
                 // классу D, производится во время исполнения

   pd1 = pb;     // ошибка: несоответствие типов
   pd2 ?= pb;    // правильно, проверка того, что *pb принадлежит
                 // классу D, производится во время исполнения
}
```

Однако я решил, что в реальном коде отыскать символ `?` будет слишком сложно, а вся конструкция провоцирует ошибки, так как после нее будут забывать проверять результат. Кроме того, иногда применение данной конструкции сопряжено с введением именованной переменной, предназначенной только для этой цели. Альтернатива – разрешить использование `?=` только в условиях – выглядела очень привлекательно:

```
void f(B* pb)
{
   D* pd1 ?= pb; // ошибка: нет проверки
                 // условная инициализация

   if (D* pd2 ?= pb) {  // правильно: есть проверка
                        // условная инициализация
      // ...
   }
}
```

Однако пришлось бы отличать случай, когда при ошибке возбуждается исключение, от случая, когда возвращается `0`. Кроме того, оператор `?=` не так неудачен, как приведения типов, поэтому провоцирует на неправильное употребление.

Разрешив объявления в условиях (см. раздел 3.11.5.2), я сделал возможным использование `dynamic_cast` в стиле этой альтернативы:

```
void f(B *pb)
{
   if (D* pd2 = dynamic_cast<D*>(pb)) {    // правильно: контролируется
      // ...
   }
}
```

14.3. Новая нотация для приведения типов

Синтаксически и семантически приведения типов – одна из самых неудачных особенностей C и C++. Поэтому поиск других вариантов не прекращался. Неявные преобразования аргументов в объявлениях функций (см. раздел 2.6), шаблоны (см. раздел 14.2.3) и ослабление правил перегрузки для виртуальных функций (см. раздел 13.7) – все это позволяет устранить некоторые виды приведений. С другой стороны, оператор `dynamic_cast` (см. раздел 14.2.2) в отдельных случаях – более безопасная альтернатива старым приведениям. Он и натолкнул на мысль пойти по другому пути – вычленить логически различные применения приведений типов и поддержать их с помощью операторов, похожих на `dynamic_cast`:

```
static_cast<T>(e)       // "хорошие" приведения типов
reinterpret_cast<T>(e)  // приведения, дающие значения, которые для
                        // безопасного использования нужно привести
                        // к исходному типу
const_cast<T>(e)        // отбрасывание const
```

В данном разделе анализируются проблемы, свойственные старым приведениям типов, и описывается синтез нового средства. Во многом определения операторов появились в процессе бурных споров среди членов рабочей группы по расширениям. Особенно конструктивные предложения внесли Дэг Брюк, Джерри Шварц и Эндрю Кениг. Новые операторы приведения были одобрены на заседании комитета в Сан-Хосе в ноябре 1993 г.

Поскольку объем книги ограничен, здесь рассматривается только самая большая трудность – указатели. Проработка остальных вопросов: арифметических типов, указателей на члены, ссылок и т.д. – оставлена как упражнение читателю.

14.3.1. Недостатки старых приведений типов

Выражение `(T)expr` даст (за очень немногими исключениями) значение типа `T`, тем или иным способом полученное на основе значения `expr`. Допускаю, что для этого потребуется иная интерпретация битов `expr`, могут также произойти сужение или расширение диапазона значений, арифметические операции над адресами для навигации по иерархии классов, отключение атрибута `const` или `volatile` и др. Видя перед собой изолированное выражение с приведением типа, пользователь не в состоянии определить, что имел в виду его автор. Например:

```
const X* pc = new X;
// ...
pv = (Y*) pc;
```

Хотел ли программист получить указатель на тип, никак не связанный с `X`? Или убрать атрибут `const`? Или то и другое одновременно? Может быть, он намеревался получить доступ к классу `Y`, базовому для `X`?

Более того, безобидное, на первый взгляд, изменение объявления способно без всяких предупреждений полностью изменить смысл выражения, например:

```
class X : public A, public B { /* ... */ };

void f(X* px)
{
   ((B*)px)->g();    // вызывается g из класса B
   px->B::g();   // более явный и, значит, лучший способ
}
```

Изменим определение класса `X` так, чтобы `B` больше не являлся для него базовым классом, и смысл `(B*)px` станет совершенно другим, а компилятор не сможет обнаружить ошибку.

Помимо связанных со старыми приведениями семантических проблем, неудобна и нотация. Она близка к минимальной и использует только скобки, то есть синтаксическую конструкцию, которая в C и так употребляется чрезмерно часто. Поэтому пользователю трудно найти в программе все приведения. И с помощью инструментального средства типа `grep` осуществлять поиск нелегко. К тому же синтаксис приведения типов – одна из основных причин усложнения грамматического разбора программы на C++.

Итак, старые приведения типов:

- трудны для понимания, поскольку предоставляют одну и ту же нотацию для различных слабо связанных между собой операций;
- провоцируют ошибки, так как почти любая комбинация типов имеет какую-то допустимую интерпретацию;
- с трудом отыскиваются в исходном тексте как вручную, так и с помощью простых утилит;
- усложняют грамматику C и C++.

Новые операторы приведения призваны распределить функциональность старых по разным категориям. Они должны уметь выполнять все те же операции, что и прежние операторы, иначе будут приведены доводы в пользу использования старых и в дальнейшем. Было найдено только одно исключение: с помощью старого синтаксиса можно приводить от производного класса к его закрытому базовому классу. Я не вижу оснований для такой операции – она опасна и бесполезна. Невозможно получить доступ к полностью закрытому представлению объекта, да это и не нужно. Тот факт, что старые приведения типов позволяли добраться до части представления закрытого класса, – досадная случайность. Например:

```
class D : public A, private B {
private:
   int m;
   // ...
};

void f(D* pd) // f() - не член и не дружественная функция D
{
   B* pb1 = (B*)pd; // получаем доступ к закрытому базовому
                    // классу B. Не годится!
   B* pb2 = static_cast<B*>(pd);     // ошибка: нет доступа к закрытому
                                     // классу. Верно!
}
```

Если исключить манипуляции с `pd` как с указателем на неинициализированную память, то функция `f()` не получит доступ к `D::m`. Таким образом, новые операторы приведения закрывают «дыру» в правилах доступа и обеспечивают большую логическую непротиворечивость.

Длинные названия и подобный шаблонам синтаксис новых приведений не внушает доверия некоторым пользователям. Но, может, это и к лучшему, поскольку одна из целей, которые мы ставили, включая данные средства в язык, – напомнить, что приведение типов – дело рискованное. Кстати, наибольшее недовольство по данному поводу высказывали те, кто пользуется C++ преимущественно как диалектом C и полагает, что приводить типы нужно как можно чаще. Нотация кажется странной и не привыкшим к шаблонам.

14.3.2. Оператор static_cast

Нотация `static_cast<T>(e)` призвана заменить `(T)e` для преобразований от `Base*` (указатель на базовый класс) к `Derived*` (указатель на производный класс). Такие преобразования не всегда безопасны, но часто встречаются и могут разумно определяться даже в отсутствие проверок во время исполнения. Например:

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

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

void f(B* pb, D* pd)
```

```
{
   D* pd2 = static_cast<D*>(pb);     // раньше писали (D*)pb

   B* pb2 = static_cast<B*>(pb);     // безопасное преобразование
   // ...
}
```

Можно представлять себе `static_cast` как явное обращение операции неявного преобразования типов. Если на время забыть о том, что `static_cast` сохраняет константность, то он позволяет выполнить `S->T` при условии, что преобразование `T->S` может быть выполнено неявно. Отсюда следует, что в большинстве случаев результат `static_cast` можно использовать без дальнейшего приведения. В этом отношении он отличается от `reinterpret_cast` (см. раздел 14.3.3).

Кроме того, `static_cast` вызывает преобразования, которые можно выполнить неявно (например, стандартные и определенные пользователем).

В противоположность `dynamic_cast` для применения `static_cast` к `pb` не требуется никаких проверок во время исполнения. Объект, на который указывает `pb`, может и не принадлежать к классу `D`, тогда результаты использования `*pd2` не определены и, возможно, пагубны.

В отличие от старых приведений типов указательные и ссылочные типы должны быть полными. Это означает, что попытка использовать `static_cast` для преобразования к указателю на тип или от него в случае, когда компилятор еще не встречал объявления этого типа, приведет к ошибке. Например:

```
class X;  // X - неполный тип
class Y;  // Y - неполный тип

void f(X* px)
{
   Y* p = (Y*)px;   // разрешено, но опасно
   p = static_cast<Y*>(px);   // ошибка: X и Y не определены
}
```

Тем самым устраняется еще один источник ошибок. Если вам нужно приведение к неполным типам, пользуйтесь оператором `reinterpret_cast` (см. раздел 14.3.3), чтобы ясно показать, что вы не пытаетесь осуществлять навигацию по иерархии, или оператором `dynamic_cast` (см. раздел 14.2.2).

14.3.2.1. Статические и динамические приведения

Применение `static_cast` и `dynamic_cast` к указателям на классы приводит к навигации по иерархии классов. Однако `static_cast` опирается только на статическую информацию и поэтому может работать некорректно. Рассмотрим пример:

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

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

void f(B* pb)
```

```
{
   D* pd1 = dynamic_cast<D*>(pb);
   D* pd2 = static_cast<D*>(pb);
}
```

Если в действительности pb указывает на D, то pd1 и pd2 получают одно и то же значение. Так же, как и в случае, когда pb==0. Но если pb указывает на B (и только), то у dynamic_cast достаточно информации, чтобы вернуть 0, а static_cast должен «поверить» программисту, что pb указывает на D, и вернуть указатель на объект, предположительно принадлежащий классу D. Рассмотрим другой пример:

```
class D1 : public D { /* ... */ };
class D2 : public B { /* ... */ };
class X : public D1, public D2 { /* ... */ };

void g()
{
   D2* pd2 = new X;
   f(pd2);
}
```

Здесь g() вызовет f(), передав объект B, который не является подобъектом D. Следовательно, dynamic_cast правильно найдет «соседний» подобъект типа D, тогда как static_cast вернет указатель на какой-то неподходящий подобъект X. Насколько я помню, первым, кто обратил мое внимание на этот феномен, был Мартин О'Риордан.

14.3.3. Оператор reinterpret_cast

Нотация reinterpret_cast<T>(e) призвана заменить (T)e для преобразований вида char* в int* и Some_class* в Unrelated_class*, которые внутренне небезопасны и зачастую зависят от реализации. При этом Some_class и Unrelated_class не связаны друг с другом отношением наследования. По сути дела, оператор reinterpret_cast возвращает значение, полученное в результате новой принудительной интерпретации своего аргумента. Например:

```
class S;
class T;

void f(int* pi, char* pc, S* ps, T* pt, int i)
{
   S* ps2 = reinterpret_cast<S*>(pi);
   S* ps3 = reinterpret_cast<S*>(pt);
   char* pc2 = reinterpret_cast<char*>(pt);
   int* pi2 = reinterpret_cast<int*>(pc);
   int i2 = reinterpret_cast<int>(pc);
   int* pi3 = reinterpret_cast<int*>(i);
}
```

Оператор reinterpret_cast позволяет преобразовать указатель в любой другой, а также любой интегральный тип в любой указательный тип и наоборот.

Все эти операции небезопасны, зависят от реализации. Если только нужное преобразование не является по существу низкоуровневым и небезопасным, программисту лучше воспользоваться какими-то другими приведениями.

В отличие от `static_cast`, на результаты работы `reinterpret_cast` нельзя полагаться ни в одном из случаев за исключением приведения к исходному типу. Все остальные применения в лучшем случае непереносимы. Вот почему преобразования между указателем на функцию и указателем на член относятся к `reinterpret_cast`, а не к `static_cast`. Например:

```
void thump(char* p) { *p = 'x'; }

typedef void (*PF)(const char*);
PF pf;

void g(const char* pc)
{
   thump(pc);// ошибка: неправильный тип аргумента

   pf = &thump;  // ошибка

   pf = static_cast<PF>(&thump);     // ошибка!

   pf = reinterpret_cast<PF>(&thump);      // допускается,
                                           // но за последствия
                                           // отвечаете только вы
   pf(pc);       // правильная работа не гарантируется!
}
```

Понятно, что присваивать `pf` указатель на `thump` опасно, так как это действие в обход системы типов. С помощью данной операции адрес константы может быть передан какой-то функции, которая сделает попытку ее модифицировать. Поэтому и нужно использовать приведение, причем именно оператор `reinterpret_cast`. Но для многих неожиданно, что вызов `thump` через `pf` все равно не гарантирован (в C++ так же, как и в C). Дело в том, что компилятору разрешено использовать разные соглашения о вызове для функций разных типов. В частности, есть веские основания для того, чтобы константные и неконстантные функции вызывались по-разному.

Заметим, что `reinterpret_cast` не осуществляет навигацию по иерархии классов. Например:

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

void f(B* pb)
{
   D* pd1 = reinterpret_cast<D*>(pb);
   D* pd2 = static_cast<D*>(pb);
}
```

Здесь pd1 и pd2, как правило, получат различные значения. При вызове

```
f(new D);
```

pd2 будет указывать на начало переданного объекта D, тогда как pd1 – на начало подобъекта B в объекте D.

Операция reinterpret_cast<T>(arg) почти так же неудачна, как и (T)arg. Однако она более заметна в тексте программы, никогда не осуществляет навигации по классам и не отбрасывает атрибут const. Для данной операции есть альтернативы, reinterpret_cast – средство для низкоуровневых преобразований, обычно зависящих от реализации, – и только!

14.3.4. Оператор const_cast

Труднее всего было найти замену для старых приведений типов, чтобы корректно обрабатывался атрибут const. В идеале нужно было гарантировать, что константность никогда не снимается сама собой. По этой причине каждый из операторов reinterpret_cast, dynamic_cast и static_cast не изменял этот атрибут.

Нотация const_cast<T>(e) призвана заменить (T)e для преобразований, которым нужно получить доступ к данным с модификатором const или volatile. Например:

```
extern "C" char* strchr(char*, char);

inline const char* strchr(const char* p, char c)
{
   return strchr(const_cast<char*>p, char c);
}
```

В const_cast<T>(e) тип аргумента T должен совпадать с типом аргумента e во всем, кроме модификаторов const и volatile. Результатом будет то же значение, что у e, только его типом станет T.

Отметим, что результат отбрасывания const для объекта, который первоначально был объявлен с этим модификатором, не определен (см. раздел 13.3).

14.3.4.1. Проблемы защиты const

В системе типов есть некоторые тонкости, которые открывают «дыры» в защите от неявного нарушения константности. Рассмотрим пример:

```
const char cc = 'a';
const char* pcc = &cc;
const char** ppcc = &pcc;
void* pv = ppcc; // никакого приведения не нужно:
                 // ppcc - не константный указатель, он
                 // лишь указывает на таковой, но
                 // константность исчезла!
char** ppc - (char**)pv;   // указывает на pcc

void f()
{
   **ppc = 'x';  // теперь можем изменять константную переменную!
}
```

Однако оставление `void*` небезопасным можно считать приемлемым, поскольку все знают – или, по крайней мере, должны знать, – что приведения из типа `void*` – это хитрость.

Подобные примеры приобретают практический интерес, когда вы начинаете строить классы, которые могут содержать много разных указателей (например, чтобы минимизировать объем сгенерированного кода, см. раздел 15.5).

Объединения (union) и использование многоточия для явного подавления проверки типов – это тоже «дыры» в механизме защиты от неявного нарушения константности. Однако я предпочитаю систему, в которой есть несколько «прорех», той, которая вообще не предоставляет никакой защиты. Как и в случае с `void*`, программисты должны понимать, что объединения и отказ от контроля аргументов опасны, что их следует по возможности избегать, а применять с особой осторожностью и только в случае необходимости.

14.3.5. Преимущества новых приведений типов

Новые операторы приведения типов направлены на минимизацию и локализацию небезопасных и подверженных ошибкам приемов программирования. В данном разделе рассматриваются ситуации, связанные с этой проблематикой (старые приведения типов, неявные сужающие преобразования и функции преобразования), и возможности преобразования существующего кода с учетом новых операторов приведения.

14.3.5.1. Старые приведения типов

Вводя новый стиль приведения типов, я хотел полностью заменить нотацию `(T)e`, объявив ее устаревшим синтаксисом. В таком случае комитет должен был предупредить пользователей о том, что нотацию `(T)e` могут исключить из следующего стандарта C++. Я видел прямые параллели между такой акцией и введением в стандарт ANSI/ISO C прототипов функций в стиле C++, когда неконтролируемые вызовы были объявлены устаревшими. Однако предложение не нашло должного отклика у большинства членов комитета, поэтому очистка C++, по-видимому, никогда не произойдет.

Важнее, однако, то, что новые операторы приведения дают некоторую возможность уйти от небезопасных конструкций, если есть мнение, что безопасность важнее обратной совместимости с C. Новый синтаксис приведения может быть также поддержан предупреждениями компилятора против использования старых приведений типов.

Благодаря новым операторам приведения становится возможен более безопасный, но не менее эффективный стиль программирования. Очевидно, важность этого аспекта начнет возрастать по мере того, как будет повышаться общее качество кода, а новые инструментальные средства, ориентированные на безопасность типов, найдут широкое применение. Это относится как к C++, так и к другим языкам.

14.3.5.2. Неявные сужающие преобразования

Идея уменьшить число нарушений статической системы типов и сделать их по возможности очевидными лежит в основе всей работы над новым синтаксисом

приведений. Разумеется, в этом контексте еще раз рассматривалась возможность устранения неявных сужающих преобразований, таких как `long` в `int` и `double` в `char` (см. раздел 2.6.1). К сожалению, их полный запрет не только недостижим, но и вреден. Главная проблема в том, что при выполнении арифметических операций может происходить переполнение:

```
void f(char c, short s, int i)
{
   c++;   // результат может не поместиться в char
   s++;   // результат может не поместиться в short
   i++;   // может произойти переполнение
}
```

Если запретить неявное сужение диапазона, то операции `c++` и `s++` станут недопустимыми, поскольку объекты типа `char` и `short` преобразуются в `int` перед выполнением арифметических действий. Если требовать, чтобы сужающие преобразования всегда задавались явно, этот пример пришлось бы переписать так:

```
void f(char c, short s, int i)
{
   c = static_cast<char>(c+1);
   c = static_cast<short>(s+1);
   i++;
}
```

Я не надеюсь, что такая нотация приживется, если у нее нет какого-либо очевидного преимущества. А в чем преимущество такой записи? Загрязнение кода явными приведениями никак не улучшает его ясности и даже не уменьшает число ошибок, поскольку пользователи будут применять операторы приведения, не слишком задумываясь. Конструкция `i++` также небезопасна из-за возможности переполнения. Добавление явных приведений бывает даже вредным, поскольку компилятор по умолчанию может вставлять код для обработки переполнений во время исполнения, а явное приведение будет подавлять этот механизм. Лучше было бы определить `dynamic_cast` так, чтобы данный оператор во время исполнения осуществлял проверку значения числового операнда. Тогда пользователи, для которых подобный контроль важен, могли бы при необходимости применять это средство. Также допустимо написать функции для проверки (см. раздел 15.6.2), например:

```
template<class V, class U> V narrow(U u)
{
   V v = u;
   if (v!=u) throw bad_narrowing;
   return v;
}
```

Несмотря на то что полный запрет сужающих преобразований невозможен и потребовал бы серьезного пересмотра правил выполнения арифметических операций, остается еще целый ряд преобразований, против которых компилятор мог

бы предупреждать достаточно уверенно: преобразование типа с плавающей точкой в интегральный, `long` в `short` и `long` в `char`. В Cfront так всегда и происходило. Остальные потенциально сужающие преобразования, например, `int` в `float` и `int` в `char` часто безвредны, чтобы пользователи смирились с предупреждениями компилятора.

14.3.5.3. Нотация вызова конструктора

C++ поддерживает нотацию конструктора `T(v)` как синоним `(T)v` старого приведения типов. Было бы лучше переопределить `T(v)` как аналог допустимого конструирования объекта (как во время инициализации):

```
T val(v);
```

Такое изменение, для которого, к сожалению, не придумано подходящего названия, потребовало бы времени, поскольку из-за него перестает работать существующий код (к тому же результату приводит и предложение объявить `(T)v` устаревшей конструкцией). Но данное предложение не получило поддержки в комитете. Правда, пользователи, которые хотят применить явную форму неявного преобразования (допустим, для устранения неоднозначности), могут написать для этого шаблон класса (см. раздел 15.6.2).

14.3.5.4. Использование новых приведений

Можно ли пользоваться новыми приведениями, не понимая всех описанных выше тонкостей? Удастся ли без особых сложностей преобразовать код, где использовались старые приведения типов, к новому стилю? Чтобы новый синтаксис вытеснил прежний, ответ на оба вопроса должен быть положительным.

Проще всего было бы во всех случаях подставить `static_cast` и посмотреть, как отреагирует компилятор. Каждую ошибку, если таковые возникнут в ходе компиляции, придется анализировать отдельно. Если проблема связана с нарушением константности, посмотрите, действительно ли приведение нарушает систему типов; если не нарушает, следует использовать `const_cast`. В ситуации, когда сложности возникли с неполными типами, указателями на функции или приведением между несоотносимыми типами, убедитесь, что получившийся указатель все-таки приводится к исходному типу. Если же неприятность касается преобразования указателя на `int` (или схожих действий), то стоит лишний раз подумать, зачем это было сделано; если устранить такое преобразование не удается, `reinterpret_cast` сделает то же самое, что в таких случаях делали старые приведения типов.

Обычно подобный анализ и устранение старых приведений может выполнить не слишком сложная программа. Но все равно было бы лучше удалить все имеющиеся приведения.

Глава 15. Шаблоны

Ничто не дается с таким трудом, не оставляет столько сомнений в успехе и не сопряжено с большим риском, чем установление нового порядка вещей.

Никколо Макиавелли

15.1. Введение

Шаблоны и исключения были специально упомянуты в статье «Whatis?» [Stroustrup, 1986b] как желательные в C++ (см. раздел 3.15). Проектирование этих средств описано в работах [Stroustrup, 1988b], [Koenig, 1989b], [Koenig, 1990] и в ARM, а их включение в язык отражено в предложениях по стандартизации C++.

Первопричина появления шаблонов – желание параметризовать контейнерные классы. Механизм же исключений есть в языке, потому что изначально хотелось располагать стандартным способом обработки ошибок во время выполнения. В обоих случаях C предоставлял лишь очень примитивные средства, которые не позволяли программисту явно выразить свои намерения и не слишком хорошо совмещались с ключевыми концепциями C++. Еще со времен C with Classes мы пользовались макросами для параметризации контейнеров (см. раздел 2.9.2), но макросы в C не могут работать с областями действия и типами и плохо увязываются с инструментальными средствами. Механизмы, которые в ранних версиях C++ использовались для обработки ошибок – имеются в виду `setjmp/longjmp` и индикаторы типа `errno`, – плохо сочетались с конструкторами и деструкторами.

Отсутствие таких механизмов приводило к неудачному проектированию, кодированию на излишне низком уровне и трудностям при совместном использовании библиотек, полученных из различных источников, то есть затрудняло работу на нужном (высоком) уровне абстракции.

На мой взгляд, шаблоны и исключения – это две стороны одной медали. Первые позволяют уменьшить число ошибок во время выполнения, расширяя спектр задач, с которыми может справиться статическая система типов. В свою очередь, исключения дают механизм для обработки оставшихся ошибок. С помощью шаблонов до разумного уровня можно довести число ошибок времени выполнения, которые обрабатываются с помощью исключений. Исключения предоставляют библиотекам, основанным на шаблонах, возможность уведомить вызывающую программу об ошибках.

15.2. Зачем нужны шаблоны

В первоначальном проекте C++ параметризованные типы рассматривались, но их пришлось отложить из-за нехватки времени на тщательное изучение вопросов проектирования и реализации, а также из-за боязни слишком усложнить компилятор. В частности, меня беспокоило, что неудачное проектирование может стать причиной замедления компиляции и компоновки. Я также считал, что удачная поддержка параметризованных типов значительно увеличит время, необходимое для переноса на другие платформы. К несчастью, мои опасения полностью подтвердились.

Шаблоны считались чрезвычайно важными средствами для правильного проектирования контейнерных классов. Впервые проект шаблонов был представлен на конференции USENIX по C++, состоявшейся в 1988 г. в Денвере [Stroustrup, 1988b]. В конце своего выступления я сказал:

«В контексте C++ вопросы ставятся следующим образом:

- можно ли сделать параметризацию типа простой для применения?
- удастся ли объекты параметризованного типа использовать так же эффективно, как и объекты «жестко заданного» типа?
- можно ли интегрировать в C++ параметризованные типы общего вида?
- возможно ли реализовать параметризованные типы так, чтобы скорость компиляции и компоновки была сравнима с той, которую обеспечивает система разработки, не поддерживающая параметризацию типов?
- удастся ли сделать такую систему компиляции простой и переносимой?»

Таковы были мои критерии проектирования шаблонов. Конечно, сам я был уверен, что на все эти вопросы удастся ответить положительно. Я утверждал, что фундаментальные альтернативы выглядят так:

«При написании стандартной библиотеки сталкиваешься с серьезной проблемой – в C++ не предоставлено общих средств определения «контейнерных классов»: списков, векторов, ассоциативных массивов и т.д. Есть два подхода к реализации таких классов:

- в Smalltalk – опора на динамическую систему типов и наследование;
- в Clu – на статическую систему типов и средства для работы с аргументами обобщенного типа *type*.

Первый подход очень гибок, но связан с высокими затратами и, что важнее, не позволяет статической системе типов обнаружить ошибки интерфейса. Второй традиционно ведет к довольно сложной реализации; средства компиляции и компоновки при этом тоже излишне сложны. Этому подходу также свойственна недостаточная гибкость, поскольку языки, в которых он использовался, прежде всего Ada, не имеют механизма наследования.

Изначально нам хотелось иметь в C++ механизм, так же хорошо структурированный, как в Clu, и имеющий столь же хорошие характеристики по времени исполнения и расходу памяти, но значительно быстрее компилируемый. Еще он должен быть таким же гибким, как в Smalltalk. Первое легко осуществимо, ко второму во многих случаях удается приблизиться».

Итак, ключевые вопросы выстраивались следующим образом: удобство нотации, эффективность выполнения и безопасность с точки зрения типов. Основные ограничения – переносимость и приемлемая производительность компилятора

и компоновщика, включая инстанцирование шаблонов классов и функций, прямо или косвенно используемых в программе.

Основные идеи о предполагаемых функциях параметризованных типов рождались при написании программ, где шаблоны имитировались с помощью макросов. Кроме меня, этим занимались Эндрю Кениг, Джонатан Шопиро и Алекс Степанов. Мы написали много макросов в стиле шаблонов, чтобы понять, какие языковые средства необходимы для поддержки данного стиля программирования. Размышляя над шаблонами, я не отводил главное место языку Ada, который лишь вызывал у меня раздражение своими операторами инстанцирования шаблонов (см. раздел 15.10.1). Однако Алекс Степанов хорошо знал данный язык, поэтому некоторые из характерных для Ada приемов, вероятно, перешли в C++.

Ранняя версия шаблонов реализована в варианте Cfront в 1989 г., автор – Сэм Харадхвала из компании Object Design Inc. Поддерживались только шаблоны классов. Затем Стэн Липпман расширил версию до полной реализации и поддержал механизм инстанцирования шаблонов, спроектированный Гленом МакКласки (Glan McCluskey) при участии Тони Хансена, Эндрю Кенига, Роба Мюррея и моем [McCluskey, 1992]. Мэри Фонтана и Мартин Нит (Martin Neath) из Texas Instruments написали свободно распространяемый препроцессор, в котором был реализован один из вариантов шаблонов [Fontana, 1991].

Несмотря на полученный опыт, мы все еще опасались включать в стандарт то, что не вполне понято, поэтому в ARM механизм шаблонов сознательно был определен в сокращенном виде. Уже тогда стало понятно, что это слишком усеченный вид, но гораздо труднее исключить неудачные средства, чем добавить новые.

Механизм шаблонов, изложенный в ARM, комитет ANSI C++ одобрил в июле 1990 г. Важным доводом в пользу включения шаблонов в предварительный стандарт стало сделанное членами комитета наблюдение, что уже имелось более полумиллиона строк реально использовавшегося кода на C++, где применялись шаблоны и их заменители.

Появление шаблонов стали переломным моментом в процессе перехода от одной стратегии проектирования нового средства в C++ к другой. Ранее применялись реализации, использования, обсуждения и повторные реализации. После включения в язык шаблонов все новые средства подробно обсуждались в комитете по стандартизации, а реализация шла параллельно этому обсуждению. Дискуссия по поводу шаблонов не была достаточно всесторонней, и в результате многие аспекты шаблонов пришлось пересматривать, базируясь на опыте последующей реализации и применения.

Цель, ради которой создавались шаблоны, была достигнута. В частности, данное средство позволяет проектировать эффективные, компактные и безопасные с точки зрения типов контейнерные классы и удобно ими пользоваться. Без шаблонов проектировщику пришлось бы прибегать к слабо или динамически типизированным вариантам, что отрицательно сказалось бы на структуре и эффективности программы.

Но я был слишком осторожен при специфицировании свойств шаблонов. Ведь такие возможности, как явная спецификация аргументов шаблона функции (см. раздел 15.6.2), выведение аргументов шаблона функции, не являющихся

типами (см. раздел 15.6.1), и вложенные шаблоны (см. раздел 15.9.3), можно было включить в язык сразу. Это бы несильно отразилось на сложности компилятора, а пользователям помогло. С другой стороны, я не предложил достаточной поддержки разработчикам компиляторов в плане инстанцирования шаблонов (см. раздел 15.10).

Предложение было передано на рассмотрение комитета ANSI/ISO по стандартизации. Правила привязки имен, механизм явного инстанцирования, ограничения на специализацию и явная квалификация вызовов шаблонов функций одобрены для включения в C++ на заседании комитета в Сан-Хосе в ноябре 1993 г. как часть общего плана совершенствования определения шаблонов.

15.3. Шаблоны классов

Ключевые конструкции излагались следующим образом [Stroustrup, 1988b]:

«В C++ параметризованный тип называется шаблоном класса. Он описывает, как можно конструировать отдельные классы, точно так же, как сам класс описывает конструирование индивидуальных объектов. Шаблон класса вектора можно было бы объявить так:

```
template<class T> class vector {
   T* v;
   int sz;
public:
   vector(int);
   T& operator[](int);
   T& elem(int i) { return v[i]; }
};
```

Префикс `template<class T>` дает понять, что объявляется шаблон и что внутри объявления будет использоваться аргумент `T` типа type (тип, переданный при объявлении). После этого в области действия объявления шаблона `T` можно использовать точно так же, как любое имя любого другого типа. Объявленные векторы допускается применять следующим образом:

```
vector<int> v1(20);
vector<complex> v2(30);

typedef vector<complex> cvec; // cvec - синоним для vector<complex>

cvec v3(40); // v2 и v3 имеют один и тот же тип

void f()
{
   v1[3] = 7;
   v2[3] = v3.elem(4) = complex(7,8);
}
```

Пользоваться шаблонами не сложнее, чем классами. Полные имена экземпляров шаблона класса, такие как `vector<int>` или `vector<complex>`, легко читаются. Быть может, данная запись даже понятнее, чем нотация для массива встроенного типа: `int[]` и `complex[]`. Если полное имя оказывается слишком длинным, то можно ввести сокращение с помощью `typedef`.

Объявить шаблон класса лишь немного сложнее, чем сам класс. Ключевое слово `class` используется для того, чтобы обозначить аргументы типа type. С одной стороны, оно вполне подходит для этого, с другой – отпадает необходимость вводить новое ключевое слово. В данном контексте `class` означает «любой тип», а не только «тип, определенный пользователем».

Угловые скобки `<...>` используются вместо фигурных `{...}` с целью подчеркнуть иную природу аргументов шаблона, а также потому, что `{...}` используются в C++ слишком часто.

Ключевое слово `template` введено, чтобы объявления шаблонов находились с легкостью, а шаблоны классов и функций были обеспечены единым синтаксисом».

Шаблоны – это механизм генерирования типов. Сами по себе они не являются типами, никак не представлены во время исполнения, поэтому не оказывают влияния на модель размещения объекта в памяти.

Я хотел, чтобы новое средство не уступало в эффективности макросам, ведь по идее шаблоны должны использоваться для представления таких низкоуровневых типов, как массивы и списки. Поэтому важной возможностью представлялось встраивание. В частности, я полагал, что наличие стандартных шаблонов для массива и вектора – единственный разумный способ поместить низкоуровневую концепцию массива C в глубь языка. Но более высокоуровневые альтернативы – массив с контролем выхода за границу и оператором `size()`, многомерный массив, векторный тип с надлежащей семантикой операций и копирования и т.д. – пользователи примут только в случае, если быстрота действия, расход памяти и удобство нотации будут не хуже, чем для встроенных массивов.

Другими словами, механизм реализации параметризованных типов должен быть таким, чтобы программист мог заменить обычные массивы стандартным библиотечным классом (см. раздел 8.5). Естественно, встроенные массивы попрежнему останутся в языке: они необходимы части пользователей и применяются в миллионах строк кода. Однако я хотел предложить эффективную альтернативу тем, кто ценит удобство и безопасность типов больше совместимости.

Кроме того, в C++ поддерживаются виртуальные функции, а следовательно, любая концепция, для которой в очевидной реализации потребовалась бы таблица переходов. Например, «истинно абстрактное» множество, состоящее из элементов типа `T`, можно было бы реализовать как шаблон абстрактного класса с виртуальными функциями, применяемыми к объектам `T` (см. раздел 13.2.2). Поэтому я начал искать решения, где основная нагрузка ложилась бы на компилятор, обеспечивая при этом близкую к оптимальной производительность в процессе выполнения как по времени, так и по памяти.

15.3.1. Аргументы шаблонов, не являющиеся типами

Кроме аргументов-типов C++ допускает в шаблонах и аргументы, не являющиеся типами. Предполагалось, что они будут использоваться главным образом для задания размеров и верхних границ контейнерных классов. Например:

```
template<class T, int i> class Buffer {
   T v[i];
   int sz;
public:
   Buffer() : sz(i) {}
   // ...
};
```

Такие шаблоны важны, если речь идет о конкуренции с массивами и структурами C в плане эффективности и компактности представления. Передача размера позволяет разработчику контейнера избежать использования свободной памяти.

Если бы аргументы, не являющиеся типами, не поддерживались, то пришлось бы передавать размер через тип массива, как в следующем примере:

```
template<class T, class A> class Buf {
   A v;
   int sz;
public:
   Buf() : sz(sizeof(A)/sizeof(T)) {}
   // ...
};

Buf<int, int[700]> b;
```

Это решение неудачно, подвержено ошибкам и не подходит ни к каким типам, кроме целых. А я еще хотел для большей гибкости дать возможность передавать в качестве аргумента шаблона указатель на функцию!

В первоначальном проекте в качестве аргументов не могли выступать шаблоны и пространства имен. Теперь я не вижу причин запрещать такие аргументы, тем более что их полезность очевидна. Передача шаблонов классов в качестве аргументов шаблонов одобрена на заседании комитета в Сан-Диего в марте 1994 г.

15.4. Ограничения на аргументы шаблонов

На аргументы шаблонов не накладывается ограничений. Проверка типов откладывается до момента инстанцирования шаблона [Stroustrup, 1988b]:

«Стоит ли требовать от пользователя, чтобы он задавал набор операций, применимых к аргументу шаблона типа type? Например:

```
// Для аргумента шаблона типа T должны быть заданы
// операции =, ==, < и <=

template <
   class T {
      T& operator=(const T&);
      int operator==(const T&, const T&);
      int operator<=(const T&, const T&);
      int operator<(const T&, const T&);
   };
>
class vector {
   // ...
};
```

Нет. Требование обязательно задавать такую информацию снижает гибкость средств параметризации, не упрощая реализацию и не повышая безопасность. <...> Высказывалось мнение, что читать и понимать параметризованные типы будет проще, если задан полный набор

операций над типами параметров. Я вижу здесь две проблемы: списки операций могут оказаться очень длинными, а для многих приложений понадобится объявлять больше шаблонов, чем это реально необходимо».

Видимо, я недооценил важность ограничений для удобства чтения и раннего выявления ошибок, но, с другой стороны, обнаружились и дополнительные проблемы при выражении таковых: тип функции специфичен, чтобы быть эффективным ограничением. Если рассматривать тип функции «буквально», то он слишком ограничивает решение. В примере с шаблоном класса `vector` логично ожидать, что подходит любая операция <, принимающая два аргумента типа `T`. Однако кроме встроенного оператора < есть еще несколько разумных альтернатив:

```
int X::operator<(X);
int Y::operator<(const Y&);
int operator<(Z,Z);
int operator<(const ZZ&, const ZZ&);
```

При буквальной интерпретации допустимым аргументом для `vector` является только вариант с `ZZ`.

Возможность наложить ограничения на шаблоны обдумывалась неоднократно:

- некоторые пользователи полагают, что при наличии ограничений на аргументы шаблонов можно сгенерировать оптимальный код, – я в это не верю;
- другие думают, что в отсутствие ограничений ставится под угрозу статическая проверка типов. На самом деле только некоторые этапы проверки откладываются до момента компоновки и на практике это является проблемой;
- кто-то считает, что при наличии ограничений объявление шаблона было бы проще понять. Часто это действительно так.

В следующих пунктах описываются два способа выражения ограничения. В некоторых случаях альтернативами ограничениям являются генерирование функций-членов, когда они действительно нужны (см. раздел 15.5), и специализация (см. раздел 15.10.3).

15.4.1. Ограничения за счет наследования

Дуг Леа, Эндрю Кениг, Филипп Готрон (Philippe Gautron), я и многие другие независимо обнаружили, как можно воспользоваться синтаксисом наследования для выражения ограничений. Например:

```
template <class T> class Comparable {
   T& operator=(const T&);
   int operator==(const T&, const T&);
   int operator<=(const T&, const T&);
   int operator<(const T&, const T&);
};

template <class T : Comparable>
   class vector {
      // ...
   };
```

Это имеет смысл. Предложения отличаются деталями, но позволяют перенести обнаружение ошибок и диагностику на этап компиляции отдельной единой трансляции. Возможные варианты пока обсуждаются в группе по стандартизации C++.

У меня, впрочем, есть принципиальные возражения против выражения ограничений с помощью наследования. Данная возможность будет подталкивать программистов к такой организации программ когда все то, что может являться разумным ограничением, выносится в класс, а наследование рассматривается как средство выражения любых ограничений. Например, вместо того чтобы сказать «в классе `T` должен быть оператор меньше», придется говорить «класс `T` должен быть производным от класса `Comparable`». Это непрямой и негибкий способ выражения ограничений, который легко ведет к злоупотреблению наследованием.

Поскольку между встроенными типами (`int`, `double` и т.д.) нет отношений наследования, то использовать его для выражения ограничений на такие типы нельзя. Недопустимо также применение наследования для выражения ограничений, которые можно использовать одновременно со встроенным и определенным пользователем типами. Например, с помощью наследования нельзя сказать, что `int` и `complex` допустимы в качестве аргументов шаблона.

Далее, автор шаблона не может предвидеть всех возможных применений средства. Поэтому сначала на аргументы шаблона будут накладываться чересчур строгие ограничения, а затем – на основе опыта – они начнут излишне ослабляться. Логическим следствием использования метода «ограничений за счет наследования» будет введение универсального базового класса, выражающего идею «отсутствия ограничений». Но наличие подобного базового класса породит небрежное отношение к программированию как в контексте использования шаблонов, так и в других ситуациях (см. раздел 14.2.3).

Применение наследования в качестве средства ограничений на аргументы шаблонов не позволяет программисту написать два шаблона с одним и тем же именем, один из которых используется для указательных, а другой – для неуказательных типов. Проблему – мое внимание к ней привлек Кит Горлен – можно решить с помощью перегрузки шаблонов функций (см. раздел 15.6.3.1).

Фундаментальное основание для критики этого подхода: наследование используется для того, что не является порождением подтипов. По-моему, наследование применяется для выражения ограничений не в силу глубоких внутренних причин. Отношения наследования – не единственные полезные отношения, имеющиеся в языке.

15.4.2. Ограничения за счет использования

Когда я впервые работал с компилятором, реализующим шаблоны, то решил проблему ограничений, выразив их в виде встраиваемой функции. Например:

```
template<class T> class X {
   // ...
   void constraints(T* tp)
   {                  // T должен иметь:
      B* bp = tp;     //  доступный базовый класс B
      tp->f();        //  функцию-член f
```

```
        T a(1);    // конструктор из int
        a = *tp;   // оператор присваивания
        // ...
    }
};
```

К сожалению, здесь используется одна деталь, характерная для определенного компилятора: Cfront выполняет полный синтаксический и семантический контроль всех встраиваемых функций в момент инстанцирования объявленного шаблона. Однако вариант функции с конкретным набором аргументов не следует инстанцировать, если она фактически не вызывается (см. раздел 15.5).

Такой подход позволяет автору шаблона задать ограничивающую функцию, а пользователь может проверить выполнение ограничения, вызвав ее в удобный для себя момент.

С другой стороны, автор может вызывать функцию `constraints()` и сам из каждого конструктора. Но это утомительно, если конструкторов много и не все из них встраиваются.

Эту концепцию можно было бы формализовать, введя в язык специальное средство:

```
template<class T> {
    constraints {
        T* bp;         // T должен иметь:
        B* bp = tp;    // доступный базовый класс B
        tp->f();       // функцию-член f
        T a(1);        // конструктор из int
        a = *tp;       // оператор присваивания
        // ...
    }
}
class X {
    // ...
};
```

Формализация позволила бы налагать ограничения и на аргументы шаблонов функций, но вряд ли такой подход стоит реализовывать. Однако из всех известных мне систем эта – единственная приближающаяся к моему принципу не вводить слишком жесткие ограничения на аргументы шаблонов, сохраняя в то же время единство, лаконичность, доступность и простоту реализации.

Примеры важных ограничений, выраженных функциями-членами шаблонов, см. в разделах 15.9.1 и 15.9.2.

15.5. Устранение дублирования кода

Устранение ненужного расхода памяти, вызванного слишком большим числом инстанцирований, считалось первоочередной проблемой проектирования языка, а не деталью реализации. Правила, требующие позднего инстанцирования функций-членов шаблона (см. разделах 15.10 и 15.10.4), гарантируют, что код не будет

дублироваться, если шаблон используется с одними и теми же аргументами в разных единицах трансляции. Я считал маловероятным, что при раннем (или даже позднем) инстанцировании шаблона возможно будет поискать, не инстанцирован ли тот же шаблон с другими аргументами, и определить, когда можно разделять инстанцированный код полностью или частично. И все же было чрезвычайно важно избежать такого неоправданного увеличения кода, которое встречается при расширении макросов и в языках с примитивным механизмом инстанцирования [Stroustrup, 1988b]:

«Среди прочего наследование гарантирует разделение кода между различными типами (код невиртуального базового класса разделяется всеми производными от него классами). Экземпляры шаблона не разделяют код, если только не применяется какая-либо своеобразная техника компиляции. Я не питаю надежд на скорое появление такой техники. Но можно ли воспользоваться наследованием для решения проблемы дублирования кода, возникающей из-за применения шаблонов? Для этого потребовалось производить шаблон от обычного класса. Например:

```
template<class T> class vector {     // обобщенный тип вектора
   T* v;
   int sz;
public:
   vector(int);
   T& elem(int i) { return v[i]; }
   T& operator[](int i);
   // ...
};

template<class T> class pvector : vector<void*> {
   // вектор указателей
   // производный от vector<void*>
public:
   pvector(int i) : vector<void*>(i) {}
   T*& elem(int i)
       { return (T*&) vector<void*>::elem(i); }
   T*& operator[](int i)
       { return (T*&) vector<void*>::operator[](i); }
   // ...
};

pvector<int*> pivec(100);
pvector<complex*> icmpvec(200);
pvector<char*> pcvec(300);
```

Реализации всех трех классов вектора указателей полностью разделяются. Все они написаны исключительно с помощью наследования и встраивания на основе класса `vector<void*>`**. Реализация же** `vector<void*>` **– один из серьезных кандидатов на включение в стандартную библиотеку».**

Благодаря описанной технике удалось предотвратить неоправданное увеличение объема кода. Те, кто не пользовался чем-то подобным (в C++ или в других языках со сходными средствами параметризации типов), сталкивались с неприятным

сюрпризом: на дублированный код могли уходить мегабайты памяти даже в программе скромного размера.

С другой стороны, я считал важным, чтобы компилятор инстанцировал лишь те функции-члены шаблона, которые использовались реально. Например, если есть шаблон `T` с функциями `f` и `g`, то компилятор должен инстанцировать только `f`, если `g` для данных аргументов шаблона не вызывается.

К тому же если вариант функции-члена будет генерироваться при данном наборе аргументов шаблона, только когда эта функция действительно вызывается, то мы повысим гибкость программы [Stroustrup, 1988b]:

«Рассмотрим `vector<T>`. Если мы хотим реализовать операцию сортировки, необходимо потребовать, чтобы для типа `T` было определено некоторое отношение порядка. Так обстоит дело не для всех типов. Если бы набор операций над `T` нужно было задавать в объявлении `vector`, пришлось бы иметь два типа векторов: один для объектов, имеющих отношение порядка, другой – для остальных. Если же множество операций над `T` задавать необязательно, достаточно и одного типа. Разумеется, нельзя будет сортировать векторы из объектов типа `glob`, для которых отношение порядка не определено. Даже при вашей попытке сделать это компилятор отвергнет сгенерированную функцию `vector<glob>::sort()`».

15.6. Шаблоны функций

Данное средство введено из-за необходимости иметь функции-члены в шаблонах классов, а также потому, что сама концепция шаблонов без него выглядела незаконченной. Конечно, были еще и хрестоматийные примеры, вроде функции `sort()`. Эндрю Кениг и Алекс Степанов предложили много примеров, доказывающих необходимость шаблонов функций. Самым важным стоит считать пример сортировки массива:

```
// объявление шаблона функции:
template<class T> void sort(vector<T>&);

void f(vector<int>& vi, vector<String>& vs)
{
   sort(vi); // sort(vector<int>& v);
   sort(vs); // sort(vector<String>& v);
}

// определение шаблона функции
template<class T> void sort(vector<T>& v)
/*
   Отсортировать элементы в порядке возрастания

   Алгоритм: пузырьковая сортировка (неэффективный, но очевидный)
*/
{
   unsigned int n = v.size();

   for (int i = 0; i<n-1; i++)
       for (int j=n-1; i<j; j--)
```

```
        if (v[j] < v[j-1]) { // переставить местами v[j] и v[j-1]
            T temp = v[j];
            v[j] = v[j-1];
            v[j-1] = temp;
        }
}
```

Как и ожидалось, шаблоны функций оказались незаменимы для поддержки шаблонов классов, когда сервисы предоставлялись обычными функциями, а не функциями-членами (например дружественными функциями, см. раздел 3.6.1).

Далее рассматриваются детали реализации шаблонов функций.

15.6.1. Выведение аргументов шаблона функции

Для шаблонов функций не нужно задавать аргументы шаблона – компилятор сам выводит их по фактическим параметрам, переданным при вызове. Разумеется, каждый аргумент шаблона, не специфицированный явно (см. раздел 15.6.2), должен однозначно определяться исходя из фактического параметра. В ходе стандартизации стало ясно, что необходимо точно определить, насколько «умно» должен вести себя компилятор при выведении аргументов шаблона из фактических параметров функции. Например, допустимо ли следующее:

```
template<class T, int i>
   T lookup(Buffer<T,i>& b, const char* p);

int f(Buffer<int,128>& buf, const char* p)
{
   return lookup(buf,p);   // использовать lookup(), где
                           // T - это int, а i - 128
}
```

Ранее ответ на вопрос был отрицательным, поскольку аргументы, не являющиеся типами, нельзя было вывести. Это означает, что невозможно определить не являющуюся членом невстраиваемую функцию, которая применялась бы к шаблону класса, принимающего в качестве аргумента не-тип. Например:

```
template<class T, int i> class Buffer {
   friend T lookup(Buffer&, const char*);
   // ...
};
```

Здесь требуется функция, определение которой раньше было незаконно.

После пересмотра данного предложения перечень конструкций, допустимых в списке аргументов шаблона функции, стал выглядеть так:

```
T
const T
volatile T
T*
T&
T[n]
```

```
some_type[I]
CT<T>
CT<I>
T (*)(args)
some_type (*) (args_containing_T)
some_type (*) (args_containing_I)
T C::*
C T::*
```

Здесь `T` – аргумент-тип шаблона, `I` – аргумент шаблона, который не является типом, `CT` – имя ранее объявленного шаблона класса, `args_containing_T` – список аргументов, из которого можно определить `T`, применяя эти правила, а `C` – имя класса. Теперь пример с функцией `lookup()` становится корректным. Пользователям не надо заучивать этот перечень, так как он просто формализует очевидный синтаксис.

Вот другой пример:

```
template<class T, class U> void f(const T*, U(*)(U));

int g(int);

void h(const char* p)
{
   f(p,g);    // T - это char, U - это int
   f(p,h);    // ошибка: невозможно вывести U
}
```

Глядя на фактические параметры в первом вызове `f()`, мы легко можем вывести фактические аргументы шаблона. Смотря на второй вызов `f()`, видим, что `h()` не соответствует образцу `U(*)(U)`, поскольку типы аргумента и возвращаемого значения различаются.

В прояснении этого и многих других подобных вопросов существенную помощь оказал Джон Спайсер (John Spicer).

15.6.2. Задание аргументов шаблона функции

Проектируя шаблоны, я думал о том, чтобы разрешить явное задание аргументов для шаблона функции точно так же, как можно задавать аргументы шаблонов классов. Например:

```
vector<int> v(10);  // класс, аргумент шаблона 'int'
sort<int>(v);       // функция, аргумент шаблона 'int'
```

Однако от этой идеи пришлось отказаться, потому что в большинстве примеров явно задавать аргументы шаблона не требовалось. Также, я опасался неоднозначностей и трудностей при синтаксическом анализе. Скажем, как следует разбирать этот пример?

```
void g()
{
   f<1>(0);  // (f) < (1>(0)) или (f<1>) (0) ?
}
```

Теперь я не считаю это проблемой. Если `f` – имя шаблона, то `f<` – начало квалифицированного имени и последующие лексемы должны интерпретироваться с учетом этого факта; в противном случае < означает «меньше».

Явное задание может быть полезно, поскольку мы не можем вывести тип возвращаемого значения по вызову шаблона функции:

```
template<class T, class U> T convert(U u) { return u; }
void g(int i)
{
   convert(i);   // ошибка: нельзя вывести T
   convert<double>(i);  // T - double, U - int
   convert<char,double>(i);    // T - char, U - double
   convert<char*,double>(i);  // T - char*, U - double
          // ошибка: нельзя преобразовать
          // double в char*
}
```

Как и для аргументов функции по умолчанию, в списке явно заданных аргументов шаблона можно опускать только крайние правые элементы.

Явное задание аргументов шаблона позволяет определить семейства функций преобразования и создания объектов. Явное преобразование, которое выполняет то, чего можно добиться одним лишь неявным преобразованием, например функция `convert()` в приведенном примере, достаточно часто требуется и хорошо подходит для включения в библиотеку. Еще один вариант – применить проверку, которая гарантировала бы, что для любого сужающего преобразования можно будет обнаружить ошибку во время выполнения.

Мы осознанно сделали похожим синтаксис новых операторов приведения (см. раздел 14.3) и явно квалифицированных вызовов шаблона функции. С помощью новых операторов приведения выражаются действия, которые нельзя описать другими средствами языка. Аналогичные операции, например `convert()`, можно выразить в виде шаблонов функций, поэтому они необязательно должны быть встроенными операторами.

Еще одно применение явно заданных аргументов шаблона функции – управление работой алгоритма за счет задания типа или значения локальной переменной. Например:

```
template<class TT, class AT> void f(AT a)
{
   TT temp = a;  // используем TT для управления
                 // точностью вычислений
   // ...
}

void g(Array<float>& a)
{
   f<float>(a);
   f<double>(a);
   f<Quad>(a);
}
```

Включение в C++ явного задания аргументов шаблона функции одобрено на заседании комитета в Сан-Хосе в ноябре 1993 г.

15.6.3. Перегрузка шаблона функции

Коль скоро существуют шаблоны функций, встает вопрос, как следует разрешать их перегрузку. Для данного средства допустимы только точные соответствия, а при разрешении перегрузки предпочтение отдается обычной функции с тем же именем:

«Разрешение перегрузки для шаблонов функций и других функций с тем же именем выполняется в три этапа [ARM]:

- **поиск точного соответствия [ARM, §13.2] среди функций; называние функции, если она найдена;**
- **нахождение шаблона функции, из которого можно инстанцировать функцию, точно соответствующую параметрам вызова; вызов этой функции, если шаблон найден;**
- **попытка применить обычную перегрузку [ARM, §13.2] для функций; вызов функции, если она найдена.**

Если соответствие не найдено, вызов считается ошибкой. Если на первом шаге отыскивается более одного соответствия, то вызов неоднозначен и также считается ошибкой».

Теперь такой подход кажется узкоспециализированным. Хотя он и работает, но служит почвой для многих мелких сюрпризов и неприятностей.

Уже в то время мне было ясно, что лучше как-то унифицировать правила для обычных функций и шаблонов. Но я не знал как. Вот приблизительный вариант альтернативного подхода, сформулированный Эндрю Кенигом:

«Для данного вызова найти множество функций, которые в принципе можно было бы подставить. В стандартном случае оно будет содержать функции, сгенерированные из разных шаблонов. Применить обычные правила разрешения к этому множеству функций».

Такое решение позволило бы применять преобразования к аргументам шаблонов функций, и мы получили бы общую схему перегрузки для любых функций. Например:

```
template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };

template<class T> void f(B<T>*);

void g(B<int>* pb, D<int>* pd)
{
   f(pb); // f<int>(pb)
   f(pd); // f<int>((B<int>*)pd);
          // используется стандартное преобразование
}
```

Это необходимо для того, чтобы шаблоны функций правильно взаимодействовали с наследованием. Другой пример:

```
template<class T> T max(T,T);

const int s = 7;
```

```
void k()
{
   max(s,7); // max(int(s),7); используется тривиальное преобразование
}
```

В ARM я предвидел, что возникнет необходимость ослабить правило, запрещающее применение каких бы то ни было преобразований. Многие из ныне существующих компиляторов допускают приведенные выше примеры. Но этот вопрос еще предстоит формально согласовать.

15.6.3.1. Условные выражения в шаблонах

При написании шаблона функции иногда желательно, чтобы определение могло зависеть от свойств аргумента шаблона. Так, в [Stroustrup, 1988b] читаем:

«Рассмотрим, как можно было бы написать функцию печати для типа вектора, которая перед выдачей сортирует элементы, но лишь тогда, когда сортировка возможна. Хорошо бы иметь некоторое средство, которое выясняет, можно ли к объектам данного типа применить данную операцию, скажем, <. Например:

```
template<class T> void vector<T>::print()
{
   // если в T есть операция <, отсортировать перед печатью

   if (?T::operator<) sort();
   for (int i=0; i<sz; i++) { /* ... */ }
}
```

При печати вектора, элементы которого можно сравнивать, вызывается функция `sort()`**, в противном случае ее вызов пропускается».**

Я решил не предоставлять такое средство для опроса типа, поскольку был убежден, – как убежден и сейчас, – что оно стало бы причиной написания плохо структурированного кода. В некоторых отношениях эта техника сочетает худшие черты макросов и чрезмерного использования RTTI (см. раздел 14.2.3).

Вместо этого для конкретных типов аргументов шаблона можно воспользоваться специализацией (см. раздел 15.10.3). Или же те операции, выполнение которых нельзя гарантировать для всех возможных типов аргументов, вынести в отдельные функции-члены, вызываемые лишь тогда, когда это возможно (см. раздел 15.5). Наконец, можно применить и перегрузку шаблонов функций, чтобы предоставить реализации для разных типов. В качестве примера рассмотрим шаблон функции `reverse()`, которая изменяет порядок элементов в контейнере на противоположный, если ей переданы итераторы, идентифицирующие первый и последний элементы. Пользовательский код должен был бы вызывать ее так:

```
void f(ListIter<int> l1, ListIter<int> l2, int* p1, int* p2)
{
   reverse(p1,p2);
   reverse(l1,l2);
}
```

где `ListIterator` используется для доступа к элементам в некотором определенном пользователем контейнере, а с помощью `int*` можно получать доступ

к обычному массиву целых. Чтобы сделать это, для каждого из двух вызовов нужно выполнять различные действия в зависимости от типов аргумента функции `reverse()`.

Шаблон функции `reverse()` просто выбирает реализацию на основе типа аргумента:

```
template<class Iter>
inline void reverse(Iter first, Iter last)
{
   rev(first,last,IterType(first));
}
```

Для выбора `IterType` используется разрешение перегрузки:

```
class RandomAccess { };

template<class T> inline RandomAccess IterType(T*)
{
   return RandomAccess();
}

class Forward { };

template<class T> inline Forward IterType(ListIterator<T>)
{
   return Forward();
}
```

Здесь для `int*` будет выбран `RandomAccess`, а для `ListIter` – `Forward`. Тип итератора определяет, какой вариант `rev()` использовать:

```
template <class Iter>
inline void rev(Iter first, Iter last, Forward)
{
   // ...
}
template <class Iter>
inline void rev(Iter first, Iter last, RandomAccess)
{
   // ...
}
```

Заметим, что третий аргумент в `rev()` фактически не применяется; он нужен только для правильной работы механизма перегрузки.

Важно, что каждое свойство типа или алгоритма можно представить типом (бывает, специально определенным для данной цели). Сделав это, мы можем использовать такой тип, чтобы указать механизму перегрузки, какую зависящую от этого свойства функцию выбрать. Если использованный тип не соответствует какому-либо действительно важному свойству, такая техника выглядит несколько искусственной, но остается общей и эффективной.

Благодаря встраиванию разрешение делается на стадии компиляции, поэтому подходящая функция `rev()` будет вызвана напрямую, без каких бы то ни было издержек во время выполнения. Заметьте, что механизм расширяемый, то есть новую реализацию `rev()` можно добавить, не трогая старый код. Данный пример основан на идеях из работы Алекса Степанова [Stepanov,1993].

Иногда может оказаться полезной идентификация типа во время исполнения (см. раздел 14.2.5).

15.7. Синтаксис

Первоначально я хотел поместить аргумент шаблона сразу после его имени:

```
class vector<class T> {
   // ...
};
```

Но такой синтаксис не всегда мог использоваться с шаблонами функций [Stroustrup, 1988b]:

«На первый взгляд, синтаксис функций выглядит логично и без нового ключевого слова:

```
T& index<class T>(vector<T>& v, int i) { /* ... */ }
```

Обычно параллель с шаблонами классов не нужна, поскольку аргументы шаблонов функций, как правило, явно не задаются:

```
int i = index(vi,10);
char* p = index(vpc,29);
```

Однако из-за такого упрощенного синтаксиса возникают сложности. Объявление шаблона оказывается трудно найти в программе, поскольку его аргументы слишком глубоко размещены в синтаксисе функций и классов, так что разбор шаблонов функций чрезвычайно труден. Можно написать синтаксический анализатор для C++, способный обрабатывать такие объявления шаблонов функций, в которых аргумент используется до того, как определен (см. выше пример с функцией `index()`). Я написал такой анализатор, но это было нелегко, а техника разбора там далека от традиционной. Если бы не было введено новое ключевое слово и не требовалось объявлять аргумент шаблона перед его использованием, это привело бы к тем же сложностям, что возникают из-за чересчур запутанного синтаксиса объявлений в C и C++».

В окончательном варианте синтаксис объявления функции `index()` принимает такой вид:

```
template<class T> T& index(vector<T>& v, int i) { /* ... */ }
```

В то время я серьезно обдумывал вариант синтаксиса, когда возвращаемое значение функции помещается после аргументов. Например:

```
index<class T>(vector<T>& v, int i) return T& { /* ... */ }
```

или

```
index<class T>(vector<T>& v, int i) : T& { /* ... */ }
```

С помощью данного варианта можно было решить проблемы синтаксического разбора, но многим пользователям удобно, когда имеется отдельное ключевое

слово, помогающее распознать шаблон, так что такого рода ухищрения оказались излишними.

Угловые скобки < . . . > были выбраны вместо фигурных по ряду причин: во-первых, многим пользователям казалось, что они легче читаются; во-вторых, как уже отмечалось выше, фигурные скобки и так часто используются в синтаксисе C и C++.

Тем не менее существовала определенная проблема. В предложении

```
List<List<int>> a;
```

на первый взгляд, объявляется список списков целых. На самом деле, это синтаксическая ошибка, поскольку лексема >> (сдвиг вправо или оператор вывода) – не то же самое, что две лексемы >. Разумеется, простой лексический трюк решил бы данную проблему, но я решил не запутывать ни грамматику, ни лексический анализатор. Однако с тех пор эта ошибка встречалась так часто, что теперь я испытываю сильнейшее желание ликвидировать проблему.

15.8. Методы композиции

Шаблоны поддерживают несколько безопасных и весьма мощных приемов композиции. Например, это средство можно применять рекурсивно:

```
template<class T> class List { /* ... */ };

List<int> li;
List< List<int> > lli;
List< List< List<int> > > llli;
```

Если нужны конкретные составные типы, то их легко определить с помощью наследования:

```
template<class T> class List2 : public List< List<T> > { };
template<class T> class List3 : public List2< List<T> > { };

List2<int> lli2;
List3<int> llli3;
```

Подобное использование наследования несколько необычно, поскольку не добавляется никаких новых членов. Наследование не сопряжено с дополнительными расходами времени или памяти; это всего лишь метод композиции. Если бы композицию нельзя было осуществить с помощью наследования, то для шаблонов пришлось бы изобрести какие-то особые механизмы композиции, иначе язык оказался бы гораздо беднее.

Переменные данных составных типов можно использовать точно так же, как и соответствующие определенные типы, но не наоборот:

```
void f()
{
   lli = lli2;   // правильно
   lli2 = lli;   // ошибка
}
```

Причина – открытое наследование определяет отношение подтипа.

Для того чтобы разрешить присваивание в обоих направлениях, понадобилось бы расширение языка, вводящее настоящие параметризованные синонимы. Например:

```
template<class T> typedef List< List<T> > List4;

void (List< List<T> >& lst1, List4& lst2)
{
   lst1 = lst2;
   lst2 = lst1;
}
```

Технически такое расширение реализовать несложно, но вряд ли стоит вводить еще одно средство для переименования.

Также благодаря наследованию можно частично задавать аргументы шаблонов в определении нового типа:

```
template<class U, class V> class X { /* ... */ };
template<class U> class XX : public X<U,int> { };
```

В обычном случае наследование от шаблона класса позволяет «подправить» базовый класс с помощью информации, нужной производному классу. Благодаря этому композиция становится более гибкой. Например:

```
template<class T> class Base { /* ... */ };

class Derived : public Base<Derived> { /* ... */ };
```

Описанная методика позволяет информации о производном классе попадать в определение базового класса. См. также раздел 14.2.7.

15.8.1. Представление стратегии реализации

Еще одно применение наследования и шаблонов для композиции – метод передачи объектов, представляющих стратегии реализации. Например, семантику сравнения для сортировки или средства распределения и освобождения памяти для контейнера можно было бы задать с помощью аргументов шаблона [2nd]:

«Один из возможных способов – использовать шаблон для составления нового класса из интерфейса для нужного контейнера и класса распределителя памяти, применяя технику размещения, описанную в [2nd, §6.7.2]:

```
template<class T, class A> class Controlled_container
   : public Container<T>, private A {

   // ...
   void some_function()
   {
      // ...
      T* p = new(A::operator new(sizeof(T))) T;
      // ...
   }
   // ...
};
```

Здесь необходимо использовать шаблон, поскольку мы проектируем контейнер. Наследование от класса `Container` необходимо, чтобы `Controlled_container` можно было использовать в качестве контейнера. Использование аргумента шаблона `A` позволяет применять различные распределители. Например:

```
class Shared : public Arena { /* ... */ };
class Fast_allocator { /* ... */ };
class Persistent : public Arena { /* ... */ };

Controlled_container<Process_descriptor,Shared> ptbl;

Controlled_container<Node,Fast_allocator> tree;

Controlled_container<Personnel_record,Persistent> payroll;
```

Это обычный способ передать нетривиальную информацию о реализации производному классу. Преимущества приема – систематичность и возможность использовать встраивание. Правда, в данном случае нередко возникают довольно длинные имена. Но для них можно ввести синонимы с помощью `typedef`».

В компонентах Буча [Booch, 1993] такой способ композиции используется повсеместно.

15.8.2. Представление отношений порядка

Рассмотрим задачу сортировки. У нас есть шаблон контейнера, тип элемента и функция, сортирующая элементы в контейнере.

Мы не можем поместить критерий сортировки внутрь контейнера, поскольку обычно хранящиеся в нем элементы не должны зависеть от него. У нас также нет возможности поместить этот критерий и внутри типа элементов, поскольку их можно сортировать разными способами.

Итак, критерий сортировки не встраивается ни в контейнер, ни в тип элемента. Вместо этого он передается в виде операции, которую следует выполнить. Если в качестве примера взять строки, состоящие из слов – личных шведских имен, тогда какую схему упорядочения применить для сравнения? Для шведского языка обычно применяются две разные схемы. Но нет сомнения, что нельзя давать сведения о соглашениях, принятых для такой операции, ни общему типу строки, ни общему алгоритму сортировки.

Таким образом, любое общее решение включает выраженный в общих терминах алгоритм сортировки, который можно определить не для конкретного типа, а для конкретного использования данного типа. Давайте, например, обобщим стандартную библиотечную функцию `strcmp()` для работы со строками любого типа `T`.

Сначала определяется шаблон класса с семантикой сравнения объекта типа `T` по умолчанию:

```
template<class T> class CMP {
public:
   static int eq(T a, T b) { return a==b; }
   static int lt(T a, T b) { return a<b; }
};
```

В шаблоне функции compare() для сравнения аргументов типа basic_string используется такая форма:

```
template,<class T> class basic_string {
   // ...
};

template<class T, class C = CMP<T> >
int compare(const basic_string<T>& str1,
      const basic_string<T>& str2)
{
   for(int i=0; i<str1.length() && i<str2.length(); i++)
      if (!C::eq(str1[i],str2[i]))
         return C::lt(str1[i],str2[i]);
   return str2.length()-str1.length();
}

typedef basic_string<char> string;
```

Имея в своем распоряжении шаблоны-члены (см. раздел 15.9.3), функцию compare() можно было бы определить и в виде члена класса basic_string.

Если требуется, чтобы C<T> производил сравнение без учета регистра, но с учетом специфики конкретного языка, возвращал наибольшее значение в коде unicode, когда аргументы неравны (имеется в виду C<T>::eq()) и т.д., то нужно лишь правильным способом определить C<T>::eq() и C<T>::lt() через характерные для типа T операторы. Тогда любой алгоритм (сравнения, сортировки и т.п.) можно выразить в терминах операций, предоставляемых классом CMP и контейнером. Например:

```
class LITERATE {
   static int eq(char a, char b) { return a==b; }
   static int lt(char,char);  // использовать книжный порядок
};
void f(string swede1, string swede2)
{
   compare(swede1,swede2); // обычный (телефонный) порядок
   compare<char,LITERATE>(swede1,swede2); // книжный порядок
}
```

Я передаю критерий сравнения в виде параметра шаблона, поскольку именно так можно передать операции без лишних затрат во время выполнения. В частности, операторы сравнения eq() и lt() легко встроить. Аргумент по умолчанию используется, чтобы не обременять пользователей громоздкой нотацией. Другие варианты этой техники рассматриваются в [2nd, §8.4].

Более характерный пример – это сравнение с учетом и без учета регистра:

```
void f(string s1, string s2)
{
   compare(s1,s2);                  // с учетом регистра
   compare<char,NOCASE>(s1,s2);     // без учета регистра
}
```

Отметим, что шаблон класса CMP никогда не используется для определения объектов; все его члены статические и открытые. Поэтому его следовало бы сделать пространством имен (см. главу 17):

```
template<class T> namespace CMP {
   int eq(T a, T b) { return a==b; }
   int lt(T a, T b) { return a<b; }
}
```

К сожалению, шаблоны-пространства имен (пока еще) не включены в C++.

15.9. Соотношения между шаблонами классов

Шаблон стоит рассматривать как спецификацию для создания конкретных типов. Другими словами, реализация шаблона – это механизм генерирования типов, указанных пользователем.

Согласно правилам языка C++ два класса, сгенерированные из одного и того же шаблона, никак не связаны между собой. Например:

```
template<class T> class Set { /* ... */ };

class Shape { /* ... */ };
class Circle : public Shape { /* ... */ };
```

Видя подобные объявления, пользователи зачастую трактуют Set<Circle> как класс, производный от Set<Shape>, или Set<Circle*> – как производный от Set<Shape*>. Например:

```
void f(Set<Shape>&);

void g(Set<Circle& s)
{
   f(s);
}
```

Этот пример не будет компилироваться, поскольку не существует встроенного преобразования из Set<Circle>& в Set<Shape>&. Да и не должно его быть; полагать, что Set<Circle> – частный случай Set<Shape>, – принципиальная (и не такая уж редкая) концептуальная ошибка. В частности, класс Set<Circle> гарантирует, что все его элементы принадлежат Circle (окружность), и значит, пользователи могут безопасно и эффективно применять к ним все операции, определенные для окружностей, например запрашивать значение радиуса. Если бы мы разрешили трактовать Set<Circle> как Set<Shape>, то уже не могли бы дать такой гарантии, поскольку в множество Set<Shape> можно поместить и другие геометрические фигуры, например, треугольники.

15.9.1. Отношения наследования

Следовательно, по умолчанию между классами, сгенерированными из одного и того же шаблона, не может быть никаких отношений. Но иногда такое отношение полезно. Нужна ли специальная операция для выражения такого рода отношений?

Я отверг эту идею, поскольку многие полезные преобразования можно выразить с помощью отношений наследования или обычных операторов-конверторов. Однако в результате не существует способа выразить некоторые отношения. Например, располагая

```
template<class T> class Ptr { // указатель на T
   // ...
};
```

часто хотелось бы для таких определенных пользователем указателей `Ptr` иметь такие же отношения наследования, к которым мы привыкли при работе со встроенными указателями. Например:

```
void f(Ptr<Circle> pc)
{
   Ptr<Shape> ps = pc;  // имеет ли это смысл?
}
```

Желательно, чтобы это было разрешено только тогда, когда `Shape` действительно является непосредственным или косвенным открытым базовым классом для `Circle`. Дэвид Джордан (David Jordan) от имени консорциума поставщиков объектно-ориентированных баз данных просил комитет по стандартизации обеспечить такое свойство для «умных» указателей.

Решение дают шаблоны-члены, которые пока не включены в C++:

```
template<class T1> class Ptr {// указатель на T1
   // ...
   template<class T2> operator Ptr<T2> ();
};
```

Нам нужно определить конвертор таким образом, чтобы преобразование `Ptr<T1>` в `Ptr<T2>` было допустимо только в случаях, когда `T1*` можно присвоить `T2*`. Это можно сделать, предоставив для `Ptr` дополнительный конструктор:

```
template<class T> class Ptr { // указатель на T
   T* p;
public:
   Ptr(T*);
   template<class T2> operator Ptr<T2> ();
      return Ptr<T2>(p);   // работает только тогда, когда
                           // p можно преобразовать в T2*
   }
   // ...
};
```

В этом решении не используются приведения типов. Предложение `return` будет компилироваться, только когда `p` может являться аргументом для конструктора `Ptr<T2>`. В приведенном примере `p` имеет тип `T1*`, а конструктору необходим аргумент типа `T2*`. Это применение метода ограничения через использование (см. раздел 15.4.2). Если вы предпочитаете иметь закрытый конструктор, то можете воспользоваться приемом, предложенным Джонатаном Шопиро:

```
template<class T> class Ptr { // указатель на T
   T* tp;
   Ptr(T*);
   friend template<class T2> class Ptr<T2>;
public:
   template<class T2> operator Ptr<T2> ();
   // ...
};
```

Шаблоны-члены описаны в разделе 15.9.3.

15.9.2. Преобразования

С вышеописанной проблемой тесно связана другая: не существует единого способа определить преобразования между различными классами, сгенерированными из одного и того же шаблона. Рассмотрим, например, шаблон `complex`, который определяет комплексные числа для некоторого множества скалярных типов:

```
template<class scalar> class complex {
   scalar re, im;
public:
   // ...
};
```

Мы можем использовать `complex<float>`, `complex<double>` и т.д., но при этом желательно, чтобы существовало преобразовывание из типа `complex` с низкой точностью в тип `complex` с высокой точностью. Например:

```
complex<double> sqrt(complex<double>);

complex<float> c1(1.2f,6.7f);
complex<double> c2 = sqrt(c1);// ошибка: несоответствие типов:
                              // ожидается complex<double>
```

Было бы неплохо, если бы существовал какой-то способ сделать вызов `sqrt` допустимым. Для этого программисты отказываются от шаблонов в пользу дублирования определения классов:

```
class float_complex {
   float re, im;
public:
   // ...
};
class double_complex {
   double re, im;
public:
   double_complex(float_complex c) : re(c.re), im(c.im) {}
   // ...
};
```

Цель такого дублирования – определить конструктор, который и задает преобразование.

И в этом случае, на мой взгляд, возможны только решения, требующие вложенных шаблонов в сочетании с той или иной формой ограничений. Фактически ограничение может быть неявным:

```
template<class scalar> class complex {
   scalar re, im;
public:
   template<class T2> complex(const complex<T2>& c)
      : re(c.re), im(c.im) { }
   // ...
};
```

Другими словами, сконструировать `complex<T1>` из `complex<T2>` удается только в том случае, когда есть возможность инициализировать `T1` с помощью `T2`. Это представляется разумным. Такое определение включает и обычный копирующий конструктор.

В данной ситуации приведенный выше пример с `sqrt()` становится законным. К сожалению, определение допускает и сужающие преобразования комплексных чисел просто потому, что в C++ допустимы сужающие преобразования для скаляров. Естественно, если принять такое определение `complex`, то компилятор, предупреждающий о сужающих преобразованиях для скаляров, будет предупреждать и о сужающих преобразованиях для значений типа `complex`.

Мы можем получить «привычные» имена с помощью `typedef`:

```
typedef complex<float> float_complex;
typedef complex<double> double_complex;
typedef complex<long double> long_double_complex;
```

Думается, что варианты без `typedef` читаются лучше.

15.9.3. Шаблоны-члены

Единственная причина, по которой, согласно ARM, шаблоны не могут являться членами классов, – я не смог убедить себя, что такую возможность будет легко реализовать. Шаблоны-члены входили в первоначальный проект шаблонов, и в целом я за то, чтобы любые конструкции, вводящие область действия, имели вложенные формы (см. разделы 3.12 и 17.4.5.4). Но если бы я безо всяких ограничений разрешил в C++ шаблоны-члены, то нечаянно разрушил бы модель размещения объекта в памяти, что отбросило бы нас немного назад. Рассмотрим решение задачи двойной диспетчеризации (см. раздел 13.8.1):

```
class Shape {
   // ...
   template<class T>
      virtual Bool intersect(const T&) const =0;
};

class Rectangle : public Shape {
   // ...
   template<class T>
      virtual Bool intersect(const T& s) const;
};
```

```
template<class T>
virtual Bool Rectangle::intersect(const T& s) const
{
   return s.intersect(*this); // *this - это Rectangle
                              // разрешаем в зависимости от s
}
```

Этот пример просто обязан быть незаконным, иначе пришлось бы вводить дополнительный элемент в таблицу виртуальных функций для класса `Shape` всякий раз, как кто-то вызовет `Shape::intersect()` с новым типом аргумента. Это означало бы, что создавать таблицы виртуальных функций и размещать в них указатели на функции может только компоновщик. Поэтому шаблон-член не может быть виртуальным.

Я обнаружил это уже после выхода ARM, так что ограничение, согласно которому шаблоны можно определять только в глобальной области действия, оказалось просто спасением. С другой стороны, упомянутые в этом разделе проблемы преобразования из-за отсутствия шаблонов-членов не имеют решения. Шаблоны-члены одобрены комитетом на заседании в Сан-Диего в марте 1994 г.

Во многих случаях альтернативой вложенным шаблонам классов служит явное задание аргументов шаблонов функций (см. раздел 15.6.2).

15.10. Инстанцирование шаблонов

Первоначально ([Stroustrup, 1988b], [ARM]) в C++ не было оператора для «инстанцирования» шаблона, то есть операции явного генерирования объявления класса и определений функций для конкретного набора аргументов шаблона. Лишь имея полную программу, можно узнать, какие шаблоны следует инстанцировать. Часто они определяются в библиотеках, а инстанцирование производится в результате действий пользователей, даже не подозревающих о наличии таких средств. Поэтому запрос инстанцирования, например, с помощью оператора `new` из Ada или подобного ему пользователю представлялся неразумным. Более того, если бы существовал оператор инстанцирования шаблона, он должен был бы грамотно обрабатывать случай, когда две не связанных между собой части программы требуют инстанцировать одну и ту же функцию с одним и тем же набором аргументов. В этом случае приходилось бы избегать дублирования кода и не потерять возможность динамического связывания.

В ARM приводятся комментарии на данную тему, но окончательного ответа не дается:

«Решение о том, какие функции генерировать из определения шаблона функции, нельзя принять, пока нет полной программы, то есть до тех пор, пока не станет ясно, какие функции нужны.

Принято, что обнаружение ошибок откладывается до последнего момента: когда после начальной стадии компоновки уже сгенерированы определения из шаблонов функций. Но многим может показаться, что это слишком поздно.

Кроме того, правило возлагает максимальную ответственность на среду программирования. Именно системе поручается найти определения шаблонов классов, шаблонов функций и классов, необходимых для генерирования определений функций по шаблонам. Для некоторых сред это слишком сложно.

Обе проблемы можно было бы смягчить, если ввести механизм, указывающий программисту, где сгенерировать определенные функции по шаблону с определенными аргументами. Это достаточно просто сделать в любой среде программирования. Тогда ошибки, относящиеся к конкретному определению шаблона функции, будут обнаруживаться в точке запроса.

Неясно, однако, следует ли считать такой механизм частью языка или среды. Было решено, что необходимо накопить опыт, а пока – хотя бы на время – такие механизмы воспринимаются как часть среды.

Простейший способ гарантировать правильную генерацию определений шаблонов функций – оставить решение на усмотрение программиста. Тогда компоновщик сообщит нужные определения, а файл, содержащий невстраиваемые определения шаблонов функций, можно будет откомпилировать, указав, какие будут использоваться аргументы шаблона. На базе такого подхода можно строить и более сложные системы».

Теперь имеется множество разных реализаций. Практика показала, что сложность проблемы не была переоценена и что нынешние реализации не вполне удовлетворительны.

В компиляторе Cfront [McCluskey, 1992] инстанцирование шаблонов автоматизировано, как и предполагалось в первоначальном проекте [Stroustrup, 1988b] и в ARM. Принцип таков: запускается компоновщик, и если оказывается, что какой-то шаблон функции не инстанцирован, то снова вызывается компилятор, который генерирует отсутствующий объектный код. Этот процесс повторяется, пока не будут инстанцированы все шаблоны. Определения шаблонов и типов аргументов (когда необходимо) базируются на соглашении об именовании файлов. При необходимости вместе с таким соглашением можно использовать отдельный задаваемый пользователем файл, в котором имена шаблонов и классов ставятся в соответствие файлам, где находятся их определения. У компилятора есть специальный режим для обработки инстанцирований шаблонов. Часто эта стратегия себя оправдывает, но в некоторых ситуациях возникают следующие проблемы:

- низкая производительность компиляции и компоновки. Если компоновщик определяет, что нужно инстанцирование, то приходится вызывать компилятор для генерирования необходимых функций. Затем компоновщик нужно вызвать снова. В системе, где компилятор и компоновщик не работают постоянно[1], это может оказаться очень накладно. Хороший механизм библиотек может существенно уменьшить число запусков компилятора;
- плохая интеграция с системами контроля версий, имеющими четко определенные представления о том, что такое исходный код, ка^к из него получается объектный. Подобные системы плохо взаимодействуют с системами разработки, в которых для получения полной программы совместно используются компилятор, компоновщик и библиотека (способ описан в первом пункте);
- слабое сокрытие деталей реализации. Если в реализации библиотеки используются шаблоны, то для того, чтобы пользователь мог скомпоновать свою программу с моей библиотекой, я должен предоставить исходные

[1] То есть не находятся в дисковом кэше операционной системы. – *Прим. перев.*

тексты шаблонов. Причина в том, что необходимость в инстанцировании шаблонов выявляется только на последнем этапе компоновки. Описанную проблему можно обойти, если заранее каким-то образом удастся сгенерировать объектный код, содержащий все версии шаблонов, которые могут быть инстанцированы для моей библиотеки. В свою очередь, это может привести к неоправданному увеличению объектного кода, если разработчик попытается предусмотреть все возможные способы использования, – а ведь в каждом конкретном приложении будет задействована лишь малая их часть. Отметим также, что, если инстанцирование прямо зависит от того, какие шаблоны пользователь инстанцирует, оно по необходимости должно быть поздним.

15.10.1. Явное инстанцирование

Перспективным вариантом решения указанных проблем представляется необязательная возможность явного инстанцирования. Это мог бы быть внеязыковой механизм, прагма компилятора или директива в самом языке. С тем или иным успехом опробовались все эти способы. Меньше всего из них мне нравится `#pragma`. Если уж нам нужен в языке механизм явного инстанцирования, то он должен быть общедоступен и иметь четко определенную семантику.

От необязательного оператора инстанцирования получим следующие преимущества:

- пользователю удастся задавать окружение для инстанцирования;
- он также сможет заранее создавать библиотеки наиболее часто инстанцируемых шаблонов способом, мало зависящим от компилятора;
- предварительно созданные библиотеки будут зависеть не от изменений в среде программы, которая их использует, а только от контекста инстанцирования.

Описанный здесь механизм инстанцирования по запросу одобрен на заседании комитета в Сан-Хосе; он основан на предложении Эрвина Унру (Erwin Unruh). Выбранный синтаксис соответствует тому, который используется при явном задании аргументов шаблона класса (см. раздел 15.3), вызовах шаблонов функций (см. раздел 15.6.2), в новых операторах приведения типов (см. разделы 14.2.2 и 4.3) и в специализациях шаблонов (см. раздел 15.10.3). Запрос на инстанцирование выглядит так:

```
template class vector<int>;                    // класс
template int& vector<int>::operator[](int);    // член
template int convert<int,double>(double);      // функция
```

Было решено использовать уже имеющееся ключевое слово `template`, а не вводить новое – `instantiate`. Объявление шаблона отличается от запроса на инстанцирование по списку аргументов шаблона: определение шаблона начинается с `template<`, а запрос на инстанцирование – просто с `template`. Предпочтение было отдано полностью специфицированной, а не сокращенной форме вроде:

```
// не C++

template vector<int>::operator[];    // член
template convert<int,double>;        // функция
```

Данное решение позволяет избежать неоднозначностей для перегруженных шаблонов функций и дает компилятору некоторую избыточную информацию, чтобы он мог выполнить проверки на непротиворечивость. Кроме того, запросы на инстанцирование встречаются достаточно редко, так что излишне лаконичная запись не очень нужна. Однако при вызовах шаблонов функций можно опускать аргументы шаблона, которые выводятся из фактических параметров функции (см. раздел 15.6.1). Например:

```
template int convert<int>(double);   // функция
```

Если шаблон класса инстанцируется явно, то инстанцируется и каждая его функция-член (см. раздел 15.10.4). Отсюда следует, что явное инстанцирование можно использовать для проверки ограничений (см. раздел 15.4.2).

Инстанцирование по запросу позволяет существенно снизить время компоновки и число повторных компиляций. Есть примеры, когда перемещение всех инстанцирований шаблонов в одну единицу трансляции сокращают время сборки программы с нескольких часов до нескольких минут. Ради такого ускорения можно принять механизм оптимизации вручную.

Что должно происходить, когда шаблон дважды явно инстанцируется с одним и тем же набором аргументов? (Вопрос считался фундаментальным.) Если воспринимать это как безусловную ошибку, то явное инстанцирование становится серьезным препятствием на пути сборки программ из независимо разработанных частей. Именно это и послужило первоначальным доводом против включения оператора явного инстанцирования. С другой стороны, подавление лишнего явного инстанцирования в обычной ситуации может оказаться очень трудной задачей.

По вышеуказанному вопросу комитет решил дать свободу разработчикам компиляторов: множественное инстанцирование диагностировать необязательно. Это позволяет сложному компилятору игнорировать лишние инстанцирования и тем самым решить вышеупомянутую проблему составления программы из библиотечных модулей. Однако компилятор не всегда бывает настолько «умным». Пользователи более простых компиляторов должны самостоятельно избегать множественного инстанцирования, но даже в самом худшем случае программа просто не будет загружаться; незаметно для пользователя семантика не изменится.

Как и раньше, язык не требует инстанцировать шаблон явно. Явное инстанцирование – всего лишь оптимизация процесса компиляции и компоновки вручную.

15.10.2. Точка инстанцирования

Самый трудный аспект определения шаблонов – точное указание, к каким объявлениям относятся имена, использованные в определении шаблона. Часто это называют проблемой привязки имен.

Описываемые здесь пересмотренные правила привязки имен – плод многолетнего труда разработчиков, в особенности членов рабочей группы по расширениям –

Эндрю Кенига, Мартина О'Риордана, Джонатана Шопиро. К моменту принятия (ноябрь 1993 г., Сан-Хосе) правила уже были всесторонне испытаны на практике.

Рассмотрим пример:

```
#include <iostream.h>
#include <vector.h>

void db(double);
                                                        // # 1
template<class T> T sum(vector<T>& v)
{
   T t = 0;
   for (int i = 0; i<v.size(); i++) t = t + v[i];
   if (DEBUG) {
       cout << "сумма равна " << t << '\n';
       db(t);
       db(i);
   }
   return t;
}
// ...

#include <complex.h>
                                                        // # 2
void f(vector<complex>& v)
{
   complex c = sum(v);
}
```

В первоначальном определении говорилось, что любые имена, встречающиеся в шаблоне, привязываются в точке инстанцирования, она находится прямо перед глобальным объявлением, в котором шаблон используется в первый раз (в примере выше – `#2`). У этого определения есть, по меньшей мере, три нежелательных свойства:

- в точке определения шаблона ничего нельзя проверить. Например, если `DEBUG` в этой точке не определено, то не будет выдано сообщение об ошибке;
- могут быть найдены и использованы имена, объявленные после определения шаблона. Часто, но не всегда это является неожиданностью для пользователя, читающего определение шаблона. Например, естественно ожидать, что вызов `db(i)` будет разрешен в пользу объявленной выше функции `db(double)`, но если фрагмент `...` содержит объявление `db(int)`, то в соответствии с общими правилами перегрузки используется именно `db(int)`. С другой стороны, если в файле `complex.h` определена функция `db(complex)`, мы бы хотели, чтобы вызов `db(t)` был разрешен в пользу `db(complex)`, а не признан ошибкой из-за того, что он является некорректным вызовом функции `db(double)`, видимой в определении шаблона;
- множество имен, видимых в точке инстанцирования, различается, когда `sum` используется в двух разных компиляциях. Если `sum(vector<complex>&)`

при этом получает два разных определения, то результирующая программа незаконна с точки зрения правила одного определения (см. раздел 2.5). Однако в таких случаях проверить выполнение правила одного определения не под силу традиционному компилятору C++.

Ко всему прочему в оригинальном правиле не учтен случай, когда определение шаблона функции не находится в данной единице трансляции. Например:

```
template<class T> T sum(vector<T>& v);

// ...

#include <complex.h>
                                                    // # 2
void f(vector<complex>& v)
{
   complex c = sum(v);
}
```

Ни разработчики компилятора, ни пользователи не получали указаний на то, как искать определения шаблонов функции `sum()`. Поэтому в компиляторах применялись разные стратегии.

Общая проблема состоит в том, что в инстанцировании шаблона участвуют три контекста и четко разделить их нельзя:

- контекст определения шаблона;
- контекст объявления типа аргумента;
- контекст использования шаблона.

Основная цель проектирования шаблонов – гарантировать наличие достаточного объема информации при определении шаблона, чтобы его разрешалось инстанцировать по фактическим аргументам, не полагаясь на «случайные» обстоятельства, сложившиеся в точке вызова.

В первоначальном проекте делалась попытка обеспечить разумное поведение, полагаясь исключительно на правило одного определения (см. раздел 2.5). Считалось, что если случай и повлияет на определение сгенерированной функции, то обстоятельства вряд ли будут складываться одинаково при любом использовании шаблона функции. Предположение обоснованное, но компиляторы обычно не выполняют проверку на непротиворечивость (и тому есть веские причины). Так или иначе, грамотно написанные программы работают. Но желающие использовать шаблоны как макросы могут написать такую программу, которая будет извлекать нежелательные, на мой взгляд, преимущества из контекста вызова. Кроме того, у разработчика компилятора возникают серьезные трудности, когда он хочет синтезировать контекст для определения функции, чтобы ускорить инстанцирование.

Уточнить понятие точки инстанцирования так, чтобы оно было лучше исходного и одновременно сохраняло работоспособность написанных программ, было необходимо, хотя и непросто.

Сначала мы решили потребовать, чтобы каждое встречающееся в шаблоне имя определялось в точке определения шаблона. При этом шаблон стал бы легко читаться, появилась бы гарантия, что при инстанцировании не будет выбрано ничего случайного, а кроме того, обеспечивалось бы раннее обнаружение ошибок. Но это не позволило бы шаблону применять операции к объектам собственного класса. В примере выше `+`, `f()` и конструктор `T` не определены в точке определения шаблона. Объявление в шаблоне тоже недопустимо, так как нельзя задать их типы. Например, `+` может быть встроенным оператором, функцией-членом или глобальной функцией. Если это функция, она может принимать аргументы типа `T`, `const T&` и т.д. Это не что иное, как проблема задания ограничений на аргументы шаблона (см. раздел 15.4).

Принимая во внимание тот факт, что ни в точке определения шаблона, ни в точке его использования нет достаточного контекста для инстанцирования, мы должны найти компромиссное решение. Оно заключается в том, чтобы разделить встречающиеся в определении шаблона имена на две категории:

- зависящие от аргументов шаблона;
- все остальные.

Последние можно привязать в контексте определения шаблона, а первые – в контексте инстанцирования. Данная концепция не имеет изъянов, если только удастся четко определить, что обозначает фраза «зависит от аргументов шаблона».

15.10.2.1. Определение зависимости от T

Фраза «зависит от аргумента шаблона `T`», вероятно, может обозначать «является членом `T`». Если `T` – встроенный тип, то встроенные операторы будут считаться членами. К сожалению, этого недостаточно. Вот пример:

```
class complex {
   // ...
   friend complex operator+(complex,complex);
   complex(double);
};
```

Чтобы этот пример работал, в определение «зависимости от аргумента шаблона `T`» надо включить, по крайней мере, друзей `T`. Но и этого мало, поскольку ответственные функции, не являющиеся членами, не обязательно должны быть дружественными:

```
class complex {
   // ...
   complex operator+=(complex);
   complex(double);
};

complex operator+(complex a, complex b)
{
   complex r = a;
   return r+=b;
}
```

Было бы неразумно требовать, чтобы проектировщик класса предоставлял все функции, которые в будущем, возможно, пригодятся автору шаблона. Предвидеть все невозможно.

Поэтому понятие «зависит от аргумента шаблона `T`» должно опираться на контекст в точке инстанцирования, по крайней мере, в такой степени, чтобы уметь находить глобальные функции, используемые вместе с `T`. При этом неизбежно появляется возможность случайно подобрать какую-нибудь лишнюю функцию. Но эта проблема не слишком серьезна. Мы определим «зависимость от аргумента шаблона `T`» наиболее стандартным способом. Именно так, вызов функции зависит от аргумента шаблона, если был разрешен по-другому или вообще не разрешен в случае отсутствия в программе фактического типа шаблона. Для компилятора проверить такое условие относительно просто. Вот примеры вызовов, зависящих от аргумента шаблона `T`:

- у вызванной функции есть формальный параметр, зависящий от `T` в соответствии с правилами выведения типа (см. раздел 15.6.1). Например, `f(T)`, `f(vector<T>)`, `f(const T*)`;
- тип фактического аргумента зависит от `T` в соответствии с правилами выведения типа (см. раздел 15.6.1). Так, `f(T(1))`, `f(t)`, `f(g(t))` и `f(&t)`, если предположить, что `t` – это `T`;
- вызов разрешается использованием преобразования к типу `T`, хотя ни фактический аргумент, ни формальный параметр вызываемой функции не принадлежат типу, зависящему от `T`, так, как в первом и втором случаях.

Последний пример взят из реальной программы, и зависящий от этого правила код был вполне удачен. Вызов `f(1)`, на первый взгляд, не зависит от `T`, как не зависит от `T` и функция `f(B)`, к которой идет обращение. Но тип `T` аргумента шаблона имел конструктор из типа `int` и являлся производным от `B`, поэтому `f(1)` разрешалось как `f(B(T(1)))`.

15.10.2.2. Неоднозначности

Что надо было бы сделать, если в точке `#1` (точка определения шаблона в примере из раздела 15.10.2) и в точке `#2` (точка использования) обнаруживаются различные функции? Мы могли бы:

- отдать предпочтение `#1`;
- предпочесть `#2`;
- выдать ошибку.

Отметим, что в точке `#1` можно искать только не-функции и функции, для которых типы аргументов уже известны в точке использования в определении шаблона. Поиск же остальных имен откладывается до точки `#2`.

Первоначальное правило требует предпочесть точку `#2`, откуда следует, что применимы обычные требования разрешения неоднозначности. Ведь только в случае, когда в точке `#2` найдено лучшее соответствие, могут возникнуть расхождения с тем, что найдено в точке `#1`. К сожалению, при этом приходится не верить собственным глазам, читая определение шаблона. Например:

```
double sqrt(double);

template<class T> void f(T t)
{
   double sq2 = sqrt(2);
   // ...
}
```

Кажется очевидным, что `sqrt(2)` вызовет `sqrt(double)`. Но в точке `#2` вполне может обнаружиться функция `sqrt(int)`. В большинстве случаев это неважно, так как правило «должно зависеть от аргумента шаблона» гарантирует использование именно «очевидного» разрешения в пользу `sqrt(double)`. Однако если бы `T` был равен `int`, то вызов `sqrt(2)` зависел бы от аргумента шаблона, так что вызов разрешился бы в пользу `sqrt(int)`. Это неустранимое следствие того, что мы принимаем во внимание точку `#2`, но, по-моему, оно вызывает много путаницы. Хотелось бы как-то решить эту проблему.

С другой стороны, я считал необходимым отдавать предпочтение именно точке `#2`, ибо только тогда можно разрешить использование членов базового класса таким же способом, как при работе с обычными (нешаблонными) классами. Например:

```
void g();

template<class T> class X : public T {
   void f() { g(); }
   // ...
};
```

Если в `T` есть функция-член `g()`, следовало бы вызывать именно эту `g()`, поскольку так ведут себя нешаблонные классы:

```
void g():

class T { public: void g(); };

class Y : public T {
   void f() { g(); }    // вызывается T::g
   // ...
};
```

С другой стороны, в самых типичных случаях то, что найдено в точке `#1`, обычно корректно. Так работает в C++ поиск глобальных имен, именно такая модель позволяет на ранних стадиях обнаруживать большую часть ошибок, предварительно компилировать большинство шаблонов и именно такой механизм защищает от «случайного» заимствования имен в контексте, неизвестном автору шаблона. Несколько разработчиков компиляторов, особенно Билл Гиббонс, убедительно доказывали, что предпочтение следует отдать точке `#1`.

Какое-то время я склонялся к тому, чтобы считать ошибкой нахождение разных функций в двух данных точках, но это лишь усложняет задачу разработчиков

компиляторов, не давая ощутимых выгод пользователям. Кроме того, оказалась бы возможной ситуация, когда употребление определенных имен в контексте использования шаблона портит его удачный код, написанный программистом, думающим, что будут использоваться имена из области действия в точке определения шаблона. В конце концов нашелся довод, который окончательно склонил меня отдавать предпочтение тому, что было найдено в точке #1. Он состоял в следующем: некоторый весьма запутанный пример мог быть тривиально разрешен автором шаблона. Сравните:

```
double sqrt(double);

template<class T> void f(T t)
{
   // ...
   sqrt(2);      // разрешается в точке #1
   sqrt(T(2));   // очевидно зависит от T
                 // привязка в точке #2
   // ...
}
```

и

```
int g();

template<class T> class X : public T {
   void f()
   {
      g();       // разрешается в точке #1
      T::g();    // очевидно зависит от T
                 // привязка в точке #2
   }
   // ...
};
```

От автора шаблона требуется более явно выражать свое намерение, когда он хочет использовать некоторую функцию, не видимую в определении шаблона. Похоже, мы достигаем разумного поведения по умолчанию.

15.10.3. Специализация

Шаблон описывает, как определяется функция или класс при любых значениях его аргументов. Например, шаблон

```
template<class T> class Comparable {
   // ...
   int operator==(const T& a, const T& b) { return a==b; }
};
```

означает, что для каждого типа T элементы сравниваются с помощью оператора ==. К сожалению, это слишком ограничительное условие. В частности, в C строки, представленные типом char*, обычно сравниваются функцией strcmp().

Во время первоначального проектирования мы нашли множество таких примеров, а также обнаружили, что «особые случаи» зачастую чрезвычайно важны. Благодаря им удается придать единообразие языку и повысить производительность. Строки в стиле C дают прекрасный пример.

Поэтому я пришел к выводу, что нужен механизм для специализации шаблонов. Это можно было сделать, либо приняв общие правила перегрузки, либо с помощью специального подхода. Я выбрал последнее, так как думал, что решаю в основном проблему нерегулярности, берущую свое начало в C, а также потому, что предложение о перегрузке неизменно встречает массу протестов. В первоначальном проекте специализация была определена как ограниченная и аномальная форма перегрузки и плохо увязывалась с остальными частями языка.

Шаблон класса или функции можно специализировать. Например, если есть шаблон

```
template<class T> class vector {
   // ...
   T& operator[](int i);
};
```

то можно ввести специализации, то есть отдельные объявления, скажем, для `vector<char>` или `vector<complex>::operator[](int)`:

```
class vector<char> {
   // ...
   char& operator[](int i);
};

complex& vector<complex>::operator[](int i) { /* ... */ }
```

Это позволяет программисту вводить специализированные реализации для классов, которые либо особенно важны с точки зрения производительности, либо имеют отличающуюся от стандартной семантику. Грубый, но очень эффективный механизм.

Исходная моя идея состояла в том, чтобы поместить такие специализации в библиотеку и автоматически вызывать их по мере необходимости без вмешательства программиста. Оказалось, что цена такой услуги велика, а ценность сомнительна. Специализация приводила к трудностям для понимания и реализации, так как заранее было неизвестно, что будет подставлено для конкретного набора аргументов шаблона, – даже если у нас перед глазами имелось его определение, – ибо этот шаблон мог быть специализирован в другой единице трансляции. Например:

```
template<class T> class X {
   T v;
public:
   T read() const { return v; }
   void write(int vv) { v = vv; }
};
```

```
void f(X<int> r)
{
   r.write(2);
   int i = r.read();
}
```

Кажется естественным предположить, что `f()` использует определенную выше функцию-член. Но это не гарантируется. В какой-то другой единице трансляции могла бы быть определена функция `X<int>::write()`, которая делает нечто совершенно иное.

Специализацию можно считать «дырой» в системе защиты C++, так как специализированная функция-член может получать доступ к закрытым данным шаблона класса таким способом, который невозможно идентифицировать, просто читая определение шаблона. Были и другие технические проблемы.

Я счел, что специализация в своем первоначальном виде реализована неудачно, но при этом предоставляет существенную функциональность. Как же можно сохранить эту функциональность, устранив все недостатки? После многих сложных рассуждений я предложил очень простое решение, которое было одобрено на заседании в Сан-Хосе: специализация должна быть объявлена перед использованием. Это ставит ее в один ряд с правилами обычной перегрузки. Если в области действия, где специализация используется, не видно ее объявления, то применяется обычное определение шаблона. Например:

```
template<class T> void sort(vector<T>& v)
   { /* ... */ }

void sort<char*>(vector<char*>& v); // специализация

void f(vector<char*>& v1, vector<String>& v2)
{
   sort(v1); // используется специализация
             // sort(vector<char*>&)

   sort(v2); // используется общий шаблон
             // sort(vector<T>&), где T - это String
}

void sort<String>(vector<String>& v);     // ошибка: специализация
                                          // после использования

void sort<>(vector<double>& v);           // правильно: sort(double)
                                          // еще не использовалась
```

Мы думали о явном ключевом слове для обозначения специализации. Например:

```
specialise void sort(vector<String>&);
```

но комитет на заседании в Сан-Хосе был настроен решительно против новых ключевых слов. К тому же на этом собрании, где присутствовали люди разных

национальностей, мы никогда не смогли бы прийти к единому мнению о том, как надо правильно писать: `specialise` или `specialize`.

15.10.4. Нахождение определений шаблонов

Традиционно программа на C++, как и на C, представляет собой множество файлов, которые собираются в единицы трансляции, компилируются и связываются с помощью набора различных программ, работающих на основе общих соглашений. Например, файлы с расширением `.c` – это исходные тексты; они включают `.h`-файлы для получения информации о других частях программы. Из `.c`-файлов компилятор генерирует объектные файлы, обычно имеющие расширение `.o`. Исполняемая программа получается путем связывания всех `.o`-файлов. Архивы и динамические подключаемые библиотеки несколько усложняют дело, но не изменяют картины в целом.

Шаблоны не очень хорошо укладываются в описанную схему. Отсюда множество проблем, связанных с их реализацией. Шаблон – не просто исходный код (скорее, можно назвать исходным кодом то, что получается в результате инстанцирования шаблона), поэтому определениям шаблонов не место в `.c`-файлах. С другой стороны, шаблоны не являются и просто типами или определениями интерфейсов, так что в `.h`-файлы их тоже помещать не стоит.

На этот счет в ARM не было однозначных указаний разработчикам (см. раздел 15.10), в результате появилось множество разных схем, которые препятствовали переносимости. Для некоторых компиляторов требовалось, чтобы шаблоны находились в `.h`-файлах. Это может негативно отразиться на производительности, поскольку в каждую единицу трансляции входит слишком много информации и каждая единица становится зависимой от всех шаблонов, входящих в ее `.h`-файлы. Вообще-то шаблоны не являются частью заголовочных файлов. Другие компиляторы требуют, чтобы шаблоны находились в `.c`-файлах. Это усложняет нахождение определения шаблона функции, когда его надо инстанцировать, а также синтез контекста для инстанцирования.

Видимо, любое решение указанных проблем должно основываться на признании того факта, что C++-программа – не просто (и не только) набор не связанных между собой единиц трансляции. Это верно даже на этапе компиляции. Каким-то образом нужно сформулировать концепцию центральной точки, в которой доступна информация о шаблонах и других элементах, относящихся сразу к нескольким единицам трансляции. Пока назовем эту точку репозитарием, поскольку ее основное назначение – хранить информацию, необходимую компилятору между трансляциями отдельных частей программы.

Можно представлять себе репозитарий как хранящуюся на диске таблицу символов, в которой для каждого шаблона есть один элемент. Компилятор использует ее для получения информации об объявлениях, определениях, специализациях, использовании и т.д. С учетом такой концепции можно охарактеризовать модель инстанцирования следующим образом: поддерживает все языковые средства, согласуется с применяемыми принципами использования `.h` и `.c`-файлов, не требует от пользователя знаний о репозитарии и предоставляет возможности для проверки ошибок, оптимизации и повышения эффективности компиляции

и компоновки, о которых просят разработчики компиляторов. Заметим, что это модель системы инстанцирования, а не правило языка и не характеристика конкретной реализации. Возможно несколько альтернативных реализаций, но я полагаю, что пользователь может игнорировать детали (почти всегда) и представлять себе систему так, как описано выше.

Рассмотрим предполагаемую работу компилятора в различных ситуациях. Как обычно, на вход компилятора подаются `.c`-файлы. Они содержат директивы `#include` для включения `.h`-файлов. Компилятору известно только о переданном ему коде. То есть в файловой системе никогда не ищется определение шаблона, которое не дано. Однако компилятор использует репозитарий, как бы запоминая, какие шаблоны уже известны и откуда они взялись. Эту схему легко расширить с учетом архивов. Вот краткое описание того, как происходит работа компилятора в некоторых ключевых точках:

- распознано объявление шаблона. Теперь шаблон можно использовать. Он помещается в репозитарий;
- в `.c`-файле распознано определение шаблона функции. Шаблон обрабатывается с целью помещения в репозитарий. Если он уже находится там, появляется ошибка повторного определения за исключением случаев, когда это новая версия того же шаблона;
- в `.h`-файле распознано определение шаблона функции. Шаблон обрабатывается с целью помещения в репозитарий. Если он уже там находится, то осуществляется проверка, был ли на самом деле ранее помещенный шаблон добавлен именно из этого заголовочного файла. Если это не так, диагностируем ошибку повторного определения. Проверяем, не нарушено ли правило одного определения. Для этого надо убедиться, что старое определение совпадает с новым. В противном случае диагностируем ошибку повторного определения, если только это не новая версия того же шаблона;
- распознано объявление специализации шаблона функции. Если необходимо, выдаем ошибку об использовании до объявления. Теперь специализацию можно использовать. Помещаем объявление в репозитарий;
- распознано определение специализации шаблона функции. При необходимости выдаем ошибку об использовании до объявления. Теперь специализацию можно использовать. Помещаем определение в репозитарий;
- распознано использование. Заносим в репозитарий запись об использовании шаблона с данным набором аргументов. Смотрим, есть ли в репозитарии определение общего шаблона или его специализации. Если да, то допустимо выполнить контроль ошибок или оптимизацию. Если шаблон еще не использовался с таким набором аргументов, то допустимо сгенерировать код сейчас или отложить это до этапа компоновки;
- распознан запрос явного инстанцирования. Проверяем, определен ли шаблон. Если нет, выдаем сообщение о неопределенном шаблоне. Проверяем, определена ли специализация. Если да, выдаем сообщение «инстанцирован и специализирован». Проверяем, был ли уже инстанцирован шаблон с данным набором аргументов. Если да, можно либо выдать сообщение о повторном

инстанцировании, либо игнорировать запрос. В противном случае допустимо сгенерировать код сейчас или отложить это до этапа компоновки. В любом случае код генерируется для каждой функции-члена шаблона класса, которую видит компилятор;

- программа компонуется. Сгенерировать код для каждого использованного шаблона, код которого не сгенерирован ранее. Повторить этот процесс, пока не будут обработаны все инстанцирования. Выдать сообщение «использована, но не определена» для всех отсутствующих шаблонов функций.

При генерации кода шаблона для данного набора аргументов используется алгоритм поиска, рассмотренный в разделе 15.10.2. Разумеется, должны выполняться проверки некорректного использования, недопустимой перегрузки и т.д.

В компилятор могут быть заложены более или менее жесткие требования к исполнениям правила одного определения и правила, запрещающего многократное инстанцирование. Такая диагностика не является обязательной, поэтому поведение в этих случаях следует считать лишь вопросом качества реализации.

15.11. Последствия введения шаблонов

Отсутствие шаблонов в раннем C++ имело негативные последствия для использования языка. Теперь, когда шаблоны получили широкое распространение, какие задачи можно решать лучше?

Из-за отсутствия шаблонов в C++ не было способа реализовать контейнерные классы, не прибегая к интенсивному использованию приведения типов и манипуляции объектами через указатели на общие базовые классы или `void*`. Теперь от всего этого можно отказаться. Но неправильное использование наследования, берущее начало в бездумном переносе в C++ методов из Smalltalk (см., например, пункт 14.2.3) и злоупотребление слабой типизацией, заимствованной из C, выкорчевать будет очень трудно.

С другой стороны, я ожидаю, что потихоньку удастся избавиться от небезопасной практики работы с массивами. В стандартной библиотеке ANSI/ISO есть шаблон класса динамического массива `dynarray` (см. раздел 8.5.), так что можно будет пользоваться либо им, либо каким-то самодельным шаблоном и не прибегать к массивам без контроля выхода за границы. Часто раздаются критические замечания, что в C и C++ не контролируются индексы массива. В большинстве случаев это недовольство беспочвенно, поскольку люди забывают, что, хотя возможность ошибиться, выйдя за пределы массива, и есть, делать такую ошибку вовсе не обязательно. Шаблоны массивов переносят детали работы с низкоуровневыми массивами в глубь реализации, где им и надлежит находиться. По мере того как массивы в стиле C станут применяться все реже из-за того, что работа с ними будет перенесена в классы и шаблоны, число связанных с массивами ошибок резко уменьшится. Это и так постепенно происходит, а наличие шаблонов, особенно входящих в состав библиотек, ускорит процесс.

Третий важный аспект работы с шаблонами состоит в том, что в сочетании с наследованием они открывают совершенно новые возможности для проектирования библиотек (см. раздел 15.8).

Хотя компиляторов, поддерживающих шаблоны, и немало, но все же они пока не стали доступными повсеместно. Да и те, что есть, еще недостаточно совершенны.

15.11.1. Отделение реализации от интерфейса

Механизм шаблонов задействован исключительно на этапах компиляции и компоновки. Он не нуждается ни в какой поддержке во время исполнения. Конечно, это сознательное решение, но остается одна проблема: как классы и функции, сгенерированные (инстанцированные) из шаблонов, могут зависеть от информации, известной только на этапе выполнения. Обычный для C++ ответ – пользуйтесь виртуальными функциями.

Многие выражали озабоченность тем, что шаблоны слишком сильно зависят от наличия исходного кода. В этом видели два нежелательных побочных эффекта:

- нельзя засекретить коммерческую программу;
- если реализация шаблона изменяется, пользователь должен перекомпилировать свою программу.

Разумеется, так обстоит дело лишь в самых примитивных реализациях, но и в этом случае шаблон класса, который является производным от классов, предоставляющих ясный интерфейс, позволяет уменьшить неприятные последствия. Часто шаблон содержит просто интерфейсный код к чему-то «секретному», что можно изменять, не затрагивая интерфейс. Примеры такой техники – шаблон `pvector` из раздела 15.5, шаблонная версия класса `set` из раздела 13.2.2. Для решения проблем можно воспользоваться концепцией виртуальных функций, так что мне не нужно предоставлять еще один вариант таблицы переходов.

Можно также придумать форму хранения шаблонов в частично откомпилированном виде, тогда секреты производителя останутся в такой же мере безопасными, что и обычный объектный код.

Для некоторых проблема состоит в том, как гарантировать, что пользователю не удастся – прямо или косвенно – инстанцировать новые версии шаблонов, которые по идее должны быть секретными. Но для этого достаточно просто не предоставлять исходный код. Данный подход реализуем, если производитель может заранее инстанцировать (см. раздел 15.10.1) все необходимые версии. Тогда эти (и только эти) версии можно поставлять в виде библиотек объектного кода.

15.11.2. Гибкость и эффективность

Поскольку шаблоны призваны напрямую конкурировать с макросами, требования к их гибкости и эффективности очень высоки. Теперь уже можно сказать, что получившийся механизм обладает нужными свойствами в высшей степени и не жертвует при этом статическим контролем типов. Когда дело доходит до выражения алгоритмов, мне иногда хочется иметь средства более высокого порядка, но никогда – контроль типов во время исполнения. Думается, что предлагаемые «улучшения» шаблонов за счет введения ограничений отрицательно сказались бы на их полезности, не увеличив безопасности, простоты или эффективности. Процитирую Алекса Степанова, который подвел итог опыту написания большой библиотеки структур данных и алгоритмов:

«C++ – довольно мощный язык. Это первый язык в нашей практике, позволивший конструировать обобщенные программные компоненты, обладающий математической точностью, изяществом и абстрактностью. При этом получающиеся компоненты не уступают в эффективности необобщенному коду».

Еще только предстоит раскрыть всю мощь сочетания шаблонов, абстрактных классов, обработки исключений и т.д. Я не думаю, что десятикратная разница в размере реализации компонент Буча [Booch, 1993b] на C++ и на Ada – это нехарактерный пример (см. раздел 8.4.1).

15.11.3. Влияние на другие компоненты C++

Аргумент шаблона может иметь встроенный или определенный пользователем тип. Это заставляло постоянно думать о том, чтобы встроенные и пользовательские типы выглядели и вели себя по возможности одинаково. К сожалению, добиться полного тождества между ними нельзя. Причина – оставаясь в рамках совместимости с C, невозможно исключить нерегулярность встроенных типов. Но в некоторых отношениях встроенные типы все же выиграли от результатов работы над шаблонами.

Когда я только еще думал о шаблонах и позже, когда начал их применять, то обнаружил несколько случаев, в которых встроенные типы трактовались несколько иначе, чем классы. Это стало препятствием к написанию шаблонов, которые можно было бы использовать с аргументами как встроенных, так и пользовательских типов. Поэтому была поставлена цель обеспечить одинаковое поведение всех типов в семантическом и синтаксическом отношениях. Эта работа не окончена и по сей день.

Рассмотрим пример:

```
vector v(10);// вектор из 10 элементов
```

Раньше такой синтаксис инициализации был неприменим к встроенным типам. Чтобы разрешить его, для встроенных типов введены понятия конструктора и деструктора, например:

```
int a(1); // до версии 2.1 ошибка, теперь инициализирует a единицей
```

Я подумывал о том, чтобы пойти дальше и разрешить наследование от встроенных типов и явное объявление для них встроенных операторов. Но сдержался.

Если разрешить наследование от `int`, программист на C++ фактически не получит ничего принципиально нового по сравнению с членом типа `int`. Ведь в классе `int` нет виртуальных функций, которые производный класс мог бы заместить. С другой стороны, правила преобразования в C настолько хаотичны, что никак не удается представить, будто `int`, `short` и т.д. – обычные классы с предсказуемым поведением. Они либо сохраняют совместимость с C, либо подчиняются четким правилам C++ для классов, но не то и другое одновременно.

Разрешить переопределять встроенные операторы – `operator+(int,int)`, например, – значило заодно сделать язык подверженным мутациям. Однако идея разрешить синтез таких функций, чтобы можно было передавать указатели на них или ссылаться как-то иначе, довольно привлекательна.

Концептуально встроенные типы имеют конструкторы и деструкторы. Например:

```
template<class T> class X {
   T a;
   int i;
   complex c;
public:
   X() : a(T()), i(int()), c(complex()) { }
   // ...
};
```

Конструктор `X` инициализирует каждый член, вызывая для него конструктор по умолчанию. Умалчиваемый конструктор любого типа `T` по определению дает то же значение, какое имела бы глобальная переменная типа `T`, не инициализированная явно. Это уточнение ARM, где считалось, что `X()` возвращает неопределенное значение, если только для `X` не был определен конструктор по умолчанию.

Глава 16. Обработка исключений

Не паникуй!

Руководство для путешествующих по Галактике автостопом

16.1. Введение

В первоначальном проекте C++ исключения рассматривались, но были отложены из-за нехватки времени на внимательное изучение вопросов проектирования и реализации, а также из опасения, что они могут сильно усложнить компилятор (см. раздел 3.15). Было понятно, что плохой проект может вызвать большие издержки во время выполнения и намного увеличить время, необходимое для переноса на другую платформу. Исключения считались важными для обработки ошибок в программах, составленных из независимо спроектированных библиотек.

Проектирование механизма исключений для C++ растянулось на несколько лет (с 1984 по 1989 гг.). Это была первая часть C++, которая проектировалась на глазах заинтересованной публики. Помимо бесчисленных обсуждений, через которые проходило любое включенное в C++ средство, несколько вариантов были изложены в статьях и подверглись широкому обсуждению. На последних этапах активное участие в работе принимал Эндрю Кениг, вместе которым мы написали статьи [Koenig, 1989a] и [Koenig, 1990]. Существенные фрагменты окончательной схемы были разработаны по пути на конференцию USENIX по C++, которая состоялась в ноябре 1987 г. в Санта-Фе. Промежуточные варианты проекта излагались также на встречах в компаниях Apple, DEC (Спринг Брук), Microsoft, IBM (Альмадена), Sun и в других местах. Я нашел людей, имевших реальный опыт работы с системами, поддерживающими обработку исключений. Первое серьезное обсуждение обработки исключений в C++ состоялось в Оксфорде летом 1983 г. Основные темы беседы с Тони Вильямсом (Tony Williams) из Лаборатории Резерфорда – проектирование отказоустойчивых систем и ценность статического контроля типов в механизме обработки исключений.

К моменту начала дебатов по поводу обработки исключений в комитете ANSI C++ опыт их использования в языке ограничивался библиотечными реализациями, предпринятыми компанией Apple, Майком Миллером [Miller,1988] и некоторыми другими. Единственную реализацию на уровне компилятора выполнил Майк Тиман [Tiemann, 1990]. Это вызывало беспокойство, хотя в общем все были согласны, что в какой-то форме обработка исключений очень пригодилась бы в C++. В частности, Дмитрий Ленков, основываясь на опыте компании Hewlett-Packard, выразил горячее желание иметь такой механизм. Только Дуг Макилрой считал, что наличие обработки исключений понизит надежность систем, поскольку

авторы библиотек и другие программисты станут просто возбуждать исключения, не пытаясь понять, в чем проблема и как устранить ее. Время покажет, в какой мере оправдалось пророчество Дуга. Естественно, что никакое языковое средство не может запретить программисту писать плохие программы.

Первые реализации обработки исключений в том виде, который описан в ARM, начали появляться весной 1992 г.

16.2. Цели и предположения

При проектировании были сделаны предположения о том, что:

- исключения используются преимущественно для обработки ошибок;
- обработчики исключений встречаются реже, чем определения функций;
- по сравнению с вызовами функций исключения возникают редко;
- исключения – это понятие уровня языка, а не компилятора, и не стратегия обработки ошибок в узком смысле.

Данные формулировки, равно как и изложенный ниже перечень желаемых возможностей, взяты со слайдов, на которых демонстрировалась эволюция проектирования с 1988 г.

Обработка исключений:

- задумывалась не как простая альтернатива механизму возврата – так предлагали некоторые, в особенности Дэвид Черитон (David Cheriton), – а как специальный механизм для поддержки построения отказоустойчивых систем;
- предназначается не для того, чтобы превратить каждую функцию в отказоустойчивую единицу. Задумана как механизм, с помощью которого подсистема может выдерживать отказы даже тогда, когда составляющие ее функции написаны без соблюдения единой стратегии обработки ошибок;
- не должна навязывать проектировщику одно единственное «правильное» представление о том, как следует обрабатывать ошибки. Цель обработки исключений – сделать язык более выразительным.

На протяжении всей работы по проектированию возрастало влияние проектировщиков разнообразных систем и уменьшалось число предложений от сообщества пользователей языка. За прошедшее с тех пор время наибольшее воздействие на проектирование обработки исключений в C++ оказала работа по отказоустойчивым системам, начатая в университете Ньюкасла в Англии Брайаном Рэнделлом и его коллегами и продолженная в других местах.

В процессе проектирования обработки исключений были выработаны приведенные ниже критерии.

1. Передача произвольного объема информации из точки, где возбуждено исключение, обработчику с сохранением данных о типе.
2. Отсутствие дополнительных издержек по скорости и по памяти в коде, не возбуждающем исключения.
3. Гарантии того, что любое возбужденное исключение будет перехвачено каким-то обработчиком.

4. Возможность группировать исключения так, чтобы можно было написать обработчик, перехватывающий не только одно, но и сразу несколько исключений.
5. Механизм, который по умолчанию будет правильно работать в многопоточной среде.
6. Механизм, допускающий взаимодействие с другими языками и особенно с C.
7. Простота использования.
8. Простота реализации.

Большая часть этих критериев была воплощена в жизнь, но третий и восьмой принципы всегда оказывались либо требующими больших затрат, либо слишком ограничительными, поэтому в результате мы лишь приблизились к их реализации. Я считаю это неплохим результатом, принимая во внимание то, что обработка исключений – трудная задача, для решения которой программисту понадобится вся помощь, которую только можно получить. Чрезмерно ревностный проектировщик языка мог бы включить средства или ограничения, которые лишь усложнили бы проектирование и реализацию отказоустойчивой системы.

Думается, что отказоустойчивая система должна быть многоуровневой. Отдельная часть системы не может восстановиться после любого мыслимого сбоя и некорректных воздействий извне. Возможны ведь и крайние случаи: отказ питания или непроизвольное изменение содержимого ячейки памяти.

В какой-то момент отдельный модуль системы может отказать, и тогда ситуация будет решаться на более высоком уровне этой системы. Например, вызванная функция может сообщить о катастрофической ошибке вызывающей; процесс может завершиться аномально и «поручить» другому процессу разобраться с последствиями; процессор может «обратиться за помощью» к другому процессору. Компьютер может затребовать помощь у оператора-человека. Подчеркнем в связи с этим: обработку ошибок следует проектировать так, чтобы у относительно простой программы, пользующейся сравнительно несложными средствами обработки исключений, был шанс выполнить свою задачу.

Попытка дать такие средства, которые позволили бы одной монолитной программе восстановиться после любых ошибок, – неверный шаг, который ведет к стратегиям обработки настолько сложным, что они сами становятся источником ошибок.

16.3. Синтаксис

Как обычно, синтаксису было уделено достаточное внимание, и в результате я остановился на не слишком лаконичном варианте, в котором было три ключевых слова и обилие скобок:

```
int f()
{
   try {      // начало try-блока
      return g();
   }
   catch (xxii) {    // начало обработчика исключения
```

```
            // сюда попадаем только тогда, когда случилось 'xxii'
        error("ошибка в g(): xxii");
        return 22;
    }
}

int g()
{
    // ...
    if (что_то_случилось) throw xxii();     // возбудить исключение
}
```

Ключевое слово `try`, очевидно, избыточно, равно как и скобки `{}`, если только `try`-блок или обработчик не состоят из нескольких предложений. Например, совсем несложно было бы разрешить такой синтаксис:

```
int f()
{
    return g() catch (xxii) {   // не C++
        error("ошибка в g(): xxii");
        return 22;
    };
}
```

Однако его так трудно объяснить, что я решил пойти на избыточность, дабы избавить персонал службы технической поддержки от вопросов запутавшихся пользователей. Из-за традиционной нелюбви пользователей C к новым ключевым словам я всячески пытался уйти от этого, но все схемы, которые приходили на ум, оказывались слишком хитроумными или сбивающими с толку. Так, я пробовал использовать одно слово `catch` и для возбуждения и для перехвата исключения. Это можно было бы сделать логически непротиворечиво, но объяснить такую схему я бы не взялся.

Слово `throw` было выбрано отчасти из-за того, что более очевидные слова `raise` и `signal` уже были заняты под функции из стандартной библиотеки C.

16.4. Группировка

Поговорив с десятком пользователей десятка разных систем, поддерживающих в том или ином виде обработку исключений, я пришел к выводу, что необходимо уметь группировать исключения. Например, пользователь должен иметь возможность перехватить любое исключение библиотеки ввода/вывода, даже не зная точно, какими они бывают. Есть, конечно, и обходные пути в ситуации, когда механизм группировки отсутствует. Например, тип исключения можно закодировать в виде данных, передаваемых вместе с единственным исключением. Допустимо просто перечислять исключения, логически входящие в одну группу, всякий раз, когда нужно обработать их все. Однако любая такая уловка расценивалась бы если и не всеми, то большинством людей как осложняющая сопровождения.

Мы с Эндрю Кенигом попробовали было схему, основанную на том, что группы создавались динамически с помощью конструкторов для объектов исключений.

Однако это выпадало из стиля, принятого в других частях C++, и многие, в том числе Тед Голдстейн и Питер Дойч (Peter Deutsch), отмечали, что такие группы по сути эквивалентны иерархии классов. Поэтому мы остановились на схеме, навеянной языком ML: возбужденное исключение-объект перехватывается обработчиком, в объявлении которого говорится, что он может принимать объекты такого типа. При этом обычные правила инициализации C++ позволяют обработчику объектов типа `B` перехватывать объекты любого класса `D`, производного от `B`. Например:

```
class Matherr { };
class Overflow : public Matherr { };
class Underflow : public Matherr { };
class Zerodivide : public Matherr { };
// ...

void g()
{
   try {
      f();
   }
   catch (Overflow) {
      // обработать исключение типа Overflow или любого производного
      // от него
   }
   catch (Matherr) {
      // обработать любое исключение типа Matherr, кроме Overflow
   }
}
```

Позже обнаружилось, что множественное наследование (см. главу 12) дает очень красивое решение трудных задач классификации. Например, можно было объявить ошибку при работе с файлом на сетевом диске так:

```
class network_file_err
   : public network_err , public file_system_err { };
```

Исключение типа `network_file_err` может обработать как обработчик сетевых ошибок, так и ошибок в файловой системе. Первым на это мне указал Дэниэль Уэйнреб (Daniel Weinreb).

16.5. Управление ресурсами

Главным в проекте обработки исключений был вопрос об управлении ресурсами. Скажем, если функция захватывает некоторый ресурс, может ли язык выполнить часть работы по освобождению ресурса при выходе, даже если произошло исключение? Рассмотрим пример, взятый из [2nd]:

```
void use_file(const char* fn)
{
   FILE* f = fopen(fn,"w");    // открыть файл с именем fn
```

```
    // использовать f

    fclose(f);// закрыть файл fn
}
```

Выглядит разумно. Но если между вызовами `fopen()` и `fclose()` произойдет ошибка, из-за возникшего исключения выход из функции `use_file` может быть выполнен без вызова `fclose()`. То же самое может случиться и в языках, не поддерживающих исключения. Например, к таким же печальным последствиям приведет вызов функции `longjmp()` из стандартной библиотеки C. Если мы хотим разрабатывать отказоустойчивые системы, эту проблему придется снимать. Примитивное решение выглядит так:

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn,"r");    // открыть файл с именем fn
    try {
        // использовать f
    }
    catch (...) { // перехватить все исключения
        fclose(f);// закрыть файл fn
        throw; // возбудить повторно
    }
    fclose(f);// закрыть файл fn
}
```

Весь код для работы с файлом заключен в блок `try`, который перехватывает любое исключение, закрывает файл и возбуждает то же исключение повторно.

К сожалению, данное решение многословно и потенциально имеет большие затраты. К тому же необходимость писать длинные повторяющиеся фрагменты утомляет программиста и он может наделать ошибок. Можно было слегка сократить код, предоставив в распоряжение программиста некоторый механизм финальной обработки. Это позволило бы уйти от дублирования кода освобождения ресурса (в данном случае `fclose(f)`), но никак не решило бы принципиальную проблему: для написания отказоустойчивого кода требуется специальная и более сложная, чем обычно, техника.

Впрочем, есть и более изящное решение. В общем виде задача выглядит так:

```
void use()
{
    // захватить ресурс 1
    // ...
    // захватить ресурс n

    // использовать ресурсы

    // освободить ресурс n
    // ...
    // освободить ресурс 1
}
```

Как правило, важно, чтобы ресурсы освобождались в порядке, обратном их захвату. Это весьма напоминает поведение локальных объектов, которые создаются конструкторами и уничтожаются деструкторами. Стало быть, мы можем решить проблему захвата и освобождения ресурсов путем разумного использования объектов классов, имеющих конструкторы и деструкторы. Например, удалось бы определить класс `FilePtr`, ведущий себя как `FILE*`:

```
class FilePtr {
   FILE* p;
public:
   FilePtr(const char* n, const char* a) { p = fopen(n,a); }
   FilePtr(FILE* pp) { p = pp; }
   ~FilePtr() { fclose(p); }

   operator FILE*() { return p; }
};
```

Мы можем сконструировать `FilePtr`, зная либо `FILE*`, либо аргументы, которые обычно передаются `fopen()`. В любом случае объект `FilePtr` будет разрушен при выходе из области действия, а его деструктор закроет файл. Теперь наша программа сокращается до минимального размера:

```
void use_file(const char* fn)
{
   FilePtr f(fn,"r");   // открыть файл с именем fn
   // использовать f
}  // файл fn закрывается неявно
```

Деструктор вызывается независимо от того, как произошел выход из функции: обычно или в результате исключения.

Данный прием я называю «захват ресурса – это инициализация». Он может работать с частично сконструированными объектами и таким образом решает довольно трудную проблему: что делать, если ошибка произошла в конструкторе (см. [Koenig, 1990] или [2nd]).

16.5.1. Ошибки в конструкторах

Для некоторых пользователей важнейшим аспектом исключений является то, что они дают общий механизм извещения об ошибках, произошедших в конструкторе. Рассмотрим конструктор класса `FilePtr`: в нем не проверяется, открыт ли файл корректно или нет. Вот наиболее аккуратная запись:

```
FilePtr::FilePtr(const char* n, const char* a)
{
   if ((p = fopen(n,a)) == 0) {
       // файл не открылся - что делать?
   }
}
```

Без исключений нет простого способа сообщить об ошибке: конструктор не возвращает значения. Поэтому приходилось изобретать обходные пути, например переводить «наполовину сконструированные» объекты в ошибочное состояние,

оставлять флаги в глобальных переменных и т.д. Как ни странно, на практике это редко становилось проблемой. Однако исключения дают общее решение:

```
FilePtr::FilePtr(const char* n, const char* a)
{
   if ((p = fopen(n,a)) == 0) {
       // ой! файл не открылся
       throw Open_failed(n,a);
   }
}
```

Механизм же обработки исключений в C++ гарантирует, что частично сконструированные объекты корректно уничтожаются, то есть полностью сконструированные подобъекты разрушаются, а те подобъекты, которые остались несконструированными, – нет. Это позволяет автору конструктора сосредоточиться на обработке ошибки в том объекте, в котором она обнаружена. Подробнее см. [2nd, §9.4.1].

16.6. Возобновление или завершение?

При проектировании механизма обработки исключений наиболее спорным оказался вопрос, следует ли поддержать семантику возобновления или прекращения исполнения. Иными словами, может ли обработчик исключения потребовать возобновления исполнения с того момента, где было возбуждено исключение? Например, было бы хорошо, если бы подпрограмма, вызванная для обработки ошибки нехватки памяти, могла найти дополнительную память и вернуть управление? Или если бы подпрограмма, вызванная для обработки деления на нуль, вернула некоторое определенное пользователем значение? А если бы подпрограмма, вызванная потому, что в дисководе нет гибкого диска, могла попросить пользователя вставить диск, а затем продолжила выполнение?

Поначалу я думал так: «Почему бы и нет? Ведь сразу видно несколько ситуаций, в которых я мог бы воспользоваться возобновлением». Но в последующие четыре года моя точка зрения некоторым образом изменилась, и потому механизм обработки исключений в языке C++ поддерживает так называемую модель с завершением.

Главные споры по поводу возобновления и завершения разгорелись в комитете ANSI C++. Вопрос обсуждался и на заседаниях всего комитета, и в рабочей группе по расширениям, и на вечерних технических обсуждениях, и в списках для рассылки по электронной почте. Споры продолжались с декабря 1989 г., когда был сформирован комитет ANSI C++, до ноября 1990 г. Тема вызвала большой интерес и у всего сообщества пользователей C++. В комитете семантику возобновления отстаивали преимущественно Мартин О'Риордан и Майк Миллер. Эндрю Кениг, Майк Вило, Тед Голдстейн, Дэг Брюк, Дмитрий Ленков и я высказывались за семантику завершения. Являясь председателем рабочей группы по расширениям, я направлял дискуссию. Результат был таков: 22 человека против одного за семантику завершения. Достичь результата удалось после долгого заседания, на котором представители компаний DEC, Sun, Texas Instruments и IBM демонстрировали

экспериментальные данные. Затем последовало одобрение этого предложения на заседании всего комитета (4 человека из 30 против, остальные – за), в таком виде, то есть с семантикой завершения, оно и опубликовано в ARM.

После длительных дебатов на встрече в Сиэтле, состоявшейся в июле 1990 г., я свел воедино все доводы в пользу семантики возобновления:

- более общий механизм (более мощный, включает и семантику завершения);
- унифицирует сходные концепции и реализации;
- необходим для очень сложных динамичных систем (например, для OS/2);
- ненамного более сложен и расточителен при реализации;
- если механизма, включающего семантику возобновления нет, его приходится имитировать;
- дает простое решение проблемы истощения ресурсов.

Аналогично я обобщил доводы, высказанные в пользу семантики завершения:

- более простой, концептуально чистый и дешевый способ;
- позволяет строить системы, более легкие в управлении;
- обладает достаточной мощностью для решения любых задач;
- позволяет избежать неудачных или не очень законных программных приемов;
- противостоит негативному опыту, накопленному при работе с семантикой возобновления.

В обоих перечнях намного упрощена суть споров, основательных и изобилующих техническими деталями. Иногда полемика носила весьма эмоциональный характер. Менее сдержанные участники кричали, что сторонники семантики завершения хотят навязать произвольные ограничения на приемы программирования. Очевидно, что вопрос о возобновлении и завершении затрагивает глубинные принципы проектирования программного обеспечения. Число сторонников той и другой точек зрения никогда не было одинаковым, однако на любой встрече отстаивающих семантику завершения было примерно в четыре раза больше.

Среди аргументов в пользу возобновления чаще других повторялись такие:

- поскольку возобновление – более общий механизм, чем завершение, именно его и следует принять, даже если есть сомнения в его полезности;
- есть важные случаи, когда подпрограмма оказывается блокированной из-за отсутствия ресурса (например, памяти или гибкого диска в накопителе). В такой ситуации семантика возобновления позволила бы подпрограмме возбудить исключение, дождаться, пока обработчик найдет ресурс, а затем продолжить выполнение, как если бы ресурс был доступен в самом начале.

А вот как выглядели наиболее популярные и убедительные аргументы в пользу завершения:

- семантика завершения намного проще семантики возобновления. Фактически для возобновления нужны базовые механизмы продолжения и вложенных функций, но никакой другой пользы из этих механизмов не извлекается;

- методика восстановления в случае нехватки ресурсов, предлагаемая во втором доводе в пользу возобновления, неудачна в принципе. Она порождает тесные связи между библиотечным и пользовательским кодами, которые трудны для понимания и могут приводить к ошибкам;
- во многих предметных областях написаны весьма крупные системы, в которых применялась именно семантика завершения, так что возобновление просто необязательно.

Последний довод был подкреплен еще и теоретическим рассуждением Флавия Кристиана (Flaviu Cristian), который доказал, что при наличии завершения семантика возобновления не нужна [Cristian, 1989].

Потратив два года на споры, я вынес впечатление, что можно привести убедительные логические доводы в пользу любой точки зрения. Они имелись даже в специальной работе [Goodenough, 1975] по обработке исключений. Мы оказались в положении все тех же античных философов, которые так яростно и умно спорили о природе Вселенной, что как-то забыли о ее изучении. Поэтому я просил любого, кто имел реальный опыт работы с большими системами, представить конкретные факты. На стороне семантики возобновления оказался Мартин О'Риордан, сообщивший, что «компания Microsoft накопила многолетний позитивный опыт применения обработки исключений с возобновлением работы». Но у О'Риордана не было конкретных примеров; также весьма сомнительно, что первая версия операционной системы OS/2 по-настоящему ценна. Опыт применения ловушек ON в языке PL/I приводился в качестве как аргумента «за», так и аргумента «против» семантики возобновления.

Затем, на заседании в Пало Альто в ноябре 1991 г. мы услышали блестящий доклад Джима Митчелла (Jim Mitchell), работавшего в компании Sun, а перед этим – в Xerox PARC. Доклад содержал доводы в пользу семантики завершения, подкрепленные личным опытом и фактическими данными. На протяжении 20 лет Митчелл пользовался обработкой исключений в разных языках и на первых порах отстаивал семантику возобновления в качестве одного из главных проектировщиков и разработчиков системы Cedar/Mesa от компании Xerox. Его вывод звучал так:

«Завершение следует предпочесть возобновлению. Это вопрос не вкуса, а многолетней практики. Возобновление выглядит соблазнительно, но это порочный метод».

Свое утверждение Митчелл подкрепил рассказом о работе над несколькими операционными системами. Самым главным был пример системы Cedar/Mesa, которую написали программисты, любившие и умевшие пользоваться семантикой возобновления. Однако через десять лет в системе из полумиллиона строк остался лишь один случай использования возобновления – в запросе контекста. Поскольку и в данной ситуации оно фактически было не нужно, механизм возобновления исключили полностью, после чего скорость работы этой части системы значительно возросла. Во всех тех случаях, когда применялась операция возобновления (а это более десяти лет эксплуатации), появлялись определенные проблемы и приходилось искать более подходящий механизм. По сути дела, все

применения возобновления были связаны с неумением отделить друг от друга различные уровни абстракции.

Мэри Фонтана представила аналогичные данные по системе Explorer компании Texas Instruments, в которой возобновление использовалось только для отладки. Эрон Инсинга (Aron Insinga) познакомил нас с примерами очень ограниченного и не слишком важного применения семантики возобновления в системе DEC VMS, а Ким Кнутилла (Kim Knutilla) поведал точно такую же историю, как Джим Митчелл, только речь в ней шла о двух больших внутренних проектах IBM, просуществовавших достаточно долго. В пользу завершения свидетельствовал также опыт компании L.M.Ericsson, о котором рассказал Дэг Брюк.

Итак, комитет по стандартизации C++ высказался за семантику завершения.

16.6.1. Обходные пути для реализации возобновления

Похоже, что большую часть преимуществ, которые способна дать семантика возобновления, можно получить комбинированием вызова функции и исключения (с семантикой завершения). Рассмотрим функцию, которая вызывается для захвата некоторого ресурса `X`:

```
X* grab_X()      // захватить ресурс X
{
   for (;;) {
      if (can_acquire_an_X) {
         // ...
         return some_X;
      }

      // не удалось захватить ресурс, попробуем возобновить выполнение

      grab_X_failed();
   }
}
```

Задача функции `grab_X_failed()` – обеспечить возможность захвата ресурса `X`. Если она не в состоянии это сделать, то возбуждает исключение:

```
void grab_X_failed()
{
   if (can_make_X_available) {// восстановление
      // сделать X доступным
      return;
   }

   throw Cannot_get_X;  // исключение
}
```

Это обобщение техники использования функции `new_handler` для обработки нехватки памяти (см. раздел 10.6). У данного приема, конечно, есть много вариантов. Мне больше всего нравится указатель на функцию, которую пользователь может установить сам, чтобы реализовать собственную политику восстановления.

Этот метод не обременяет систему сложным механизмом, необходимым для реализации возобновления. Зачастую он даже не оказывает такого негативного влияния на организацию системы, как общая семантика возобновления.

16.7. Асинхронные события

Механизм обработки исключений в C++ не предназначен для обработки асинхронных событий, по крайней мере, напрямую:

«Можно ли применить исключения для обработки, скажем, сигналов? В большинстве систем разработки на C почти наверняка нет. Проблема в том, что в C используются нереентерабельные функции типа `malloc`. Если прерывание возникает в середине `malloc` и возбуждает исключение, то нет никакого способа запретить обработчику исключения вызвать `malloc` еще раз.

В компиляторе C++, в котором механизм вызова функций и вся библиотека времени выполнения спроектированы с учетом реентерабельности, сигналы могли бы возбуждать исключения. Но пока такие компиляторы не стали широко доступными (если это вообще когда-нибудь произойдет), мы рекомендуем четко отделять сигналы от исключений. Во многих случаях было бы разумно построить взаимодействие сигналов и исключений так: обработчик сигнала сохраняет где-то информацию, которая регулярно опрашивается некоторой функцией, а уже эта функция может возбудить исключение в зависимости от того, что обнаружено» [Koenig, 1990].

Большинство пользователей C/C++, употребляющих исключения, считают, что для создания надежной системы надо на как можно более ранней стадии придумать механизм для отображения асинхронных событий на модель процесса. Если исключения могут возникать в любой момент во время выполнения и если приходится прерывать обработку одного исключения, с тем чтобы уделить внимание другому, то хаоса не избежать. Низкоуровневая система обработки прерываний должна как можно дальше отстоять от обычных программ. Я разделяю данное мнение.

Приняв эту точку зрения, вы не будете напрямую использовать исключения, например, для обработки нажатия клавиши DEL и не станете заменять сигналы в ОС UNIX исключениями. В таких случаях некоторая низкоуровневая процедура обработки прерываний должна выполнить минимальную работу и, возможно, отобразить прерывание на обработчик, способное возбудить исключение в некоторой четко определенной точке программы. Отметим, что в соответствии со стандартом C обработчикам сигналов не разрешено вызывать функции, поскольку не гарантируется, что в этот момент работа компьютера в достаточной мере стабильна, чтобы успешно вызвать функцию и вернуться к выполнению программы.

Предполагается, что такие низкоуровневые события, как арифметические переполнения и деление на нуль, обрабатываются специализированными низкоуровневыми средствами, а не с помощью исключений. Это делает поведение C++ в области арифметических операций совместимым с другими языками и позволяет избежать проблем, возникающих в компьютерах с конвейерной архитектурой, где такие события, как деление на нуль, асинхронны. Синхронная обработка деления на нуль и т.п. возможна не на всех компьютерах. А там, где она возможна, сброс конвейера, необходимый для того чтобы такие события можно было перехватить до начала нового вычисления, замедляет работу компьютера (иногда на порядок).

16.8. Распространение на несколько уровней

Есть веские причины для того, чтобы позволить исключению неявно распространяться не далее функции, вызвавшей ту, где оно возникло. Однако для C++ это не подходит:

- имеются миллионы написанных на C++ функций, которые никто не будет модифицировать для обработки исключения или передачи его дальше;
- бессмысленно пытаться сделать каждую функцию барьером для всех ошибок. Лучшей стратегией является та, где обработкой нелокальных ошибок занимаются специально предназначенные для этого интерфейсы;
- в программах, где используются разные языки, нельзя требовать некоего специфического действия от конкретной функции, поскольку она может быть написана на другом языке. Так, допустимо, что C++-функция, возбуждающая исключение, вызывается из функции, написанной на C, которая, в свою очередь, вызвана из функции на C++, готовой это исключение обработать.

Первая причина имеет чисто прагматический характер, зато две последние принципиальны: вторая – это утверждение о стратегии проектирования, а третья о предположительных средах использования C++.

16.9. Статическая проверка

В C++ разрешено многоуровневое распространение исключений, но из-за этого утрачена одна из возможностей статического контроля. Нельзя, просто посмотрев на код функции, определить, какие исключения она может возбуждать. Фактически функция способна возбуждать любые исключения, даже если в тексте нет ни одного предложения `throw`, ибо исключения могут возбуждаться вызванными функциями.

Несколько программистов, в особенности Майк Пауэлл (Mike Powell), горько жаловались на эту особенность и думали над тем, как для исключений в C++ предоставить более твердые гарантии. Наилучший вариант – каждое возбужденное исключение перехватывается тем или иным обработчиком, написанным пользователем. Также часто требуется гарантия, что покинуть функцию могут только исключения из явно заданного списка. Механизм задания списка исключений, возбуждаемых функцией, в основных чертах спроектировали на доске Майк Пауэлл, Майк Тиман и я примерно в 1989 г.

«По существу, запись

```
void f() throw (e1, e2)
{
   // предложения
}
```

эквивалентна такой:

```
void f()
{
   try {
       // предложения
```

```
    }
    catch (e1) {
        throw; // возбудить повторно
    }
    catch (e2) {
        throw; // возбудить повторно
    }
    catch (...) {
        unexpected();
    }
}
```

Преимущество явного объявления исключений по сравнению с эквивалентной проверкой в тексте не только в том, что нужно набирать меньше символов. Важнее то, что объявление функции является частью интерфейса, видимого пользователю. Определение же функции доступно не всем, и, даже если имеются исходные тексты всех библиотек, лучше бы заглядывать в них пореже.

Еще одно преимущество состоит в том, что на практике может оказаться полезным обнаруживать неперехваченные исключения еще на стадии компиляции» [Koenig, 1990].

Спецификации исключений следовало бы проверять на стадии компиляции, но для этого все функции должны были бы вести себя согласованно, а это вряд ли возможно. К тому же такая статическая проверка легко могла бы послужить причиной многократной повторной компиляции программы. Более того, перекомпилировать программу могли бы лишь пользователи, обладающие всеми исходными текстами:

«Например, потенциально функцию необходимо изменить и перекомпилировать всякий раз, как в функции, которую она вызывает (прямо или косвенно), изменяется множество перехватываемых или возбуждаемых исключений. Это может привести к серьезным срывам в сроках поставки программы, составленной (пусть даже частично) из нескольких независимо разработанных библиотек. В таких библиотеках де-факто должен был бы согласовываться набор используемых исключений. Например, если подсистема X обрабатывает исключения, возбуждаемые в подсистеме Y, а поставщик Y вводит новый тип исключения, то код X пришлось бы модифицировать. Пользователь и X, и Y не сможет поставить новую версию своего продукта, пока обе подсистемы не будут модифицированы. В случаях когда используется много подсистем, это может привести к целой цепочке задержек. Но даже и там, где пользователь прибегает к услугам только одного поставщика, это может стать причиной каскадной модификации кода и многократных перекомпиляций.

Из-за такого рода проблем многие стараются вовсе не прибегать к механизму спецификации исключений или как-то обойти его» [Koenig, 1990].

Поэтому мы решили поддержать только проверку во время исполнения, а статический контроль оставить специальным инструментальным средствам.

«Аналогичная трудность возникает также при использовании динамической проверки. Однако в этом случае ее можно решить с помощью механизма группировки исключений, описанного в разделе 16.4. При наивном подходе новое исключение, добавленное к подсистеме Y, осталось бы неперехваченным или было бы преобразовано в вызов функции `unexpected()` некоторым явно вызываемым интерфейсом. Однако если подсистема Y построена правильно, то все ее исключения были бы производными от класса `Yexception`. Например:

```
class newYexception : public Yexception { /* ... */ };
```

При этом функция, объявленная как

```
void f() throw (Xexception, Yexception, IOexception);
```

смогла бы обработать исключение `newYexception`, передав его функции, вызвавшей `f()`».

Более подробное обсуждение см. в [2nd, §9].

В 1995 г. была найдена схема, которая позволяет в какой-то мере выполнить статический контроль спецификации исключений, не порождая описанных выше проблем. Поэтому теперь спецификации исключений проверяются, так что инициализация и присваивание указателю на функцию, а также замещение виртуальных функций не могут нарушить защиту. Некоторые неожиданные исключения все же возможны, и они, как и раньше, перехватываются во время исполнения.

16.9.1. Вопросы реализации

Как обычно, эффективности уделялось самое пристальное внимание. Очевидно, что можно было спроектировать такой механизм обработки исключений, реализовать который удалось бы лишь за счет явных затрат при вызове функций или неявных, связанных с невозможностью некоторых видов оптимизации. Похоже, нам удалось этого избежать, поскольку, по крайней мере, теоретически механизм исключений в C++ можно реализовать без каких бы то ни было временных издержек в программах, которые не возбуждают исключений. Компилятор можно построить так, что все временные издержки будут проявляться лишь тогда, когда исключение действительно возбуждается [Koenig, 1990]. Ограничить расход памяти также не составляет особой сложности, но очень трудно одновременно избежать и временных, и пространственных затрат. Сейчас есть уже несколько компиляторов, поддерживающих исключения, поэтому последствия компромиссов можно сравнить (см., например, [Cameron, 1992]).

Как ни странно, обработка исключений почти не отражается на модели размещения объекта в памяти. Для обеспечения взаимодействия между точкой возбуждения исключения и обработчиком необходимо уметь представлять тип во время выполнения. Однако это можно сделать с помощью специализированного механизма, не затрагивающего объект в целом, используя структуры данных, поддерживающие идентификацию типа во время исполнения (см. раздел 14.2.6). Гораздо важнее то, что особенно значимым становится отслеживание точного времени жизни всех автоматических объектов. Прямолинейная реализация может привести к неоправданному увеличению кода, несмотря даже на то, что число дополнительных команд, которые действительно выполняются, невелико.

Техника реализации, представляющаяся оптимальной, заимствована из работ по языкам Clu и Modula-2+ [Rovner, 1986]. Основная идея состоит в том, чтобы составить таблицу диапазонов адресов кода, соответствующих состоянию вычислений, относящихся к обработке исключений. Для каждого диапазона регистрируются деструкторы и обработчики исключений, которые следует вызвать. Когда возбуждается исключение, механизм его обработки сравнивает текущее значение счетчика команд с адресами в таблице диапазонов. Если оно попадает в один из диапазонов, то предпринимаются соответствующие действия; в противном случае производится раскрутка стека и в таблице ищется значение счетчика команд для вызвавшей функции.

16.10. Инварианты

Поскольку язык C++ еще сравнительно молод, продолжает развиваться и в то же время широко используется, постоянно появляются предложения об улучшениях и расширениях. Как только в каком-нибудь языке появлялось новомодное средство, его рано или поздно начинали предлагать включить в C++. Бертран Мейер (Bertrand Meyer) популяризировал старую идею о пред- и постусловиях и ввел для нее прямую поддержку в языке Eiffel [Meyer, 1988]. Естественно, то же самое предлагалось сделать и для C++.

Определенная часть пользователей C всегда интенсивно работала с макросом `assert()`, но не было приемлемого способа сообщить о нарушении утверждения во время выполнения. Такой способ дают исключения, в то время как шаблоны позволяют отказаться от макросов. Например, вот как можно написать шаблон `Assert()`, который имитирует C-макрос `assert()`:

```
template<class T, class X> inline void Assert(T expr,X x)
{
   if (!NDEBUG)
      if (!expr) throw x;
}
```

Он будет возбуждать исключение `x`, если выражение `expr` ложно и мы не отключили проверку, задав ненулевое значение переменной `NDEBUG`. Например:

```
class Bad_f_arg { };

void f(String& s, int i)
{
   Assert(0<=i && i<s.size(), Bad_f_arg());
   // ...
}
```

Это наименее структурированный вариант такой техники. Я предпочитаю определять инварианты для классов в виде функций-членов, а не использовать утверждения напрямую. Например:

```
void String::check()
{
   Assert(p
          && 0<=sz
          && sz<TOO_LARGE
          && p[sz-1]==0 , Invariant);
}
```

Простота, с которой в уже существующем языке C++ можно определять и проверять утверждения и инварианты, намного уменьшила число тех, кто непременно желает включить расширение для прямой поддержки верификации программ. Поэтому большая часть работы, относящейся к подобным приемам, проходила по разряду предложений по методике стандартизации [Gautron,1992], или была связана с более серьезными системами верификации [Lea, 1990], или просто велась в рамках имеющегося языка.

Глава 17. Пространства имен

Всегда рассматривайте то, что вы проектируете,
в более широком контексте.

Элиэль Сааринен

17.1. Введение

В C имеется единое глобальное пространство для всех имен, что не совсем удачно сочетается с одной функцией, структурой или единицей трансляции. Это порождает конфликты имен. Попытка решить проблему в первоначальном проекте C++ сводилась вот к чему: по умолчанию все имена объявлялись локальными для единицы трансляции, а чтобы они прочитывались в других единицах, следовало явно воспользоваться словом `extern`. Как отмечалось в разделе 3.12, с одной стороны, этого было мало для решения исходной задачи, с другой стороны, такое решение было недостаточно совместимым. Итак, идея провалилась.

Занимаясь механизмом типобезопасного связывания (см. раздел 11.3), я снова вернулся к этой проблеме. Было замечено, что небольшое изменение синтаксиса, семантики и способа реализации конструкции

```
extern "C" { /* ... */ }
```

позволило бы нам считать, что

```
extern XXX { /* ... */ }
```

означает следующее: имена, объявленные внутри `XXX`, принадлежат отдельной области действия `XXX` и доступны в других областях действия лишь при наличии квалификатора `XXX::`. Следовательно, метод был в точности таким же, как при доступе к статическим членам класса извне этого класса.

По разным причинам и прежде всего из-за нехватки времени описанная идея долгое время пролежала в столе и была извлечена на свет во время обсуждений на заседаниях комитета ANSI/ISO в начале 1991 г. Сначала Кит Роуи (Keith Rowe) из Microsoft предложил использовать нотацию

```
bundle XXX { /* ... */ };
```

для определения именованной области действия и оператор `use` для импорта имен из области `bundle` в другую область действия. В дискусси на эту тему приняли участие несколько членов рабочей группы по расширениям, в том числе Стив Дович (Steve Dovich), Дэг Брюк, Мартин О'Риордан и я. В конечном итоге Фолькер Баух (Volker Bauche), Роланд Хартингер (Roland Hartinger) и Эрвин

Унру, представляющие компанию Siemens, выдвинули предложение, в котором новых ключевых слов не было:

```
:: XXX :: { /* ... */ };
```

Это стало поводом для серьезных споров в группе по расширениям. В частности, Мартин О'Риордан показал, что нотация `::` приводит к неоднозначному толкованию использования `::` для членов классов и глобальных имен.

К началу 1993 г. мне удалось сформулировать логически непротиворечивое предложение. Огромный вклад в техническое обсуждение вопроса о пространствах имен внесли Дэг Брюк, Джон Бранс, Стив Дович, Билл Гиббонс, Филипп Готрон, Тони Хансен, Питер Джуль, Эндрю Кениг, Эрик Крон (Eric Krohn), Дуг Макилрой, Ричард Миннер (Richard Minner), Мартин О'Риордан, Джон Скэллер (John Skaller), Джерри Шварц, Марк Террибиль (Mark Terribile) и Майк Вило. Кроме того, Вило ратовал за немедленное воплощение этих идей в конкретное предложение, чтобы иметь нужные средства для решения неминуемой проблемы имен в стандартной библиотеке ISO C++. Включение пространств имен в C++ одобрено на заседании комитета в Мюнхене в июле 1993 г. На заседании в Сан-Хосе (ноябрь 1993 г.) решено использовать пространства имен для контроля за именами в стандартных библиотеках C и C++.

17.2. Для чего нужны пространства имен

При наличии только одного глобального пространства имен оказывается трудно писать фрагменты программ, которые будут связываться вместе, и при этом не опасаться возникновения конфликта имен. Например:

```
// my.h:
char f(char);
int f(int);
class String { /* ... */ };

// your.h
char f(char);
double f(double);
class String { /* ... */ };
```

Третья сторона не сможет одновременно использовать файлы `my.h` и `your.h`, содержащие такие объявления.

Заметим, что некоторые из этих имен будут присутствовать в объектном коде, а программы нередко поставляются без исходных текстов. Отсюда следует, что подмены одного имени другим с помощью макросов и т.п. недостаточно, так как при этом не меняется имя, доступное компоновщику.

17.2.1. Обходные пути

Существует несколько обходных путей. Например:

```
// my.h:
char my_f(char);
```

```
int my_f(int);
class my_String { /* ... */ };

// your.h
char yo_f(char);
double yo_f(double);
class yo_String { /* ... */ };
```

Такой подход не является необычным, но он совершенно неудачен, а если префиксы к тому же не очень короткие, то и неудобен для пользователя. Другая сложность состоит в том, что существует лишь несколько сотен двухбуквенных префиксов, а на C++ уже написаны сотни библиотек. Это одна из самых старых проблем из числа рассматриваемых в данной книге. Программисты, давно работающие на C, вспомнят то время, когда в имена членов структур включались одно- или двубуквенные суффиксы, чтобы избежать конфликтов с членами других структур.

Использование макросов может сделать такое решение еще хуже (или лучше, если вам макросы нравятся):

```
// my.h:
#define my(X) myprefix_##X

char my(f)(char);
int my(f)(int);
class my(String) { /* ... */ };

// your.h
#define yo(X) your_##X

char yo(f)(char);
double yo(f)(double);
class yo(String) { /* ... */ };
```

Идея состоит в том, чтобы разрешить длинные префиксы в именах, видимых компоновщику, оставив короткими имена, используемые в исходном тексте. Как и все схемы, основанные на макросах, эта также создает проблемы для инструментальных средств: отслеживать отображение имен должен либо инструмент (за счет существенного усложнения), либо пользователь (что усложняет программирование и сопровождение).

Альтернативный подход – его обычно предпочитают те, кто не любит макросы, – поместить всю необходимую информацию внутри класса:

```
// my.h:
class My {
public:
   static char f(char);
   static int f(int);
   class String { /* ... */ };
};
```

```
// your.h
class Your {
public:
   static char f(char);
   static double f(double);
   class String { /* ... */ };
};
```

К сожалению, и этот подход не лишен многих мелких неудобств. Не все глобальные объявления можно так просто перенести в класс, а некоторые от этого меняют семантику. Например, чтобы избежать изменений семантики, глобальные функции и переменные должны объявляться как статические члены, а тела функций и инициализаторы обычно следует отделять от объявлений.

17.3. Какое решение было бы лучшим?

Для решения проблем, связанных с пространством имен, можно воспользоваться разными механизмами. На самом деле, во многих языках имеются, по крайней мере, зачатки средств такого рода. Например, в C есть статические функции, в Pascal – вложенные области действия, в C++ – классы, но более полные решения следует искать в таких языках, как PL/I, Ada, Modula-2, Modula-3 [Nelson, 1991], ML [Wilkstrom, 1987] и CLOS [Kiszales, 1992].

Итак, что должен был бы дать языку C++ хороший механизм пространств имен? Продолжительная дискуссия рабочей группы по расширениям при комитете ANSI/ISO позволила составить целый список преимуществ. Вот эти возможности:

- связь с двумя библиотеками без конфликтов имен;
- введение имени без конфликта с чужими именами (например, из библиотеки, о которой я никогда не слышал, или неизвестными мне именами из библиотеки, которую, как мне казалось, я знаю);
- добавление новое имя в библиотеку без ущерба интересов пользователей;
- использование имен из двух разных библиотек, даже если эти имена совпадают;
- разрешение конфликтов имен без модификаций самих функций (только за счет манипулирования разрешением имен на уровне объявлений);
- добавление нового имени в пространство имен, не опасаясь вызвать незаметного для пользователя изменения смысла кода, использующего другие пространства имен (мы не можем дать таких гарантий для кода, где используется пространство имен, в которое добавлено имя);
- отсутствие конфликтов между именами самих пространств имен; в частности, реальное (видимое компоновщику) имя может быть длиннее имени, использованного в тексте программы;
- применение механизма пространств к стандартным библиотекам;
- совместимость с C и C++;
- отсутствие издержек на этапе компоновки и во время выполнения при использовании пространств имен;

- лаконичность пространств имен, не меньшая, чем при использовании глобальных имен;
- явное указание в коде, из какого пространства должно быть взято имя.

Хорошее решение также должно быть простым. Под простотой я понимаю следующее:

- чтобы механизм можно было сразу использовать для серьезных задач, на объяснение его работы должно уйти не более десяти минут. Чтобы объяснение удовлетворило языковых пуристов, может потребоваться гораздо больше времени;
- разработчик компилятора должен быть способен реализовать этот механизм менее чем за две недели.

Разумеется, простоту здесь нельзя понимать строго, ведь людям с разной подготовкой и способностями для усвоения одного и того же материала требуется неодинаковое время.

Есть также несколько свойств, которые мы сознательно решили исключить из списка критериев, хотя об их включении часто просили:

- возможность взять два объектных файла с конфликтующими именами и связать их вместе. Это могут сделать инструментальные средства в любой системе, но неясно, как можно было бы реализовать для этого языковую поддержку без значительных затрат. Существует слишком много компоновщиков и форматов объектных файлов, чтобы рассчитывать на их приведение к общему стандарту. Чтобы решение было полезным для C++, оно должно зависеть только от возможностей, предоставляемых практически всеми современными компоновщиками;
- возможность назначать произвольные синонимы именам, используемым в библиотеках. Существующие механизмы – `typedef`, ссылки и макросы – позволяют это делать в некоторых ситуациях, а универсальным средствам переименования я не доверяю (см. раздел 12.8).

Отсюда следует, что механизм устранения неоднозначности должен быть встроен в объектный код авторами отдельных фрагментов программы. В частности, поставщикам библиотек придется использовать какой-то метод, который дал бы возможность пользователям разрешить неоднозначность. К счастью, именно поставщики библиотек больше всего выигрывают от систематического использования пространств имен.

Разумеется, этот перечень можно продолжать, и многие наверняка не согласны с оценкой относительной важности каждого критерия. Тем не менее список дает представление о сложности задачи и о требованиях, которым должно удовлетворять решение.

Изложив все критерии, я получил возможность проверить, в какой мере проектирование пространств имен отвечает требованиям простоты. Питер Джуль выполнил первую реализацию за пять дней, а мне менее чем за десять минут удалось объяснить основы механизма пространств имен нескольким пользователям. Заданные вопросы показали, что мои «ученики» усвоили суть и смогли самостоятельно

додуматься до таких применений пространств имен, которые я вообще не объяснял. Итак, механизм оказался достаточно простым. Опыт последующей реализации, обсуждение концепции пространств имен и некоторые их приложения утвердили меня в этом мнении.

17.4. Решение: пространства имен

Принятое решение в принципе очень просто. Оно дает четыре новых механизма:

- механизм определения области действия, обобщающий глобальные объявления в C и C++: пространства имен. Такие области действия разрешается именовать, а к их членам можно обращаться с помощью обычного для членов класса синтаксиса: `namespace_name::member_name`, где `namespace_name` – имя пространства имен, а `member_name` – имя члена. Фактически область действия класса можно считать частным случаем области действия пространства имен;
- механизм определения локального синонима для имени из пространства имен;
- механизм, позволяющий обратиться к одному члену пространства имен, не указывая имя пространства имен, то есть опуская префикс `namespace_name::`. Это достигается использованием using-объявления;
- механизм, позволяющий обращаться ко всем членам пространства имен, не указывая явно имя самого пространства имен. Достигается использованием using-директивы.

Тем самым удовлетворяются все критерии, приведенные в разделе 17.3. Кроме того, решается давний вопрос о том, как обратиться к членам базового класса из области действия производного класса (см. разделы 17.5.1 и 17.5.2). Заодно становится избыточным атрибут `static` для глобальных имен (см. раздел 17.5.3).

Рассмотрим пример:

```
namespace A {
   void f(int);
   void f(char);
   class String { /* ... */ };
   // ...
}
```

Имена, объявленные внутри фигурных скобок, принадлежат пространству имен `A` и не конфликтуют ни с глобальными именами, ни с именами из других пространств имен. Объявления внутри пространства имен (в том числе, и определения) имеют в точности ту же семантику, что и глобальные объявления, но их область действия ограничена этим пространством имен.

Программист может использовать такие имена, добавляя явный квалификатор:

```
A::String s1 = "Annemarie";

void g1()
{
   A::f(1);
}
```

Вместо этого можно явно указать, что использование квалификатора является излишним для отдельного имени из конкретного пространства имен. Для этого служит using-объявление:

```
using A::String;
String s2 = "Nicholas";// то же, что A::String
void g2()
{
   using A::f;   // вводим локальный синоним для f из A
   f(2);         // то же, что A::f
}
```

Using-объявление вводит синоним для некоторого имени в локальную область действия.

Вместо этого мы можем сделать доступными без квалификации все имена из некоторой библиотеки с помощью using-директивы:

```
using namespace A;      // сделать доступными все имена из A
String s3 = "Marian";   // то же, что A::s3

void g3()
{
   f(3);      // то же, что A::f
}
```

Using-директива не вводит имена из указанного пространства имен в локальную область действия, а просто делает их доступными.

В моем первоначальном проекте у using-директив был более простой и лаконичный синтаксис:

```
using A;  // означает "using namespace A;"
```

Это привело к путанице: вместо using-директив использовали using-объявления и наоборот. С введением более явного синтаксиса неразбериха исчезла. Также упростился синтаксический анализатор.

Я предвидел, что пользователи захотят избегать длинных имен. Поэтому в первоначальном проекте в одном using-объявлении было разрешено указывать имена нескольких членов:

```
using X::(f,g,h);
```

Однако это неудачно, равно как и все рассмотренные нами альтернативы. Точнее, каждая из обсуждавшихся альтернатив хоть кем-то была признана неудачной. Получив некоторый опыт работы с пространствами имен, я понял, что потребность в таких списках возникает куда реже, чем ожидалось. Кроме того, при чтении кода я обычно не обращал внимания на такие списки, так как они слишком похожи на объявления функций. Поэтому вместо них я стал употреблять повторные using-объявления:

```
using X::f;
using X::g;
using X::h;
```

Вот почему в окончательном варианте для задания списка имен в using-объявлении не осталось специального синтаксиса.

Пространства имен – это пример средства, которое после экспериментирования значительно упростилось. Реализовать его также оказалось легко, поскольку пространства имен прекрасно укладываются в общую концепцию классов и областей действия C++.

17.4.1. Мнения по поводу пространств имен

После неоднократных попыток согласовать противоречивые взгляды на то, что именно требуется от пространств имен в больших программах, выкристаллизовались три способа доступа к именам. Одни пользователи настаивали, что надежную и поддающуюся сопровождению программу можно построить только в том случае, если каждое применение нелокального имени будет явно квалифицировано. Естественно, что сторонники данной точки зрения весьма сомневались в полезности using-объявлений и уж тем более using-директив.

Другие осуждают явную квалификацию за излишнюю многословность, которая якобы затрудняет изменение кода, ограничивает гибкость и лишает практического смысла переход к использованию пространств имен. Разумеется, такие люди выступают за using-директивы и иные механизмы, позволяющие заменять длинные квалифицированные имена короткими синонимами.

Я с сочувствием отношусь к смягченным вариантам той и другой точек зрения, поэтому при работе с пространствами имен допустим любой стиль. Как обычно, ничто не мешает конкретной организации составить руководство по стилю программирования, в котором будут описаны корпоративные требования. Но заставлять всех программистов им следовать, вводя некоторое языковое правило, было бы неразумно.

Многие не без оснований опасаются исчезновения обычных неквалифицированных имен. Речь идет о привязке имени не к тому объекту или функции, которые имел в виду программист. Каждый программист на C хотя бы однажды сталкивался с данным явлением. Явная квалификация практически снимает эту проблему. К тому же ряду относится опасение, что будет трудно найти объявление имени и догадаться о смысле содержащего его выражения. Явная квалификация содержит столь очевидные указания, что часто и искать-то объявление не нужно: сочетания имени библиотеки с именем функции бывает вполне достаточным для понимания смысла выражения. Поэтому необычные или нечасто используемые нелокальные имена лучше квалифицировать явно; код от этого только станет яснее.

С другой стороны, явная квалификация имен, о которых все знают (или должны знать), и тех, которые часто используются, может быстро надоесть. Например, запись `stdio::printf`, `math::sqrt`, `iostream::cout` вряд ли хоть чем-то поможет человеку, знакомому с C++, а никому не нужная громоздкость становится источником ошибок. Это сильный аргумент в пользу using-объявлений и using-директив. При этом using-объявления – более избирательный и потому гораздо менее опасный механизм. Using-директива

```
using namespace X;
```

делает видимым неизвестное множество имен. В частности, сегодня это множество может быть одним, а завтра, после внесения изменений в X, – совершенно другим. Те, кому такой подход не нравится, могут явно указать нужные им имена из X в using-объявлениях:

```
using X::f;
using X::g;
using X::h;
```

Но иногда возможность получить доступ к каждому имени из пространства имен, не перечисляя их все, с автоматическим изменением множества доступных имен после модификации определения X – как раз то, что нужно.

17.4.2. Внедрение пространств имен

Принимая во внимание наличие миллионов строк кода на C++, зависящих от использования глобальных имен и существующих библиотек, я полагал, что самый важный вопрос о пространствах имен – как их внедрить? Неважно, насколько изящно выглядит код, в котором применяются пространства имен, если ни у пользователей, ни у поставщиков библиотек нет пути для перехода к применению нового механизма. Требовать масштабного переписывания кода – это, конечно, не вариант.

Рассмотрим каноническую программу на C, с которой всегда начинают обучение этому языку:

```
#include <stdio.h>

int main()
{
   printf("Hello, world\n");
}
```

Эта программа не должна переставать работать. Не нравится мне и предложение считать стандартные библиотеки особым случаем. Я признаю очень важной возможность применения механизма пространств имен к стандартным библиотекам на общих основаниях. Но комитет по стандартизации не вправе требовать для своих библиотек привилегий, которые не в состоянии предоставить поставщикам других библиотек.

Ключом к решению этой задачи являются using-директивы. Например, файл stdio.h помещается в пространство имен:

```
// stdio.h:

namespace std {
   // ...
   int printf(const char* ... );
   // ...
}
using namespace std;
```

Этим достигается обратная совместимость, а для тех, кто не хочет, чтобы имена были доступны неявно, определяется новый заголовочный файл `stdio`:

```
// stdio:

namespace std {
   // ...
   int printf(const char* ... );
   // ...
}
```

Те, кому не нравится дублирование объявлений, конечно, просто включат `stdio` в определение `stdio.h`:

```
// stdio.h:

#include <stdio>
using namespace std;
```

Думаю, что using-директивы – это, в первую очередь, инструмент, облегчающий переход к пространствам имен. Большинство программ станут яснее, если при обращении к именам из внешних пространств будут использоваться явная квалификация и using-объявления.

Конечно, имена из пространства имен, в котором они определены, не требуют никакой квалификации:

```
namespace A {
   void f();
   void g()
   {
      f();       // вызывается A::f; квалификация не нужна
      // ...
   }
}

void A::f()
{
   g();          // вызывается A::g; квалификация не нужна
   // ...
}
```

В этом отношении пространства имен ведут себя так же, как классы.

17.4.3. Псевдонимы пространства имен

Если пользователь дает своим пространствам имен короткие имена, возможен конфликт, например:

```
namespace A {// короткое имя пространства имен
             // когда-нибудь обязательно возникнет конфликт
   // ...
};
```

```
A::String s1 = "asdf";
A::String s2 = "lkjh";
```

Но длинные имена писать утомительно:

```
namespace American_Telephone_and_Telegraph {    // слишком длинное для
          // использования в реальном коде
   // ...
}

American_Telephone_and_Telegraph::String s3 = "asdf";
American_Telephone_and_Telegraph::String s4 = "lkjh";
```

Эту дилемму можно разрешить, введя короткий псевдоним для длинного имени пространства имен:

```
// использование псевдонима для длинного имени пространства имен

namespace ATT = American_Telephone_and_Telegraph;

ATT::String s3 = "asdf";
ATT::String s4 = "lkjh";
```

Такая возможность позволяет пользователю обращаться к библиотеке, не указывая каждый раз ее настоящего имени. Пространства имен можно применять для составления интерфейсов, в которые входят имена, принадлежащие сразу нескольким пространствам:

```
namespace My_interface {
   using namespace American_Telephone_and_Telegraph;
   using My_own::String;
   using namespace OI;
      // разрешить конфликт между определениями 'Flags'
      // из OI и American_Telephone_and_Telegraph
   typedef int Flags;
   // ...
}
```

17.4.4. Использование пространств имен для управления версиями

В качестве примера применения пространств имен показывается, как поставщик библиотеки мог бы воспользоваться ими для решения проблемы несовместимых отличий между разными версиями. Этот прием мне впервые продемонстрировал Тандж Бенетт (Tanj Bennett). Вот первая версия `release1`:

```
namespace release1 {
   // ...
   class X {
      Impl::Xrep* p;
   public:
```

```
        virtual void f1() = 0;
        virtual void f2() = 0;
        // ...
    };
    // ...
}
```

`Impl` – это некоторое пространство имен, в котором находятся детали реализации.

Пользовательский код мог бы выглядеть так:

```
class XX : public release1::X {
    int xx1;
    // ...
public:
    void f1();
    void f2();
    virtual void ff1();
    virtual void ff2();
    // ...
};
```

Это значит, что поставщик библиотеки не может изменить размер объектов класса `release1::X` (например, добавить новые данные-члены), добавить или изменить порядок следования виртуальных функций и т.д., поскольку это потребовало бы перекомпиляции пользовательского кода, чтобы учесть новое размещение объекта в памяти. Существуют реализации C++, защищающие пользователя от таких изменений, но пока не получившие широкого распространения, так что разработчик библиотеки не может полагаться на них, если не хочет привязать себя к единственному компилятору. Я мог бы посоветовать пользователям не наследовать подобным библиотечным классам таким образом, но ведь, несмотря на все предупреждения, они все равно будут поступать по-своему и жаловаться на необходимость перекомпиляции.

Нужно найти другое, более удачное решение. Если воспользоваться пространствами имен для различения версий, вторая версия `release2` могла бы выглядеть так:

```
namespace release1 {    // версия 1 оставлена для совместимости
    // ...
    class X {
        Impl::Xrep* p;  // Impl1::Xrep приведена в соответствии
                        // с версией 2
    public:
        virtual void f1() = 0;
        virtual void f2() = 0;
        // ...
    };
    // ...
}
```

```
namespace release2 {
   // ...
   class X {
      Impl::Xrep* p;
   public:
      virtual void f2() = 0;  // новый порядок следования
      virtual void f3() = 0;  // новая функция
      virtual void f1() = 0;
      // ...
   };
   // ...
}
```

Старый код пользуется версией `release1`, новый – версией `release2`. При этом оба кода работают и не вступают в конфликт. Заголовочные файлы для версий `release1` и `release2` различаются, так что для обеспечения минимальной функциональности пользователю нужен только один `#include`. Чтобы облегчить переход на новую версию, можно воспользоваться псевдонимом пространства имен для локализации последствий изменения версии. Всего один файл

```
// lib.h:
   namespace lib = release1;
   // ...
```

может инкапсулировать все зависимости от версии и использоваться всюду:

```
#include "lib.h"

class XX : public lib::X {
   // ...
};
```

При переходе на новую версию достаточно изменить лишь одну строку:

```
// lib.h:
   namespace lib = release2;
   // ...
```

Такое изменение делается лишь тогда, когда появились основательные причины перейти к версии `release2`, и нашлось время, чтобы перекомпилировать программу и разобраться с возможными несовместимостями между обеими версиями на уровне исходных текстов.

17.4.5. Технические детали

Здесь рассматриваются технические детали, касающиеся разрешения областей действия, глобальной области действия, перегрузки, вложенных пространств имен и составления пространства имен из отдельных частей.

17.4.5.1. Удобство и безопасность

Using-объявление добавляет имена в локальную область действия. Using-директива этого не делает, она лишь обеспечивает доступность имен. Например:

```
namespace X {
   int i, j, k;
}

int k;

void f1()
{
   int i = 0;
   using namespace X;   // делает доступными имена из X
   i++;                 // локальное i
   j++;                 // X::j
   k++;                 // ошибка: X::k или глобальное k?
   ::k++;               // глобальное k
   X::k++;              // k из X
}

void f2()
{
   int i = 0;
   using X::i;   // ошибка: i дважды объявлена в f2()
   using X::j;
   using X::k;   // скрывает глобальное k

   i++;
   j++;       // X::j
   k++;       // X::k
}
```

Тем самым сохраняется важное свойство: локально объявленное имя (введенное либо с помощью обычного, либо с помощью using-объявления) скрывает нелокальные объявления того же имени, и любая некорректная перегрузка имени обнаруживается в точке объявления.

Как видно из вышеприведенного примера, если отдавать приоритет глобальной области действия над пространствами имен, которые в нее включены, то удается в какой-то мере защититься от случайного конфликта имен.

С другой стороны, нелокальные имена видимы в контексте, где были объявлены, и трактуются так же, как любые другие нелокальные имена. В частности, ошибки, связанные с using-директивой, обнаруживаются только в точке использования. Это защищает программиста от сообщений компилятора о потенциальных ошибках. Например:

```
namespace A {
   int x;
}

namespace B {
   int x;
}
```

```
void f()
{
   using namespace A;
   using namespace B;   // правильно: здесь ошибки нет

   A::x++;        // правильно
   B::x++;        // правильно
   x++;           // ошибка: A::x или B::x?
}
```

17.4.5.2. Глобальная область действия

С появлением пространств имен глобальная область действия стала лишь еще одним пространством имен. Особенность глобального пространства имен состоит только в том, что его имя не обязательно явно квалифицировать. Запись `::f` означает «`f` объявлено в глобальном пространстве имен», тогда как `X::f` означает «`f` объявлено в пространстве имен `X`». Рассмотрим пример:

```
int a;

void f()
{
   int a = 0;
   a++;       // локальное a
   ::a++;     // глобальное a
}
```

Если поместить этот код в пространство имен и добавить еще одну переменную `a`, получим:

```
int a;

namespace X {
   int a;

   void f()
   {
       int a = 0;
       a++;       // локальное a
       X::a++;    // X::a
       ::a++;     // X::a или глобальное a? - глобальное a
   }
}
```

Другими словами, квалификация с помощью унарного оператора `::` означает «глобальное» пространство имен, а не то, в котором объявлена переменная. Последнее означало бы, что при погружении произвольного кода в пространство имен его смысл не изменяется. Но тогда глобальная область действия не имела бы никакого имени, а это противоречит нашему желанию, чтобы глобальное пространство было лишь частным случаем обычного пространства имен, хотя и с несколько странным именем. Поэтому мы и решили, что оператор `::` будет адресовать переменную `a`, объявленную в глобальной области действия.

Думается, что применять глобальные имена станут значительно реже. Правила для пространств имен специально были составлены так, чтобы не давать никаких преимуществ ленивым программистам по сравнению с теми, кто сознательно стремится не засорять глобальную область действия.

Обратите внимание, что using-директива не объявляет имена в той области действия, где употребляется:

```
namespace X {
   int a;
   int b;
   // ...
}

using namespace X;  // сделать доступными все имена из X
using X::b;         // объявить локальный синоним для X::b

int i1 = ::a;// ошибка: в глобальной области 'a' не объявлена
int i2 = ::b;// правильно: будет найден локальный синоним X::b
```

Отсюда следует, что старый код, в котором явно использовался оператор `::`, перестанет работать, если погрузить библиотеку в пространство имен. Нужно либо модифицировать код, явно указав новое имя библиотечного пространства имен, либо воспользоваться подходящим глобальным using-объявлением.

17.4.5.3. Перегрузка

Самым запутанным аспектом предложения о пространствах имен был вопрос, следует ли разрешать перегрузку имен, объявленных в разных пространствах. Рассмотрим пример:

```
namespace A {
   void f(int);
   // ...
}
using namespace A;

namespace B {
   void f(char);
   // ...
}
using namespace B;

void g()
{
   f('a');       // вызывается B::f(char);
}
```

Пользователь, невнимательно ознакомившийся с пространством имен `B`, может решить, что будет вызвана `A::f(int)`. Сложности возникнут и у пользователя, который тщательно изучил программу год назад, но не обратил внимания, что в последней версии в пространство имен `B` было добавлено объявление `f(char)`.

Однако эта проблема возникает лишь тогда, когда вы сопровождаете программу, в которой предложение `using namespace` используется дважды для одной и той же области действий – для вновь разрабатываемых продуктов такая практика не рекомендуется. Кроме того, компилятор вполне может выдать предупреждение о том, что вызов функции разрешен двумя способами за счет объявлений из разных пространств имен, даже если обычные правила перегрузки позволяют выбрать только один вариант. Я рассматриваю using-директивы в основном как средство, применимое только в переходный период; авторы новых программ смогут избежать многих теоретических и некоторых практических затруднений, если будут по возможности употреблять квалифицированные имена и using-объявления.

Перегрузка имен, объявленных в разных пространствах, разрешена по двум причинам. С одной стороны, это проще всего, поскольку в таком случае применяются общие правила перегрузки. С другой стороны, удается обойтись минимальными изменениями кода при включении пространств имен в существующие библиотеки (по крайней мере, ничего лучшего придумать не удалось). Например:

```
// старый код:

void f(int); // из A.h
// ...

void f(char);// из B.h
// ...

void g()
{
   f('a');       // вызывается f из B.h
}
```

Для добавления в этот код пространств имен не надо менять ничего, кроме заголовочных файлов.

17.4.5.4. Вложенные пространства имен

Одно из очевидных применений пространства имен – помещение в него полного набора объявлений и определений:

```
namespace X {
   // все мои объявления
}
```

Обычно список объявлений также может содержать пространства имен. Поэтому из прагматических соображений вложенные пространства имен допускаются. Это согласуется и с той точкой зрения, что любые конструкции должны вкладываться, если нет веских доводов против такой возможности. Например:

```
void h();

namespace X {
   void g();
   // ...
```

```
    namespace Y {
        void f();
        void ff();
        // ...
    }
    // ...
}
```

Применимы обычные правила областей действия и квалификации:

```
void X::Y::ff()
{
    f(); g(); h();
}

void X::g()
{
    f();          // ошибка: в X нет f()
    Y::f();
}

void h()
{
    f();          // ошибка: нет глобальной f()
    Y::f();       // ошибка: нет глобального Y
    X::f();       // ошибка: в X нет f()
    X::Y::f();
}
```

17.4.5.5. Пространства имен открыты

Пространство имен открыто в том смысле, что его можно составлять из нескольких объявлений, расположенных в разных местах. Например:

```
namespace A {
    int f();      // сейчас в A есть член f()
};

namespace A {
    int g();      // а сейчас в A есть члены f() и g()
};
```

Это делается для того, чтобы большие фрагменты программы можно было поместить в одно пространство имен. Ведь находятся же в одном пространстве имен современные библиотеки и приложения! Необходимо распределить определение пространства имен по нескольким заголовочным и исходным файлам. Такая открытость заодно упрощает и переход к использованию пространств имен. Например, текст

```
// мой заголовочный файл:
extern void f(); // моя функция
// ...
```

```
#include <stdio.h>
extern int g();  // моя функция
// ...
```

можно переписать, не меняя порядок объявлений:

```
// мой заголовочный файл:

namespace Mine {
   void f();  // моя функция
   // ...
}

#include <stdio.h>

namespace Mine {
   int g();      // моя функция
   // ...
}
```

Многие, в том числе и я, предпочитают использовать несколько маленьких пространств имен, вместо того чтобы помещать крупные фрагменты кода в одно пространство. Подобный стиль можно навязать, если требовать, чтобы все члены входили в единственное объявление пространства имен точно так же, как в одном объявлении были бы объявлены все члены класса. Но я не видел смысла приносить многие мелкие удобства, которые сопутствуют открытым пространствам имен, в жертву ограничительной системе только для того, чтобы угодить преобладающим на данный момент вкусам.

17.5. Классы и пространства имен

Не думаю, что однажды поступившее предложение сделать пространство имен разновидностью класса – хорошая мысль, поскольку многие возможности, предоставляемые классами, лишь поддерживают концепцию определенного пользователем типа данных. Например, средства создания объектов и манипулирования ими имеют мало общего с областями действия.

Противоположная точка зрения – класс является частным случаем пространства имен – почти не вызывает сомнений. Класс действительно есть пространство имен в том смысле, что все операции, поддерживаемые для пространств имен, применимы с той же семантикой и к классам, если только конкретная операция для классов явно не запрещена. Это позволяет достичь простоты и единства языка, одновременно сводя к минимуму затраты на реализацию. Такой взгляд подтверждается легкостью, с которой пространства имен вошли в C++, и тем, как естественно с их помощью решаются давно стоящие и, казалось бы, не связанные между собой проблемы.

17.5.1. Производные классы

Рассмотрим известную проблему, связанную с тем, что член производного класса скрывает член базового класса с тем же именем:

```
class B {
public:
   f(char);
};

class D : public B {
public:
   f(int);       // скрывает f(char)
};

void f(D& d)
{
   d.f('c'); // вызывается D::f(int)
}
```

Разумеется, появление пространств имен не изменяет семантики таких примеров. Но возможна новая интерпретация: поскольку `D` – это класс, его область действия представляет собой пространство имен. Пространство имен `D` вложено в `B`, поэтому `D::f(int)` скрывает `B::f(char)`. Следовательно, вызывается `D::f(int)`. Если такое разрешение имени вас не устраивает, можете воспользоваться using-объявлением, чтобы включить `f()` из `B` в область действия:

```
class B {
public:
   f(char);
};

class D : public B {
public:
   f(int);       // ввести B::f в D, разрешив перегрузку
   using B::f;
};

void f(D& d)
{
   d.f('c'); // вызывается D::f(char)
}
```

Как обычно, имена из двух базовых классов, которым множественно наследует производный класс, могут приводить к неоднозначности (независимо от того, что обозначают имена):

```
struct A { void f(int); };
struct B { void f(double); };

struct C : A, B {
   void g() {
      f(1);      // ошибка: A::f(int) или B::f(double)
      f(1.0);    // ошибка: A::f(int) или B::f(double)
   }
};
```

Но теперь для разрешения таких неоднозначностей мы можем добавить пару using-объявлений, чтобы ввести `A::f` и `B::f` в область действия `C`:

```
struct C : A, B {
   using A::f;
   using B::f;

   void g() {
      f(1);      // A::f(1)
      f(1.0);    // B::f(1.0)
   }
};
```

Явный механизм такого рода неоднократно предлагался на протяжении нескольких лет. Я помню, как мы обсуждали эту тему с Джонатаном Шопиро во время работы над версией 2.0, но отказались от ее реализации, поскольку механизм казался слишком специализированным и узким. С другой стороны, using-объявления – общий механизм, который по счастливой случайности решил и эту проблему.

17.5.2. Использование базовых классов

Во избежание путаницы using-объявление, являющееся членом класса, должно именовать член базового (непосредственного или косвенного) класса. Чтобы не было проблем с правилом доминирования (см. раздел 12.3.1), использование using-директив в качестве членов классов запрещено.

```
struct D : public A {
   using namespace A;   // ошибка: using-директива в качестве члена
   using ::f;           // ошибка: ::f - не член базового класса
};
```

Using-объявление, именующее член базового класса, играет важную роль для повышения единообразия доступа:

```
class B {
public:
   f(char);
};

class D : private B {
public:
   using B::f;
};
```

Это более общий и понятный способ добиться того же, что раньше делалось с помощью объявлений доступа (см. раздел 2.10):

```
class D : private B {
public:
   B::f;  // старый способ: объявление доступа
};
```

Using-объявление делает такое специализированное объявление доступа излишним. Поэтому объявления доступа провозглашены устаревшими, следовательно, в будущем, когда пользователи перепишут старые тексты, объявления доступа будут исключены из языка.

17.5.3. Исключение глобальных статических объявлений

Часто бывает полезно поместить набор объявлений в пространство имен, просто чтобы не смешивать их с объявлениями в заголовочных файлах или с глобальными объявлениями в других единицах трансляции. Например:

```
#include <header.h>
namespace Mine {
   int a;
   void f() { /* ... */ }
   int g() { /* ... */ }
}
```

Во многих случаях неважно, как именно называется пространство имен, лишь бы его имя не конфликтовало с именами других пространств. Чтобы справиться с этой проблемой более корректно, разрешим использовать безымянное пространство имен:

```
#include <header.h>
namespace {
   int a;
   void f() { /* ... */ }
   int g() { /* ... */ }
}
```

Если не считать перегрузки имен из заголовочного файла, это можно эквивалентно переписать следующим образом:

```
#include <header.h>

static int a;
static void f() { /* ... */ }
static int g() { /* ... */ }
```

Обычно такая перегрузка нежелательна, но все-таки ее можно обеспечить:

```
namespace {
#include <header.h>
   int a;
   void f() { /* ... */ }
   int g() { /* ... */ }
}
```

Таким образом, пространства имен позволяют объявить устаревшим использование ключевого слова `static` для управления видимостью глобальных имен. Это оставляет за `static` единственную семантику: статически распределенная, недублирующаяся переменная.

Безымянное пространство имен ничем не отличается от любого другого, только не нужно писать его имя. По сути дела,

```
namespace { /* ... */ }
```

семантически эквивалентно

```
namespace unique_name { /* ... */ }
using namespace unique_name;
```

Все безымянные пространства имен в одной и той же области действия имеют одно и то же уникальное имя. В частности, все глобальные безымянные пространства имен в одной единице трансляции являются частями одного и того же пространства имен и отличаются от глобальных безымянных пространств имен в других единицах.

17.6. Совместимость с C

Функцию с C-компоновкой можно поместить в пространство имен:

```
namespace X {
   extern "C" void f(int);
   void g(int);
}
```

Это позволяет использовать функции с C-компоновкой точно так же, как любые другие члены пространства имен. Например:

```
void h()
{
   X::f();
   X::g();
}
```

Однако в одной программе не может быть двух разных функций с C-компоновкой, имеющих одно и то же имя, даже если они находятся в разных пространствах имен. Небезопасные правила компоновки, принятые в C, затрудняют нахождение такой ошибки.

Можно было бы вообще запретить помещение в пространство имен функций с C-компоновкой. Но тогда люди, которым нужен интерфейс с C, совсем не стали бы пользоваться пространствами имен, а предпочли бы засорять глобальное пространство. Так что это не выход.

Другая альтернатива – гарантировать, что две функции с одним и те же именем, но в разных пространствах имен, на самом деле оказываются различными, даже если имеют C-компоновку. Например:

```
namespace X {
   extern "C" void f(int);
}

namespace Y {
   extern "C" void f(int);
}
```

Проблема состоит в том, как вызвать такую функцию из программы на C. Поскольку в языке C нет механизма разрешения неоднозначности на основе пространств имен, пришлось бы полагаться на соглашение об именовании (почти наверняка зависящее от реализации). Например, программа могла бы вызывать функции `__X__f` и `__Y__f`. Такое решение было признано неприемлемым, поэтому приходится смириться с небезопасными правилами C. Язык C засоряет пространство имен компоновщика, но не глобальные пространства имен единицы трансляции C++.

Обратите внимание, что это проблема C, а не пространств имен C++. Связывание с программой на языке, где есть механизм, аналогичный пространствам имен в C++, реализуется простым и безопасным способом. Например, я считаю, что

```
namespace X {
   extern "Ada" void f(int);
}

namespace Y {
   extern "Ada" void f(int);
}
```

это естественный способ обратиться из C++-программы к функциям в разных пакетах языка Ada.

Глава 18. Препроцессор C

Cpp должен быть разрушен.

Почти Катон Старший

Среди средств, приемов и идей, которые C++ унаследовал от C, был и препроцессор Cpp. Ориентация препроцессора на обработку файлов принципиально не согласуется с языком программирования, в основе которого лежат области действия, типы и интерфейсы. Рассмотрим на первый взгляд безобидный фрагмент:

```
#include <stdio.h>
extern double sqrt(double);

main()
{
   printf("Квадратный корень из 2 равен %g\n", sqrt(2));
   fflush(stdout);
   return(0);
}
```

Очевидно, что он печатает следующее:

```
Квадратный корень из 2 равен 1.41421
```

Это кажется вполне разумным, но я откомпилировал строку таким образом:

```
cc -Dsqrt=rand -Dreturn=abort
```

после чего было напечатано

```
Квадратный корень из 2 равен 7.82997e+28
abort - core dumped
```

и программа завершилась с дампом памяти.

Пожалуй, это чересчур необычный пример и вряд ли вы станете использовать флаги компилятора для определения макросов Cpp, но суть дела от этого не меняется. Макроопределения могут быть скрыты в среде разработки, в директивах компилятора и в заголовочных файлах. Макроподстановки игнорируют границы областей действия и могут даже изменять структуру области действия, вставляя фигурные скобки, кавычки и т.д. Это позволяет программисту менять текст, видимый компилятору, не трогая ни единой строки кода. Конечно, иногда даже самые экстремальные способы использования Cpp могут оказаться полезными, но его средства до такой степени не структурированы, что являются источниками постоянных трудностей для программистов, отвечающих за разработку, сопровождение

и перенос кода на другие платформы, а также для авторов инструментальных средств.

Быть может, худшей из всех особенностей Cpp является то, что он сдерживал создание сред программирования на C. Невозможность управлять работой Cpp, в основе которого лежит посимвольный разбор файла, делает нетривиальные инструментальные средства для C и C++ более громоздкими, медленными и менее изящными, чем они могли бы быть.

Cpp нельзя даже назвать хорошим макропроцессором. Поэтому я поставил себе целью изжить Cpp. Но задача оказалась труднее, чем представлялось вначале. Cpp, возможно, и неудачен, но трудно найти ему лучше структурированную и более эффективную замену.

У препроцессора C есть четыре основных директивы [1]:

- `#include` копирует исходный текст из другого файла;
- `#define` определяет макрос (с аргументами или без);
- `#ifdef` включает последующие строки в зависимости от условия;
- `#pragma` влияет на поведение компилятора способом, зависящим от реализации.

Эти директивы применяются для решения различных задач программирования:

`#include`
- сделать доступным определение интерфейса;
- составить исходный текст из различных частей.

`#define`
- определить:
  - символические константы;
  - открытые подпрограммы;
  - обобщенные подпрограммы;
  - обобщенные типы;
- переименовать;
- связать строки;
- определить специализированный синтаксис;
- сделать общую макрообработку.

`#ifdef`
- осуществить контроль версий;
- произвести комментирование кода.

`#pragma`
- осуществить управление размещением структур данных в памяти;
- информировать компилятор о необычном потоке управления.

Все эти задачи решаются с помощью Cpp довольно неудачно, с использованием непрямых методов, но тем не менее эффективно, и зачастую для практических

[1] Директивы `#if`, `#line` и `#undef` тоже важны, но в контексте данного обсуждения значения не имеют.

целей такой работы достаточно. Важнее то, что Cpp имеется везде, где есть C. Это хорошо известно и часто оказывается полезнее, чем применение гораздо лучшего, но менее распространенного макропроцессора. Данный аспект важен еще и потому, что препроцессор C часто используется для решения задач, не имеющих ничего общего с языком C.

В C++ имеются альтернативы для большинства основных применений директивы `#define`:

- модификатор `const` для определения констант (см. раздел 3.8);
- модификатор `inline` для открытых подпрограмм (см. раздел 2.4.1);
- шаблоны для функций, параметризованных типами (см. раздел 15.6);
- шаблоны для параметризованных типов (см. раздел 15.3);
- пространства имен для обобщения именования (см. главу 17).

Альтернативы директиве `#include` в C++ нет, хотя пространства имен предоставляют механизм областей действия, который поддерживает композицию таким образом, что поведение `#include` становится более структурированным.

Я предлагал добавить директиву `include` в сам язык C++ в качестве замены одноименной директиве Cpp. Директива `include` отличалась бы от препроцессорной следующим:

- если файл включен дважды, то второй `include` игнорируется. Это устраняет часто встречающуюся проблему, которая сейчас решается искусственно и неэффективно с помощью директив `#define` и `#ifdef`;
- макрос, определенный вне включенного с помощью `include` текста, не расширяется внутри этого текста, что обеспечивает защиту информации от случайного воздействия макросов;
- макрос, определенный внутри включенного с помощью `include` текста, не расширяется в тексте, который обрабатывается после включенного. Тем самым гарантируется, что макросы во включенном тексте не вводят зависимость от порядка в ту единицу трансляции, которая включает файл. Также это защищает от сюрпризов, которые могут преподнести макросы.

Такой механизм был бы очень полезен в системах, где осуществляется предварительная компиляция заголовочных файлов, да и вообще для тех программистов, которые составляют программы из независимо разработанных частей. Хочу, впрочем, отметить, что это только идея, а не одобренное средство.

Остаются директивы `#ifdef` и `#pragma`. Без последней можно вполне обойтись, поскольку я ни разу еще не встречал понравившейся мне прагмы. Слишком часто `#pragma` используется для того, чтобы украдкой подменить семантику языка и внедрить чрезмерно специализированные расширения, к тому же с нелепым синтаксисом. Для директивы `#ifdef` у нас пока нет подходящей замены. В частности, использование предложений `if` и константных выражений – это неполное решение. Например:

```
const C = 1;

// ...
```

```
if (C) {
   // ...
}
```

Этот прием не годится для управления объявлениями, а предложение `if` должно быть синтаксически корректным даже в том случае, если некоторая его ветвь никогда не будет исполняться.

Мне бы хотелось застать время, когда Cpp не будет использоваться. Однако единственный реальный и ответственный путь для достижения этой цели – сначала сделать Cpp ненужным, потом убедить людей пользоваться более удачной альтернативой и только после этого – через много-много лет – отправить Cpp в среды разработки, где ему и место наряду с остальными внеязыковыми средствами.

Алфавитный указатель

А

Абстрактные
  класс 267, 284
    и библиотеки 191
    и шаблоны 385
  тип 286
Автоматические
  инстанцирование шаблона 369
  память 41
Автопрототипирование 50
Аппаратура, специализированная 144
Аргументы
  именованные 164
  ограничения на аргументы шаблонов 348
  ослабление правил 301
  по умолчанию 68
  правила соответствия 235
  проверка во время выполнения 302
  шаблонов
    выведение 354
    зависимость 375
    не являющиеся типами 347
    функций 355
Асинхронные события 398
Ассемблер 133
Ассоциативный массив 328
  стандартный 202

Б

Базовая библиотека 199
Базовый класс 58
  виртуальный 265
    размещение в памяти 272
  доступ 307
  и производный 84
  инициализация 282
  приведение
    из виртуального базового класса 318
    к закрытому базовому классу 335
  сокрытие членов 421
Базы данных 200
Безопасность 138
  и исключения 389
  и совместимость 340
Безопасные
  преобразование 234
  приведение типов 315
Библиотеки 191
  complex 76
  базовые 199
  в стиле Smalltalk 191
  для поддержки
    многозадачности 192, 196
    параллельности 196
    устойчивости объектов 200
  для численных расчетов 200
  и RTTI 190
  и языковые средства 189
  инициализация 107
  компонентов Буча 199
  поставщики и пространства имен 407
  потокового ввода/вывода 193
  проектирование 190
    и шаблоны 383
  специализированные 200
  стандартные 201
  стандартных компонентов 192
  языковая поддержка 190
Блокировка 198
Булев тип bool 261
Буч
  библиотека компонентов 199
  компоненты 363, 385

В

Ввод/вывод 193
  в языке C 107
  объектов 327
  символов из расширенных наборов 172
Вектор
  стандартный 202
  шаблон 347

Версии C++ 80
Видимость и доступ 64
Виртуальные функции 82
  вызов в конструкторах 288
  деструктор 223
  и множественное наследование 271
  и шаблоны 347
  копирование 248
  оптимизация 246
    вызова 128
  память 109
  перекомпиляции 86
  реализация 86
  эффективность 58, 86
Вложенные
  класс 113, 140
    локальность 130
  функция 162
Возведение в степень 254
Возвращаемое значение
  в виде ссылки 98
  оператором new() 225
  оптимизация 308
  тип 300
Возобновление 394
Временные объекты 153
Время выполнения
  инициализация 107, 292, 332
  информация о типе 312
    и библиотеки 191
  поддержка в C with Classes 58
  эффективность 41, 330
Вспомогательный класс 98
Встроенные
  системы 210
  типы 385
Выбор языка программирования 184
Выведение аргументов шаблона 354
Вызовы
  виртуальной функции в конструкторе 288
  необъявленной функции 50
  нотация вызова конструктора 342
  по значению 97
  по ссылке 97
  соглашения 145, 243

Г

Глобальные
  имя 294
    статическое устаревшее 424
  область действия 416
  переменная 41
Группировка исключений 390, 400

Д

Двоичный литерал 162
Двойная диспетчеризация 306
  с помощью шаблонов 368
Декремент, оператор -- 252
Делегирование
  и множественное наследование 277
  и оператор -> 249
Деструкторы 105
  delete() 66
  виртуальный 223
  для встроенного типа 385
  и delete 220
  и исключения 393
  и сборка мусора 229
  явный вызов 224
Диалекты 111, 141
Диграфы 171
Динамические
  инициализация 108, 292
  контроль исключений 400
  память 41
  связывание 214
Директива инстанцирования шаблона 371
Доступ
  и видимость 63
  к базовому классу 308
  контроль 40, 307
    в конструкторах 105
    и статические члены 295
    имен 63
  предоставление 63
Дружественные функции
  и инкапсуляция 63
  и члены 92

Е

Единицы
  защиты 64
  трансляции 67

З

Завершение 394
Зависимость
  от аргумента шаблона 375
  от библиотек 145
  от реализации 144
Заголовочный файл 43, 214, 242
  предварительно откомпилированный 429
Закрытое наследование реализации 62
Закрытый базовый класс 307

Запрет
  копирования 244
  наследования 245
  размещения 244
Затраты
  во время выполнения 240
  множественного наследования 276
Защита
  в Smalltalk 307
  класс как единица 64
  модель 62
  модель C++ 33
Защищенный член 307

И

Идентификатор типа 322
Идентификация типа 85, 312
Иерархия классов 30
Именованные аргументы 164
Имя
  аргумента 166
  глобальное 294
  кодирование типа 241
  контроль доступа 64
  конфликт
    и глобальная область действия 404
    и множественное наследование 280
  правила привязки 373
  префиксы 405
  пространства имен 412
  разрешение 149
    в шаблонах 376
  сокрытие 87
  схема кодирования 145
  эквивалентность 45
Инициализатор =0 285
Инициализация 47
  cin 107
  cout 107
  stdin 107
  stdout 107
  библиотек 107
  динамическая 108, 292
  и виртуальная память 109
  и выделение памяти 219
  и присваивание 33
  константного члена 162
  контролируемая 332
  порядок 107, 282
  синтаксис 385
  статическая 106, 292
Инкапсуляция и дружественные функции 63
Инкремент, оператор ++ 252
Инстанцирование 370
  позднее 351
Инструментальные средства
  C++ 75
  для проектирования языка 114
  и Cpp 428
  критерии 32
  семантически-ориентированные 214
  символьно-ориентированные 214
  синтаксически-ориентированные 214
  специализированные 212
Интегрированная система 123
Интерфейс 40
  и приведение типов 191
  и реализация 287
  и реализация для шаблонов 384
  использование const 101
  отдельный 44
  открытое наследование 63
  с другим языком 243
  составление 413
Исключения
  гарантии перехвата 388
  группировка 390, 400
  динамическая проверка 400
  затраты 388
  и C 389
  и библиотеки 191
  и большие системы 396
  и деструкторы 393
  и иерархия классов 391
  и конструкторы 391
  и нехватка памяти 394
  и переполнение 398
  и повторная компиляция 400
  и размещение объекта в памяти 401
  и старый код 398
  и управление ресурсами 391
  и уровни абстракции 397
  и шаблоны 385
  и эффективность 401
  обработка 119, 387
  распространение на несколько уровней 399
  синтаксис 389
  спецификация 400
  статическая проверка 399
Использование 328
  dynamic_cast 303, 327
  RTTI 317, 319
  множественного наследования 278, 328
  новых приведений типов 342
  пространств имен 408, 411
  шаблонов 329

К

Квалификация 105
  явная 410
Класс
  абстрактный 267, 284
  базовый 58
    и производный 84
  в языке Simula 32, 53
  виртуальный базовый 266
  вложенный 140, 295
    опережающее объявление 296
  вспомогательный 98
  доступ к базовому 307
  зависимость от порядка объявления членов 149
  и struct 86
  и пространство имен 422
  иерархия 30
  иерархия и перегрузка 233
  как единица защиты 64
  как определенный пользователем тип 39
  контейнерный 343
  концепция 40
  корневой 267
  модель размещения в памяти 61
  опережающая ссылка на член 151
  производный 58, 307
  универсальный корневой 191
  член 295
  шаблон 346, 365
  шаблон и абстрактный 385
Ключевое слово
  and 171
  and_eq 171
  bitand 171
  bitor 171
  compl 171
  instantiate 371
  not 171
  not_eq 171
  or 171
  or_eq 171
  postfix 252
  prefix 252
  specialise 380
  template 346
  xor 171
  xor_eq 171
Код
  генерация 308
  дублирование 351
Кодирование
  низкоуровневое 343
  типа 241
Комбинирование методов 274
Комментарий // 53, 104
Компилятор 177
  встраивание функций 42
  неполного цикла Cfront 76
  предупреждения 51, 342
Компиляция
  однопроходная 151
  раздельная 32, 43
Композиция шаблонов 361
Компоновка (связывание)
  в C 239
  затраты 240
  и перегрузка 239
  и указатели на функции 243
  модель 43
  опыт реализации 241
  программ на C и C++ 239
  производительность и шаблоны 370
  с программами
    на Fortran, Pascal, PL/I 243
    на других языках 43
  типобезопасная 44, 239
    синтаксис 403
Конструктор 104
  вызов виртуальной функции 288
  для встроенного типа 385
  и библиотеки 191
  и исключения 393
  и оператор new 219
  и распределение памяти 66
  и шаблоны 385
  контроль доступа 105
  копирующий 247
  нотация для вызова 342
  по умолчанию 68, 386
Контейнеры 202
Контекст 374
Контроль 401
  доступа 40, 307
  исключений 400
  множественного наследования 269
  нехватки памяти 225
  ошибок в шаблонах 374
Конференции 175
Копирование
  виртуальная функция 248
  глубокое 248
  конструктор 247
  контроль допустимости 244
  объекта 299
  побитовое 246
  поверхностное 247

почленное 246
указателя 247
Корневой класс 192, 267
Критерии
разработки шаблонов 344
расширения C++ 159
Куча 41

Л

Литерал
двоичный 162
типа char 232
Локальность 129
вложенных классов 130
Локальные
переменные 40
по умолчанию 112
статические массивы в ANSI C 140

М

Макросы
Cpp 130, 427
вместо обобщенных типов 61
для имитации шаблонов 345
и контейнерные классы 343
Массив
ассоциативный 202, 328
локальный статический в ANSI C 140
освобождение памяти 224
распределение памяти 220
шаблон 347
Менеджер ресурсов 222
Метаобъект 330
Метод
без ограничений 332
комбинирование 274
Многопоточность 196
Многопроцессорные системы 196
Многоточие 340
Множественное наследование 119, 263, 265
в C++ 276
и Simula 264
и библиотеки 191, 264
и виртуальные функции 271
и делегирование 277
и конфликт имен 280
и неоднозначность 264
и обработка исключений 391
и размещение объекта в памяти 270
и сборка мусора 277
издержки 276
использование 278, 328
статический контроль 269
Модульность 113
Мультиметод 303, 331

Н

Набор команд 145
Надежность и обработка исключений 387
Наследование
delete() 220
new() 220
запрет 245
и перегрузка 232
и шаблоны 365
интерфейса 62
множественное 119
ограничения за счет 349
от встроенного типа 385
реализации 62
Неоднозначность 234
в шаблонах 376
множественного наследования 264
управление 233
Неявные
int 55
инстанцирования шаблонов 355
преобразования 90, 230
приведения типов 355
сужающие преобразования 340
Низкоуровневое
кодирование 343
программирование 133, 209
средства в C with Classes 38
Нотация
. и :: 106
вызова конструктора 342
конструктора 104
объявлений, линейная 55
приведения типов 333, 340
Нулевой указатель 236

О

Область действия
глобальная 417
и Cpp 427
конфликты имен 404
Объединение иерархий классов 280
Объект
в стеке 103
ввод/вывод 282
копирование 299
представление 40, 286
размещение в памяти 310
статический 103
частично сконструированный 393

Объектно-ориентированные
база данных 200
программирование 84, 182
и C++ 205
изучение 180
проектирование 125, 182
Объявление доступа 63
и using-объявление 424
Объявления
overload 238
в Algol68 109
в условиях 110
в циклах for 110
вложенного класса, опережающие 296
линейная нотация 55
реализация 44
синтаксис в C 54
специализация 379
Ограничения 348
Ограниченные указатели 168
Однопроходный синтаксический анализатор 149
Операторы
, 254
--, декремента 252
--, префиксный и постфиксный 252
**, возведение в степень 254
*^, возведение в степень 256
++, инкремента 252
++, префиксный и постфиксный 252
-> 248
->* 254
. 249
.* 254
<< 194
= 246
delete 41, 223
наследование 220
delete[] 221
new 41, 219
возвращаемое значение 225
наследование 220
new[] 221
закрытый 244
определенный пользователем 257
опроса типа 330
перегрузка 88
составной 258
сравнения типов 331
функция 93
эквивалентность 249
Опережающие
объявления вложенных классов 295
ссылки на члены 151
Определение
встроенного типа 385
формальное 114
функции 40
шаблон, контекст 374
языка C 20
Опрос типа во время компиляции 358
Оптимизация
виртуальных функций 246
возврата значений 308
глобальная 128
инстанцирования 373
таблицы виртуальных функций 308
Освобождение памяти
для массива 224
и оператор delete 219
Отказоустойчивые системы 389
Открытое наследование интерфейса 62
Отладка 51
Отладчик, стандартизация 145
Ошибки
и Cpp 428
контроль 383
обнаружение и ограничения 349
обработка 225
в C 343
многоуровневая 389
повторного определения 382
потенциальные 125
привязки имен в шаблонах 373

П

Память
автоматическая 41
арена 222
виртуальная и инициализация 109
динамическая 41
для объекта класса 41
использование в Cfront 76
куча 41
неформатированная 233
нехватка 225
и исключения 394
постоянная 292
свободная 41
статическая 41
стек 41
управление 101
специализированный распределитель 218
Параллельность 30, 36
поддержка 196
Параметризованный тип 118
Перегрузка 88
для нескольких аргументов 235
и const 232
и аргументы по умолчанию 69

и зависимость от порядка 233
и иерархия классов 233
и компоновка 239
и наследование 232
и преобразования 230
и пространства имен 419
и эффективность 89
на базе перечислений 261
оператора присваивания 67
операторов 88
разрешение 231
соответствие 235
членов базового
и производного классов 422
шаблона функции 357
Передача параметров 96
Переименование 280, 407
Переключение по типу 319
Переменные 206
глобальные 41
локальные 41
неинициализированные 110
Переносимость 32, 137
Переполнение и исключения 398
Перечисления
в C 259
перегрузка операторов 261
ПЗУ 292
Поверхностное копирование 247
Поддержка
в C with Classes 38
отладки 51
параллельности 196
построения библиотек 119, 190, 311
Подкласс
в Smalltalk 58
и суперкласс 84
Полиморфизм без виртуальных функций 59
Полиморфный тип 318
Порядок
зависимость 130
от порядка объявления членов 149
и перегрузка 233
инициализации 107
членов 282
конструирования 289
Постоянная память 292
Постфиксный
-- и префиксный 252
++ и префиксный 252
Правила 120
нулевых издержек 133
одного определения 46
переопределения 150
переписывания 150
пересечения 236
пересмотра 153
переупорядочения 153
поддержки проектирования 125
привязки имен 373
соответствия аргументов 235
специализации 380
Предоставление доступа 63
Представление
объекта 40
сокрытие 286
Предупреждения
в Cfront 52
компилятора 51, 340
Преобразования
безопасные 234
в void* 233
граф 234
естественные 234
и перегрузка 230
и совместимость с C 234
и шаблоны 367
неявные 91, 234
static_cast 336
сужающие 340
стандартные 235
сужающие 52, 234
типов с плавающей точкой в интегральные 52
функции 93
Препроцессор
#define 428
#ifdef 428
if как альтернатива 429
#include 428
include как альтернатива 429
#pragma 428
инстанцирования шаблона 371
Cpp 131, 427
Cpre для C with Classes 36
Прерывание 398
Префиксный
-- и постфиксный 252
++ и постфиксный 252
Префиксы имен 405
Приведение типов
безопасное 315
и const 339
и RTTI 314
и игнорирование const 291
и интерфейс 191
и неполные типы 336
и указатель на функцию 338
из void* 340

к закрытому базовому классу 335
неявное 356
синтаксис 315
Присваивание 246
void * 237
и инициализация 33
перегрузка 67
указателю this 102, 219
Программа
корректная 144
начальная загрузка 108
Программирование 209
объектно-ориентированное 84, 182
и C++ 205
Проектирование
C++ 70
RTTI 329
библиотеки 189
и проверка типов 118
объектно-ориентированное 126, 182
отказоустойчивых систем 389
правила поддержки 125
пространств имен 406
с использованием C++ 181
уточнений шаблонов 345
шаблонов 344
и библиотек 383
языка 125
Производительность
Simula 31, 41
начальной загрузки программы 108
оператора new 218
Производный класс 58, 307
и базовый 84
и перегрузка членов базового класса 422
размещение объекта в памяти 61
сокрытие членов 421
Пространства имен 295, 408
безымянное 424
в стандартной библиотеке 404
вложенное 419
глобальное 403, 417
и библиотеки 191
и класс 422
и кодирование имен 425
и перегрузка 418
и совместимость с C 425
и управление версиями 413
как аргумент шаблона 348
открытое 420
псевдонимы 412
реализация 407
синтаксис 409
шаблоны 365
Прототип 50

P

Размещение в памяти 145, 221
и RTTI 327
и виртуальные функции 85
и исключения 401
объекта
в C with Classes 48
виртуального базового класса 272
производного класса 61
при множественном наследовании 270
совместимость с C 37
таблицы виртуальных функций в Cfront 327
Разрешение
имен 150
в шаблонах 376
перегрузки 231
Распределение памяти
для массивов 220
запрет размещения 244
в свободной памяти 245
и инициализация 219
и конструктор 66
и оператор new 219
и системы реального времени 218
Расширения
архитектурно-зависимые 169
и поддержка параллельности 196
и стабильность 137
и стандарты 145
предлагавшиеся 161
принятые 161
распознаваемые 145
Расширенная информация о типе 324
Расширенный набор символов 172
Реализация
C with Classes 38
Simula 31
виртуальной функции 86
закрытого наследования 63
и интерфейс 287
объявлений 44
оператора new в Cfront 66
переносимая 32
пространства имен 407
системы и языка 214
функции-члена 48
шаблонов в Cfront 370
Рекурсивный спуск 79
Репозитарий для шаблонов 381
Ресурс
захват как инициализация 393
требования Cfront 75
управление и исключения 391

С

Сборка мусора 154, 210
автоматическая 226
и деструктор 229
и множественное наследование 277
необязательная 206, 226
специализированная 224
стандартизация 228
Семантика
возобновления 394
завершения 394
ссылки 206
указателя 206
Сигналы 398
Символы 169
ограничения на число 80
Симулятор распределенной системы 30
Синтаксис 131
>> 361
задания компоновки 403
избыточность 390
инициализации 385
обработки исключений 389
объявлений в C 54
приведений типов 315
пространств имен 409
указателя 310
шаблонов 355
Система
встроенная 210
и реализация языка 214
интегрированная 123
многопроцессорная 196
отказоустойчивая, проектирование 389
смешанная 212
Совместимость
C и C with Classes 47
C и C++ 132, 141, 142, 232, 236, 247
и безопасность 340
с компоновщиками 132
Соглашения о вызове 243
Сокрытие
и замещение 87
имен 87
представления 286
реализации шаблона 370
членов базового класса 87, 421
Соответствие 235
Сортировка 363
Составление
интерфейсов 413
программ 127
Специализация 378
Специализированные
аппаратура 144
библиотеки 201
инструментальные средства 211
язык 209, 211
Спецификации
исключений 400
компоновки 241
Списки 202
Среда
выполнения 227
отделение от языка 205
программирования на C++ 178
разработки C++ 178
стандартная для приложений 146
Ссылка 96
THIS в Simula 49
в Algol68 96
возврат 98
и указатель 96
константная 97
перегрузка в Algol68 53
привязка 96
семантика 206
«умная» 249
Стандарт 140
Стандартизация 145
ISO C++ 140
RTTI 321
Стандартные
алгоритмы 201
библиотека 411
ANSI/ISO 201
и пространство имен 404
вектор 202
итераторы 202
контейнеры 202
набор команд 145
преобразования 235
сборщик мусора 228
соглашения о вызове 145
среды для приложений 146
Статические
инициализация 106, 292
контроль
множественного наследования 269
типов 40, 117, 330
объект 103
память 41
проверка исключений 399
распределение памяти, запрет 244
система типов 128
функция-член 295
Стек 41, 103

Структура
Cfront 76
тэг 56
Сужающее преобразование 50, 234
неявное 340
Суперкласс 84

Т

Таблица виртуальных функций
дублирование 308
оптимизация 308
размещение в памяти 326
Типы
bool 261
абстрактный 286
безопасность 103
встроенный 385
и определенный пользователем 41, 129
идентификация 322
информация во время исполнения 312
кодирование 241
контроль на этапе компиляции 40
литерал типа char 232
нарушения системы типов 128
параметризованный 118, 346
перечисления 259
полиморфный 318
расширенная информация о 324
ссылочный при возврате 98
статический и динамический контроль 118
статический контроль 205
шаблон как параметризованный тип 344
явное поле 59
Типобезопасное связывание 241
Точка инстанцирования 372
Триграфы 170

У

Указатели 248
this 49
и не-указатели 41
и ссылки 96
ограниченные 168
копирование 247
на функцию 85, 338, 243
на член 310
нулевой 236
синтаксис 310
«умный» 366
Универсальный корневой класс 192
Уничожение временных объектов 155
Условные выражения в шаблонах 358
Устаревшие средства
глобальные статические объявления 424
неявный int 55
объявления доступа 424

Ф

Файл
.c и шаблоны 381
.h и шаблоны 381
заголовочный 44, 214, 242
исходный 214
Фрагментация 218
Функции 162
:after и return() 67, 274
:before и call() 67, 274
new() 40
виртуальные 82
и множественное наследование 271
и шаблоны 347
и модель размещения объекта в памяти 85
оптимизация 246
перекомпиляция 86
реализация 86
эффективность 58, 86
встраиваемая (inline) 41
константная функция-член 291
обратного вызова 310
операторная 93
определение 40
переходник 251
преобразования 93
указатель и приведение типов 337
член 39
статическая функция-член 295
шаблон 353
аргументы 355
перегрузка 357

Ч

Численные расчеты 169, 211
библиотеки 200
Член
зависимости от порядка объявления 149
защищенный 307
и дружественные функции 92
инициализация константного 162
класса 295
константная функция 291
опережающее объявление 151
порядок инициализации 282
реализация функции-члена 48
сокрытие членов базового класса 87

статический 295
указатели 309
функция 39
шаблон 368

Ш

Шаблон
complex 367
аргументы, не являющиеся типами 347
в Cfront 345
вектора 347
виртуальный член 369
вложенный 368
выведение аргументов 354
двойная диспетчеризация 368
директива инстанцирования 371
зависимость от аргумента 375
заимствование имен 377
и .c-файлы 381
и .h-файлы 381
и Cpp 429
и typedef 362
и абстрактный класс 385
и виртуальные функции 347
и встраивание 347
и встроенные типы 385
и исключения 385
и исходный код 381
и компоновка 370
и конструктор 385
и контейнерный класс 343
и макросы 345
и наследование 365
и правило одного определения 374
и преобразования 367
и проектирование библиотек 383
и пространство имен 365
и размещение объекта в памяти 347
и системы контроля версий 370
и этап компиляции 370
инстанцирование 371
использование 329
как параметризованный тип 344
класса 365
контроль ошибок 374
критерии проектирования 344
массива 347
методы композиции 361
неоднозначности 376
ограничения на аргументы 348
пространство имен
в качестве аргумента 348
разрешение имен 376
реализация в Cfront 370
репозитарий 381
синтаксис 355
сокрытие реализации 370
специализация 378
условные выражения в 358
функции 353
аргументы 355
перегрузка 357
шаблоны в качестве аргументов 348
явные аргументы 355
Шлюз и указатель this 271

Э

Эквивалентность
имен 45
операторов 249
структур 45
Эффективность
ввода/вывода 193
виртуальных функций 58, 86
во время выполнения 42, 330
действий над матрицами 259
и гибкость 384
и исключения 401
и перегрузка 94

Я

Явные
квалификация 410, 417
задание аргумента шаблона 355
инстанцирование шаблона 371
поле типа 59
вызов деструктора 224
Язык
и библиотеки 189
и система 46
инструменты для проектирования 114
интерфейс с другими языками 243
поддержка построения библиотек 190
принципы и правила проектирования 120
специализированный 209, 211
средства и приемы программирования 180

A

Ada 21, 53, 77, 116, 118, 185, 344
Algol 22, 33, 49, 95, 111, 116, 119
and, ключевое слово 171
and_eq, ключевое слово 171
ANSI
C 79, 140
стандартная библиотека 201

ANSI/ISO
правила разрешения имен 152
ARM 135, 182
изучение C++ 139
правила разрешения имен 150
справочное руководство 138
Assert() 402

B

bad_cast 202
bad_typeid 202
basic_string 364
BCPL 32, 74
bitand, ключевое слово 171
bitor, ключевое слово 171
bits<N> 202
bitstring 202
BLAS 169
bool, булев тип 261
Buffer 347, 354

C

C 33, 74
ввод/вывод 107
диалекты 111
и C++ 111, 185, 209
и Simula 16
и исключения 389
и модификатор const 100
обработка ошибок 343
переменные 206
переносимость 53
перечисления 259
препроцессор 78, 131, 427
расширения 157
связывание 239
сгенерированный код 49
семантика 114
синтаксис объявлений 54
совместимость 234, 247
функция printf 193
численные приложения 168
C with Classes 36
C++
библиотеки 75
дизайн 70
и Ada 200
и C 111, 185, 209
и Fortran 185
и Simula 117
и Smalltalk 117
и абстракции данных 205
и ассемблер 133
инструментальные средства 75
контроль типов 103
множественное наследование 278
модель защиты 33
потоковый ввод/вывод 193
проектирование 181
синтаксический анализ 79
среда программирования 177
стандарт ANSI 140
стандартизация в ISO 141
статический контроль типов 205
call() и :before 67, 274
catch 389
cerr 194
Cfront 136
char
и int 231
и перегрузка 230
константы типа 232
char_ref 99
cin, инициализация 107
clone() 299
CLOS 274, 278, 305
Clu 21, 53, 116, 118, 401
CMP 363
compl, ключевое слово 171
complex 89, 202, 375
библиотека 76
шаблон 367
const 33
в C и C++ 100
и static_cast 336
и перегрузка 232
инициализация 292
константные ссылки 97
константные функции-члены 291
const_cast 339
constraints 351
Container, шаблон 321
Controlled_container 363
convert() 356
copy() 87
cout 195
CPL 74
Cpp 427
Cpre 80

D

delete
и деструктор 220
и освобождение памяти 219
и функция free() 67
оператор 41, 219, 223

delete(), деструктор 66
delete[] 221
double и float 231
dynamic_cast 313
  и static_cast 336
  применение 303, 326
  синтаксис 315
dynarray 202

E

EBCDIC, IBM 171
Eiffel 21, 77, 185, 210, 332, 402
explicit 91
extern C 241

F

f(void) 50
false 261
float
  и double 231
  и перегрузка 231
fopen() 392
Fortran 43, 132, 211
free(), delete 67

G

GNU
  библиотека 200
  компилятор 177

H

Hchar 172
histogram 72
Hstring 172

I

inherited, ключевое слово 297
inline
  и Cpp 429
  и шаблоны 347
  ключевое слово 43
  функция 41
  функция-член 151
INSPECT 85, 315
instantiate, ключевое слово 371
int
  и char 231
  неявный, устаревшее средство 55
  наследование 385
Interviews, библиотека 176, 199, 308
isKindOf, Smalltalk 331

J

Jchar 172
Jstring 172

L

link 60, 72, 265
Lisp 47, 77, 185
list 72
lvalue и rvalue 98

M

malloc(), и new 66
Map 328
Matherr 391
Matrix 258
ML 114, 116, 118, 391
Modula-2 44, 53, 114, 185, 212
Modula-3 21, 77, 187, 210
monitor 66
mutable 293

N

name() 323
narrow() 341
NCEG 168
NDEBUG 402
network_file_err 391
new
  Ada 369
  и malloc() 66, 102
  и конструктор 219
  и распределение памяти 219
  обработчик 226
  оператор 41, 219
  распределение памяти и конструктор 66
  реализация в Cfront 66
new()
  конструктор 65
  функция 40
new[], оператор 221
new_handler 102
NIH
  библиотека 176, 199
No_free_store 245
noalias 168
not, ключевое слово 171
not_eq, ключевое слово 171
NULL 237
Num 250

O

OLE2 214
OOPSLA 305
or, ключевое слово 171
or_eq, ключевое слово 171
OS/2 176, 396
overload 238

P

Pascal 146, 212
postfix, ключевое слово 252
prefix, ключевое слово 252
printf, ввод/вывод в C 193
Ptr 249, 366
pvector 352

Q

queue 72

R

raise 390
readonly 100
RefNum 250
reinterpret_cast 337
  и static_cast 338
restrict 168
return() и :after 67, 274
RHALE++, библиотека 200
RTTI 85, 312
rvalue и lvalue 98

S

Season 261
self, Smalltalk 49
set 286
set_new_handler 202, 226
Shape, intersect 303
signal 390
Simula 30, 41, 49, 72, 116
size_t 219
slist_set 286
Smalltalk 47, 53, 77, 185, 332
sort() 353
specialise, ключевое слово 380
stack 39
static_cast 335
  и dynamic_cast 336
  и reinterpret_cast 338
  и неявные преобразования 336
stdin 107
stdio.h 411
stdout 107
STL 202
String 92, 94, 97
strtok 232
struct и class 86
sum() 373

T

task 72, 264, 295
template, ключевое слово 346
terminate() 202
this
  присваивание 102, 219
  указатель 49
THIS, Simula 49
true 261
try 389
type_info 202, 323
typedef 56
  и шаблоны 362
typeid 322

U

unexpected() 202, 400
Unicode, набор символов 172
union 340
Usable 246
Usable_lock 246
Using-директива 408
Using-объявление 409

V

vec 59, 283
vector 59, 283
void* 219
volatile, ANSI C 140
vptr 85
vtbl 85

W

wchar_t 172
wordlink 60
writeonly 100
wstring 202

X

xor, ключевое слово 171
xor_eq, ключевое слово 171

Бьерн Страуструп

Дизайн и эволюция C++

Главный редактор	*Захаров И. М.*
Перевод с английского	*Слинкин А. А.*
Научный редактор	*Шалаев Н. В.*
Литературный редактор	*Петроградская А. В.*
Технический редактор	*Прока С. В.*
Верстка	*Сучкова Н. А.*
Графика	*Бахарев А.А.*
Дизайн обложки	*Антонов А. И.*

Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 28. Зак. №

Издательство «ДМК Пресс».

Отпечатано в полном соответствии
с качеством предоставленных диапозитивов
в ППП «Типография «Наука»
121099, Москва, Шубинский пер., 6.