

Охватывает C++11, C++14 и C++17



Шаблоны

C++

Справочник разработчика

ВТОРОЕ ИЗДАНИЕ

C++

Дэвид ВАНДЕВУРД

Николаи М. ДЖОСАТТИС

Дуглас ГРЕГОР



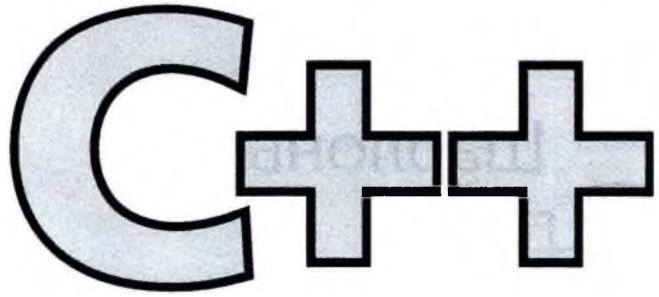


Шаблоны

C++

Справочник
разработчика

ВТОРОЕ ИЗДАНИЕ



Templates

The Complete Guide

SECOND EDITION

David VANDEVOORDE
Nicolai M. JOSUTTIS
Douglas GREGOR

◆ Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Шаблоны

C++

Справочник
разработчика

ВТОРОЕ ИЗДАНИЕ

Дэвид ВАНДЕВУРД
Николаи М. ДЖОСАТТИС
Дуглас ГРЕГОР



Москва • Санкт-Петербург
2018

ББК 32.973.26-018.2.75

В17

УДК 681.3.07

Компьютерное издательство "Диалектика"

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

Научный консультант канд. физ.-мат. наук *Е.А. Зуев*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:

info@dialektika.com, http://www.dialektika.com

Вандевурд, Дэвид, Джосаттис, Николаи М., Грегор, Дуглас.

B17 Шаблоны C++. Справочник разработчика, 2-е изд. : Пер. с англ. — Спб. : ООО "Альфа-книга", 2018. — 848 с. : ил. — Парал. тит. англ.

ISBN 978-5-9500296-8-4 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

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

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Inc., Copyright © 2018 Pearson Education, Inc.

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

Russian language edition published by Dialektika-Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2018

Научно-популярное издание

Дэвид Вандевурд, Николаи М. Джосаттис, Дуглас Грегор

Шаблоны C++. Справочник разработчика 2-е издание

Подписано в печать 14.05.2018. Формат 70x100/16

Гарнитура Times

Усл. печ. л. 68,37. Уч.-изд. л. 43,3

Тираж 400 экз. Заказ № 3994

Отпечатано в АО "Первая Образцовая типография"

Филиал "Чеховский Нечатный Двор"

142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО «Альфа-книга», 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-9500296-8-4 (рус.)

© Компьютерное издательство "Диалектика", 2018,

перевод, оформление, макетирование

© 2018 Pearson Education, Inc.

ISBN 978-0-321-71412-1 (англ.)

Оглавление

Предисловие	22
О книге	27
Часть I. Основы	35
Глава 1. Шаблоны функций	37
Глава 2. Шаблоны классов	57
Глава 3. Нетиповые параметры шаблонов	79
Глава 4. Вариативные шаблоны	89
Глава 5. Основы работы с шаблонами	101
Глава 6. Семантика перемещения и <code>enable_if<></code>	125
Глава 7. По значению или по ссылке?	141
Глава 8. Программирование времени компиляции	159
Глава 9. Применение шаблонов на практике	173
Глава 10. Основные термины в области шаблонов	189
Глава 11. Обобщенные библиотеки	197
Часть II. Углубленное изучение шаблонов	217
Глава 12. Вглубь шаблонов	219
Глава 13. Имена в шаблонах	261
Глава 14. Инстанцирование	293
Глава 15. Вывод аргументов шаблона	321
Глава 16. Специализация и перегрузка	379
Глава 17. Дальнейшее развитие	409
Часть III. Шаблоны и проектирование	425
Глава 18. Полиморфная мощь шаблонов	427
Глава 19. Реализация свойств типов	443
Глава 20. Перегрузка свойств типов	525
Глава 21. Шаблоны и наследование	551
Глава 22. Статический и динамический полиморфизм	581
Глава 23. Метапрограммирование	593
Глава 24. Списки типов	613
Глава 25. Кортежи	637
Глава 26. Контролируемые объединения	665
Глава 27. Шаблоны выражений	693
Глава 28. Отладка шаблонов	715
Приложение А. Правило одного определения	727
Приложение Б. Категории значений	737
Приложение В. Разрешение перегрузки	745
Приложение Г. Стандартные утилиты для работы с типами	761
Приложение Д. Концепты	803
Библиография	815
Глоссарий	821
Предметный указатель	833

Содержание

Предисловие	22
Благодарности ко второму изданию	23
Благодарности Дэвида ко второму изданию	23
Благодарности Нико ко второму изданию	24
Благодарности Дуга ко второму изданию	24
Благодарности к первому изданию	25
Благодарности Нико	26
Благодарности Дэвида	26
О книге	27
Что необходимо знать, приступая к чтению этой книги	28
Структура книги в целом	28
Как читать эту книгу	29
Некоторые замечания о стиле программирования	30
Стандарты C++11, C++14 и C++17	31
Примечание редактора перевода	33
Исходные тексты примеров и дополнительная информация	33
Обратная связь с авторами	34
Ждем ваших отзывов!	34
Часть I. Основы	35
Зачем нужны шаблоны	35
Глава 1. Шаблоны функций	37
1.1. Первое знакомство с шаблонами функций	37
1.1.1. Определение шаблона	37
1.1.2. Применение шаблонов	38
1.1.3. Двухэтапная трансляция	40
1.2. Вывод аргумента шаблона	41
1.3. Несколько параметров шаблона	43
1.3.1. Параметр шаблона для возвращаемого типа	44
1.3.2. Вывод возвращаемого типа	45
1.3.3. Возвращаемый тип как общий тип	47

	Содержание	7
1.4. Аргументы шаблона по умолчанию	47	
1.5. Перегрузка шаблонов функций	49	
1.6. А не должны ли мы?..	54	
1.6.1. Передача по значению или по ссылке	54	
1.6.2. Об использовании <code>inline</code>	55	
1.6.3. Об использовании <code>constexpr</code>	55	
1.7. Резюме	56	
Глава 2. Шаблоны классов	57	
2.1. Реализация шаблона класса <code>Stack</code>	57	
2.1.1. Объявление шаблона класса	58	
2.1.2. Реализация функций-членов	59	
2.2. Использование шаблона класса <code>Stack</code>	61	
2.3. Частичное использование шаблонов классов	62	
2.3.1. Концепты	63	
2.4. Друзья	64	
2.5. Специализации шаблонов классов	66	
2.6. Частичная специализация	67	
2.7. Аргументы шаблона класса по умолчанию	70	
2.8. Псевдонимы типов	71	
2.9. Вывод аргументов шаблона класса	74	
2.10. Шаблонизированные агрегаты	77	
2.11. Резюме	78	
Глава 3. Нетиповые параметры шаблонов	79	
3.1. Параметры шаблонов классов, не являющиеся типами	79	
3.2. Параметры шаблонов функций, не являющиеся типами	82	
3.3. Ограничения на параметры шаблонов, не являющиеся типами	83	
3.4. Тип параметра шаблона <code>auto</code>	84	
3.5. Резюме	88	
Глава 4. Вариативные шаблоны	89	
4.1. Шаблоны с переменным количеством аргументов	89	
4.1.1. Примеры вариативных шаблонов	89	
4.1.2. Перегрузка вариативных и невариативных шаблонов	90	
4.1.3. Оператор <code>sizeof...</code>	91	
4.2. Выражения свертки	92	
4.3. Применения шаблонов с переменным количеством аргументов	94	
4.4. Вариативные шаблоны классов и вариативные выражения	95	
4.4.1. Вариативные выражения	96	

4.4.2. Вариативные индексы	97
4.4.3. Вариативные шаблоны класса	97
4.4.4. Вариативные правила вывода	98
4.4.5. Вариативные базовые классы и <code>using</code>	99
4.5. Резюме	100

Глава 5. Основы работы с шаблонами 101

5.1. Ключевое слово <code>typename</code>	101
5.2. Инициализация нулем	102
5.3. Использование <code>this-></code>	104
5.4. Шаблоны для массивов и строковых литералов	105
5.5. Шаблоны-члены	108
5.5.1. Конструкция <code>.template</code>	114
5.5.2. Обобщенные лямбда-выражения и шаблоны членов	114
5.6. Шаблоны переменных	115
5.7. Шаблонные параметры шаблонов	118
5.8. Резюме	123

Глава 6. Семантика перемещения и `enable_if<>` 125

6.1. Прямая передача	125
6.2. Шаблоны специальных функций-членов	129
6.3. Отключение шаблонов с помощью <code>enable_if<></code>	132
6.4. Использование <code>enable_if<></code>	134
6.5. Применение концептов для упрощения выражений <code>enable_if<></code>	138
6.6. Резюме	139

Глава 7. По значению или по ссылке? 141

7.1. Передача по значению	142
7.2. Передача по ссылке	144
7.2.1. Передача с помощью константной ссылки	144
7.2.2. Передача с помощью неконстантной ссылки	146
7.2.3. Передача с помощью передаваемой ссылки	148
7.3. Использование <code>std::ref()</code> и <code>std:: cref()</code>	149
7.4. Работа со строковыми литералами и массивами	151
7.4.1. Специальные реализации для строковых литералов и обычных массивов	152
7.5. Работа с возвращаемыми значениями	153
7.6. Рекомендуемые объявления параметров шаблона	155
7.7. Резюме	158

Глава 8. Программирование времени компиляции	159
8.1. Шаблонное метапрограммирование	159
8.2. Вычисления с использованием <code>constexpr</code>	161
8.3. Выбор пути выполнения с помощью частичной специализации	163
8.4. SFINAE	165
8.4.1. Выражение SFINAE с <code>decltype</code>	169
8.5. Инструкция <code>if</code> времени компиляции	170
8.6. Резюме	172
Глава 9. Применение шаблонов на практике	173
9.1. Модель включения	173
9.1.1. Ошибки компоновщика	173
9.1.2. Шаблоны в заголовочных файлах	175
9.2. Шаблоны и <code>inline</code>	176
9.3. Предкомпилированные заголовочные файлы	177
9.4. Расшифровка романов об ошибках	180
9.5. Некоторые замечания	187
9.6. Резюме	188
Глава 10. Основные термины в области шаблонов	189
10.1. “Шаблон класса” или “шаблонный класс”	189
10.2. Подстановка, инстанцирование и специализация	190
10.3. Объявления и определения	191
10.3.1. Полные и неполные типы	192
10.4. Правило одного определения	193
10.5. Аргументы и параметры шаблонов	193
10.6. Резюме	195
Глава 11. Обобщенные библиотеки	197
11.1. Вызываемые объекты	197
11.1.1. Поддержка функциональных объектов	198
11.1.2. Работа с функциями-членами	
и дополнительными аргументами	200
11.1.3. Оборачивание вызовов функций	202
11.2. Другие утилиты для реализации обобщенных библиотек	205
11.2.1. Свойства типов	205
11.2.2. <code>std::addressof()</code>	207
11.2.3. <code>std::declval()</code>	207
11.3. Прямая передача временных значений	208
11.4. Ссылки в качестве параметров шаблонов	209

11.5. Откладывание вычислений	212
11.6. О чем следует подумать при написании обобщенных библиотек	214
11.7. Резюме	215
Часть II. Углубленное изучение шаблонов	217
Глава 12. Вглубь шаблонов	219
12.1. Параметризованные объявления	219
12.1.1. Виртуальные функции-члены	224
12.1.2. Связывание шаблонов	224
12.1.3. Первичные шаблоны	226
12.2. Параметры шаблонов	227
12.2.1. Параметры типа	227
12.2.2. Параметры, не являющиеся типами	228
12.2.3. Шаблонные параметры шаблонов	229
12.2.4. Пакеты параметров шаблонов	231
12.2.5. Аргументы шаблона по умолчанию	233
12.3. Аргументы шаблонов	234
12.3.1. Аргументы шаблонов функций	235
12.3.2. Аргументы типов	237
12.3.3. Аргументы, не являющиеся типами	237
12.3.4. Шаблонные аргументы шаблонов	240
12.3.5. Эквивалентность	242
12.4. Вариативные шаблоны	243
12.4.1. Раскрытие пакета	244
12.4.2. Где может происходить раскрытие пакета	246
12.4.3. Пакеты параметров функций	248
12.4.4. Множественные и вложенные раскрытия пакетов	250
12.4.5. Раскрытия пакетов нулевой длины	251
12.4.6. Выражения свертки	252
12.5. Друзья	254
12.5.1. Дружественные классы шаблонов классов	254
12.5.2. Дружественные функции шаблонов классов	256
12.5.3. Дружественные шаблоны	258
12.6. Заключительные замечания	258
Глава 13. Имена в шаблонах	261
13.1. Систематизация имен	261
13.2. Поиск имен	263
13.2.1. Поиск, зависящий от аргументов	265

13.2.2. ADL объявлений друзей	267
13.2.3. Внесение имен классов	268
13.2.4. Текущие инстанцирования	270
13.3. Синтаксический анализ шаблонов	271
13.3.1. Зависимость от контекста в нешаблонных конструкциях	272
13.3.2. Зависимые имена типов	275
13.3.3. Зависимые имена шаблонов	278
13.3.4. Зависимые имена в объявлениях <code>using</code>	279
13.3.5. ADL и явные аргументы шаблонов	281
13.3.6. Зависимые выражения	282
13.3.7. Ошибки компиляции	284
13.4. Наследование и шаблоны классов	285
13.4.1. Независимые базовые классы	285
13.4.2. Зависимые базовые классы	286
13.5. Заключительные замечания	290
Глава 14. Инстанцирование	293
14.1. Инстанцирование по требованию	293
14.2. Отложенное инстанцирование	295
14.2.1. Частичное и полное инстанцирование	296
14.2.2. Инстанцированные компоненты	296
14.3. Модель инстанцирования C++	300
14.3.1. Двухфазный поиск	300
14.3.2. Точки инстанцирования	301
14.3.3. Модель включения	305
14.4. Схемы реализации	306
14.4.1. Жадное инстанцирование	308
14.4.2. Инстанцирование по запросу	309
14.4.3. Итеративное инстанцирование	311
14.5. Явное инстанцирование	312
14.5.1. Ручное инстанцирование	313
14.5.2. Объявления явного инстанцирования	315
14.6. Инструкции <code>if</code> времени компиляции	316
14.7. В стандартной библиотеке	318
14.8. Заключительные замечания	319
Глава 15. Вывод аргументов шаблона	321
15.1. Процесс вывода	321
15.2. Выводимые контексты	323
15.3. Особые ситуации вывода	325

15.4. Списки инициализаторов	326
15.5. Пакеты параметров	327
15.5.1. Шаблоны оператора литерала	329
15.6. Ссылки на г-значения	330
15.6.1. Правила свертки ссылок	330
15.6.2. Передаваемые ссылки	331
15.6.3. Прямая передача	332
15.6.4. Сюрпризы вывода	335
15.7. SFINAЕ	336
15.7.1. Непосредственный контекст	337
15.8. Ограничения вывода	339
15.8.1. Допустимые преобразования аргументов	340
15.8.2. Аргументы шаблона класса	341
15.8.3. Аргументы вызова по умолчанию	342
15.8.4. Спецификации исключений	342
15.9. Явные аргументы шаблонов функций	343
15.10. Вывод из инициализаторов и выражений	346
15.10.1. Спецификатор типа <code>auto</code>	346
15.10.2. Запись типа выражения с помощью <code>decltype</code>	352
15.10.3. <code>decltype(auto)</code>	354
15.10.4. Особые случаи вывода <code>auto</code>	357
15.10.5. Структурированное связывание	360
15.10.6. Обобщенные лямбда-выражения	364
15.11. Шаблоны псевдонимов	367
15.12. Вывод аргументов шаблонов классов	368
15.12.1. Правила вывода	369
15.12.2. Неявные правила вывода	371
15.12.3. Прочие тонкости	373
15.13. Заключительные замечания	376
Глава 16. Специализация и перегрузка	379
16.1. Когда обобщенный код недостаточно хорош	379
16.1.1. Прозрачная настройка	380
16.1.2. Семантическая прозрачность	381
16.2. Перегрузка шаблонов функций	382
16.2.1. Сигнатуры	383
16.2.2. Частичное упорядочение перегруженных шаблонов функций	385
16.2.3. Правила формального упорядочения	386

16.2.4. Шаблоны и нешаблоны	388
16.2.5. Вариативные шаблоны функций	391
16.3. Явная специализация	393
16.3.1. Полная специализация шаблона класса	394
16.3.2. Полная специализация шаблона функции	398
16.3.3. Полная специализация шаблона переменной	400
16.3.4. Полная специализация члена	400
16.4. Частичная специализация шаблона класса	403
16.5. Частичная специализация шаблонов переменных	407
16.6. Заключительные замечания	408
Глава 17. Дальнейшее развитие	409
17.1. Ослабленные правила применения <code>typename</code>	410
17.2. Обобщенные параметры шаблонов, не являющиеся типами	410
17.3. Частичная специализация шаблонов функций	413
17.4. Именованные аргументы шаблонов	414
17.5. Перегруженные шаблоны классов	415
17.6. Вывод неконечных раскрытий пакетов	416
17.7. Регуляризация <code>void</code>	417
17.8. Проверка типов для шаблонов	418
17.9. Рефлексивное метапрограммирование	420
17.10. Работа с пакетами	422
17.11. Модули	423
Часть III. Шаблоны и проектирование	425
Глава 18. Полиморфная мощь шаблонов	427
18.1. Динамический полиморфизм	427
18.2. Статический полиморфизм	430
18.3. Сравнение динамического и статического полиморфизма	433
18.4. Применение концептов	435
18.5. Новые виды проектных шаблонов	437
18.6. Обобщенное программирование	437
18.7. Заключительные замечания	441
Глава 19. Реализация свойств типов	443
19.1. Пример: суммирование последовательности	443
19.1.1. Фиксированные свойства	444
19.1.2. Свойства-значения	447
19.1.3. Параметризованные свойства	452

19.2. Стратегии и классы стратегий	452
19.2.1. Различие между свойствами и стратегиями	454
19.2.2. Шаблоны членов и шаблонные параметры шаблонов	456
19.2.3. Комбинирование нескольких стратегий и/или свойств	457
19.2.4. Накопление с обобщенными итераторами	458
19.3. Функции типов	459
19.3.1. Типы элементов	460
19.3.2. Преобразующие свойства	462
19.3.3. Свойства-предикаты	468
19.3.4. Свойства типов результатов	471
19.4. Свойства на основе SFINAE	474
19.4.1. Принцип SFINAE и перегрузки функций	474
19.4.2. SFINAE и частичные специализации	479
19.4.3. Применение обобщенных лямбда-выражений со SFINAE	480
19.4.4. SFINAE и свойства	484
19.5. <code>IsConvertible</code>	488
19.6. Обнаружение членов	491
19.6.1. Обнаружение членов-типов	491
19.6.2. Обнаружение произвольных членов-типов	493
19.6.3. Обнаружение членов, не являющихся типами	494
19.6.4. Использование обобщенных лямбда-выражений для обнаружения членов	498
19.7. Прочие методы работы со свойствами	500
19.7.1. If-Then-Else	500
19.7.2. Обнаружение операций, не генерирующих исключения	503
19.7.3. Повышение удобства свойств	506
19.8. Классификация типов	508
19.8.1. Определение фундаментальных типов	509
19.8.2. Определение составных типов	511
19.8.3. Идентификация типов функций	515
19.8.4. Обнаружение типов классов	517
19.8.5. Обнаружение типов перечислений	518
19.9. Свойства стратегий	518
19.9.1. Типы параметров только для чтения	519
19.10. В стандартной библиотеке	522
19.11. Заключительные замечания	523
Глава 20. Перегрузка свойств типов	525
20.1. Специализация алгоритма	525
20.2. Диспетчеризация дескрипторов	527

20.3. Включение/отключение шаблонов функций	528
20.3.1. Предоставление нескольких специализаций	531
20.3.2. Откуда берется <code>EnableIf</code> ?	533
20.3.3. <code>if</code> времени компиляции	534
20.3.4. Концепты	536
20.4. Специализация класса	537
20.4.1. Включение/отключение шаблонов классов	538
20.4.2. Диспетчеризация дескрипторов для шаблонов классов	539
20.5. Шаблоны, безопасные по отношению к инстанцированию	542
20.6. В стандартной библиотеке	547
20.7. Заключительные замечания	548
Глава 21. Шаблоны и наследование	551
21.1. Оптимизация пустого базового класса	551
21.1.1. Принципы размещения	551
21.1.2. Члены как базовые классы	554
21.2. Странно рекурсивный шаблон проектирования	557
21.2.1. Метод Бартона–Нэкмана	559
21.2.2. Реализации операторов	562
21.2.3. Фасады	563
21.3. Миксины	570
21.3.1. Странные миксины	572
21.3.2. Параметризованная виртуальность	573
21.4. Именованные аргументы шаблона	574
21.5. Заключительные замечания	578
Глава 22. Статический и динамический полиморфизм	581
22.1. Функциональные объекты, указатели и <code>std::function<></code>	581
22.2. Обобщение указателей на функции	583
22.3. Интерфейс моста	586
22.4. Стирание типа	586
22.5. Условная передача владения	588
22.6. Вопросы производительности	591
22.7. Заключительные замечания	591
Глава 23. Метапрограммирование	593
23.1. Состояние современного метапрограммирования	593
23.1.1. Метапрограммирование значений	593
23.1.2. Метапрограммирование типов	595

23.1.3. Гибридное метапрограммирование	596
23.1.4. Гибридное метапрограммирование с типами единиц	599
23.2. Размерности рефлексивного метапрограммирования	602
23.3. Стоимость рекурсивного инстанцирования	603
23.3.1. Отслеживание всех инстанцирований	605
23.4. Вычислительная полнота	606
23.5. Рекурсивное инстанцирование	
и рекурсивные аргументы шаблонов	607
23.6. Значения перечислений и статические константы	608
23.7. Заключительные замечания	610
Глава 24. Списки типов	613
24.1. АнATOMия списков типов	613
24.2. Алгоритмы над списками типов	615
24.2.1. Индексация	615
24.2.2. Поиск наилучшего соответствия	616
24.2.3. Добавление в список типов	619
24.2.4. Обращение порядка типов в списке	621
24.2.5. Преобразование списка типов	622
24.2.6. Накопление списков типов	623
24.2.7. Сортировка вставками	626
24.3. Списки нетиповых параметров	629
24.3.1. Выводимые нетиповые параметры	632
24.4. Оптимизация алгоритмов с помощью раскрытий пакетов	632
24.5. Списки типов в стиле LISP	634
24.6. Заключительные замечания	636
Глава 25. Кортежи	637
25.1. Базовый дизайн Tuple	637
25.1.1. Хранилище	637
25.1.2. Создание кортежа	640
25.2. Базовые операции кортежей	641
25.2.1. Сравнение	641
25.2.2. Вывод	642
25.3. Алгоритмы для работы с кортежами	643
25.3.1. Кортежи как списки типов	643
25.3.2. Добавление элементов в кортеж и удаление их оттуда	644
25.3.3. Обращение порядка элементов кортежа	646
25.3.4. Индексные списки	647

25.3.5. Обращение с использованием индексных списков	648
25.3.6. Тасование и выбор	650
25.4. Распаковка кортежей	653
25.5. Оптимизация кортежей	654
25.5.1. Кортежи и оптимизация пустого базового класса	654
25.5.2. <code>get()</code> с константным временем работы	659
25.6. Индексы кортежа	660
25.7. Заключительные замечания	663
Глава 26. Контролируемые объединения	665
26.1. Хранилище	666
26.2. Дизайн	668
26.3. Запрос и извлечение значения	672
26.4. Инициализация, присваивание и уничтожение элементов	673
26.4.1. Инстанцирования	673
26.4.2. Уничтожение	674
26.4.3. Присваивание	675
26.5. Посетители	680
26.5.1. Возвращаемый тип <code>visit()</code>	683
26.5.2. Общий возвращаемый тип	684
26.6. Инициализация и присваивание <code>Variant</code>	686
26.7. Заключительные замечания	690
Глава 27. Шаблоны выражений	693
27.1. Временные объекты и раздельные циклы	693
27.2. Программирование выражений в аргументах шаблонов	699
27.2.1. Операнды шаблонов выражений	700
27.2.2. Тип <code>Array</code>	703
27.2.3. Операторы	705
27.2.4. Подведем итог	707
27.2.5. Присваивание шаблонам выражений	709
27.3. Производительность и ограничения шаблонов выражений	710
27.4. Заключительные замечания	711
Глава 28. Отладка шаблонов	715
28.1. Поверхностное инстанцирование	715
28.2. Статические утверждения	718
28.3. Архетипы	719

28.4. Трассировщики	721
28.5. Оракулы	725
28.6. Заключительные замечания	726
Приложение А. Правило одного определения	727
A.1. Единицы трансляции	727
A.2. Объявления и определения	728
A.3. Детали правила одного определения	729
A.3.1. Ограничения “одно на программу”	729
A.3.2. Ограничения “одно на единицу трансляции”	732
A.3.3. Ограничения эквивалентности единиц перекрестной трансляции	733
Приложение Б. Категории значений	737
B.1. Традиционные l- и r-значения	737
B.1.1. Преобразования l-значений в r-значения	738
B.2. Категории значений, начиная с C++11	738
B.2.1. Временная материализация	741
B.3. Проверка категорий значений с помощью decltype	743
B.4. Ссылочные типы	743
Приложение В. Разрешение перегрузки	745
B.1. Когда используется разрешение перегрузки	745
B.2. Упрощенное разрешение перегрузки	746
B.2.1. Неявный аргумент для функций-членов	749
B.2.2. Улучшение точного соответствия	751
B.3. Детали перегрузки	752
B.3.1. Предпочтение нешаблонных функций или более специализированных шаблонов	752
B.3.2. Последовательности преобразований	753
B.3.3. Преобразования указателей	754
B.3.4. Списки инициализаторов	755
B.3.5. Функции-функции-суррогаты	758
B.3.6. Другие контексты перегрузки	759
Приложение Г. Стандартные утилиты для работы с типами	761
Г.1. Использование свойств типов	761
Г.1.1. std::integral_constant и std::bool_constant	762
Г.1.2. Что вы должны знать при использовании свойств	764

Г.2. Основные и составные категории типов	765
Г.2.1. Проверка основных категорий типов	766
Г.2.2. Проверка составных категорий типов	770
Г.3. Характеристики и операции над типами	771
Г.3.1. Прочие характеристики типов	771
Г.3.3. Взаимоотношения между типами	790
Г.4. Построение типов	792
Г.5. Прочие свойства	796
Г.6. Комбинирование свойств типов	799
Г.7. Прочие утилиты	801
Приложение Д. Концепты	803
Д.1. Использование концептов	803
Д.2. Определение концептов	806
Д.3. Перегрузка ограничений	808
Д.3.1. Поглощение ограничений	808
Д.3.2. Ограничения и диспетчеризация дескрипторов	810
Д.4. Советы	811
Д.4.1. Проверка концептов	811
Д.4.2. Гранулированность концептов	811
Д.4.3. Бинарная совместимость	812
Библиография	815
Форумы	815
Книги и веб-сайты	816
Глоссарий	821
Предметный указатель	833

Алессандре и Кассандре
— Дэвид

Тем, кто заботится о людях и человечестве
— Нико

Эми, Тессе и Молли
— Дуг

Предисловие

Концепции шаблонов в C++ более 30 лет. Шаблоны C++ были описаны еще в 1990 году в книге *The Annotated C++ Reference Manual* (ARM; см. [39]), а до этого рассматривались в ряде более специализированных публикаций. Однако даже через десять лет ощущался недостаток литературы, которая бы фокусировалась на основных концепциях и передовых методах этой увлекательной, сложной и мощной возможности C++. Первым изданием данной книги мы хотели (возможно, несколько самонадеянно) решить эту проблему.

Со времени публикации первого издания в конце 2002 года в C++ изменилось очень многое. В стандарт C++ были добавлены новые возможности, кардинально изменившие сам язык, а непрерывные исследования сообщества программистов на C++ обнаружили новые методы программирования на основе шаблонов. Поэтому, сохранив цели первого издания, мы, по сути, написали новую книгу — о шаблонах в *современном C++*.

Авторы подошли к написанию этой книги с различным уровнем подготовки и с различными намерениями. Дэвид, опытный разработчик компиляторов и активный участник рабочих групп Комитета по стандартизации C++, которые развиваются основы языка, заинтересован в точном и подробном описании всей мощи (и проблем) шаблонов. Нико, “обычный” программист и член рабочей группы Комитета по стандартизации библиотеки C++, заинтересован в таком понимании всех методов применения шаблонов, которое помогло бы активно и с выгодой их использовать. Дуг, разработчик библиотеки шаблонов, превратившийся в разработчика компиляторов и проектировщика языка, заинтересован в сборе, классификации и оценке многочисленных методов, используемых для построения библиотек шаблонов. Кроме того, все мы хотели бы поделиться этими знаниями с вами и всем сообществом, чтобы помочь избежать дальнейших недоразумений, путаниц или опасений при работе с шаблонами C++.

Как следствие, книга содержит и концептуальное введение с примерами из повседневной практики, и подробное описание точного поведения шаблонов. По дороге от основных принципов применения шаблонов до “искусства программирования шаблонов” вам будут открываться (или переоткрываться заново) такие методы, как статический полиморфизм, свойства типов, метaprogramмирование и шаблоны выражений. Вы получите более глубокое понимание стандартной библиотеки C++, в которой почти весь код включает шаблоны.

Во время написания этой книги авторы узнали много нового и получили массу удовольствия. Надеемся, что с читателями произойдет то же самое. Приятного чтения!

Благодарности ко второму изданию

Написание книги — трудная работа. Ее поддержка — работа еще сложнее. Нам потребовалось более пяти лет, “размазанных” по последнему десятилетию, чтобы написать это новое издание книги, и эта работа не могла бы быть выполнена без поддержки и терпения множества людей.

В частности, это программисты, которые дали свои отзывы, сообщили о найденных ошибках и предложили возможные улучшения к первому изданию за последние 15 лет. Их просто слишком много, чтобы перечислить всех... Вы сами знаете, кто вы, и мы действительно благодарны вам за время, найденное для того, чтобы написать нам свои мысли и замечания. Пожалуйста, простите, если наши ответы были не всегда оперативны.

Мы хотели бы также поблагодарить всех, кто просматривал черновики этой книги и предоставил нам свои ценные отзывы и уточнения. Эти обзоры подняли книгу на значительно более высокий уровень качества и в очередной раз продемонстрировали, что хорошие вещи требуют вклада многих мудрых людей. Поэтому мы очень признательны таким людям, как Стив Дьюхарст (Steve Dewhurst), Говард Хиннант (Howard Hinnant), Микель Кипеляйнен (Mikael Kilpeläinen), Дитмар Кюль (Dietmar Kühl), Даниэль Крюглер (Daniel Krügler), Невин Либер (Nevin Lieber), Андреас Найзер (Andreas Neiser), Эрик Ниблер (Eric Niebler), Ричард Смит (Richard Smith), Эндрю Саттон (Andrew Sutton), Хьюбер Тонг (Hubert Tong) и Вилль Вутилайнен (Ville Voutilainen).

Конечно же, спасибо всем поддерживавшим нас сотрудникам издательства Addison-Wesley/Pearson. В наше время профессиональная поддержка автора книги — большая редкость, но все эти люди были терпеливы и мудры и оказывались в нужное время в нужном месте со своими знаниями и професионализмом. Мы благодарим таких сотрудников издательства, как Питер Гордон (Peter Gordon), Ким Бодихаймер (Kim Boedigheimer), Грег Денч (Greg Doench), Джуди Нагил (Julie Nahil), Dana Wilson и Carol Lallier.

Огромное спасибо всему сообществу \LaTeX за отличную текстовую систему и отдельно Фрэнку Миттельбаху (Frank Mittelbach) за помощь в решении проблем, связанных с \LaTeX .

Благодарности Дэвида ко второму изданию

Выход второго издания книги ожидался очень долгое время, и теперь, когда книга готова, я благодарен всем тем людям из моего окружения, кто сделал это возможным. В первую очередь это относится к моей жене Карине (Karina) и дочерям Александре (Alessandra) и Кассандре (Cassandra), которые дружно позвоили мне занять под работу над книгой значительное время из моего “семейного расписания”, особенно в последний год работы. Всегда проявляли интерес к моей книге и мои родители, и всякий раз, когда я бывал у них в гостях, активно интересовались состоянием этого проекта.

Очевидно, что это техническая книга, и ее содержание отражает знания и опыт в программировании. Однако одних знаний и опыта недостаточно, чтобы

довести такую работу до завершения. Поэтому я очень благодарен Нико за принятное на себя (в дополнение к его вкладу как автора) решение производственных вопросов и вопросов менеджмента. Если эта книга полезна для вас, и вы в один прекрасный день встретитесь с Нико — не забудьте его за это поблагодарить. Я также благодарен Дугу за согласие вступить в команду и не выйти из нее несмотря на кучу своей собственной работы.

Своими находками с нами делились многие программисты на C++, и я благодарен всем им. Однако хочу выразить особую благодарность Ричарду Смиту (Richard Smith), который на протяжении многих лет всегда терпеливо и со знанием дела отвечал на мои электронные письма с массой технических вопросов. Спасибо также моим коллегам Джону Спайсеру (John Spicer), Майку Миллеру (Mike Miller) и Майку Херрику (Mike Herrick) за обмен знаниями и создание атмосферы, способствующей дальнейшим исследованиям.

Благодарности Нико ко второму изданию

В первую очередь я хочу поблагодарить двух суперэкспертов в области языка, Дэвида и Дуга, которым я, как прикладной программист и эксперт в области библиотеки, задавал много глупых вопросов и от которых узнал много нового. Теперь я иногда чувствую себя тоже экспертом (конечно, только до того момента, пока у меня не возникнет очередной вопрос). Парни, это было здорово!

Все остальные благодарности — Ютте Экштейн (Jutta Eckstein). Ютта обладает прекрасной способностью поддерживать людей в их идеалах, идеях и целях. Хотя большинство людей испытывают это лишь изредка при встрече с ней или общей работе в области информационных технологий, мне повезло пользоваться этим в повседневной жизни. После всех этих лет я все же надеюсь, что это никогда не закончится.

Благодарности Дуга ко второму изданию

Моя сердечная благодарность — моей прекрасной жене Эми (Amy) и нашим девочкам Молли (Molly) и Тесса (Tessa). Их любовь и общение приносят мне ежедневно радость и уверенность, так необходимую для решения серьезнейших проблем в жизни и работе. Я хотел бы также поблагодарить моих родителей, которые привили мне любовь к учебе и поощряли меня к этой работе на протяжении всех этих лет.

Было очень приятно работать с Дэвидом и Нико, которые, будучи во многом противоположностями, хорошо дополняют друг друга. Дэвид привносит большую ясность в технические тексты, оттачивая их и делая предельно точными и информативными. Нико, с его исключительными организаторскими способностями, являлся силой, цементирующей наш коллектив, и привносил в него уникальную способность разбирать любые сложные технические вопросы и делать их более простыми, доступными и существенно более понятными.

Благодарности к первому изданию

Эта книга вобрала в себя мысли, концепции, решения и примеры из множества различных источников, и теперь мы хотели бы выразить свою признательность всем, кто оказывал нам помошь и поддержку на протяжении последних нескольких лет.

Прежде всего огромное спасибо нашим рецензентам, а также тем, кто высказывал свое мнение по самым первым вариантам рукописи. Без их участия нам не удалось бы довести книгу до такого уровня, какой она имеет сегодня. Нашиими рецензентами были Кайл Блейни (Kyle Blaney), Томас Гшвинд (Thomas Gschwind), Деннис Менкл (Dennis Mancl), Патрик Мак-Киллен (Patrick McKillen), Ян Христиан ван Винкель (Jan Christian van Winkel). Особая благодарность Дитмару Кюлю (Dietmar Kühl), который тщательно прорецензировал и отредактировал всю книгу. Обратная связь, которую обеспечил Дитмар, помогла значительно повысить уровень книги.

Хотелось бы также выразить признательность всем людям и организациям, которые предоставили нам возможность протестировать вошедшие в книгу примеры на разных платформах с помощью различных компиляторов. Большое спасибо Edison Design Group за замечательный компилятор и его поддержку. Сотрудники этой группы помогали нам не только создавать эту книгу, но и приводить ее в соответствие со стандартами. Большое спасибо всем разработчикам свободно распространяемых компиляторов GNU и egcs (особая благодарность Джесону Меррилу (Jason Merril) за отзывчивость), а также компании Microsoft за бета-версию Visual C++ (здесь мы контактировали с Джонатаном Кейвсом (Jonathan Caves), Гербом Саттером (Herb Sutter) и Джесоном Ширком (Jason Shirk)).

То, что сегодня составляет “ноосферу C++”, — плод коллективного творчества сетевого сообщества C++. Львиную долю этих знаний обеспечивают модерируемые конференции Usenet — comp.lang.c++.moderated и comp.std.c++. Поэтому особое спасибо активным модераторам этих групп, которые смогли сделать обсуждение полезным и конструктивным. Мы хотим также поблагодарить каждого из тех, кто не один год подряд выкраивал время для описания и объяснения своих идей, желая сделать их нашим общим достоянием.

Трудно переоценить тот вклад, который внесли в работу над книгой сотрудники издательства Addison-Wesley. Мы выражаем особую признательность нашему редактору Дебби Лафферти (Debbie Lafferty) за ее деликатные “пинки”, дальние советы и добросовестную упорную работу над книгой. Спасибо также другим сотрудникам издательства — Тайрелль Олбах (Tyrell Albaugh), Банни Эймс (Bunny Ames), Мелани Бак (Melanie Buck), Жаклин Дюсетт (Jacquelyn Doucette), Чанде Лири-Коту (Chanda Leary-Coutu), Кэтрин Охала (Catherine Ohala) и Марти Рабинович (Marty Rabinowitz). Искренне благодарим Марину Ланг (Marina Lang), которая способствовала изданию этой книги в Addison-Wesley, а также Сьюзан Винер (Susan Winer), выполнившую первое редактирование, которое помогло очертить контуры будущей книги.

Благодарности Нико

Прежде всего мне хотелось бы передать личную благодарность и бесчисленное количество поцелуев своей семьи: Улли (Ulli), Лукасу (Lucas), Анике (Anica) и Фредерику (Frederic) — за их заботу, предупредительность и поддержку, которые так помогали мне во время работы над книгой.

Кроме того, я хотел бы сказать спасибо Дэвиду. Его знания и опыт огромны, но терпение оказалось поистине безграничным (временами я задавал ему на редкость глупые вопросы). Работать с ним очень интересно.

Благодарности Дэвида

Тем, что мне удалось завершить работу над этой книгой, я обязан своей жене Карине (Karina). Я чрезвычайно благодарен ей за ту роль, которую она играет в моей жизни. Когда в твоем ежедневном расписании множество одинаково первоочередных дел, написание книги “в свободное время” быстро превращается в утопию. Именно Карина помогала мне справляться с этим расписанием, учила меня говорить “нет”, чтобы выкроить время для работы и обеспечить постоянное продвижение вперед. А самое главное — она была потрясающей движущей силой этого проекта. Я каждый день благодарю Бога за ее дружбу и любовь.

И еще: я очень рад, что мне пришлось работать с Нико. Его вклад в создание книги измеряется не только непосредственно написанным текстом. Именно опыт и дисциплинированность Нико позволили нам перейти от моих графоманских попыток к хорошо организованному изданию.

“Мистер Шаблон” Джон Спайсер (John Spicer) и “мистер Перегрузка” Стив Адамчик (Steve Adamczyk) — замечательные друзья и коллеги. А также, по моему мнению, этот дуэт является последней инстанцией во всем, что касается основ языка C++. Именно они внесли ясность во многие сложные вопросы, освещенные в данной книге, и если вам случится найти ошибку в описании какого-то элемента языка C++, значит, по этому вопросу я не смог проконсультироваться с ними.

И наконец, я хотел бы выразить признательность всем тем, кто в разной степени помогал нам в работе над этим проектом. Многие из них не внесли непосредственный вклад в этот проект, но их участие и поддержка послужили для него огромной движущей силой. Это прежде всего мои родители; их любовь и ободрение были для меня чрезвычайно важны. Источником вдохновения были для меня и мои друзья, которые постоянно интересовались, как продвигается работа над книгой. Спасибо вам всем: Майкл Бэкманн (Michael Beckmann), Бретт и Джули Бин (Brett and Julie Beene), Джарран Кэрр (Jarran Carr), Симон Чанг (Simon Chang), Хо и Сара Чо (Ho and Sarah Cho), Кристофф Де Динечин (Christophe De Dinechin), Ева Дилман (Eva Deelman), Нейл Эберли (Neil Eberle), Сэссан Хазеги (Sassan Hazeghi), Викрам Кумар (Vikram Kumar), Джим и Линдсей Лонг (Jim and Lindsay Long), Р.Дж. Морган (R.J. Morgan), Майк Пуритано (Mike Puritano), Рагу Рагавендра (Ragu Raghavendra), Джим и Фуонг Шарп (Jim and Phuong Sharp), Грег Вогн (Gregg Vaughn) и Джон Вигли (John Wiegllay).

О книге

Первое издание этой книги было опубликовано 15 лет назад. Мы пытались написать полное руководство по шаблонам C++, надеясь на то, что оно будет полезно для практикующих программистов C++. Этот проект оказался успешным: было чрезвычайно приятно получать отзывы читателей, которым пригодился наш материал и которые обращались к книге снова и снова, как к справочнику.

Увы, первое издание уже устарело, и хотя большая часть изложенного в нем материала полностью соответствует современным концепциям C++, нет никаких сомнений, что эволюция языка, приведшая к понятию “современного C++” — стандартам C++11, C++14 и C++17, настоятельно требует существенного пересмотра материала из первого издания.

Во втором издании наша цель “верхнего уровня” остается неизменной: создание руководства по шаблонам C++, которое было бы и надежным справочником, и доступным учебником. Но в этот раз мы работаем с современным языком программирования C++, который представляет собой нечто значительно большее, чем язык, доступный во времена предыдущего издания.

Мы также понимаем, что ресурсы, посвященные программированию на C++, со времени первого издания существенно изменились (в лучшую сторону). Появились несколько книг, которые весьма глубоко разбираются в конкретных приложениях с использованием шаблонов. Что еще более важно, сейчас в Интернете имеется гораздо больше информации о шаблонах C++ и методах их применения, как и примеров их использования. Так что в этом издании мы решили подчеркнуть широту методов, которые могут использоваться в различных приложениях.

Некоторые из представленных в первом издании методов устарели, потому что язык C++ теперь предлагает куда более прямые пути достижения того же результата. Эти методы убраны из книги (или низведены до небольших примечаний), и вместо них вы найдете новые методы, которые показывают текущее состояние дел при использовании новых возможностей (можно даже сказать — нового) языка.

Даже теперь, после того как мы прожили бок о бок с шаблонами C++ более 20 лет, программисты по-прежнему регулярно находят новые фундаментальные идеи, которые могут идеально вписаться в современные потребности в области развития программного обеспечения. Цель нашей книги — поделиться этими знаниями и обеспечить читателя всей необходимой информацией для развития нового понимания основных методик программирования на C++, а возможно, и новых открытий в этой области.

Что необходимо знать, приступая к чтению этой книги

Чтобы получить максимальную пользу от работы с книгой, читатель должен быть знаком с C++. В данной книге дается детальное описание конкретных возможностей языка программирования, но не основ самого языка. Предполагается знакомство читателя с концепцией классов и наследования, а также умение писать программы на C++ с использованием таких компонентов, как потоки ввода-вывода и контейнеры из стандартной библиотеки C++. Читатель должен быть также знаком с основными возможностями современного C++, такими как `auto`, `decltype`, семантика перемещения и лямбда-выражения. Кроме того, при необходимости в данной книге рассматриваются различные тонкие вопросы, которые могут не иметь прямого отношения к шаблонам. Таким образом обеспечивается доступность изложенного здесь материала как для квалифицированных специалистов, так и для программистов среднего уровня.

Изложение материала основано в первую очередь на стандартах языка C++, принятых в 2011, 2014 и 2017 годах. Однако на момент написания этой книги еще не высохли чернила на стандарте C++17, так что мы не ожидаем, что большинство читателей будет хорошо с ним знакомо. Все упомянутые стандарты оказали существенное влияние на поведение и использование шаблонов. Поэтому мы предоставляем краткое введение с описанием новых возможностей, которые имеют наибольшее отношение к нашей теме. Однако наша цель — не введение в современные стандарты C++ и не предоставление исчерпывающего описания отличий новых стандартов от предыдущих версий ([25] и [26]). Мы сосредоточиваемся в первую очередь на том, как шаблоны проектируются и используются в программах на языке C++, соответствующем современным стандартам ([27], [28] и [29]), и время от времени подчеркиваем ситуации, когда современные стандарты C++ позволяют или поощряют использовать технологии, отличные от предлагавшихся более ранними стандартами.

Структура книги в целом

Цель данной книги — предоставить читателю информацию, необходимую для работы с шаблонами и использования в полной мере их преимуществ; кроме того, книга призвана обеспечить читателей информацией, которая позволит опытным программистам преодолеть современные ограничения в этой области. Исходя из этого, мы разбили материал книги на части.

- Часть I, “Основы”, представляет собой введение в основные концепции, положенные в основу шаблонов. Эта часть написана в стиле учебника.
- Часть II, “Углубленное изучение шаблонов”, предоставляет детальные сведения о языке. Эта часть является неплохим справочником по конструкциям, связанным с шаблонами.
- Часть III, “Нетиповые параметры шаблонов”, поясняет фундаментальные методы проектирования и кодирования, поддерживаемые шаблонами C++. Они простираются от почти тривиальных идей до сложнейших идиом.

Каждая из перечисленных частей книги состоит из нескольких глав. Кроме того, книга включает несколько приложений, которые охватывают материал, относящийся не только к шаблонам (например, вопросы перегрузки в C++). Дополнительное приложение охватывает концепты, которые являются фундаментальными расширениями возможностей шаблонов и которые будут включены в будущие стандарты (вероятно, C++20).

Главы, входящие в состав первой части книги, требуют последовательного изучения. Например, глава 3 основана на материале, рассмотренном в главе 2. Однако в других частях книги связь между главами выражена не столь явно. Перекрестные ссылки помогут вам при необходимости перемещаться между различными темами книги.

Наконец, в книге имеется предметный указатель, который предоставляет еще одну возможность читать книгу не последовательно.

Как читать эту книгу

Если вы являетесь программистом на C++ и хотите получить общее представление о концепции шаблонов и поближе познакомиться с ней, то вам следует тщательно изучить часть I, “Основы”. С этим материалом имеет смысл хотя бы бегло ознакомиться даже тем, кто с шаблонами уже “на ты”, чтобы прочувствовать стиль и освоиться с используемой в книге терминологией. Эта часть также охватывает некоторые “материально-технические” аспекты, касающиеся организации исходного кода, содержащего шаблоны.

В зависимости от того, какой метод изучения материала вы предпочитаете, можно либо основательно изучить детальную информацию о шаблонах из части II, “Углубленное изучение шаблонов”, либо познакомиться с приемами практического программирования в части III, “Шаблоны и проектирование” (обращаясь к части II, если возникнут какие-либо вопросы). Последнее представляется особенно целесообразным в случае, если вы приобрели эту книгу для конкретных практических целей.

Приложения содержат большое количество полезной информации, на которую сделано много ссылок в основной части книги. Кроме того, мы старались сделать их интересными и в качестве самостоятельного материала.

Опыт подсказывает, что лучше всего новые знания усваиваются на примерах. Поэтому их вы найдете в книге большое количество. Иногда это всего лишь несколько строк кода, иллюстрирующих теоретическое положение, иногда — полноценные программы, реализующие конкретное применение материала. В последнем случае примеры снабжены комментариями C++ с описанием пути к файлу, в котором содержится код программы. Все эти файлы можно найти на сайте данной книги по адресу <http://www.tplbook.com>.

Некоторые замечания о стиле программирования

У каждого программиста на C++ свой стиль программирования, и авторы данной книги также не составляют исключения. Понятие стиля включает обычные вопросы: где помещать пробелы, разделители (скобки, фигурные скобки) и т.п. В целом мы старались придерживаться единого стиля, хотя иногда по ходу изложения приходилось делать исключения. Например, чтобы придать коду больше наглядности, в разделах руководства были широко использованы пробелы и осмысленные имена, в то время как при рассмотрении более сложных вопросов предпочтение отдавалось компактности.

Хотелось бы обратить внимание читателя на то, что в данной книге применяется несколько необычный подход к записи объявлений типов, параметров и переменных. Очевидно, что при объявлении возможно использование нескольких стилей:

```
void foo(const int &x);
void foo(const int& x);
void foo(int const &x);
void foo(int const& x);
```

Хотя этот порядок записи менее распространен, для обозначения целочисленной константы мы решили использовать `int const` вместо `const int`. Сделано это было по двум причинам. Во-первых, такой порядок обеспечивает более очевидный ответ на вопрос: “Что именно является константой?” “Что” — это всегда то, что находится перед квалификатором `const`. В самом деле, хотя выражение

```
const int N = 100;
```

эквивалентно выражению

```
int const N = 100;
```

не существует аналогичной эквивалентной записи для

```
int* const bookmark; // Указатель не может меняться; значение
                     // же, на которое он указывает - может.
```

в которой можно было бы разместить квалификатор `const` до `*`. В этом примере константой является сам указатель, а не значение типа `int`, на которое он указывает.

Вторая причина связана с синтаксической подстановкой, часто встречающейся при работе с шаблонами. Рассмотрим два следующих определения типов:

```
typedef char* CHARS;
typedef CHARS const CPTR; // Константный указатель на char
```

или с использованием ключевого слова `using`:

```
using CHARS = char *;
using CPTR = CHARS const; // Константный указатель на char
```

Смысль второго объявления сохраняется при текстуальной замене `CHARS` тем, что это слово означает:

```
typedef char* const CPTR; // Константный указатель на char
```

или:

```
using CPTR = char* const; // Константный указатель на char
```

Однако если мы напишем `const` до квалифицируемого типа, то этот принцип окажется неприменим. Рассмотрим альтернативу двум представленным ранее первым определениям типа:

```
typedef char* CHARS;
typedef const CHARS CPTR; // Константный указатель на char
```

Текстуальная замена `CHARS` приводит к типу с иным значением:

```
typedef const char* CPTR; // Указатель на константный char
```

Очевидно, что сказанное выше справедливо и для спецификатора `volatile`.

Что касается расстановки пробелов, то мы решили помещать пробел между амперсандом и именем параметра:

```
void foo(int const& x);
```

Поступая таким образом, мы подчеркиваем разделение между типом параметра и именем параметра. Однако при такой записи еще более запутанными становятся объявления наподобие

```
char* a, b;
```

Здесь согласно правилам, унаследованным из C, `a` является указателем, а `b` — обычной символьной переменной. Чтобы исключить такого рода путаницу, мы просто стараемся избегать объявления нескольких переменных приведенным образом.

Эта книга посвящена в первую очередь возможностям языка программирования. Однако многие методики, возможности и вспомогательные шаблоны в настоящее время входят в стандартную библиотеку C++. Для их объединения мы демонстрируем шаблонные методы, иллюстрируя, как они применяются для реализации некоторых компонентов библиотеки, и используем возможности стандартной библиотеки для создания более сложных примеров. Поэтому мы применяем не только такие заголовочные файлы, как `<iostream>` и `<string>` (которые содержат шаблоны, но не так существенны для определения других шаблонов), но и `<cstddef>`, `<utilities>`, `<functional>` и `<type_traits>` (предоставляющие “кирпичи” для построения более сложных шаблонов).

Кроме того, приложение Г, “Стандартные утилиты для работы с типами”, содержит вспомогательные возможности, предоставляемые стандартной библиотекой C++, а также детальное описание всех стандартных свойств типов. Они широко используются в сложном программировании шаблонов.

Стандарты C++11, C++14 и C++17

Исходный стандарт C++ был опубликован в 1998 году и дополнен *техническими поправками* в 2003 году, которые внесли незначительные исправления

и уточнения в первоначальный стандарт. Эти “старые стандарты C++” известны как C++98 и C++03.

Стандарт C++11 был первым крупным пересмотром стандарта под руководством Комитета по стандартизации ISO C++, который привнес в язык программирования множество новых возможностей, связанных с работой с шаблонами, в частности таких.

- Вариативные шаблоны (variadic templates, шаблоны с переменным количеством параметров).
- Шаблоны псевдонимов (alias templates).
- Семантика перемещения, ссылки на r-значения и прямая передача (perfect forwarding).
- Стандартные свойства типов (type traits).

За C++11 последовали стандарты C++14 и C++17, которые добавили в язык новые возможности, хотя и не столь драматические, как внесенные стандартом C++11¹. Новые возможности, связанные с работой с шаблонами и описанные в книге, включают следующее (но не ограничиваются перечисленным).

- Шаблоны переменных (C++14).
- Обобщенные лямбда-выражения (C++14).
- Вывод аргументов шаблона класса (C++17).
- if времени компиляции (C++17).
- Выражения свертки (C++17).

Мы также рассмотрим *концепты* (интерфейсы шаблонов), которые в настоящее время планируются к включению в стандарт C++20.

На момент написания книги стандарты C++11 и C++14 достаточно широко поддерживались основными компиляторами, как и многие возможности стандарта C++17. Тем не менее компиляторы сильно отличаются в их поддержке различных возможностей языка. Одни компиляторы будут компилировать большую часть кода из этой книги, другие не смогут справиться с рядом приведенных примеров. Однако мы ожидаем, что эта проблема будет вскоре решена — по мере того как программисты будут все активнее требовать поддержку стандарта от поставщиков.

Как бы то ни было, язык программирования C++ продолжает развиваться. Эксперты сообщества C++ (независимо от их участия в работе Комитета по стандартизации C++) обсуждают различные способы улучшения языка, и несколько

¹ Заметим, что в C++ синоним типа определяет псевдоним, а не новый тип (см. раздел 2.8), например:

```
typedef int Length; //Length определяется как псевдоним int
int i = 42;
Length l = 88;
i = l;           // OK
l = i;           // OK
```

таких предлагаемых усовершенствований связаны с шаблонами. В главе 17, “Дальнейшее развитие”, описаны некоторые тенденции в этой области.

Примечание редактора перевода

На странице найденных ошибок и опечаток к данной книге имеется следующее примечание авторов книги.

Мы забыли уточнить, что до стандарта C++17 было необходимо добавлять определение к объявлению статического constexpr-члена класса:

```
struct A {  
    static constexpr int n = 5; // В C++11/C++14 – объявление,  
                            // начиная с C++17 – определение  
};
```

До стандарта C++17 этот код представляет собой только объявление n. Но начиная со стандарта C++17 это объявление является также и определением. Таким образом, работая со стандартами, предшествующими C++17, в частности в одной единице трансляции необходимо предоставить соответствующее определение (которое в ряде примеров в книге оказалось опущено):

```
// До C++17 – в одной, и только одной единице трансляции:  
constexpr int A::n; // Определение в C++11/C++14;  
                    // начиная с C++17 – не рекомендуемое  
                    // избыточное объявление
```

Причина, по которой никто из нас этого не заметил, кроется в том, что зачастую такое определение оказывается попросту ненужным. Фактически член всегда можно передать по значению. И только при передаче членов по ссылке, при условии, что компилятор не оптимизирует такой вызов, возникают ошибки компоновки:

```
std::cout << A::n;      // OK даже без определения (оператор  
                        // ostream::operator<<(int) получает A::n  
                        // по значению)  
int inc(const int& i); // Передача по ссылке!  
std::cout << inc(A::n); // Ошибка компоновки при отсутствии  
                        // определения и оптимизации
```

Просьба к читателям учесть это замечание при чтении книги.

Исходные тексты примеров и дополнительная информация

Исходные тексты всех примеров программ и более подробную информацию по книге можно найти на сайте по адресу <http://www.tplbook.com>.

Обратная связь с авторами

Мы приветствуем любые конструктивные отклики читателей — как отрицательные, так и положительные. Нам пришлось основательно потрудиться, чтобы создать для вас книгу, которую, надеемся, вы оцените достаточно высоко. Однако в определенный момент мы просто вынуждены были прервать работу над ней, поскольку подошел срок “выпуска продукта”. Следовательно, ни один из наших читателей не застрахован от того, что при изучении материала ему придется столкнуться с ошибками или несогласованностью, а также с отдельными моментами, которые нуждаются в доработке, или с тем, что отдельные темы в книге не освещены вообще. Ваши отклики дают нам возможность проинформировать всех читателей через веб-сайт книги о найденных вами “узких местах” и улучшить таким образом ее последующие издания.

Связываться с нами лучше всего по электронной почте, которую вы найдете на веб-сайте книги (<http://www.templbook.com>); однако, прежде чем посыпать сообщение, удостоверьтесь, пожалуйста, что найденная вами неточность отсутствует в списке опечаток.

Заранее благодарим вас за сотрудничество.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

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

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

ЧАСТЬ I

Основы

Эта часть книги знакомит читателя с общими концепциями и языковыми средствами шаблонов C++. Она начинается с обсуждения основных задач и концепций на примерах шаблонов функций и шаблонов классов. В последующих главах рассматриваются такие дополнительные фундаментальные приемы работы с шаблонами, как параметры шаблонов, не являющиеся типами (нетиповые параметры шаблонов), шаблоны с переменным количеством параметров, ключевое слово `template` и шаблоны-члены. Рассматривается работа с семантикой перемещения, объявление параметров и использование обобщенного кода для программирования времени компиляции. В завершение приведены некоторые подсказки, касающиеся терминологии и применения шаблонов на практике — как прикладными программистами, так и авторами обобщенных библиотек.

Зачем нужны шаблоны

C++ требует от нас объявления переменных, функций и большинства других видов сущностей с использованием конкретных типов. Однако во многих случаях код для различных типов выглядит практически одинаково. Например, реализация алгоритма *быстрой сортировки* структурно одинакова для таких разных типов данных, как массив целочисленных значений или вектор строк — лишь бы содержащиеся в контейнерах элементы можно было сравнивать между собой.

Если используемый вами язык программирования не поддерживает специальных языковых средств для такого рода обобщенности, существуют только следующие (не самые лучшие) альтернативы.

1. Можно вновь и вновь реализовывать одно и то же поведение для каждого типа данных, нуждающихся в таком поведении.
2. Можно написать обобщенный код для общего базового типа, такого как `Object` или `void*`.
3. Можно использовать специальные препроцессоры.

Если вы пришли из других языков программирования, вероятно, вам уже приходилось проделывать подобные действия. Однако каждый из описанных выше подходов имеет свои недостатки.

1. Каждый раз заново реализуя одно и то же поведение, мы, по сути, снова и снова изобретаем велосипед. Мы делаем одни и те же ошибки и, чтобы не наделать их еще больше, стараемся избегать более сложных, но зато и более эффективных алгоритмов.
2. Если мы пишем обобщенный код для общего базового класса, то теряем при этом преимущество проверки типов. Кроме того, в разных ситуациях может потребоваться порождение от различных базовых классов, что еще более затрудняет поддержку кода.

3. При использовании специального препроцессора код заменяется некоторым “тупым механизмом замены текста”, который не имеет представления ни об области видимости, ни о типах, что может привести к странным семантическим ошибкам.

Шаблоны являются решением данной проблемы, лишенным указанных недостатков. Шаблон представляет собой функцию или класс, написанные для одного или нескольких типов данных, которые пока еще не известны. При использовании шаблона в качестве аргументов ему явно или неявно передаются конкретные типы данных. Поскольку шаблоны являются средствами языка, для них обеспечивается полная поддержка проверки типов и областей видимости.

Шаблоны получили широкое применение в современном программировании. Например, практически весь код стандартной библиотеки C++ состоит из шаблонов. Библиотека обеспечивает алгоритмы сортировки объектов и значений определенного типа, структуры данных (именуемые также *классами контейнеров*) для управления элементами определенного типа, строки, для которых тип символа является параметризованным, и т.п. Однако это только начало: шаблоны позволяют также параметризовать поведение, оптимизировать код и параметризовать информацию. Эти применения шаблонов рассматриваются в последующих главах. А пока начнем с некоторых простых шаблонов.

Глава 1

Шаблоны функций

Данная глава знакомит читателя с шаблонами функций. Шаблоны функций – это параметризованные функции; таким образом, шаблон функции представляет целое семейство функций.

1.1. Первое знакомство с шаблонами функций

Шаблоны функций предоставляют функциональное поведение, которое может быть вызвано для разных типов. Другими словами, шаблон функции представляет семейство функций. Шаблон очень похож на обычную функцию, различия только в том, что некоторые элементы этой функции остаются неопределенными и являются параметризованными. Чтобы проиллюстрировать сказанное выше, рассмотрим небольшой пример.

1.1.1. Определение шаблона

Ниже приведен шаблон функции, возвращающей большее из двух значений.

basics/max1.hpp

```
template<typename T>
T max(T a, T b) {
    // Если b < a, возвращаем a, в противном случае b
    return b < a ? a : b;
}
```

Это определение шаблона задает семейство функций, возвращающих большее из двух значений; эти значения передаются функции как ее параметры *a* и *b*¹. Тип этих параметров остается открытым как *параметр шаблона* *T*. Как показано в примере, параметры шаблонов объявляются с использованием следующего синтаксиса:

template< разделенный запятыми список параметров >

В нашем примере список параметров представляет собой *typename T*. Обратите внимание на то, что в качестве скобок используются символы “меньше” (<) и “больше” (>), которые мы будем называть *угловыми скобками* (*angle brackets*). Ключевое слово *typename* задает так называемый *параметр, являющийся типом*, или, для краткости – *параметр типа*, или *типовую параметр*. Это пока что наиболее распространенный вид параметров шаблонов в программах на C++, хотя

¹ Заметим, что шаблон *max()* в соответствии с [64] внутренне возвращает *b < a ? a : b*, а не *a < b ? b : a*, чтобы гарантировать, что функция корректно ведет себя, даже когда два значения эквивалентны, но не равны.

возможны и другие параметры, которые рассматриваются в книге несколько позже (см. главу 3, “Нетиповые параметры шаблонов”).

В данном примере параметр типа обозначен как `T`. В качестве имени параметра можно использовать любой идентификатор, но обычно по соглашению используется именно `T`. Параметр типа представляет произвольный тип, который определяется при вызове функции. Можно использовать любой тип (это может быть один из фундаментальных типов, класс и т.п.), который допускает применение операций, использованных в шаблоне. В нашем случае тип `T` должен поддерживать оператор `<`, поскольку он используется в теле функции для сравнения `a` и `b`. Из определения `max()` несколько менее очевидно, что значения типа `T` должны быть копируемы, чтобы быть возвращаемыми².

По историческим причинам для определения параметра типа разрешается применение вместо `typename` ключевого слова `class`. Ключевое слово `typename` в ходе эволюции языка C++ появилось относительно поздно, при разработке стандарта C++98. До этого единственным способом задания параметра типа было ключевое слово `class`, применение которого для определения параметра типа корректно и сегодня. Следовательно, эквивалентным способом определения шаблона `max()` является следующий:

```
template<class T>
T max(T a, T b) {
    return b < a ? a : b;
}
```

Семантически в данном контексте между этими двумя способами записи нет никакой разницы. Даже в случае применения ключевого слова `class` для аргументов шаблона может быть использован *любой* тип. Однако, поскольку ключевое слово `class` может ввести в заблуждение (вместо `T` можно подставлять не только тип, являющийся классом), в данном контексте следует отдавать предпочтение использованию ключевого слова `typename`. Отметим, что в отличие от объявлений типа класса ключевое слово `struct` при объявлении параметров типа вместо `typename` использовать нельзя.

1.1.2. Применение шаблонов

В приведенном ниже фрагменте кода иллюстрируется использование шаблона функции `max()`.

basics/max1.cpp

```
#include "max1.hpp"
#include <iostream>
```

²До стандарта C++17 тип `T` должен был быть копируемым и для того, чтобы его можно было передать в качестве аргументов, но, начиная с C++17, временные значения (`rvalue` — см. приложение Б, “Категории значений”) можно передавать, даже если корректного копирующего или перемещающего конструктора нет.

```
#include <string>
int main() {
    int i = 42;
    std::cout << "max(7,i): " << ::max(7, i) << '\n';

    double f1 = 3.4;
    double f2 = -6.7;
    std::cout << "max(f1,f2): " << ::max(f1, f2) << '\n';

    std::string s1 = "mathematics";
    std::string s2 = "math";
    std::cout << "max(s1,s2): " << ::max(s1, s2) << '\n';
}
```

В этой программе `max()` вызывается трижды: для двух значений типа `int`, для двух `double` и для двух `std::string`. Каждый раз вычисляется большее значение. В результате программа выводит следующую информацию:

```
max(7, i) : 42
max(f1, f2): 3.4
max(s1, s2): mathematics
```

Обратите внимание на то, что в примере каждый вызов шаблона `max()` предваряется двумя двоеточиями (`::`). Это вовсе не потому, что `max()` находится в глобальном пространстве имен. Дело в том, что в стандартной библиотеке тоже есть шаблон `std::max()`, который может быть вызван при определенных обстоятельствах или способен привести к неоднозначности³.

Шаблоны не компилируются в единую сущность, способную обработать любой тип данных. Вместо этого из шаблона генерируются различные объекты для каждого типа, для которого применяется шаблон⁴. Таким образом, `max()` компилируется для каждого из трех типов. Например, первый вызов `max()`

```
int i = 42;
... max(7, i) ...
```

использует шаблон функции с `int` в качестве параметра шаблона `T`. Таким образом, он имеет семантику вызова следующего кода:

```
int max(int a, int b) {
    return b < a ? a : b;
}
```

³ Например, если один тип аргумента определен в пространстве имен `std` (например, `string`), тогда в соответствии с правилами поиска имен C++ будут найдены оба шаблона — как глобальный, так и из пространства имен `std` (см. приложение B, “Разрешение перегрузки”).

⁴ Альтернативный способ — “один объект на все случаи жизни” — также имеет право на существование, но не используется на практике (будучи менее эффективным во время выполнения). Все правила языка основываются на принципе генерации различных объектов для различных аргументов шаблонов.

Процесс замены параметров шаблона конкретными типами называется *инстанцированием шаблона* (*instantiation*). Его результатом является *экземпляр* (*instance*) шаблона⁵.

Отметим, что процесс инстанцирования запускается простым использованием шаблона функции. Специально требовать от компилятора инстанцирования шаблона не нужно.

Аналогично другие вызовы `max()` инстанцируют шаблон `max` для `double` и `std::string` точно так же, как если бы они были объявлены и реализованы отдельно:

```
double max(double, double);
std::string max(std::string, std::string);
```

Обратите также внимание: тип `void` является корректным аргументом шаблона при условии, что получающийся код корректен; например:

```
template<typename T>
T foo(T*) {
}

void* vp = nullptr;
foo(vp); // OK: выводится void foo(void*)
```

1.1.3. Двухэтапная трансляция

Попытка инстанцирования шаблона для типа, который не поддерживает все используемые в нем операции, приведет к ошибке времени компиляции. Например:

```
std::complex<float> c1, c2; // Отсутствует оператор <
...
::max(c1, c2); // Ошибка времени компиляции
```

Таким образом, шаблоны “компилируются” в два этапа.

1. Во время определения (definition time) код шаблона проверяется на корректность без инстанцирования, с игнорированием параметров шаблонов. Этот процесс включает:
 - выявление таких синтаксических ошибок, как отсутствующие точки с запятой;
 - выявление применения неизвестных имен (имен типов, функций и т.п.), которые не зависят от параметров шаблона;
 - выполнение проверок статических утверждений, не зависящих от параметров шаблонов.
2. Во время инстанцирования код шаблона вновь проверяется на корректность. Таким образом, все части, которые зависят от параметров шаблонов, подвергаются двойной проверке.

⁵ Термины *инстанцирование* и *экземпляр* в объектно-ориентированном программировании применяются и в другом контексте, а именно для конкретного объекта класса. Однако, поскольку наша книга посвящена шаблонам, этот термин будет использоваться применительно к шаблонам, если специально не оговорено иное.

Например:

```
template<typename T>
void foo(T t) {
    undeclared(); // Если функция undeclared() неизвестна -
                  // получаем ошибку первого этапа
    undeclared(t); // Если функция undeclared(T) неизвестна -
                    // получаем ошибку второго этапа
    static_assert(sizeof(int) > 10, // Сбой всегда при
                 "int слишком мал"); // sizeof(int)<=10
    static_assert(sizeof(T) > 10, // Сбой при инстанцировании
                 "T too small"); // с типом T, размер
                           // которого <=10
}
```

Проверка имен, выполняемая дважды, называется *двуэтапным (двуфазным) поиском* (*two-phase lookup*) и обсуждается в разделе 14.3.1.

Обратите внимание на то, что некоторые компиляторы не выполняют полные проверки на первом этапе⁶. Поэтому вы можете не разглядеть общие проблемы, пока код шаблона не будет инстанцирован хотя бы раз.

Компиляция и компоновка

Двуэтапная трансляция вызывает важную проблему при практической работе с шаблонами: когда шаблон функции используется способом, запускающим его инстанцирование, компилятору (в определенный момент) потребуется определение этого шаблона. Это приводит к отходу от обычной практики, различающей компиляцию и компоновку для нешаблонных функций, когда объявления функции достаточно для компиляции кода с ее использованием. Методы решения этой проблемы обсуждаются в главе 9, “Применение шаблонов на практике”. Пока что будем использовать простой подход: реализовывать каждый шаблон в заголовочном файле.

1.2. Вывод аргумента шаблона

При вызове с некоторыми аргументами такого шаблона функции, как `max()`, параметры этого шаблона определяются передаваемыми в функцию аргументами. Если передать два значения `int`, компилятор делает вывод, что вместо `T` следует подставить тип `int`.

Однако `T` может быть только “частью” типа. Например, если мы объявим шаблон функции `max()` как использующей константные ссылки:

```
template<typename T>
T max(T const& a, T const& b) {
    return b < a ? a : b;
}
```

⁶ Например, компилятор Visual C++ в некоторых версиях (таких как Visual Studio 2013 и 2015) допускает необъявленные имена, которые не зависят от параметров шаблонов, и даже некоторые синтаксические ошибки (как, например, отсутствующие точки с запятой).

и передадим значения типа `int`, то тип `T` будет выведен как `int`, поскольку параметры функции будут соответствовать `int const&`.

Преобразования типов в процессе вывода

Обратите внимание на то, что автоматическое преобразование типов во время вывода типа ограничено.

- При объявлении вызова по ссылке при выводе типа не выполняются даже тривиальные преобразования. Два аргумента, объявленные с одним и тем же параметром шаблона `T`, должны точно совпадать.
- При объявлении параметров по значению поддерживаются только тривиальные *низводящие* (*decay*) преобразования: игнорируются квалификаторы `const` или `volatile`, ссылки преобразуются в тип, на который они ссылаются, а обычные массивы или функции преобразуются в соответствующий тип указателя. Для двух аргументов, объявленных с одним и тем же параметром шаблона `T`, *низведенные* типы должны совпадать.

Например:

```
template<typename T>
T max(T a, T b);

...
int i = 17;
int const c = 42;
max(i, c);           // OK: Т выводится как int
max(c, c);           // OK: Т выводится как int
int& ir = i;
max(i, ir);          // OK: Т выводится как int
int arr[4];
max(&i, arr);        // OK: Т выводится как int*
```

Однако следующий код вызывает ошибку времени компиляции:

```
max(4, 7.2);          // Ошибка: Т может быть выведен как int или double
std::string s;
max("hello", s);      // Ошибка: Т может быть выведен как
                      // char const[6] или std::string
```

Справиться с этими ошибками можно тремя способами.

1. Выполнить приведение аргументов, чтобы они соответствовали одному типу:

```
max(static_cast<double>(4), 7.2); // OK
```

2. Явно указать (или квалифицировать) тип `T` для того, чтобы предупредить выведение типа компилятором:

```
max<double>(4, 7.2); // OK
```

3. Указать, что параметры могут иметь разные типы.

Мы рассмотрим эти варианты в разделе 1.3. В разделе 7.2 и главе 15, “Вывод аргументов шаблона”, правила преобразования при выведении типов будут описаны подробнее.

Вывод типов для аргументов по умолчанию

Заметим также, что вывод типов не работает для аргументов по умолчанию. Например:

```
template<typename T>
void f(T = "");

***  
f(); // OK: Т выводится как int, так что вызывается f<int>(1)  
f(); // Ошибка: невозможно вывести Т
```

Для поддержки такого применения вы должны также объявить аргумент по умолчанию для параметра шаблона (о чем будет рассказано в разделе 1.4):

```
template<typename T = std::string>
void f(T = "");  
  
***  
f(); // OK
```

1.3. Несколько параметров шаблона

До сих пор мы встречались с шаблонами с двумя различными наборами параметров.

1. *Параметры шаблона*, объявленные в угловых скобках перед именем шаблона функции:

```
template<typename T> // Т – параметр шаблона
```

2. *Параметры вызова*, объявленные в круглых скобках после имени шаблона функции:

```
T max (T a, T b) // а и б – параметры вызова
```

Количество задаваемых параметров не ограничено. Например, можно определить шаблон `max()` для двух потенциально различных типов данных:

```
template<typename T1, typename T2>
T1 max(T1 a, T2 b) {
    return b < a ? a : b;
}

***  
auto m = ::max(4, 7.2); // OK, но тип первого аргумента
                        // определяет возвращаемый тип
```

Может показаться желательным иметь возможность передавать шаблону `max()` параметры различных типов, но, как демонстрирует пример, этот способ имеет свои недостатки. Если использовать один из типов параметров в качестве возвращаемого типа, аргумент другого параметра должен конвертироваться в этот же тип, независимо от намерений вызвавшего этот шаблон программиста. Таким образом, возвращаемый тип зависит от порядка аргументов вызова. Наибольшее из значений 66.66 и 42 будет `double` 66.66, в то время как наибольшим из значений 42 и 66.66 оказывается `int` 66.

C++ предоставляет различные способы решения этой проблемы.

- Ввести третий параметр шаблона для возвращаемого типа.
- Позволить компилятору самому определять возвращаемый тип.
- Объявить возвращаемый тип как “общий тип” двух типов параметров.

Все эти варианты будут рассмотрены далее.

1.3.1. Параметр шаблона для возвращаемого типа

Итак, наше обсуждение показало, что вывод аргументов шаблона позволяет вызывать шаблоны функций с синтаксисом, идентичным вызову обычной функции: нам не нужно явно указывать типы, соответствующие параметрам шаблона.

Мы также упоминали, что можно явно указать типы, используемые как параметры шаблона:

```
template<typename T>
T max(T a, T b);
***  
::max<double>(4, 7.2); // Инстанцируем T как double
```

В тех случаях, когда связи между параметрами шаблона и параметрами вызова нет, и нельзя определить параметры шаблона, необходимо явно указать аргумент шаблона при вызове. Например, можно ввести третий аргумент шаблона для определения возвращаемого типа шаблона функции:

```
template<typename T1, typename T2, typename RT>
RT max(T1 a, T2 b);
```

Однако вывод аргумента шаблона не учитывает возвращаемый тип⁷, а RT среди типов параметров вызова функции не появляется. Таким образом, тип RT не может быть выведен⁸.

Как следствие, необходимо явно указать список аргументов шаблона. Например:

```
template<typename T1, typename T2, typename RT>
RT max(T1 a, T2 b);
***  
::max<int, double, double>(4, 7.2); // OK, но утомительно
```

Пока что мы рассмотрели случаи, в которых либо явно указаны все аргументы шаблона функции, либо ни один из них. Другой подход заключается в явном указании только первых аргументов, позволяя компилятору вывести остальные. В общем случае необходимо указать все типы аргументов до последнего, который не может быть определен неявно. Таким образом, если изменить порядок

⁷ Вывод типов можно рассматривать как часть разрешения перегрузки — процесс, который не учитывает возвращаемый тип функции. Единственным исключением является возвращаемый тип членов класса, которые представляют собой операторы преобразования типов.

⁸ В C++ возвращаемый тип также не может быть выведен из контекста, в котором вызывающая функция делает вызов.

параметров шаблона в нашем примере, вызывающему коду придется указать только тип возвращаемого значения:

```
template<typename RT, typename T1, typename T2>
RT max(T1 a, T2 b);

 $\cdots$ 
::max<double>(4, 7.2) // OK: возвращаемый тип double,
// типы T1 и T2 выведены
```

В этом примере вызов `max<double>` явно устанавливает тип `RT` как `double`, а параметры `T1` и `T2` выводятся на основании переданных значений аргументов как `int` и `double`.

Обратите внимание на то, что такая измененная версия функции `max()` не приводит к значительным преимуществам. Для версии с одним параметром можно указать тип параметра (и возвращаемый тип), если переданы два аргумента разного типа. Таким образом, было бы хорошо сохранить простоту и использовать однопараметрическую версию функции `max()` (как мы и поступим в следующих разделах при обсуждении других вопросов шаблонов).

Подробнее процесс вывода аргументов шаблона рассматривается в главе 15, “Вывод аргументов шаблона”.

1.3.2. Вывод возвращаемого типа

Если тип возвращаемого значения зависит от параметров шаблона, для вывода возвращаемого типа проще и лучше позволить компилятору выяснить его самостоятельно. Начиная с C++14, это возможно — просто нужно не указывать никакой тип возвращаемого значения (но вам все равно придется объявить тип возвращаемого значения как `auto`):

`basics/maxauto.hpp`

```
template<typename T1, typename T2>
auto max(T1 a, T2 b)
{
    return b < a ? a : b;
}
```

Фактически применение `auto` для возвращаемого типа без соответствующего *завершающего возвращаемого типа* (trailing return type) (который вводится с помощью `->` в конце объявления функции) указывает, что возвращаемый тип должен быть выведен из операторов `return` в теле функции. Конечно, такой вывод типа возвращаемого значения из тела функции должен быть возможным. Таким образом, код функции должен быть доступен компилятору, а все операторы `return` в нем должны возвращать значения одного и того же типа.

До C++14 единственной возможностью позволить компилятору определить тип возвращаемого была реализация функции как часть ее объявления. В C++11 мы можем воспользоваться тем, что синтаксис завершающего возвращаемого типа позволяет использовать параметры вызова. То есть мы можем *объявить*, что возвращаемый тип является производным от того, что дает оператор `?::`:

basics/maxdecltype.hpp

```
template<typename T1, typename T2>
auto max(T1 a, T2 b) -> decltype(b < a ? a : b)
{
    return b < a ? a : b;
}
```

Здесь результирующий тип определяется правилами работы оператора `?:`, которые довольно сложны, но обычно дают интуитивно ожидаемый результат (например, если `a` и `b` имеют разные арифметические типы, то результат имеет соответствующий общий арифметический тип).

Обратите внимание на то, что

```
template<typename T1, typename T2>
auto max(T1 a, T2 b) -> decltype(b < a ? a : b);
```

является объявлением, таким образом, компилятор использует правила работы оператора `?:`, вызванного для параметров `a` и `b`, чтобы выяснить тип возвращаемого значения функции `max()` во время компиляции. Реализация функции не обязана в точности соответствовать объявлению. Например, применения значения `true` в качестве условия оператора `?:` в объявлении вполне достаточно:

```
template<typename T1, typename T2>
auto max(T1 a, T2 b) -> decltype(true ? a : b);
```

Однако в любом случае это определение имеет существенный недостаток: может случиться так, что возвращаемый тип является ссылочным, поскольку в некоторых условиях `T` может быть ссылкой. По этой причине необходимо вернуть *низведенный* тип `T`, который выглядит следующим образом:

basics/maxdecltypedecay.hpp

```
#include <type_traits>
template<typename T1, typename T2>
auto max(T1 a, T2 b) -> typename std::decay<decltype(true ? a : b)>::type
{
    return b < a ? a : b;
}
```

Здесь использовано свойство типа `std::decay<>`, которое возвращает результирующий тип как член `type`. Оно определяется в стандартной библиотеке в заголовочном файле `<type_traits>` (см. раздел Г.4 приложения Г, “Стандартные утилиты для работы с типами”). Поскольку член `type` является типом, для доступа к нему следует квалифицировать выражение с помощью ключевого слова `typename` (см. раздел 5.1).

Обратите внимание на то, что инициализация типа `auto` всегда низводится. Это также относится к возвращаемым значениям, когда возвращаемый тип указан как `auto`. `auto` в качестве возвращаемого типа ведет себя как в следующем коде, в котором `a` объявлен как низведенный тип `i`, т.е. `int`:

```
int i = 42;
int const & ir = i; // ir ссылается на i
auto a = ir;        // а объявлена как новый объект типа int
```

1.3.3. Возвращаемый тип как общий тип

Начиная с C++11, стандартная библиотека C++ предоставляет средства для указания “более общего типа”. `std::common_type<>::type` дает “общий тип” для двух (или больше) различных типов, переданных как аргументы шаблона. Например:

basics/maxcommon.hpp

```
#include <type_traits>
template<typename T1, typename T2>
std::common_type_t<T1, T2> max(T1 a, T2 b) {
    return b < a ? a : b;
}
```

`std::common_type` является свойством типа, определенным в заголовочном файле `<type_traits>`, которое представляет собой структуру, член `type` которой является результирующим типом. Таким образом, оно используется следующим образом:

```
typename std::common_type<T1, T2>::type // Начиная с C++11
```

Поскольку, начиная с C++14, можно упростить применение такого рода свойств, добавляя `_t` к имени свойства и опуская `typename` и `::type` (см. подробное изложение в разделе 2.8), определение возвращаемого типа превращается в

```
std::common_type_t<T1, T2> // Эквивалентная запись C++14
```

Реализация `std::common_type<>` использует ряд сложных технологий программирования шаблонов, которые рассматриваются в разделе 26.5.2. Внутренне этот шаблон выбирает согласно правилам языка результирующий тип оператора `:` или использует специализации для определенных типов. Таким образом, `max(4, 7.2)`, `max(7.2, 4)` дают одно и то же значение `7.2` типа `double`. Обратите внимание на то, что `std::common_type<>` также наследуется. Подробнее этот вопрос рассматривается в разделе Г.5.

1.4. Аргументы шаблона по умолчанию

Можно также определить значения по умолчанию для параметров шаблона. Эти значения называются *аргументами шаблона по умолчанию* и могут использоваться с шаблонами любого вида⁹. Они могут даже ссылаться на предыдущие параметры шаблона.

⁹ До C++11 аргументы шаблонов по умолчанию были разрешены только в шаблонах классов в силу исторических проблем в разработке шаблонов функций.

Например, если вы хотите объединить подходы для определения возвращаемого типа с возможностью иметь несколько типов параметров (как рассмотрено в предыдущем разделе), то вы можете ввести параметр шаблона RT для возвращаемого типа со значением по умолчанию, представляющим собой общий тип двух аргументов. Существует несколько возможностей.

1. Можно использовать оператор ?: непосредственно. Однако, поскольку мы вынуждены применить оператор ?: до объявления параметров вызова a и b, мы можем использовать только их типы:

basics/maxdefault1.hpp

```
#include <type_traits>
template < typename T1, typename T2,
           typename RT = std::decay_t<decltype(true?T1():T2())>>
RT max(T1 a, T2 b)
{
    return b < a ? a : b;
}
```

Обратите внимание на применение std::decay_t<> для гарантии того, что не будет возвращена ссылка¹⁰.

Обратите также внимание на то, что данная реализация требует возможности вызова конструкторов по умолчанию для передаваемых типов. Есть и другое решение, использующее std::declval, которое, однако, делает объявление еще более сложным (см. соответствующий пример в разделе 11.2.3).

2. Можно использовать свойство типа std::common_type<> для указания значения по умолчанию для возвращаемого типа:

basics/maxdefault3.hpp

```
#include <type_traits>
template<typename T1, typename T2,
          typename RT = std::common_type_t<T1,T2>>
RT max(T1 a, T2 b)
{
    return b < a ? a : b;
}
```

И вновь обратите внимание на то, что std::common_type<> обеспечивает низведение типа, так что из функции не может быть возвращена ссылка.

В любом случае вы можете использовать значение возвращаемого типа по умолчанию:

auto a = ::max(4, 7.2);

или явно указать возвращаемый тип после типов прочих аргументов:

auto b = ::max<double,int,long double>(7.2, 4);

¹⁰ В C++11 вместо std::decay_t<...> следовало использовать typename std::decay<...>::type (см. раздел 2.8).

Однако мы опять сталкиваемся с проблемой необходимости определения трех типов, чтобы иметь возможность указывать тип возвращаемого значения. Нам нужна возможность указать тип возвращаемого значения в качестве первого параметра шаблона, но при этом быть в состоянии вывести его из типов аргументов. В принципе можно иметь аргументы по умолчанию для ведущих параметров шаблона функции, даже если за ними следуют параметры без аргументов по умолчанию:

```
template<typename RT = long, typename T1, typename T2>
RT max(T1 a, T2 b)
{
    return b < a ? a : b;
}
```

С использованием этого определения можно выполнить следующие вызовы:

```
int i;
long l;
...
max(i, 1);           // Возврат long (аргумент по умолчанию
                     // параметра шаблона для возвращаемого типа)
max<int>(4, 42); // Возврат явно указанного int
```

Однако этот подход имеет смысл, только если имеется некоторый “естественный” тип по умолчанию для параметра шаблона. В данном случае нам нужен аргумент по умолчанию, который зависит от предыдущих параметров шаблона. В принципе это возможно, как вы узнаете из раздела 26.5.1, но этот метод зависит от свойств типов и усложняет определение.

По всем указанным причинам наилучшее и наиболее простое решение заключается в том, чтобы позволить компилятору вывести тип возвращаемого значения, как это предложено в разделе 1.3.2.

1.5. Перегрузка шаблонов функций

Как и обычные функции, шаблоны функций также могут быть перегружены. То есть вы можете иметь различные определения функций с одним и тем же именем функции, так что, когда это имя используется в вызове функции, компилятор C++ должен решить, какой из нескольких кандидатов должен быть вызван. Правила принятия такого решения могут стать довольно сложными даже при отсутствии шаблонов. В этом разделе мы обсудим перегрузки с использованием шаблонов. Если вы не знакомы с основными правилами перегрузки без шаблонов, пожалуйста, обратитесь к приложению B, “Разрешение перегрузки”, содержащему подробный обзор правил разрешения перегрузки.

Следующая короткая программа иллюстрирует перегрузку шаблона функции:

basics/max2.cpp

```
// Максимальное из двух значений int:
int max(int a, int b)
{
    return b < a ? a : b;
}
```

```
// Максимальное из двух значений любого типа:
template<typename T>
T max(T a, T b)
{
    return b < a ? a : b;
}

int main()
{
    ::max(7,42);           // Нешаблонная функция для двух int
    ::max(7.0,42.0);       // max<double> (вывод аргументов)
    ::max('a','b');        // max<char> (вывод аргументов)
    ::max<>(7,42);         // max<int> (вывод аргументов)
    ::max<double>(7,42);   // max<double> (вывода аргументов нет)
    ::max('a',42.7);       // Нешаблонная функция для двух int
}
```

Как показано в этом примере, нешаблонная функция может существовать с шаблоном функции с тем же именем (который может быть инстанцирован с тем же типом). При всех прочих равных факторах, процесс разрешения перегрузки предпочитает нешаблонные функции тем, которые генерируются из шаблона. Первый вызов подпадает под это правило:

```
::max(7,42); // Идеальное соответствие нешаблонной функции для двух int
```

Если шаблон может генерировать функцию с лучшим соответствием, то будет выбран шаблон. Это подтверждается вторым и третьим вызовами функции `max()`:

```
::max(7.0,42.0); // max<double> (вывод аргументов)
::max('a','b'); // max<char> (вывод аргументов)
```

Здесь шаблон соответствует вызову лучше нешаблонной функции, потому что не требует преобразования типа `double` или `char` в тип `int` (см. правила разрешения перегрузки в разделе B.2).

Можно также явно указать пустой список аргументов шаблона. Этот синтаксис указывает, что разрешить вызов может только шаблон, но параметры шаблона должны быть выведены из аргументов вызова:

```
::max<>(7, 42); // max<int> (вывод аргументов)
```

Поскольку автоматическое преобразование типов не рассматривается при выводе параметров шаблона, но учитывается для параметров обычной функции, последний вызов использует нешаблонную функцию (при этом и 'a', и 42.7 преобразуются в `int`):

```
::max('a', 42.7); // Только нешаблонная функция допускает
                  // нетривиальные преобразования
```

Интересным примером является перегрузка шаблона, в котором можно явно задать только тип возвращаемого значения:

basics/maxdefault4.hpp

```
template<typename T1, typename T2>
auto max(T1 a, T2 b)
{
    return b < a ? a : b;
}

template<typename RT, typename T1, typename T2>
RT max(T1 a, T2 b)
{
    return b < a ? a : b;
}
```

Теперь можно вызвать `max()`, например, следующим образом:

```
auto a = ::max(4, 7.2); // Использован первый шаблон
auto b = ::max<long double>(7.2, 4); // Использован второй шаблон
```

Однако при вызове

```
auto c = ::max<int>(4, 7.2); // Ошибка: соответствуют оба шаблона
```

ему соответствуют оба шаблона, что приводит к тому, что процесс перегрузки не в состоянии выбрать ни один из них, и мы получаем ошибку неоднозначности. Таким образом, при перегрузке шаблонов функций следует убедиться, что любому вызову соответствует только один из них.

Полезным примером может быть перегрузка шаблона максимума для указателей и обычных строк в стиле C:

basics/max3val.cpp

```
#include <cstring>
#include <string>

// максимальное из двух значений любого типа:
template<typename T>
T max(T a, T b)
{
    return b < a ? a : b;
}

// Максимальный из двух указателей определяется значениями,
// на которые указывают указатели, а не значениями указателей
template<typename T>
T* max(T* a, T* b)
{
    return *b < *a ? a : b;
}

// Максимальная из двух строк в стиле C:
char const* max(char const* a, char const* b)
{
    return std::strcmp(b, a) < 0 ? a : b;
}
```

```

int main()
{
    int a = 7;
    int b = 42;
    auto m1 = ::max(a, b); // max() для двух значений типа int

    std::string s1 = "hey";
    std::string s2 = "you";
    auto m2 = ::max(s1, s2); // max() для двух std::string

    int* p1 = & b;
    int* p2 = & a;
    auto m3 = ::max(p1, p2); // max() для двух указателей

    char const* x = "hello";
    char const* y = "world";
    auto m4 = ::max(x, y); // max() для двух строк в стиле C
}

```

Обратите внимание на то, что во всех перегрузках `max()` мы передаем аргументы по значению. В общем, это хорошая идея — при перегрузке шаблонов функций не изменять больше необходимого. Вы должны ограничивать свои изменения количеством параметров или явным указанием параметров шаблона. В противном случае могут возникнуть неожиданные эффекты. Например, если вы реализовали свой шаблон функции `max()` для передачи аргументов по ссылке и перегрузили его для двух строк в стиле C, передаваемых по значению, то вы не сможете использовать версию с тремя аргументами для вычисления максимальной из трех C-строк:

basics/max3ref.cpp

```

#include <cstring>

// Максимальное из двух значений (передача по ссылке)
template<typename T>
T const& max(T const& a, T const& b)
{
    return b < a ? a : b;
}

// Максимальная из двух строк (передача по значению)
char const* max(char const* a, char const* b)
{
    return std::strcmp(b, a) < 0 ? a : b;
}

// Максимальное из трех значений (передача по ссылке)
template<typename T>
T const& max(T const& a, T const& b, T const& c)
{
    return max(max(a, b), c); // Ошибка при передаче
                                // аргументов в max(a,b) по значению
}

```

```

int main()
{
    auto m1 = ::max(7, 42, 68); // ОК
    char const* s1 = "frederic";
    char const* s2 = "anica";
    char const* s3 = "lucas";
    auto m2 = ::max(s1, s2, s3); // ОШИБКА времени выполнения
                                // (неопределенное поведение)
}

```

Проблема в том, что если вы вызовете `max()` для трех строк в стиле C, инструкция

```
return max(max(a,b),c);
```

приведет к ошибке времени выполнения, потому что для C-строк `max(a,b)` создает новое, временное локальное значение, которое возвращается по ссылке, но это временное значение становится недействительным, как только оператор `return` завершается, оставляя `main()` с висячей ссылкой. К сожалению, это довольно тонкая ошибка, которая может проявляться не во всех случаях¹¹.

Заметим, что, напротив, первый вызов функции `max()` в функции `main()` от этой проблемы не страдает. В нем создаются временные переменные для аргументов 7, 42 и 68, но эти временные переменные создаются в функции `main()`, где они и хранятся до тех пор, пока инструкция не будет полностью выполнена.

Это только один пример кода, который может вести себя не так, как ожидается, из-за точного применения правил разрешения перегрузки. Кроме того, убедитесь, что все перегруженные версии функции объявлены до вызова. Тот факт, что не все перегруженные функции видимы в момент вызова, может иметь важное значение. Например, определение версии функции `max()` с тремя аргументами, которому не видимо объявление специализированной двухаргументной версии функции `max()` для `int`, приводит к тому, что в версии с тремя аргументами используется двухаргументный шаблон:

`basics/max4.cpp`

```

#include <iostream>

// Максимальное из двух значений любого типа:
template<typename T>
T max(T a, T b)

{
    std::cout << "max<T>() \n";
    return b < a ? a : b;
}

// Максимальное из трех значений любого типа:
template<typename T>
T max(T a, T b, T c)

```

¹¹ В общем случае соответствующий стандарту компилятор даже не может отклонить этот код при компиляции.

```

{
    return max(max(a, b), c); // Использует шаблонную версию даже для
}                                // int, так как следующее объявление
                                // встречается слишком поздно:

// Максимальное из двух значений типа int:
int max(int a, int b)
{
    std::cout << "max(int,int)\n";
    return b < a ? a : b;
}

int main()
{
    ::max(47, 11, 33); // Ой: использует max<T>(), а не max(int,int)
}

```

Подробнее этот вопрос будет рассматриваться в разделе 13.2.

1.6. А не должны ли мы?..

Вероятно, даже эти примеры простых шаблонов функций могут вызвать дальнейшие вопросы. Три вопроса встречаются настолько часто, что мы хотим вкратце обсудить их здесь.

1.6.1. Передача по значению или по ссылке

Вы можете спросить, почему мы в основном объявляем функции с передачей аргументов по значению вместо использования ссылок. В общем случае передача по ссылке рекомендуется для типов, отличных от дешевых простых типов (таких как фундаментальные типы или `std::string_view`), потому что при этом не создаются ненужные копии.

Однако по ряду причин передача по значению часто оказывается лучшим решением.

- Простой синтаксис.
- Лучшая оптимизация кода компилятором.
- Семантика перемещения зачастую делает копирование дешевым.
- Иногда копирование или перемещение не выполняется вовсе.

Кроме того, для шаблонов в игру вступают дополнительные аспекты.

- Шаблон может использоваться как для простых, так и для сложных типов, так что выбор подхода для сложных типов может оказаться контрпродуктивным для простых типов.
- Как автор вызывающего кода, вы все равно можете принять решение о передаче аргументов по ссылке с использованием `std::ref()` и `std::cref()` (см. раздел 7.3).

- Хотя передача строковых литералов или простых массивов всегда оказывается проблемой, их передача по ссылке часто становится еще большей проблемой.

Все эти вопросы будут подробно рассмотрены в главе 7, “По значению или по ссылке?”. Пока что в книге мы будем передавать аргументы по значению, если только не окажется, что некоторая функциональность возможна лишь при использовании ссылок.

1.6.2. Об использовании `inline`

В общем случае шаблоны функций не должны быть объявлены с использованием `inline`. В отличие от обычных невстраиваемых функций мы можем определить шаблоны функций, не являющиеся `inline`, в заголовочном файле и включать этот заголовочный файл в несколько единиц трансляции.

Единственным исключением из этого правила являются полные специализации шаблонов для конкретных типов, так что результирующий код более не является обобщенным (в нем определены все параметры шаблона). Более подробно этот вопрос рассматривается в разделе 9.2.

С точки зрения строгого определения языка `inline` означает только то, что определение функции может встречаться в программе несколько раз. Однако это также означает подсказку компилятору, что вызовы этой функции должны быть встраиваемыми; в некоторых случаях это может позволить генерировать более эффективный код, но также во многих других случаях может сделать код менее эффективным. В настоящее время компиляторы обычно способны решать такие вопросы без дополнительных намеков в виде ключевого слова `inline`. Однако пока что при принятии решений компиляторы по-прежнему учитывают наличие ключевого слова `inline`.

1.6.3. Об использовании `constexpr`

Начиная с C++11, для возможности вычисления некоторых значений во время компиляции можно использовать ключевое слово `constexpr`. Это имеет смысл для множества шаблонов.

Например, чтобы иметь возможность использовать функцию получения максимума во время компиляции, необходимо объявить ее следующим образом:

basics/maxconstexpr.hpp

```
template<typename T1, typename T2>
constexpr auto max(T1 a, T2 b)
{
    return b < a ? a : b;
}
```

При этом шаблон функции можно использовать во время компиляции в соответствующих контекстах, таких, например, как объявление размера массива:

```
int a[:max(sizeof(char), 1000u)];
```

или размера `std::array<>`:

```
std::array<std::string, ::max(sizeof(char), 1000u)> arr;
```

Обратите внимание на то, что мы передаем 1000 как значение типа `unsigned int`, чтобы избежать предупреждения о сравнении в шаблоне знаковых и беззнаковых значений.

В разделе 8.2 обсуждаются другие примеры использования `constexpr`. Однако, чтобы сосредоточиться на фундаментальных вопросах, мы обычно будем опускать ключевое слово `constexpr` при обсуждении других характеристик шаблона.

1.7. Резюме

- Шаблоны функций определяют семейство функций для различных аргументов шаблона.
- При передаче аргументов параметрам функции, зависящим от параметров шаблонов, шаблоны функций выводят параметры шаблонов для инстанцирования функций для соответствующих аргументов.
- Ведущие параметры шаблонов можно указывать явным образом.
- Для параметров шаблонов можно определить аргументы по умолчанию. Они могут ссылаться на предшествующие параметры шаблонов, а за ними могут следовать параметры, не имеющие аргументов по умолчанию.
- Шаблоны функций можно перегружать.
- При перегрузке шаблонов функций другими шаблонами функций следует гарантировать, что каждому вызову соответствует только один из них.
- При перегрузке шаблонов функций следует ограничивать вносимые изменения явным указанием параметров шаблона.
- Следует убедиться, что все перегруженные версии шаблонов функций размещены в программе до их вызовов.

Глава 2

Шаблоны классов

Подобно функциям, классы могут также быть параметризованы одним или несколькими типами. Классы контейнеров, которые используются для управления элементами определенного типа, являются типичным примером этой возможности. Такие классы контейнеров можно реализовать с помощью шаблонов классов; тип элементов при этом остается открытым. В этой главе мы используем в качестве примера шаблона класса стек.

2.1. Реализация шаблона класса `Stack`

Как и в случае шаблонов функций, мы объявляем и определяем класс `Stack<T>` в заголовочном файле следующим образом:

`basics/stack1.hpp`

```
#include <vector>
#include <cassert>
template<typename T>
class Stack
{
private:
    std::vector<T> elems;           // Элементы

public:
    void push(T const& elem);     // Внесение в стек
    void pop();                   // Снятие со стека
    T const& top() const;         // Верхний элемент стека
    bool empty() const            // Пуст ли стек
    {
        return elems.empty();
    }
};

template<typename T>
void Stack<T>::push(T const& elem)
{
    elems.push_back(elem); // Добавление копии переданного элемента
}

template<typename T>
void Stack<T>::pop()
{
    assert(!elems.empty());
    elems.pop_back();           // Удаление последнего элемента
}

template<typename T>
T const& Stack<T>::top() const
```

```
{
    assert(!elems.empty());
    return elems.back(); // Возврат последнего элемента
}
```

Как видите, шаблон класса реализован с использованием шаблона класса `vector<T>` из стандартной библиотеки C++. В результате нам не нужно реализовывать управление памятью, копирующий конструктор и оператор присваивания, так что мы можем сосредоточиться на интерфейсе данного шаблона класса.

2.1.1. Объявление шаблона класса

Объявление шаблона класса подобно объявлению шаблона функции: перед объявлением следует объявить один или несколько идентификаторов в качестве параметров типов. Обычно в качестве идентификатора используется `T`:

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

Здесь также ключевое слово `class` может быть использовано вместо ключевого слова `typename`:

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

Внутри шаблона класса имя `T` может использоваться так же, как и любой другой тип, для объявления членов-данных и функций-членов. В приведенном далее примере `T` используется для объявления типа элементов как вектора, в объявлении метода `push()` как функции, которая получает `T` в качестве аргумента, и для объявления `top()` как функции, возвращающей `T`:

```
template<typename T>
class Stack
{
private:
    std::vector<T> elems; // Элементы

public:
    void push(T const& elem); // Внесение в стек
    void pop(); // Снятие со стека
    T const& top() const; // Верхний элемент стека
    bool empty() const // Пуст ли стек
    {
        return elems.empty();
    }
};
```

Типом этого класса является `Stack<T>`, где `T` представляет собой параметр шаблона. Таким образом, `Stack<T>` придется использовать всякий раз, когда тип этого класса применяется в объявлении, за исключением тех случаев, когда аргументы шаблона могут быть выведены. Однако внутри шаблона класса имя класса, за которым не следуют аргументы шаблона, представляет класс с параметрами шаблона в качестве аргументов (подробности см. в разделе 13.2.3).

Если, например, вам нужно объявить свой собственный копирующий конструктор и оператор присваивания, обычно это выглядит так:

```
template<typename T>
class Stack
{
    ...
    Stack(Stack const&);           // Копирующий конструктор
    Stack& operator=(Stack const&); // Оператор присваивания
    ...
};
```

что формально эквивалентно следующему коду:

```
template<typename T>
class Stack
{
    ...
    Stack(Stack<T> const&);       // Копирующий конструктор
    Stack<T>& operator=(Stack<T> const&); // Оператор присваивания
    ...
};
```

Однако обычно `<T>` сигнализирует об особой обработке параметров шаблона, поэтому лучше использовать первую форму записи.

Однако вне структуры класса такое указание является необходимым:

```
template<typename T>
bool operator==(Stack<T> const& lhs, Stack<T> const& rhs);
```

Обратите внимание на то, что в местах, где требуется имя, а не тип класса, можно использовать только `Stack`. Это, в частности, относится к указанию имен конструкторов (но не их аргументов) и деструкторов.

Заметим также, что в отличие от нешаблонных классов нельзя объявлять или определять шаблоны классов внутри функций или в области видимости блока. В общем случае шаблоны могут быть определены только в глобальной области видимости, области видимости пространства имен или внутри объявлений классов (подробнее см. в разделе 12.1).

2.1.2. Реализация функций-членов

Для того чтобы определить функцию-член шаблона класса, нужно указать, что это шаблон функции; при этом необходимо использовать полностью квалифицированный тип шаблона класса. Таким образом, реализация функции-члена `push()` типа `Stack<T>` имеет следующий вид:

```
template<typename T>
void Stack<T>::push(T const& elem)
```

```
{
    elems.push_back(elem); // Добавление копии переданного элемента
}
```

В этом случае вызывается функция `push_back()` вектора элементов, которая добавляет элемент в конец вектора.

Заметим, что функция `pop_back()` вектора удаляет последний элемент, но не возвращает его, что связано с вопросами безопасности исключений. Реализовать полностью безопасную в плане исключений функцию `pop()`, возвращающую удаленный элемент, невозможно (этот вопрос впервые был рассмотрен Томом Каргиллом (Tom Cargill) в [31]; кроме того, этот вопрос детально рассматривается в [68]). Однако если игнорировать небезопасность данной функции в плане исключений, то можно реализовать функцию `pop()`, возвращающую только что удаленный элемент. Для этого мы просто используем `T` для объявления локальной переменной соответствующего типа.

```
template<typename T>
T Stack<T>::pop()
{
    assert(!elems.empty());
    T elem = elems.back(); // Сохранение копии последнего элемента
    elems.pop_back();      // Удаление последнего элемента
    return elem;           // Возврат копии сохраненного элемента
}
```

Поскольку поведение функций вектора `back()` (возвращающей последний элемент) и `pop_back()` (удаляющей последний элемент) не определено для случая, когда вектор не содержит ни одного элемента, требуется проверка, не является ли стек пустым. Если он пуст, мы завершаем программу с использованием `assert`, так как вызов `pop()` для пустого стека является ошибкой. Так же мы поступаем и в функции `top()`, которая возвращает (но не удаляет) элемент, находящийся на вершине стека:

```
template<typename T>
T const& Stack<T>::top() const
{
    assert(!elems.empty());
    return elems.back(); // Возврат последнего элемента
}
```

Конечно, точно так же, как и в случае любых других функций-членов, функции-члены шаблонов классов можно реализовать как встраиваемые функции, располагающиеся внутри объявления класса, например:

```
template<typename T>
class Stack
{
    ...
    void push(T const& elem)
    {
        elems.push_back(elem); // Добавление копии переданного элемента
    }
    ...
};
```

2.2. Использование шаблона класса `Stack`

Для того чтобы использовать объект шаблона класса, до C++17 требовалось явно указать аргументы шаблона¹. В приведенном ниже примере показано, как используется шаблон класса `Stack<>`.

`basics/stack1test.cpp`

```
#include "stack1.hpp"
#include <iostream>
#include <string>

int main()
{
    Stack<int> intStack;           // Стек элементов int
    Stack<std::string> stringStack; // Стек элементов string

    // Работа со стеком целых чисел
    intStack.push(7);
    std::cout << intStack.top() << '\n';

    // Работа со стеком строк
    stringStack.push("hello");
    std::cout << stringStack.top() << '\n';
    stringStack.pop();
}
```

Объявление `Stack<int>` указывает, что внутри шаблона класса в качестве типа `T` будет использоваться `int`. Таким образом, `intStack` создается как объект на базе вектора с элементами типа `int`, и для всех вызываемых функций-членов инстанцируется код для этого типа. Аналогично путем объявления и использования `Stack<std::string>` создается объект на базе вектора, элементами которого являются строки, и для каждой из вызываемых функций-элементов инстанцируется код для этого типа.

Заметим, что инстанцирование происходит только для *вызываемых функций (членов) шаблона*. Для шаблонов классов функции-члены инстанцируются только при их использовании. Очевидно, что такой подход позволяет сэкономить время, память и использовать шаблоны классов только частично (о чем мы поговорим подробнее в разделе 2.3).

В данном примере инстанцируются конструктор по умолчанию, а также функции `push()` и `top()` как для значений типа `int`, так и для значений типа `string`. Однако функция `pop()` инстанцируется только для строк. Если шаблон класса имеет статические члены, то они инстанцируются однократно для каждого типа, для которого используется шаблон класса.

¹ В C++17 введен вывод аргументов шаблонов класса, который позволяет опустить шаблоны аргумента, если они могут быть выведены из конструктора. Этот вопрос рассматривается в разделе 2.9.

Тип инстанцированного шаблона класса можно использовать так же, как и любой другой тип. Вы можете квалифицировать его с помощью ключевых слов `const` или `volatile`, а также создавать на его основе массивы или ссылки. Его также можно использовать как часть определения типа с помощью `typedef` или `using` (об определениях типов рассказывается в разделе 2.8) или в качестве параметра типа при создании другого шаблонного типа, например:

```
void foo(Stack<int> const& s) // Параметр s представляет собой стек
{
    using IntStack =
        Stack<int>;           // IntStack - псевдоним для Stack<int>
    Stack<int> istack[10];    // istack - массив из 10 стеков
    IntStack istack2[10];    // istack2 - такой же массив того же типа
    ...
}
```

Аргументы шаблона могут быть любого типа, такими как указатели на `float` или даже стеками элементов типа `int`:

```
Stack<float*> floatPtrStack; // Стек указателей на float
Stack<Stack<int>> intStackStack; // Стек стеков элементов int
```

Единственным требованием является то, что любая вызываемая операция должна быть возможна для этого типа.

До принятия стандарта C++11 между двумя закрывающими угловыми скобками требовалось помещать пробел:

```
Stack<Stack<int> > intStackStack; // Работает для всех версий C++
```

Если этого не сделать, получался оператор `>>`, что приводило к синтаксической ошибке:

```
Stack<Stack<int>> intStackStack; // Ошибка до C++11
```

Причиной такого поведения в старых версиях C++ было то, что при первом проходе компилятора C++ выполнялось разделение исходного кода на лексемы независимо от семантики кода. Однако, поскольку отсутствие пробела стало распространенной ошибкой, требовавшей соответствующего сообщения об ошибках, было принято решение учитывать семантику кода в любом случае, и в стандарте C++11 требование ставить пробел между двумя закрывающими угловыми скобками в шаблонах было убрано (см. раздел 13.3.1).

2.3. Частичное использование шаблонов классов

Шаблон класса обычно выполняет несколько операций над аргументами шаблона, для которых он инстанцируется (в том числе конструирование и уничтожение). Это может привести к впечатлению, что аргументы шаблона должны предоставлять все операции, необходимые для всех функций-членов шаблона класса. Но это не так: аргументы шаблона обязаны предоставлять только те операции, которые *необходимы* (а не те, которые *могут* потребоваться).

Пусть, например, класс `Stack<T>` предоставляет функцию-член `printOn()` для вывода содержимого стека, вызывающую `operator<<` для каждого элемента:

```
template<typename T>
class Stack
{
    ...
    void printOn(std::ostream& strm) const
    {
        for (T const& elem : elems)
        {
            strm << elem << ' '; // Вызов << для каждого элемента
        }
    }
};
```

При этом вы все равно можете использовать этот класс для элементов, для которых не определен `operator<<`:

```
Stack<std::pair<int, int>> ps; // std::pair<> не имеет operator<<
ps.push({4, 5}); // OK
ps.push({6, 7}); // OK
std::cout << ps.top().first << '\n'; // OK
std::cout << ps.top().second << '\n'; // OK
```

И только если вы вызовете `printOn()` для такого стека, то получите ошибку времени компиляции, потому что компилятор не сможет инстанцировать вызов `operator<<` для этого конкретного типа элемента:

```
ps.printOn(std::cout); // Ошибка: operator<< не поддерживается
                      // для данного типа элементов
```

2.3.1. Концепты

Это приводит к вопросу: как нам узнать, какие операции необходимы, чтобы шаблон мог быть инстанцирован? Термин *концепт* (concept) часто используется для обозначения множества ограничений, которые многократно требуются в библиотеке шаблонов. Например, стандартная библиотека C++ опирается на такие ограничения, как *итератор с произвольным доступом и конструируемость по умолчанию*.

В настоящее время (т.е. по стандарту C++17) концепты могут быть более или менее выражены только в документации (например, с помощью комментариев в коде). Это может стать серьезной проблемой из-за того, что нарушение ограничений может привести к жутким сообщениям об ошибках (см. раздел 9.4).

В течение многих лет были неоднократные попытки ввести определения и проверки концептов в качестве возможности языка. Однако до стандарта C++17 включительно этот подход так и не был стандартизован.

Начиная с C++11, существует возможность выполнения по крайней мере некоторых проверок основных ограничений с использованием ключевого слова `static_assert` и некоторых предопределенных свойств типов, например:

```
template<typename T>
class C
{
    static_assert(std::is_default_constructible<T>::value,
                 "Класс C требует элементы, "
                 "конструируемые по умолчанию");
    ...
};
```

Без этого статического утверждения, если требуется конструктор по умолчанию, все равно произойдет ошибка компиляции. Однако в таком случае сообщение об ошибке может содержать всю историю инстанцирования шаблона — от исходной причины неудачного инстанцирования до определения шаблона, в котором была обнаружена ошибка (см. раздел 9.4).

Однако для проверки, например, того, что объекты типа *T* предоставляют определенную функцию-член или что их можно сравнивать с помощью оператора *<*, требуется куда более сложный код. Подробный пример такого кода приведен в разделе 19.6.3.

Детальное обсуждение концептов в C++ содержится в приложении Д, “Концепты”.

2.4. Друзья

Вместо печати содержимого стека с помощью функции *printOn()* лучше реализовать *operator<<* для стека. Однако, как обычно, *operator<<* должен быть реализован как свободная функция, которая затем может вызывать функцию-член *printOn()*:

```
template<typename T>
class Stack
{
    ...
    void printOn(std::ostream& strm) const
    {
        ...
    }
    friend std::ostream& operator<< (std::ostream& strm,
                                         Stack<T> const& s)
    {
        s.printOn(strm);
        return strm;
    }
};
```

Обратите внимание: это означает, что *operator<<* для класса *Stack<T>* является не шаблоном функции, а “обычной” функцией, инстанцируемой при необходимости с использованием шаблона класса².

Однако при попытках *объявить* дружественную функцию и *определить* ее позже все оказывается более сложным. Фактически у нас есть два варианта.

²Это шаблонизированная сущность (templated entity); см. раздел 12.1.

2.5. Специализации шаблонов классов

Шаблон класса можно специализировать для конкретных аргументов шаблона. Так же, как и в случае перегрузки шаблонов функций (см. раздел 1.5), специализированные шаблоны классов позволяют оптимизировать реализации для конкретных типов или корректировать неверное поведение определенных типов при инстанцировании шаблона класса. Однако при специализации шаблона класса необходимо специализировать все его функции-члены. Хотя можно специализировать и отдельную функцию-член шаблона класса, но если сделать это, то потом нельзя будет специализировать целый шаблон класса, которому принадлежит специализированный член.

Чтобы специализировать шаблон класса, следует объявить класс с предваряющей конструкцией `template<>` и указать типы, для которых специализируется шаблон класса. Типы используются в качестве аргументов шаблона и должны задаваться непосредственно после имени класса:

```
template<>
class Stack<std::string>
{
    ...
};
```

Для таких специализаций любая функция-член должна определяться как “обычная” функция-член с заменой каждого включения `T` специализированным типом:

```
void Stack<std::string>::push(std::string const& elem)
{
    elems.push_back(elem); // Добавление копии переданного элемента
}
```

Ниже приведен завершенный пример специализации `Stack<>` для типа `std::string`.

basics/stack2.hpp

```
#include "stack1.hpp"
#include <deque>
#include <string>
#include <cassert>

template<>
class Stack<std::string>
{
private:
    std::deque<std::string> elems; // Элементы
public:
    void push(std::string const&); // Внесение элемента в стек
    void pop(); // Удаление элемента из стека
    std::string const& top() const; // Возврат последнего элемента
```

```

bool empty() const           // Проверка пустоты стека
{
    return elems.empty();
}

void Stack<std::string>::push(std::string const& elem)
{
    elems.push_back(elem); // Добавление копии переданного элемента
}

void Stack<std::string>::pop()
{
    assert(!elems.empty());
    elems.pop_back();      // Удаление последнего элемента
}

std::string const& Stack<std::string>::top() const
{
    assert(!elems.empty());
    return elems.back();   // Возврат последнего элемента
}

```

В этом примере специализация использует семантику ссылок для передачи строкового аргумента функции `push()`, которая имеет больше смысла для данного конкретного типа (еще лучше передавать универсальную ссылку; этот вопрос рассматривается в разделе 6.1).

Еще одним отличием является применение дека вместо вектора для управления элементами в стеке. Хотя здесь такая замена не дает особых преимуществ, это сделано, чтобы продемонстрировать, что реализация специализации может значительно отличаться от реализации первичного шаблона.

2.6. Частичная специализация

Шаблоны классов могут быть специализированы частично. Можно определить частные реализации для определенных условий, но при этом некоторые параметры шаблона все еще остаются задаваемыми пользователем. Например, можно определить отдельную специализацию `Stack<>` для указателей:

basics/stackpartspec.hpp

```

#include "stack1.hpp"

// Частичная специализация класса Stack<> для указателей:
template<typename T>
class Stack<T*>
{
private:
    std::vector<T*> elems; // Элементы
}

```

```

public:
    void push(T*);           // Добавление элемента в стек
    T* pop();                // Удаление элемента из стека
    T* top() const;          // Возврат последнего элемента
    bool empty() const       // Проверка пустоты стека
    {
        return elems.empty();
    }
};

template<typename T>
void Stack<T*>::push(T* elem)
{
    elems.push_back(elem); // Добавление копии переданного элемента
}

template<typename T>
T* Stack<T*>::pop()
{
    assert(!elems.empty());
    T* p = elems.back();
    elems.pop_back();        // Удаление последнего элемента и его
    return p;                // возврат (в отличие от общего случая)
}

template<typename T>
T* Stack<T*>::top() const
{
    assert(!elems.empty());
    return elems.back();     // Возврат копии последнего элемента
}

```

С помощью

```

template<typename T>
class Stack<T*>
{
};

```

мы определяем шаблон класса, все еще параметризованный с использованием `T`, но специализированный для указателей (`Stack<T*>`).

Еще раз обратите внимание на то, что специализация может предоставлять (несколько) иной интерфейс. Например, в приведенном коде функция `pop()` возвращает хранимый указатель, так что пользователь шаблона класса может вызвать `delete` для удаляемого значения, если оно было создано с помощью оператора `new`:

```

Stack<int*> ptrStack; // Стек указателей (специальная реализация)

ptrStack.push(new int{42});
std::cout << *ptrStack.top() << '\n';
delete ptrStack.pop();

```

Частичная специализация с несколькими параметрами

Шаблоны классов могут также специализировать взаимоотношения между несколькими параметрами шаблона. Например, для следующего шаблона класса:

```
template<typename T1, typename T2>
class MyClass
{
    ...
};
```

возможны такие частичные специализации:

```
// Частичная специализация: оба параметра имеют один тип
template<typename T>
class MyClass<T, T>
{
    ...
};

// Частичная специализация: второй параметр - int
template<typename T>
class MyClass<T, int>
{
    ...
};

// Частичная специализация: оба параметра - указатели
template<typename T1, typename T2>
class MyClass<T1*, T2*>
{
    ...
};
```

В приведенных ниже примерах показано, какие шаблоны используются в тех или иных объявлениях:

```
MyClass<int, float> mif; // Используется MyClass<T1, T2>
MyClass<float, float> mff; // Используется MyClass<T, T>
MyClass<float, int> mfi; // Используется MyClass<T, int>
MyClass<int*, float*> mp; // Используется MyClass<T1*, T2*>
```

Если одинаково хорошо подходят две специализации, объявление является неоднозначным:

```
MyClass<int, int> m; // Ошибка: годятся MyClass<T, T> и MyClass<T, int>
MyClass<int*, int*> m; // Ошибка: годятся MyClass<T, T> и MyClass<T1*, T2*>
```

Для разрешения второй неоднозначности можно воспользоваться специализацией для указателей одного и того же типа:

```
template<typename T>
class MyClass<T*, T*>
{
    ...
};
```

Детально частичная специализация рассматривается в разделе 16.4.

2.7. Аргументы шаблона класса по умолчанию

Как и для шаблонов функций, для параметров шаблона класса можно определять значения по умолчанию. Например, в класс `Stack<T>` можно добавить второй параметр, определяющий контейнер, который используется для хранения элементов, и в качестве значения по умолчанию указать тип `std::vector<T>`.

`basics/stack3.hpp`

```
#include <vector>
#include <cassert>
template<typename T, typename Cont = std::vector<T>>
class Stack
{
private:
    Cont elems;           // Элементы
public:
    void push(T const& elem); // Добавление элемента в стек
    void pop();           // Снятие элемента со стека
    T const& top() const; // Возврат элемента с вершины стека
    bool empty() const    // Проверка пустоты стека
    {
        return elems.empty();
    }
};

template<typename T, typename Cont>
void Stack<T, Cont>::push(T const& elem)
{
    elems.push_back(elem); // Добавление копии переданного элемента
}

template<typename T, typename Cont>
void Stack<T, Cont>::pop()
{
    assert(!elems.empty());
    elems.pop_back();      // Удаление последнего элемента
}

template<typename T, typename Cont>
T const& Stack<T, Cont>::top() const
{
    assert(!elems.empty());
    return elems.back();   // Возврат последнего элемента
}
```

Заметим, что теперь, когда у нас имеются два параметра шаблона, каждое определение функции-члена должно иметь эти два параметра.

```
template<typename T, typename Cont>
void Stack<T, Cont>::push(T const& elem)

{
    elems.push_back(elem); // Добавление копии переданного элемента
}
```

Этот стек можно использовать точно так же, как и раньше. Если шаблону передается только первый аргумент, представляющий тип элементов в стеке, то для хранения элементов этого типа используется вектор.

```
template<typename T, typename Cont = std::vector<T>>
class Stack
{
private:
    Cont elems; // Элементы
    ...
};
```

При объявлении объекта `Stack` в своей программе можно явно указать, какой именно контейнер должен использоваться для хранения элементов.

basics/stack3test.cpp

```
#include "stack3.hpp"
#include <iostream>
#include <deque>

int main()
{
    // Стек элементов типа int:
    Stack<int> intStack;

    // Стек элементов типа double с контейнером std::deque<>
    Stack<double, std::deque<double>> dblStack;

    // Работа со стеком элементов типа int
    intStack.push(7);
    std::cout << intStack.top() << '\n';
    intStack.pop();

    // Работа со стеком элементов типа double
    dblStack.push(42.42);
    std::cout << dblStack.top() << '\n';
    dblStack.pop();
}
```

С помощью выражения

`Stack<double, std::deque<double>>`

мы объявляем стек элементов типа `double`, который для работы с элементами использует контейнер `std::deque<>`.

2.8. Псевдонимы типов

Применение шаблонов классов можно сделать более удобным, определив новое имя для типа.

Применение `typedef` и `using`

Просто определить новое имя полного типа можно двумя способами.

1. С помощью ключевого слова `typedef`:

```
typedef Stack<int> IntStack; // typedef
void foo(IntStack const& s); // s - стек для int
IntStack istack[10]; // istack - массив из 10 стеков
```

Будем называть такое объявление *typedef-объявлением*³, а получающееся в результате имени — *typedef-именем*.

2. С помощью ключевого слова `using` (начиная с C++11):

```
using IntStack = Stack<int>; // Объявление псевдонима
void foo(IntStack const& s); // s - стек для int
IntStack istack[10]; // istack - массив из 10 стеков
```

Введенное в [36], это выражение называется *объявлением псевдонима*.

Обратите внимание: в обоих случаях новое имя определяется для уже существующего типа, а не для создаваемого нового типа. Таким образом, после

```
typedef Stack<int> IntStack;
```

или

```
using IntStack = Stack<int>;
```

`IntStack` и `Stack<int>` представляют собой две взаимозаменяемые записи одного и того же типа.

В качестве общего термина для обоих вариантов определения нового имени для существующего типа мы используем термин *объявление псевдонима типа*, а для этого нового имени — *псевдоним типа*.

В силу большей удобочитаемости (определенное имя типа находится слева от знака `=`) в оставшейся части книги при объявлении псевдонима типа мы предпочтаем синтаксис объявления псевдонима.

Шаблоны псевдонимов

В отличие от `typedef`, объявление псевдонима может быть шаблонизировано для того, чтобы предоставить удобное имя для семейства типов. Эта возможность также доступна, начиная с C++11, и называется шаблонным псевдонимом, или, для единообразности именования, *шаблоном псевдонима* (*alias template*)⁴.

³ Мы преднамеренно используем это слово. Ключевое слово `typedef` изначально было сокращением *type definition* (определение типа). Однако в C++ “определение типа” имеет иное значение (например, определение класса или перечисления). Здесь же речь идет об альтернативном имени (псевдониме, alias) для существующего типа, создаваемом с помощью ключевого слова `typedef`.

⁴ Шаблоны псевдонимов иногда (неверно) называют *typedef-шаблонами*, поскольку они выполняют то же, что делал бы `typedef`, если бы он мог создавать шаблоны.

Приведенный ниже шаблон псевдонима `DequeStack`, параметризованный типом элементов `T`, раскрывается в `Stack`, который хранит свои элементы в контейнере `std::deque`:

```
template<typename T>
using DequeStack = Stack<T, std::deque<T>>;
```

Следовательно, как шаблоны классов, так и шаблоны псевдонимов могут использоваться в качестве параметризованных типов. Но шаблон псевдонима просто дает новое имя существующему типу, который может использоваться и сам по себе. И `DequeStack<int>`, и `Stack<int, std::deque<int>>` представляют собой один и тот же тип.

Еще раз обратите внимание на то, что в общем случае шаблоны могут быть объявлены и определены только в глобальной области видимости и области видимости пространства имен или внутри объявления класса.

Шаблоны псевдонимов для типов-членов

Шаблоны псевдонимов особенно полезны для определения сокращений для типов, являющихся членами шаблонов классов. После

```
template <typename T> struct MyType
{
    typedef ... iterator;
    ...
};
```

или

```
template <typename T> struct MyType
{
    using iterator = ...;
    ...
};
```

определение наподобие

```
template<typename T>
using MyTypeIterator = typename MyType<T>::iterator;
```

позволяет использовать

```
MyTypeIterator<int> pos;
```

вместо⁵

```
typename MyType<T>::iterator pos;
```

Суффикс `_t` свойств типов

Начиная с C++14, стандартная библиотека использует этот метод для определения сокращений для всех свойств типов в стандартной библиотеке, которые представляют собой тип. Например, чтобы иметь возможность записать

```
std::add_const_t<T> // Начиная с C++14
```

⁵ Ключевое слово `typename` необходимо, поскольку член является типом (см. раздел 5.1).

вместо

`typename std::add_const<T>::type // начиная с C++11`

стандартная библиотека определяет:

```
namespace std
{
    template<typename T> using add_const_t = typename add_const<T>::type;
}
```

2.9. Вывод аргументов шаблона класса

До C++17 было необходимо всегда указывать все типы параметров шаблона класса (если только они не имели значения по умолчанию). Начиная с C++17, это ограничение ослаблено. Теперь вы можете не определять аргументы шаблона явно, если конструктор способен *вывести* все параметры шаблона (которые не имеют значений по умолчанию).

Так, во всех предыдущих примерах кода можно использовать копирующий конструктор без указания аргументов шаблона:

```
Stack<int> intStack1;           // Стек целых чисел
Stack<int> intStack2 = intStack1; // Работает во всех версиях
Stack intStack3 = intStack1;     // Работает, начиная с C++17
```

Предоставляя конструкторы, которые получают некоторые исходные аргументы, можно поддержать вывод типа элементов стека. Например, рассмотрим стек, который может быть инициализирован одним элементом:

```
template<typename T>
class Stack
{
private:
    std::vector<T> elems; // Элементы
public:
    Stack() = default;
    Stack(T const& elem) // Инициализация стека одним элементом
        : elems({elem})
    {
    }
    ...
};
```

Это определение позволяет объявить стек следующим образом:

```
Stack intStack = 0; // Stack<int> выводится в C++17
```

Инициализация стека целым числом 0 приводит к тому, что параметр T шаблона выводится как `int`, так что инстанцируется `Stack<int>`.

Обратите внимание на следующее.

- Из-за определения конструктора с одним элементом необходимо явное создание конструктора по умолчанию (с поведением по умолчанию), потому что конструктор по умолчанию доступен только в том случае, если ни один другой конструктор не определен:

```
Stack() = default;
```

- Аргумент `elem` передается в `elems` с использованием фигурных скобок для инициализации вектора `elems` с помощью списка инициализации с единственным элементом `elem`:

```
: elems({elem})
```

Иного конструктора для явного создания вектора с единственным параметром в качестве единственного элемента не существует⁶.

Обратите внимание: в отличие от шаблонов функций аргументы шаблона класса не могут быть выведены только частично (с явным указанием только *некоторых* из аргументов шаблона). Подробнее об этом рассказывается в разделе 15.12.

Вывод аргументов шаблона класса с использованием строковых литералов

В принципе можно инициализировать стек с помощью строкового литерала:

```
Stack stringStack = "bottom"; // Будет выведен Stack<char const[7]>
```

Но это вызывает много проблем: в общем случае при передаче аргументов типа шаблона `T` по ссылке, не выполняется *низведение* параметра (т.е. не работает механизм преобразования типа массива в соответствующий тип указателя). Это означает, что мы выполнили инициализацию

```
Stack<char const[7]>
```

и используем тип `char const[7]` везде, где встречается `T`. Например, мы не можем вносить в стек строки другого размера, поскольку это будет другой тип. Подробнее этот вопрос рассматривается в разделе 7.4.

Однако при передаче аргументов типа шаблона `T` по значению выполняется *низведение*, так что тип массива превращается в соответствующий тип указателя. Таким образом, параметр вызова `T` конструктора выводится как `char const*`, так что тип самого класса выводится как `Stack<char const*>`.

По этой причине было бы целесообразно объявить конструктор так, чтобы аргумент в него передавался по значению:

```
template<typename T>
class Stack
{
private:
    std::vector<T> elems; // Элементы
public:
    Stack(T elem)          // Инициализация одним элементом по значению
        : elems({elem})    // для низведения при выводе типа
    {
    }
    ...
};
```

⁶Хуже того — существует конструктор вектора с единственным целочисленным аргументом, представляющим размер вектора, так что без использования фигурных скобок мы бы получили стек с соответствующим переданному аргументу количеством элементов.

В таком случае приведенная ниже инициализация отлично работает:

```
Stack stringStack = "bottom"; // Вывод типа Stack<char const*>
```

Однако в данном случае лучше выполнить перемещение временного значения `elem` в стек, чтобы избежать излишнего копирования:

```
template<typename T>
class Stack
{
private:
    std::vector<T> elems; // Элементы
public:
    Stack(T elem)           // Инициализация одним элементом по значению
        : elems({std::move(elem)}) {}
    ...
};
```

Правила вывода

Вместо объявления конструктора с передачей по значению существует и другое решение: поскольку обработка обычных указателей в контейнерах является источником неприятностей, мы должны запретить автоматическое выведение таких указателей для классов контейнеров.

Можно сформулировать определенные *правила вывода* (deduction guides) для предоставления дополнительных или исправления существующих выводов аргументов шаблона класса. Например, можно указать, что всякий раз, когда передается строковый литерал или строка в стиле C, создается стек для элементов типа `std::string`:

```
Stack(char const*) -> Stack<std::string>;
```

Это правило должно находиться в той же области видимости (пространстве имен), что и определение класса. Обычно оно следует за определением класса. Мы называем тип, следующий за `->`, типом, управляемым данным правилом.

Теперь объявление

```
Stack stringStack{"bottom"}; // OK: Stack<std::string> в C++17
```

выводит стек как `Stack<std::string>`. Однако приведенный ниже код по-прежнему неработоспособен:

```
Stack stringStack = "bottom"; // Выведен Stack<std::string>,
                           // но не работает
```

Мы выводим `std::string`, так что выполняется инстанцирование `Stack<std::string>`:

```
class Stack
{
private:
    std::vector<std::string> elems; // Элементы
public:
```

```

Stack(std::string const& elem) // Инициализация одним элементом
    : elems({elem})
{
}
...
} ;

```

Однако в соответствии с правилами языка при копирующей инициализации (инициализации с использованием `=`) объекта передача строкового литерала в конструктор, ожидающий `std::string`, невозможна. Поэтому инициализировать стек необходимо следующим образом:

```
Stack stringStack{"bottom"}; // Выведен Stack<std::string>, работает
```

Обратите внимание на то, что при сомнениях вывод аргумента шаблона класса выполняет копирование. После объявления `stringStack` как `Stack<std::string>` приведенные ниже инициализации объявляют тот же тип (вызывая, таким образом, копирующий конструктор) вместо инициализации стека элементом, который представляет собой стек строк:

```

Stack stack2{stringStack}; // Выводится Stack<std::string>
Stack stack3(stringStack); // Выводится Stack<std::string>
Stack stack4 = {stringStack}; // Выводится Stack<std::string>

```

Дополнительную информацию о выводе аргументов шаблонов классов можно найти в разделе 15.12.

2.10. Шаблонизированные агрегаты

Шаблонами могут быть и агрегатные классы (классы/структуры без пользовательских, явных или унаследованных конструкторов, без закрытых или защищенных нестатических членов-данных, без виртуальных функций и без виртуальных, закрытых или защищенных базовых классов). Например:

```

template<typename T>
struct ValueWithComment
{
    T value;
    std::string comment;
};

```

определяет агрегат, параметризованный типом хранимого значения `val`. Как и для любых других шаблонов классов, можно объявлять соответствующие объекты и продолжать использовать их как агрегаты:

```

ValueWithComment<int> vc;
vc.value = 42;
vc.comment = "начальное значение";

```

Начиная с C++17, можно даже определять правила вывода для шаблонов агрегатных классов:

```
ValueWithComment(char const*, char const*)
    -> ValueWithComment<std::string>;
ValueWithComment vc2 = {"hello", "начальное значение"};
```

Без правила вывода инициализация невозможна, так как `ValueWithComment` не имеет конструктора, для которого можно было бы выполнить вывод.

Класс стандартной библиотеки `std::array<>` также является агрегатом, параметризованным типом элемента и размером. Стандартная библиотека C++17 определяет для него правило вывода, которое будет рассмотрено в разделе 4.4.4.

2.11. Резюме

- Шаблон класса — это класс, который реализован с одним или несколькими параметрами типов, остающимися открытыми.
- Чтобы использовать шаблон класса, нужно указать конкретные типы в качестве аргументов шаблона. После этого шаблон класса инстанцируется (и компилируется) для указанных типов.
- Для шаблонов классов инстанцируются только те функции-члены, которые реально вызываются в программе.
- Можно специализировать шаблоны классов для определенных типов.
- Можно выполнять частичную специализацию шаблонов классов для определенных типов.
- Начиная с C++17, аргументы шаблона класса могут быть выведены автоматически на основе конструкторов.
- Можно определять шаблоны агрегатных классов.
- Параметры вызова типа шаблона низводятся, если они объявлены как передаваемые по значению.
- Шаблоны могут быть объявлены и определены в глобальной области видимости, области видимости пространства имен или в объявлении класса.

Глава 3

Нетиповые параметры шаблонов

Параметры шаблонов классов или функций не обязаны быть типами. Они могут быть и обычными величинами. В этом случае, как и для шаблонов с параметрами типа, программист создает код, в котором некоторые детали остаются открытыми до момента, когда этот код будет использоваться. Однако эти детали представляют собой не типы, а величины. При использовании такого шаблона эти величины требуется указать явно, после чего выполняется инстанцирование кода шаблона. В данной главе эта возможность продемонстрирована на примере новой версии шаблона класса стека. Кроме того, здесь приведен пример параметров шаблона функции, не являющихся типами, и рассмотрены некоторые ограничения применения этой технологии.

3.1. Параметры шаблонов классов, не являющиеся типами

В отличие от примеров реализаций стека из предыдущих глав его можно реализовать и на базе массива с фиксированным размером, в котором будут храниться элементы. Преимущество этого метода состоит в сокращении расхода ресурсов на управление памятью, независимо от того, выполняет это управление программист или стандартный контейнер. Однако возникает другая проблема: какой размер для такого стека будет оптимальным? Если указать размер меньший, чем требуется, это приведет к переполнению стека. Если задать слишком большой размер, память будет расходоваться неэффективно. Напрашивается вполне резонное решение: оставить определение этого значения на усмотрение пользователя — он должен указать максимальный размер, необходимый для работы именно с его элементами.

Для этого определим размер в качестве параметра шаблона.

basics/stacknontype.hpp

```
#include <array>
#include <cassert>
template<typename T, std::size_t Maxsize>
class Stack
{
private:
    std::array<T, Maxsize> elems; // Элементы
    std::size_t numElems; // Текущее количество элементов
public:
    Stack(); // Конструктор
    void push(T const& elem); // Добавление в стек
    void pop(); // Снятие со стека
```

```

T const& top() const;           // Возврат верхнего элемента
bool empty() const             // Проверка пустоты стека
{
    return numElems == 0;
}
std::size_t size() const       // Текущее количество элементов
{
    return numElems;
}
};

template<typename T, std::size_t Maxsize>
Stack<T, Maxsize>::Stack()
: numElems(0)                  // Изначально элементов нет
{
    // Никаких действий
}

template<typename T, std::size_t Maxsize>
void Stack<T, Maxsize>::push(T const& elem)
{
    assert(numElems < Maxsize);
    elems[numElems] = elem;      // Добавление элемента
    ++numElems;                 // Увеличение количества элементов
}

template<typename T, std::size_t Maxsize>
void Stack<T, Maxsize>::pop()
{
    assert(!empty());
    --numElems;                 // Уменьшение количества элементов
}

template<typename T, std::size_t Maxsize>
T const& Stack<T, Maxsize>::top() const
{
    assert(!empty());
    return elems[numElems-1];   // Возврат последнего элемента
}

```

Новый второй параметр шаблона `Maxsize` имеет тип `int`. Он задает размер внутреннего массива элементов стека:

```

template<typename T, std::size_t Maxsize>
class Stack
{
private:
    std::array<T, Maxsize> elems; // Элементы
    ***
};

```

Кроме того, он используется в функции `push()` для проверки заполненности стека.

```
template<typename T, std::size_t Maxsize>
void Stack<T, Maxsize>::push(T const& elem)
{
    assert(numElems < Maxsize);
    elems[numElems] = elem; // Добавление элемента
    ++numElems;           // Увеличение количества элементов
}
```

Для того чтобы использовать этот шаблон класса, следует указать как тип элементов, так и максимальный размер стека.

basics/stacknontype.cpp

```
#include "stacknontype.hpp"
#include <iostream>
#include <string>
int main()
{
    Stack<int, 20> int20Stack;           // Стек для 20 int
    Stack<int, 40> int40Stack;           // Стек для 40 int
    Stack<std::string, 40> stringStack; // Стек для 40 string

    // Работа со стеком для 20 int
    int20Stack.push(7);
    std::cout << int20Stack.top() << '\n';
    int20Stack.pop();

    // Работа со стеком для 40 строк
    stringStack.push("hello");
    std::cout << stringStack.top() << '\n';
    stringStack.pop();
}
```

Обратите внимание на то, что каждый экземпляр шаблона имеет свой собственный тип. Таким образом, `int20Stack` и `int40Stack` — это два различных типа. Преобразование этих типов один в другой — ни явное, ни неявное — не определено. Следовательно, нельзя использовать один тип вместо другого и нельзя присваивать значение одного из этих типов другому.

Остается добавить, что для параметров шаблона можно указать значения по умолчанию:

```
template<typename T = int, std::size_t Maxsize = 100>
class Stack
{
    ***
};
```

Однако с точки зрения хорошего дизайна это может не быть корректным решением в данном примере. Значения по умолчанию должны быть интуитивно корректны. Но ни тип `int`, ни максимальный размер, равный 100, не кажутся интуитивно корректными для типа обобщенного стека. Таким образом, будет лучше, если оба эти значения программист будет указывать явно, так что эти два атрибута всегда будут документированы в объявлении.

3.2. Параметры шаблонов функций, не являющиеся типами

Можно также определить параметры, не являющиеся типами, и для шаблонов функций. Например, приведенный ниже шаблон функции определяет группу функций, предназначенных для добавления некоторого значения.

basics/addvalue.hpp

```
template<int Val, typename T>
T addValue(T x)
{
    return x + Val;
}
```

Функции такого вида могут быть полезны тогда, когда функции или операции используются в качестве параметров. Например, при работе со стандартной библиотекой шаблонов экземпляр этого шаблона функции можно использовать для добавления определенного значения к каждому элементу коллекции:

```
std::transform(source.begin(),
              source.end(),           // Начало и конец источника
              dest.begin(),          // Начало приемника
              addValue<5, int>);    // Операция
```

Последний аргумент инстанцирует шаблон функции `addValue<>` для добавления 5 к переданному значению типа `int`. Получающаяся функция вызывается для каждого элемента исходной коллекции `source`, в процессе чего она преобразуется в целевую коллекцию `dest`.

Обратите внимание: вы должны указать аргумент `int` в качестве параметра `T` шаблона `addValue<>`. Вывод работает только при непосредственном вызове, а алгоритму `std::transform()` требуется полный тип для вывода типа четвертого параметра. В языке нет поддержки подстановки/вывода только некоторых параметров шаблона с последующим рассмотрением и выводом прочих параметров.

Можно также указать, что параметр шаблона выводится из предыдущего параметра — например, чтобы получить тип возвращаемого значения из переданного параметра, не являющегося типом:

```
template<auto Val, typename T = decltype(Val)>
T foo();
```

или чтобы гарантировать, что переданное значение того же типа, что и переданный тип:

```
template < typename T, T Val = T{} >
T bar();
```

3.3. Ограничения на параметры шаблонов, не являющиеся типами

Учтите, что на параметры шаблонов, не являющиеся типами, накладываются определенные ограничения. В общем случае такими параметрами могут быть только целочисленные константы (включая перечисления), указатели на объекты/функции/члены, l-ссылки на объекты или функции либо `std::nullptr_t` (тип значения `nullptr`).

Использование чисел с плавающей точкой и объектов с типом класса в качестве параметров шаблона не разрешено.

```
template<double VAT>           // Ошибка: значения с плавающей точкой в
double process(double v)       // качестве параметров шаблонов не разрешены
{
    return v * VAT;
}

template<std::string name> // Ошибка: объекты классов в качестве
class MyClass             // параметров шаблонов не разрешены
{
    ...
};
```

При передаче в качестве аргументов шаблона указателей или ссылок объекты не должны быть строковыми литералами, временными значениями, членами-данными или иными подобъектами. Эти ограничения ослаблялись с каждой новой версией C++, и до C++17 применялись дополнительные ограничения:

- в C++98 и C++03 объекты должны были иметь внешнее связывание;
- в C++11 и C++14 объекты должны были иметь внешнее или внутреннее связывание.

Таким образом, следующий код невозможен:

```
template<char const* name>
class Message
{
    ...
};

Message<"hi"> x; // Ошибка: строковый литерал "hi" не разрешен
```

Однако есть и обходные пути (зависящие от используемой версии C++):

```
extern char const s03[] = "hi"; // Внешнее связывание
char const s11[] = "hi";       // Внутреннее связывание
int main()
{
    Message<s03> m03;           // OK (все версии)
    Message<s11> m11;           // OK, начиная с C++11
    static char const s17[] = "hi"; // Связывания нет
    Message<s17> m17;           // OK, начиная с C++17
}
```

Во всех трех случаях массив константных символов инициализируется строкой "hi", и этот объект используется в качестве параметра шаблона, объявленного как `char const*`. Этот способ работает во всех версиях C++, если объект имеет внешнее связывание (s03), в C++11 и C++14, если он имеет внутреннее связывание (s11), и, начиная с C++17, если он не имеет связывания вообще.

Подробнее этот вопрос рассматривается в разделе 12.3.3, а в разделе 17.2 освещаются возможные изменения в этой области в будущем.

Избежание неверных выражений

Аргументами параметров шаблонов, не являющихся типами, могут быть любые выражения времени компиляции. Например:

```
template<int I, bool B>
class C;

*** C < sizeof(int) + 4, sizeof(int) == 4 > c;
```

Обратите, однако, внимание на то, что если в выражении используется оператор `>`, то вы должны поместить все выражение в круглые скобки, так, чтобы вложенные `>` не были восприняты как завершение списка аргументов:

```
C<42, sizeof(int)> 4 > c;           // Ошибка: первый > завершает
                                         // список аргументов шаблона
C < 42, (sizeof(int) > 4) > c; // OK
```

3.4. Тип параметра шаблона `auto`

Начиная с C++17, можно определить параметр шаблона, не являющийся типом, как обобщенный, т.е. способный принимать любой тип, который разрешен для таких параметров. Используя эту возможность, можно предоставить еще более универсальный класс стека с фиксированным размером:

basics/stackauto.hpp

```
#include <array>
#include <cassert>

template<typename T, auto Maxsize>
class Stack
{
public:
    using size_type = decltype(Maxsize);
private:
    std::array<T, Maxsize> elems; // Элементы
    size_type numElems;          // Текущее количество элементов
public:
    Stack();                     // Конструктор
    void push(T const& elem);   // Добавление в стек
    void pop();                  // Снятие со стека
    T const& top() const;        // Возврат верхнего элемента
```

```

    bool empty() const           // Проверка пустоты стека
    {
        return numElems == 0;
    }
    size_type size() const      // Текущее количество элементов
    {
        return numElems;
    }
};

// Конструктор
template<typename T, auto Maxsize>
Stack<T, Maxsize>::Stack()
    : numElems(0)              // Изначально элементов нет
{
                                // Никаких действий
}

template<typename T, auto Maxsize>
void Stack<T, Maxsize>::push(T const& elem)
{
    assert(numElems < Maxsize);
    elems[numElems] = elem;      // Добавление элемента
    ++numElems;                 // Увеличение количества элементов
}

template<typename T, auto Maxsize>
void Stack<T, Maxsize>::pop()
{
    assert(!empty());
    --numElems;                  // Уменьшение количества элементов
}

template<typename T, auto Maxsize>
T const& Stack<T, Maxsize>::top() const
{
    assert(!empty());
    return elems[numElems - 1];   // Возврат последнего элемента
}

```

Определяя

```

template<typename T, auto Maxsize>
class Stack
{
    ***
};

```

с использованием *заместителя типа* `auto`, мы указываем, что `Maxsize` является значением, тип которого еще не указан. Он может быть любым типом, который разрешен для параметров шаблонов, не являющихся типами.

Внутри шаблона класса можно использовать как значение этого параметра:

```
std::array<T, Maxsize> elems; // Элементы
```

так и его тип:

```
using size_type = decltype(Maxsize);
```

который затем можно применить, например, в качестве возвращаемого типа функции-члена `size()`:

```
size_type size() const // Текущее количество элементов
{
    return numElems;
}
```

Начиная с C++14, `auto` можно использовать и в качестве возвращаемого типа, выводимого компилятором:

```
auto size() const // Текущее количество элементов
{
    return numElems;
}
```

При таком объявлении класса тип для возврата количества элементов определяется типом, используемым для указания максимального числа элементов при создании стека:

basics/stackauto.cpp

```
#include <iostream>
#include <string>
#include "stackauto.hpp"

int main()
{
    Stack<int, 20u> int20Stack;           // Стек для 20 целых (int)
    Stack<std::string, 40> stringStack; // Стек для 40 строк (string)

    // Работа со стеком для 20 целых (int)
    int20Stack.push(7);
    std::cout << int20Stack.top() << '\n';
    auto size1 = int20Stack.size();

    // Работа со стеком для 40 строк (string)
    stringStack.push("hello");
    std::cout << stringStack.top() << '\n';
    auto size2 = stringStack.size();

    if (!std::is_same_v<decltype(size1), decltype(size2)>)
    {
        std::cout << "Типы size различны" << '\n';
    }
}
```

При объявлении

`Stack<int,20u> int20Stack; // Стек для 20 целых (int)`

внутренний тип размера стека – `unsigned int`, поскольку передано значение `20u`.

При объявлении

`Stack<std::string,40> stringStack; // Стек для 40 строк (string)`

внутренний тип размера стека – `int`, поскольку передано значение `40`.

Функция `size()` для этих двух стеков возвращает разные типы, так что после

```
auto size1 = int20Stack.size();
...
auto size2 = stringStack.size();
```

типы переменных `size1` и `size2` различны. Используя стандартное свойство типа `std::is_same` (см. раздел Г.3.3) и ключевое слово `decltype`, мы можем в этом убедиться:

```
if (!std::is_same<decltype(size1), decltype(size2)>::value)
{
    std::cout << "Типы size различны" << '\n';
}
```

Этот код приводит к выводу на консоль

Типы size различны

Начиная с C++17, для свойств, возвращающих значения, можно использовать суффикс `_v` и опустить `::value` (см. раздел 5.6):

```
if (!std::is_same_v<decltype(size1), decltype(size2)>)
{
    std::cout << "Типы size различны" << '\n';
}
```

Обратите внимание на то, что другие ограничения на тип параметров шаблона, не являющихся типами, остаются в силе. В частности, по-прежнему остаются в силе рассматривавшиеся в разделе 3.3 ограничения, накладываемые на возможные типы аргументов шаблона, не являющихся типами. Например:

`Stack<int,3.14> sd; // Ошибка: аргумент типа с плавающей точкой`

А поскольку можно также передать строки как константные массивы (начиная с C++17, даже как локально объявленные статические — см. раздел 3.3), следующий код вполне корректен:

`basics/message.cpp`

```
#include <iostream>
template<auto T> // Значение любого разрешенного параметра,
                  // не являющегося типом (начиная с C++17)
class Message
{
public:
    void print()
    {
        std::cout << T << '\n';
    }
};

int main()
{
    Message<42> msg1;
    msg1.print();      // Инициализация значением 42 и его вывод
```

```
static char const s[] = "hello";
Message<s> msg2; // Инициализация значением char const[6] "hello"
msg2.print();    // и вывод этого значения
}
```

Заметим также, что возможна даже конструкция `template<decltype(auto) N>`, которая позволяет инстанцировать `N` как ссылку:

```
template<decltype(auto) N>
class C
{
    ...
};

int i;
C<(i)> x; // N представляет собой int&
```

Подробности представлены в разделе 15.10.1.

3.5. Резюме

- В качестве параметров шаблонов могут выступать не только типы, но и значения.
- В качестве аргументов для параметров шаблонов, не являющихся типами, нельзя использовать числа с плавающей точкой и объекты типа класса. На применение указателей/ссылок на строковые литералы, временные значения и подобъекты накладываются ограничения.
- Использование ключевого слова `auto` позволяет шаблонам иметь параметры, не являющиеся типами, которые представляют собой значения обобщенных типов.

Глава 4

Вариативные шаблоны

Начиная с C++11, параметрам шаблонов можно сопоставлять переменное количество аргументов шаблона. Это дает возможность использовать шаблоны в местах, где требуется передать произвольное количество аргументов произвольных типов. Типичное приложение этой возможности — передача произвольного количества параметров различных типов в класс. Еще одно применение — предоставить обобщенный код для обработки любого количества параметров произвольных типов.

4.1. Шаблоны с переменным количеством аргументов

Параметры шаблона могут быть определены таким образом, чтобы принимать неограниченное количество аргументов шаблона. Такие шаблоны называются *вариативными* (variadic templates).

4.1.1. Примеры вариативных шаблонов

Например, можно использовать приведенный ниже код для вызова функции `print()` для переменного количества аргументов разных типов:

`basics/varprint1.hpp`

```
#include <iostream>

void print()
{
}

template<typename T, typename... Types>
void print(T firstArg, Types... args)
{
    std::cout << firstArg << '\n'; // Печать первого аргумента
    print(args...);                // Вызов print() для остальных
}                                // аргументов
```

Если передаются один или несколько аргументов, используется шаблон функции, который, указывая отдельно первый аргумент, позволяет выводить в консоль первый аргумент, перед тем как рекурсивно применить `print()` для остальных аргументов. Остальные аргументы, представленные именем `args`, являются *пакетом параметров функции* (function parameter pack):

`void print (T firstArg, Types... args)`

`Types` указывается в *пакете параметров шаблона* (template parameter pack):

`template<typename T, typename... Types>`

Для завершения рекурсии предоставляется нешаблонная перегрузка `print()`, которая вызывается при пустом пакете параметров.

Например, вызов

```
std::string s("world");
print(7.5, "hello", s);
```

приведет к следующему выводу в консоль:

```
7.5
hello
world
```

Первый вызов расширяется до

```
print<double, char const*, std::string> (7.5, "hello", s);
```

где

- `firstArg` имеет значение `7.5`, так что тип `T` представляет собой `double`, а
- `args` является вариативным аргументом шаблона, который содержит значения `"hello"` типа `char const*` и `"world"` типа `std::string`.

После вывода `7.5` в качестве `firstArg` вновь вызывается `print()` для оставшихся аргументов. Этот вызов расширяется до

```
print<char const*, std::string> ("hello", s);
```

где

- `firstArg` имеет значение `"hello"`, так что тип `T` представляет собой `char const*`, а
- `args` является вариативным аргументом шаблона, который содержит значение типа `std::string`.

После вывода в консоль `"hello"` в качестве `firstArg` вновь вызывается `print()` для оставшихся аргументов. Этот вызов расширяется до

```
print<std::string> (s);
```

где

- `firstArg` имеет значение `"world"`, так что тип `T` представляет собой `std::string`, а
- `args` является пустым вариативным аргументом шаблона, который не содержит значений.

Таким образом, после вывода в консоль `"world"` в качестве `firstArg` вновь вызывается `print()` — без аргументов, что приводит к вызову нешаблонной перегрузки `print()`, не выполняющей никаких действий.

4.1.2. Перегрузка вариативных и невариативных шаблонов

Заметим, что приведенный выше пример можно также реализовать следующим образом:

basics/varprint2.hpp

```
#include <iostream>

template<typename T>
void print(T arg)
{
    std::cout << arg << '\n'; // Вывод переданного аргумента
}
template<typename T, typename... Types>
void print(T firstArg, Types... args)
{
    print(firstArg);           // Вызов print() для первого аргумента
    print(args...);            // Вызов print() для прочих аргументов
}
```

Если два шаблона функций отличаются только завершающим пакетом параметров, шаблон функции без такого пакета является предпочтительным¹. Правила разрешения перегрузки, применяемые в этом случае, более подробно рассматриваются в разделе B.3.1.

4.1.3. Оператор `sizeof...`

C++11 вводит новую версию оператора `sizeof` для вариативных шаблонов: `sizeof...`. Он раскрывается до количества элементов, содержащихся в пакете параметров. Таким образом, исходный текст

```
template<typename T, typename... Types>
void print(T firstArg, Types... args)
{
    std::cout << sizeof...(Types)
        << '\n';           // Количество оставшихся типов
    std::cout << sizeof...(args)
        << '\n';           // Количество оставшихся аргументов
    ...
}
```

дважды выводит в консоль количество аргументов, остающихся после первого аргумента функции `print()`. Как видите, `sizeof...` можно вызывать как для пакета параметров шаблона, так и для пакета параметров функции.

Это может навести нас на мысль, что для завершения рекурсии можно опустить вызов функции, когда больше нет необработанных аргументов:

```
template<typename T, typename... Types>
void print(T firstArg, Types... args)
{
    std::cout << firstArg << '\n';
    ...
}
```

¹Изначально в C++11 и C++14 это приводило к неоднозначности, которая была исправлена позже [33], но все компиляторы во всех версиях работают именно таким образом.

```

if (sizeof...(args) > 0) // Ошибка при sizeof...(args)==0
{
    print(args...); // функции print() без аргументов нет
}
}

```

Однако этот подход не сработает, потому что в общем случае инстанцируются обе ветви инструкции `if`. Будет ли инстанцированный код нужен — выясняется *во время выполнения*, в то время как инстанцирование вызова происходит *во время компиляции*. По этой причине, если вы вызываете шаблон функции `print()` для одного (последнего) аргумента, то инструкция с вызовом `print(args...)` все равно инстанцируется — на этот раз без аргументов, и если такой функции `print()` без аргументов нет, мы получаем ошибку.

Заметим, однако, что, начиная с C++17, доступна инструкция `if` времени компиляции, которая позволяет достичь ожидаемых результатов с помощью немного иного синтаксиса. Этот вопрос будет рассмотрен в разделе 8.5.

4.2. Выражения свертки

Начиная с C++17 в языке имеется возможность вычисления результата с применением бинарного оператора ко *всем* аргументам пакета параметров (с необязательным начальным значением).

Например, следующая функция возвращает сумму всех переданных аргументов:

```

template<typename... T>
auto foldSum(T... s)
{
    return (... + s); // ((s1 + s2) + s3) ...
}

```

Если пакет параметров пуст, выражение обычно считается некорректным (за исключением того, что для оператора `&&` оно имеет значение `true`, для оператора `||` — значение `false`, и для оператора-запятой значение для пустого пакета параметров представляет собой `void()`).

В табл. 4.1 перечислены возможные выражения свертки.

Таблица 4.1. Выражения свертки (начиная с C++17)

Выражение свертки	Вычисляет
<code>(... op pack)</code>	<code>((pack1 op pack2) op pack3) ... op packN)</code>
<code>(pack op ...)</code>	<code>(pack1 op (... (packN-1 op packN)))</code>
<code>(init op ... op pack)</code>	<code>((init op pack1) op pack2) ... op packN)</code>
<code>(pack op ... op init)</code>	<code>(pack1 op (... (packN op init)))</code>

В выражениях свертки можно использовать почти все бинарные операторы (подробнее см. раздел 12.4.6). Например, можно использовать выражение свертки для обхода бинарного дерева с использованием оператора `->*`:

basics/foldtraverse.cpp

```
// Определение структуры бинарного дерева:
struct Node
{
    int value;
    Node* left;
    Node* right;
    Node(int i = 0) : value(i), left(nullptr), right(nullptr)
    {
    }
    ...
};

auto left = &Node::left;
auto right = &Node::right;

// Обход дерева с использованием выражения свертки:
template<typename T, typename... TP>
Node* traverse(T np, TP... paths)
{
    return (np ->* ... ->* paths); // np ->* paths1 ->* paths2 ...
}

int main()
{
    // Инициализация структуры бинарного дерева:
    Node* root = new Node{0};
    root->left = new Node{1};
    root->left->right = new Node{2};
    ...

    // Обход бинарного дерева:
    Node* node = traverse(root, left, right);
    ...
}
```

Здесь конструкция

`(np ->* ... ->* paths)`

использует выражение свертки для обхода вариативных элементов `paths` из пр.

С помощью такого выражения и с использованием инициализатора можно существенно упростить рассматривавшийся ранее вариативный шаблон для вывода на консоль всех аргументов:

```
template<typename... Types>
void print(Types const& ... args)
{
    (std::cout << ... << args) << '\n';
}
```

Заметим, однако, что в этом случае нет пробелов, разделяющих аргументы из пакета параметров. Чтобы добиться этого, нужен дополнительный шаблон класса, который гарантирует, что любой вывод любого аргумента сопровождается пробелом:

basics/addspace.hpp

```
template<typename T>
class AddSpace
{
private:
    T const& ref; // Ссылка на аргумент, переданный в конструкторе
public:
    AddSpace(T const& r): ref(r)
    {
    }
    friend std::ostream& operator<< (std::ostream& os, AddSpace<T> s)
    {
        return os << s.ref << ' '; // Вывод аргумента и пробела
    }
};

template<typename... Args>
void print(Args... args)
{
    (std::cout << ... << AddSpace(args)) << '\n';
}
```

Обратите внимание на то, что выражение `AddSpace(args)` использует вывод аргумента шаблона класса (см. раздел 2.9), чтобы получить `AddSpace<Args>(args)` для того, чтобы для каждого аргумента был создан объект `AddSpace`, ссылающийся на переданный аргумент и добавляющий пробел при использовании в выражениях вывода.

Подробнее о выражениях свертки рассказывается в разделе 12.4.6.

4.3. Применения шаблонов с переменным количеством аргументов

Шаблоны с переменным количеством аргументов играют важную роль при реализации обобщенных библиотек, в частности, стандартной библиотеки C++.

Одно из типичных применений — передача переменного количества аргументов произвольного типа. Например, мы используем эту возможность в следующих ситуациях.

- Передача аргументов конструктору нового объекта в динамической памяти, которым владеет интеллектуальный указатель:

```
// Создание интеллектуального указателя shared_ptr для
// complex<float>, инициализированного значениями 4.2 и 7.7:
auto sp = std::make_shared<std::complex<float>>(4.2, 7.7);
```

- Передача аргументов в поток выполнения, запускаемый библиотекой:

```
std::thread t(foo, 42, "hello"); // Вызов foo(42, "hello")
                                // в отдельном потоке
```

- Передача аргументов в конструктор нового элемента, добавляемого к вектору:

```
std::vector<Customer> v;
...
// Добавление Customer, инициализированного тремя аргументами:
v.emplace_back("Tim", "Jovi", 1962);
```

Обычно аргументы передаются с семантикой перемещения с использованием “*прямой передачи*” (“perfect forward”) (см. раздел 6.1), так что соответствующие объявления имеют примерно следующий вид:

```
namespace std
{
    template<typename T, typename... Args> shared_ptr<T>
    make_shared(Args&& ... args);

    class thread
    {
        public:
            template<typename F, typename... Args>
            explicit thread(F&& f, Args&& ... args);
        ...
    };

    template<typename T, typename Allocator = allocator<T>>
    class vector
    {
        public:
            template<typename... Args>
            reference emplace_back(Args&&... args);
        ...
    };
}
```

Обратите внимание на то, что к вариативным параметрам шаблонов функций применимы те же правила, что и к обычным параметрам. Например, при передаче по значению аргументы копируются и низводятся (например, массивы становятся указателями), в то время как при передаче по ссылке параметры ссылаются на исходные параметры и не низводятся:

```
// args копируются с низведенными типами:
template<typename... Args> void foo(Args... args);

// args представляют собой ссылки без низведения на переданные объекты:
template<typename... Args> void bar(Args const& ... args);
```

4.4. Вариативные шаблоны классов и вариативные выражения

Помимо приведенных выше примеров пакеты параметров могут появляться в дополнительных местах, включая, например, выражения, шаблоны классов, объявления `using` и даже правила вывода. Полный список приведен в разделе 12.4.2.

4.4.1. Вариативные выражения

Вы можете сделать больше, чем просто передать все параметры. Вы можете выполнить с ними вычисления, т.е. вычисления со всеми параметрами в пакете параметров.

Например, следующая функция удваивает каждый параметр из пакета параметров `args` и передает каждый удвоенный аргумент функции `print()`:

```
template<typename... T>
void printDoubled(T const& ... args)
{
    print(args + args...);
}
```

Если, например, вы вызовете

```
printDoubled(7.5, std::string("hello"), std::complex<float>(4, 2));
```

то результатом будет следующее (за исключением побочных действий конструкторов):

```
print(7.5 + 7.5,
      std::string("hello") + std::string("hello"),
      std::complex<float>(4, 2) + std::complex<float>(4, 2));
```

Если вы хотите просто добавить 1 к каждому аргументу, учтите, что точки многоточия не могут следовать за числовым литералом непосредственно:

```
template<typename... T>
void addOne(T const& ... args)
{
    print(args +
          1...); // Ошибка: 1... является литералом со слишком
           // большим количеством десятичных точек
    print(args + 1...); // OK
    print((args + 1)...); // OK
}
```

Выражения времени компиляции могут включать пакеты параметров шаблонов таким же образом. Например, следующий шаблон функции возвращает информацию о том, одинаковы ли типы всех переданных аргументов:

```
template<typename T1, typename... TN>
constexpr bool isHomogeneous(T1, TN...)
{
    return (std::is_same<T1, TN>::value && ...); // Начиная с C++17
}
```

Здесь мы видим применение выражений свертки (см. раздел 4.2). Выражение `isHomogeneous(43, -1, "hello")`

разворачивается в

```
std::is_same<int, int>::value && std::is_same<int, char const*>::value
```

и дает в качестве результата `false`, в то время как

```
isHomogeneous("hello", " ", "world", "!")
```

дает `true`, поскольку тип для всех переданных аргументов вычисляется как `char const*` (обратите внимание на то, что, поскольку аргументы передаются по значению, выполняется низведение их типов).

4.4.2. Вариативные индексы

В качестве еще одного примера приведенная далее функция использует вариативный список индексов для доступа к соответствующим элементам аргумента, переданного первым:

```
template<typename C, typename... Idx>
void printElems(C const& coll, Idx... idx)
{
    print(coll[idx]...);
}
```

Следовательно, при вызове

```
std::vector<std::string> coll = {"good", "times", "say", "bye"};
printElems(coll, 2, 0, 3);
```

результатом окажется вызов

```
print (coll[2], coll[0], coll[3]);
```

Можно также объявить пакет параметров, не являющихся типами. Например, объявление

```
template<std::size_t... Idx, typename C>
void printIdx(C const& coll)
{
    print(coll[Idx]...);
}
```

позволяет выполнить вызов

```
std::vector<std::string> coll = {"good", "times", "say", "bye"};
printIdx<2, 0, 3>(coll);
```

действие которого совпадает с таковым для предыдущего примера.

4.4.3. Вариативные шаблоны класса

Вариативные шаблоны могут быть и шаблонами классов. Важным примером является класс, в котором произвольное количество параметров шаблона определяет типы соответствующих членов:

```
template<typename... Elements>
class Tuple;
```

```
Tuple<int, std::string, char> t; // t может хранить int, string и char
```

Этот вопрос будет рассматриваться в главе 25, “Кортежи”.

Еще одним примером может служить указание возможных типов объектов:

```
template<typename... Types>
class Variant;
```

```
Variant<int, std::string, char> v; // v может хранить int,
// string или char
```

Этот вопрос будет рассматриваться в главе 26, “Контролируемые объединения”.

Можно также определить класс, который представляет собой тип, который можно рассматривать как список индексов:

```
// Тип для произвольного количества индексов:
template<std::size_t...>
struct Indices
{
};
```

Этот класс может быть использован для определения функции, которая вызывает `print()` для элементов `std::array` или `std::tuple` с использованием доступа времени компиляции с помощью `get<>()` для указанных индексов:

```
template<typename T, std::size_t... Idx>
void printByIdx(T t, Indices<Idx...>)
{
    print(std::get<Idx>(t)...);
}
```

Этот шаблон может быть использован так:

```
std::array<std::string, 5> arr = {"Hello", "my", "new", "!", "World"};
printByIdx(arr, Indices<0, 4, 3>());
```

или так:

```
auto t = std::make_tuple(12, "monkeys", 2.0);
printByIdx(t, Indices<0, 1, 2>());
```

Это первый шаг на пути к метапрограммированию, которое будет рассматриваться в разделе 8.1 и в главе 23, “Метапрограммирование”.

4.4.4. Вариативные правила вывода

Вариативными могут быть даже правила вывода (см. раздел 2.9). Например, стандартная библиотека C++ определяет следующее правило вывода для `std::array`:

```
namespace std
{
    template<typename T, typename... U> array(T, U...)
        -> array < enable_if_t < (is_same_v<T, U>&& ...), T >,
                (1 + sizeof...(U)) >;
}
```

Инициализация наподобие

```
std::array a{42, 45, 77};
```

в правиле выводит `T` как тип элемента, а различные типы `U...` — как типы последующих элементов. Общее количество элементов, таким образом, равно `1+sizeof...(U)`:

```
std::array<int, 3> a{42, 45, 77};
```

Выражение `std::enable_if<>` для первого параметра `array` представляет собой выражение свертки, которое (как показано в `isHomogeneous()` в разделе 4.4.1) раскрывается до:

```
is_same_v<T, U1> && is_same_v<T, U2> && is_same_v<T, U3> ...
```

Если результат не равен `true` (т.е. не все типы элементов одинаковы), правило вывода отвергается и весь вывод оказывается неудачным. Таким образом стандартная библиотека гарантирует, что для успешного вывода все элементы должны быть одного и того же типа.

4.4.5. Вариативные базовые классы и `using`

Наконец, рассмотрим следующий пример.

basics/varusing.cpp

```
#include <string>
#include <unordered_set>

class Customer
{
private:
    std::string name;
public:
    Customer(std::string const& n) : name(n) { }
    std::string getName() const
    {
        return name;
    }
};

struct CustomerEq
{
    bool operator()(Customer const& c1, Customer const& c2) const
    {
        return c1.getName() == c2.getName();
    }
};

struct CustomerHash
{
    std::size_t operator()(Customer const& c) const
    {
        return std::hash<std::string>()(c.getName());
    }
};

// Определение класса, который объединяет operator()
// для вариативных базовых классов:
template<typename... Bases>
struct Overloader : Bases...
{
    using Bases::operator()...; // Начиная с C++17
};
```

```

int main()
{
    // Объединяет пользовательские хеш-функцию
    // и равенство в одном типе:
    using CustomerOP = Overloader<CustomerHash, CustomerEq>;
    std::unordered_set<Customer, CustomerHash, CustomerEq> coll1;
    std::unordered_set<Customer, CustomerOP, CustomerOP> coll2;
    ...
}

```

Здесь мы сначала определяем класс `Customer` и независимые функциональные объекты для хеширования и сравнения объектов `Customer`. Имея

```

template<typename... Bases>
struct Overloader : Bases...
{
    using Bases::operator()...; // Начиная с C++17
};

```

мы можем определить класс, производный от вариативного количества базовых классов, который вносит объявления `operator()` из каждого из этих базовых классов. Объявление

```
using CustomerOP = Overloader<CustomerHash, CustomerEq>;
```

позволяет использовать эту возможность для порождения `CustomerOP` из `CustomerHash` и `CustomerEq` и разрешает обе реализации `operator()` в порожденном классе.

Еще одно применение этой методики описано в разделе 26.4.

4.5. Резюме

- С помощью пакетов параметров шаблоны могут определяться для произвольного количества параметров шаблона произвольного типа.
- Для обработки параметров требуется рекурсия и/или соответствующие невариативные функции.
- Оператор `sizeof...` дает количество аргументов в пакете параметров.
- Типичным применением вариативных шаблонов является передача произвольного количества аргументов произвольного типа.
- Используя выражения свертки, можно применить операторы ко всем аргументам пакета параметров.

Глава 5

Основы работы с шаблонами

В данной главе продолжается рассмотрение фундаментальных свойств шаблонов, связанных с практическим использованием последних: дополнительные возможности применения ключевого слова `typename`, определение функций-членов и вложенных классов как шаблонов, шаблонные параметры шаблонов, инициализация нулем и некоторые детали, касающиеся использования строковых литералов в качестве аргументов шаблонов функций. Упомянутые аспекты не всегда можно отнести к числу простых и очевидных, однако любой программист-практик должен иметь хотя бы некоторое представление о них.

5.1. Ключевое слово `typename`

Ключевое слово `typename` введено в язык в процессе стандартизации C++ для указания того, что идентификатор в шаблоне является типом. Рассмотрим следующий пример:

```
template<typename T>
class MyClass
{
public:
    ...
    void foo()
    {
        typename T::SubType* ptr;
    }
};
```

В этом примере второе ключевое слово `typename` используется для пояснения того, что `SubType` является типом, определенным внутри класса `T`. Таким образом, `ptr` является указателем на тип `T::SubType`.

Без применения `typename` идентификатор `SubType` интерпретировался бы как член класса, не являющийся типом (например, как статический член-данные или константа перечисления). В результате выражение

```
T::SubType* ptr
```

представляло бы собой умножение статического члена `SubType` класса `T` на `ptr`, что не было бы ошибкой, поскольку для некоторых инстанций класса `MyClass<>` это могло бы быть корректным кодом.

В общем случае ключевое слово `typename` следует использовать всякий раз, когда имя, зависящее от параметра шаблона, представляет собой тип (более подробно этот вопрос рассматривается в разделе 13.3.2).

Одним из применений `typename` является объявление итераторов стандартных контейнеров в обобщенном коде.

basics/printcoll.hpp

```
#include <iostream>

// Вывод элементов контейнера STL
template<typename T>
void printcoll(T const& coll)
{
    typename T::const_iterator pos;           // Итератор для обхода coll
    typename T::const_iterator end(coll.end()); // Конечная позиция

    for (pos = coll.begin(); pos != end; ++pos)
    {
        std::cout << *pos << ' ';
    }

    std::cout << '\n';
}
```

В этом шаблоне функции параметр вызова представляет собой стандартный контейнер типа *T*. Для обхода всех элементов контейнера используется итератор, который объявлен внутри каждого класса стандартного контейнера как тип *const_iterator*.

```
class stlcontainer
{
public:
    using iterator = ...;          // Итератор для чтения/записи
    using const_iterator = ...;     // Итератор для чтения
    ...
};
```

Таким образом, для доступа к типу *const_iterator* шаблонного типа *T* нужно квалифицировать тип с использованием ключевого слова *typename*:

```
typename T::const_iterator pos;
```

Более подробно о применении ключевого слова *typename* до C++17 рассказывается в разделе 13.3.2. Обратите внимание на то, что стандарт C++20, вероятно, устранит необходимость *typename* во многих распространенных ситуациях (подробнее см. раздел 17.1).

5.2. Инициализация нулем

Для фундаментальных типов данных, таких как *int*, *double* или указатели, не существует стандартного конструктора, который инициализировал бы эти величины каким-либо полезным значением по умолчанию. Напротив, каждая неинициализированная локальная переменная имеет неопределенное значение:

```
void foo()
{
    int x;      // Значение x не определено
    int* ptr;   // ptr может указывать куда угодно (но не в никуда)
}
```

Теперь допустим, что мы пишем шаблоны и хотим иметь переменные типа шаблона, инициализированные значением по умолчанию. У нас возникает проблема: ведь с помощью простого определения для встроенных типов этого сделать нельзя:

```
template<typename T>
void foo()
{
    T x; // Если T – встроенный тип, значение x не определено
}
```

По этой причине для встроенных типов можно явно вызывать стандартный конструктор, который инициализирует их нулем (или значением `false` для величин типа `bool`, а для указателей – значением `nullptr`). Следовательно, можно обеспечить корректную инициализацию по умолчанию даже для встроенных типов с помощью кода наподобие приведенного ниже:

```
template<typename T>
void foo()
{
    T x{}; // x равно 0 (false, nullptr) если T – встроенный тип
}
```

Этот способ инициализации называется *инициализацией значением* (value initialization), что означает, что для объекта вызывается конструктор либо он *инициализируется нулем* (zero initialize). Это работает, даже если конструктор объявлен как `explicit`.

До C++11 синтаксис, гарантирующий корректную инициализацию, был следующим:

```
T x = T(); // x равно 0 (false, nullptr) если T – встроенный тип
```

До C++17 этот механизм (который все еще поддерживается) работал, только если конструктор, выбранный для копирующей инициализации, не был объявлен как `explicit`. В C++17 обязательное устранение копирования позволяет обойти это ограничение, так что работает любой вариант синтаксиса, но инициализация с помощью фигурных скобок дает возможность использовать конструктор со списком инициализации¹, если конструктор по умолчанию недоступен.

Для гарантии того, что член шаблонного класса, имеющий параметризованный тип, будет инициализирован, следует определить конструктор по умолчанию, который использует инициализатор в фигурных скобках для инициализации членов класса.

```
template<typename T>
class MyClass
{
private:
    T x;
public:
```

¹ То есть конструктор с параметром типа `std::initializer_list<X>` для некоторого типа X.

```
MyClass() : x{} // Гарантирует инициализацию x даже
{
    // для встроенных типов
}
***
```

Синтаксис для стандартов до C++11

```
MyClass() : x() // Гарантирует инициализацию x даже
{
    // для встроенных типов
}
```

также остается работоспособен.

Начиная с C++11, можно также предоставить инициализацию по умолчанию для нестатических членов, так что допустим следующий код:

```
template<typename T>
class MyClass
{
    private:
        T x{}; // Инициализация x нулем, если не указано иное
    ***
```

Однако учтите, что аргументы по умолчанию не могут использовать этот синтаксис. Например:

```
template<typename T>
void foo(T p{}) // Ошибка
{
    ***
```

Вместо этого следует написать:

```
template<typename T>
void foo(T p = T{}) // OK (до C++11 следует использовать T())
{
    ***
```

5.3. Использование `this->`

Для шаблонов классов, имеющих базовые классы, которые зависят от параметров шаблона, использование имени `x` самого по себе не всегда эквивалентно применению `this->x`, даже если член `x` является наследуемым. Например:

```
template<typename T>
class Base
{
public:
    void bar();
};

template<typename T>
class Derived : Base<T>
{
```

```

public:
    void foo()
    {
        bar(); // Вызов внешней bar() или ошибка
    }
};

```

В этом примере при разрешении имени `bar` в теле `foo()` *никогда* не рассматривается функция `bar()`, определенная в классе `Base`. Следовательно, либо будет получено сообщение об ошибке, либо будет вызвана другая функция `bar()` (например, глобальная функция `bar()`).

Более подробно этот вопрос рассматривается в разделе 13.4.2. А пока что рекомендуем использовать следующее эмпирическое правило: всегда необходимо полностью квалифицировать любое имя, объявленное в базовом классе, который каким-либо образом зависит от параметра шаблона. Для этого можно использовать конструкции `this->` или `Base<T>::`.

5.4. Шаблоны для массивов и строковых литералов

При передаче в шаблоны массивов или строковых литералов требуется принять некоторые меры. Во-первых, если параметры шаблона объявляются как ссылки, к аргументам не применяется низведение типов, так что, например, переданный аргумент "Hello" имеет тип `char const [6]`. Это может стать проблемой при передаче массивов или строковых аргументов разной длины, потому что их типы будут различными. Только при передаче аргументов по значению выполняется низведение типов, так что строковые литералы преобразуются в тип `char const*`. Этот вопрос подробно рассмотрен в главе 7, "По значению или по ссылке?".

Обратите внимание на то, что можно также написать шаблоны, которые работают конкретно с массивами или строковыми литералами. Например:

`basics/lessarray.hpp`

```

template<typename T, int N, int M>
bool less(T(&a)[N], T(&b)[M])
{
    for(int i = 0; i < N && i < M; ++i)
    {
        if (a[i] < b[i]) return true;
        if (b[i] < a[i]) return false;
    }
    return N < M;
}

```

При вызове

```

int x[] = {1, 2, 3};
int y[] = {1, 2, 3, 4, 5};
std::cout << less(x, y) << '\n';

```

`less<>()` инстанцируется с `T`, представляющим собой `int`, `N`, равным 3, и `M`, равным 5.

Можно также использовать шаблон для строковых литералов:

```
std::cout << less("ab", "abc") << '\n';
```

В этом случае `less<>()` инстанцируется с `T`, представляющим собой `char const`, `N`, равным 3, и `M`, равным 4.

Если вы хотите только предоставить шаблон функции для строковых литералов (и других массивов `char`), то можете сделать это следующим образом:

basics/lessstring.hpp

```
template<int N, int M>
bool less(char const(&a)[N], char const(&b)[M])
{
    for (int i = 0; i < N && i < M; ++i)
    {
        if (a[i] < b[i]) return true;
        if (b[i] < a[i]) return false;
    }
    return N < M;
}
```

Обратите внимание, что можно (а иногда и нужно) выполнять перегрузку или частичную специализацию для массивов с неизвестными границами. Следующая программа иллюстрирует все возможные перегрузки для массивов:

basics/arrays.hpp

```

    static void print()
    {
        std::cout << "print() для T[]\n";
    }
};

template<typename T>
struct MyClass<T(&) []> // Частичная специализация для ссылок
{
    static void print() // на массивы с неизвестными границами
    {
        std::cout << "print() для T(&) []\n";
    }
};

template<typename T>
struct MyClass<T*> // Частичная специализация для указателей
{
    static void print()
    {
        std::cout << "print() для T*\n";
    }
};

```

Здесь шаблон класса `MyClass<>` специализирован для различных типов: массивов с известными и неизвестными границами, ссылок на массивы с известными и неизвестными границами и указателей. Каждый случай отличается от других и может встретиться при использовании массивов:

`basics/arrays.cpp`

```

#include "arrays.hpp"

template<typename T1, typename T2, typename T3>
void foo()
{
    int a1[7], int a2[], // Указатели по правилам языка
    int (&a3)[42], // Ссылка на массив с известными границами
    int (&x0)[], // Ссылка на массив с неизвестными границами
    T1 x1, // Передача по значению с низведением
    T2& x2, T3&& x3) // Передача по ссылке
{
    MyClass<decltype(a1)>::print(); // Использует MyClass<T*>
    MyClass<decltype(a2)>::print(); // Использует MyClass<T*>
    MyClass<decltype(a3)>::print(); // Использует MyClass<T(&) [SZ]>
    MyClass<decltype(x0)>::print(); // Использует MyClass<T(&) []>
    MyClass<decltype(x1)>::print(); // Использует MyClass<T*>
    MyClass<decltype(x2)>::print(); // Использует MyClass<T(&) []>
    MyClass<decltype(x3)>::print(); // Использует MyClass<T(&) []>
}

int main()
{
    int a[42];
    MyClass<decltype(a)>::print(); // Использует MyClass<T[SZ]>
    extern int x[]; // Предварительное объявление массива
}

```

```

 MyClass<decltype(x)>::print(); // Использует MyClass<T[]>
 foo(a, a, a, x, x, x, x);
}
int x[] = {0, 8, 15};           // Определение предварительно
                                // объявленного массива

```

Обратите внимание на то, что *параметр вызова*, объявленный как массив (с длиной или без таковой), согласно правилам языка в действительности имеет тип указателя. Учтите также, что шаблоны для массивов с неизвестными границами могут использоваться для неполных типов, таких как

```
extern int i[];
```

При передаче по ссылке мы получаем тип `int(&) []`, который также может использоваться в качестве параметра шаблона².

Еще один пример использования различных типов массивов в обобщенном коде приведен в разделе 19.3.1.

5.5. Шаблоны-члены

Члены классов тоже могут быть шаблонами. Это справедливо как для вложенных классов, так и для функций-членов. Применение и преимущества такой возможности можно еще раз продемонстрировать на примере шаблона класса `Stack<>`. Обычно стеки можно присваивать друг другу только в том случае, если они имеют одинаковый тип, что предполагает одинаковый тип их элементов. Однако стеку невозможно присвоить стек с элементами любого другого типа, даже если для типов элементов определено неявное преобразование типов:

```

Stack<int> intStack1, intStack2; // Стеки для int
Stack<float> floatStack;       // Стек для float
...
intStack1 = intStack2;          // OK: стеки имеют одинаковые типы
floatStack = intStack1;         // Ошибка: разные типы стеков

```

Оператор присваивания по умолчанию требует, чтобы с обеих сторон оператора использовался один и тот же тип, но если типы элементов у стеков различны, то это не так.

Однако если определить оператор присваивания в виде шаблона, то присваивание стеков с элементами, для которых определено соответствующее преобразование типов, станет возможным. Для этого необходимо объявить `Stack<>`, как показано ниже.

```
basics/stack5decl.hpp
```

```

template<typename T>
class Stack

```

² Параметры вида `X(&) []` для некоторого произвольного типа `X` стали корректны только в C++17. Однако многие компиляторы допускали такие параметры в более ранних версиях языка.

```

{
    private:
        std::deque<T> elems; // Элементы
    public:
        void push(T const&); // Добавление элементов в стек
        void pop(); // Снятие со стека
        T const& top() const; // Возврат верхнего элемента
        bool empty() const // Проверка пустоты стека
        {
            return elems.empty();
        }
        // Присваивание стека с элементами типа T2
        template<typename T2>
        Stack& operator= (Stack<T2> const&);
};

```

Были сделаны два изменения.

1. Добавлено объявление оператора присваивания для стеков с элементами другого типа T2.
2. Теперь в качестве внутреннего контейнера для элементов стека используется дек `std::deque<>`. Это следствие реализации нового оператора присваивания.

Реализация нового оператора присваивания показана ниже³.

basics/stack5assign.hpp

```

template<typename T>
template<typename T2>
Stack<T>& Stack<T>::operator= (Stack<T2> const& op2)
{
    Stack<T2> tmp(op2); // Создание копии присваиваемого стека
    elems.clear(); // Удаление существующих элементов

    while (!tmp.empty()) // Копирование всех элементов
    {
        elems.push_front(tmp.top());
        tmp.pop();
    }
    return *this;
}

```

Прежде всего посмотрим на синтаксис определения шаблона-члена. Внутри шаблона с параметром `T` определяется внутренний шаблон с параметром `T2`:

```

template<typename T>
template<typename T2>
...

```

³ Это не более чем базовая реализация, демонстрирующая возможности шаблонов. Ряд вопросов, таких как корректная обработка исключений, сознательно проигнорирован.

Казалось бы, в теле функции-члена можно просто обращаться ко всем необходимым данным присваиваемого стека `op2`. Однако этот стек имеет другой тип (при инстанцировании шаблона класса для двух разных типов данных будут получены стеки двух разных типов), поэтому вы ограничены только использованием открытого интерфейса. Отсюда следует, что единственный способ обращения к элементам стека — это вызов `top()`. Однако для этого каждый элемент должен оказаться в вершине стека. Таким образом, сначала нужно сделать копию `op2`, чтобы можно было удалять элементы при помощи вызовов `pop()`. Поскольку функция `top()` возвращает последний элемент, помещенный в стек, необходимо использовать контейнер, который поддерживает вставку элементов в противоположный конец коллекции. По этой причине здесь используется дек `std::deque<>`, у которого имеется функция `push_front()`, помещающая элемент в начало коллекции.

Для доступа ко всем членам `op2` можно объявить все прочие экземпляры стеков друзьями:

basics/stack6decl.hpp

```
template<typename T>
class Stack
{
private:
    std::deque<T> elems; // Элементы
public:
    void push(T const&); // Добавление элементов в стек
    void pop(); // Снятие со стека
    T const& top() const; // Возврат верхнего элемента
    bool empty() const // Проверка пустоты стека
    {
        return elems.empty();
    }

    // Присваивание стека с элементами типа T2
    template<typename T2>
    Stack& operator= (Stack<T2> const &);

    // Для доступа к закрытым членам Stack<T2> для любого типа T2:
    template<typename> friend class Stack;
};
```

Как можно видеть, поскольку имя параметра шаблона не используется, оно может быть опущено:

`template<typename> friend class Stack;`

После такого объявления возможна следующая реализация шаблонного оператора присваивания:

basics/stack6assign.hpp

```
template<typename T>
template<typename T2>
```

```

Stack<T>& Stack<T>::operator= (Stack<T2> const& op2)
{
    elems.clear();                                // Удаление существующих элементов
    elems.insert(elems.begin(),                  // Вставка в начало
                op2.elems.begin(), // всех элементов из op2
                op2.elems.end());
    return *this;
}

```

Независимо от того, какой реализацией вы воспользуетесь, при наличии такого шаблона-члена можно присвоить стек `int` стеку `float`:

```

Stack<int> intStack;      // Стек для int
Stack<float> floatStack; // Стек для float
...
floatStack = intStack;    // OK: Стеки имеют различные типы,
                         // но int преобразуется в float

```

Разумеется, такое присваивание не изменяет типа стека и его элементов. После присваивания тип элементов `floatStack` остается `float` и, следовательно, функция `pop()` будет по-прежнему возвращать значение типа `float`.

Может показаться, что проверка типов в этой функции отсутствует вообще, так что можно выполнять присваивание стеков с элементами любого типа, но это не так. Необходимая проверка типов происходит, когда элемент (копии) исходного стека помещается в результирующий стек:

```
elems.push_front(tmp.top());
```

Если, например, стек строк присвоить стеку значений с плавающей точкой, при компиляции этой строки будет выдано сообщение об ошибке, в котором будет сказано, что строка, возвращаемая функцией `tmp.top()`, не может быть передана как аргумент функции `elems.push_front()` (в зависимости от компилятора сообщения могут быть различными, но смысл их именно такой).

```

Stack<std::string> stringStack;    // Стек строк
Stack<float> floatStack;          // Стек чисел с плавающей точкой
...
floatStack = stringStack; // Ошибка: string не преобразуется в float

```

Можно также изменить реализацию так, чтобы параметризовать тип внутреннего контейнера.

basics/stack7decl.hpp

```

template<typename T, typename Cont = std::deque<T>>
class Stack
{
private:
    Cont elems;                                // Элементы
public:
    void push(T const&); // Добавление элементов в стек
    void pop();           // Снятие со стека
    T const& top() const; // Возврат верхнего элемента
}

```

```

bool empty() const    // Проверка пустоты стека
{
    return elems.empty();
}
// Присваивание стека с элементами типа T2
template<typename T2, typename Cont2>
Stack& operator= (Stack<T2, Cont2> const&);
// Для доступа к закрытым членам Stack<T2> для любого типа T2:
template<typename, typename> friend class Stack;
};

```

Шаблон оператора присваивания будет выглядеть, как показано ниже.

basics/stack7assign.hpp

```

template<typename T, typename Cont>
template<typename T2, typename Cont2>
Stack<T, Cont>&
Stack<T, Cont>::operator= (Stack<T2, Cont2> const& op2)
{
    elems.clear();                      // Удаление существующих элементов
    elems.insert(elems.begin(),          // Вставка в начало
                op2.elems.begin(),        // всех элементов из op2
                op2.elems.end());
    return *this;
}

```

Вспомним, что у шаблонов классов инстанцируются только вызываемые функции-члены. Таким образом, если избегать присваивания стеков с элементами разных типов, в качестве внутреннего контейнера вполне можно использовать вектор:

```

// Стек для int с использованием вектора в качестве контейнера
Stack<int, std::vector<int>> vStack;
***  

vStack.push(42);
vStack.push(7);
std::cout << vStack.top() << '\n';

```

Поскольку необходимости в операторе присваивания нет, сообщение об ошибке отсутствия функции-члена `push_front()` не выдается, и программа работает корректно.

Чтобы ознакомиться с полной реализацией последнего примера, рассмотрите все файлы из подкаталога `basics` с именами, которые начинаются со `stack7`.

Специализация шаблонов функций-членов

Шаблоны функций-членов также могут быть частично или полностью специализированы. Например, для класса

basics/boolstring.hpp

```

class BoolString
{
private:
    std::string value;

```

```

public:
    BoolString(std::string const& s)
        : value(s)
    {
    }
    template<typename T = std::string>
    T get() const // Получение значения (преобразованного в T)
    {
        return value;
    }
};

```

можно предоставить полную специализацию для шаблона функции-члена следующим образом:

basics/boolstringgetbool.hpp

```

// Полная специализация BoolString::getValue<>() для bool
template<>
inline bool BoolString::get<bool>() const
{
    return value == "true" || value == "1" || value == "on";
}

```

Учтите, что вам не нужно (и нельзя) объявлять специализации; вы можете только определять их. Поскольку это полная специализация, которая находится в заголовочном файле, необходимо объявить ее как *inline*, чтобы избежать ошибок при включении определения в различные единицы трансляции.

Класс и полную специализацию можно использовать следующим образом:

```

std::cout << std::boolalpha;
BoolString s1("hello");
std::cout << s1.get() << '\n';      // Вывод hello
std::cout << s1.get<bool>() << '\n'; // Вывод false
BoolString s2("on");
std::cout << s2.get<bool>() << '\n'; // Вывод true

```

Шаблоны специальных функций-членов

Шаблоны функций-членов могут использоваться везде, где специальные функции-члены позволяют копирование или перемещение объектов. Подобно операторам присваивания, определенным выше, они могут также быть конструкторами. Однако обратите внимание на то, что шаблоны конструкторов и операторов присваивания не заменяют стандартные предопределенные конструкторы и операторы присваивания. Шаблоны-члены не рассматриваются как специальные функции-члены, которые копируют или перемещают объекты. В рассмотренном примере при присваивании стеков одного и того же типа по-прежнему вызывается оператор присваивания по умолчанию.

Это может быть и хорошо, и плохо.

- Может случиться так, что шаблон конструктора или оператора присваивания обеспечивает лучшее совпадение, чем предопределенный копирующий/перемещающий конструктор или оператор присваивания, хотя шаблонная версия используется только для инициализации других типов (подробности см. в разделе 6.2).
- Не так просто “шаблонизировать” копирующий/перемещающий конструктор, например, чтобы иметь возможность ограничить его существование (подробности см. в разделе 6.4).

5.5.1. Конструкция `.template`

Иногда необходимо явным образом квалифицировать аргументы шаблона при вызове шаблона-члена. В этом случае вы должны использовать ключевое слово `template`, чтобы гарантировать, что `<` представляет собой начало списка аргументов шаблона. Рассмотрим пример, в котором используется стандартный тип `bitset`:

```
template<unsigned long N>
void printBitset(std::bitset<N> const& bs)
{
    std::cout << bs.template to_string<char, std::char_traits<char>,
                           std::allocator<char>>();
}
```

Для битового множества `bs` мы вызываем шаблонную функцию-член `to_string()`, явно указывая подробности типа строки. Без дополнительного применения `.template` компилятор не знает, что следующий далее токен “меньше” (`<`) на самом деле является не оператором “меньше”, а началом списка аргументов шаблона. Обратите внимание: это является проблемой, только если конструкция перед точкой зависит от параметра шаблона. В нашем примере параметр `bs` зависит от параметра шаблона `N`.

Запись `.template` (и аналогичные записи наподобие `->template` и `::template`) должны использоваться только внутри шаблонов и только если они следуют за выражением, которое зависит от параметра шаблона. Более подробно этот вопрос рассматривается в разделе 13.3.3.

5.5.2. Обобщенные лямбда-выражения и шаблоны членов

Обратите внимание на то, что обобщенные лямбда-выражения, введенные в C++14, являются сокращениями шаблонов членов. Простое лямбда-выражение, вычисляющее “сумму” двух аргументов произвольного типа

```
[](auto x, auto y)
{
    return x + y;
}
```

представляет собой сокращение для конструируемого по умолчанию объекта следующего класса:

```

class SomeCompilerSpecificName
{
public:
    SomeCompilerSpecificName(); // Конструктор вызывается
                                // только компилятором
    template<typename T1, typename T2>
    auto operator()(T1 x, T2 y) const
    {
        return x + y;
    }
};

```

Подробности представлены в разделе 15.10.6.

5.6. Шаблоны переменных

Начиная с C++14, переменные также могут быть параметризованы определенным типом. Это называется *шаблоном переменной* (variable template)⁴.

Например, можно использовать приведенный ниже код для определения значения π , не определяя пока что тип этого значения:

```
template<typename T>
constexpr T pi{3.1415926535897932385};
```

Заметим, что, как и для всех шаблонов, это объявление не может находиться в области видимости функции или блока.

Чтобы использовать шаблон переменной, необходимо указать ее тип. Например, следующий код использует две различные переменные в области видимости объявления `pi<>`:

```
std::cout << pi<double> << '\n';
std::cout << pi<float> << '\n';
```

Можно также объявить шаблоны переменных, которые используются в различных единицах трансляции:

```
// ===== header.hpp:
template<typename T> T val{}; // Инициализированное нулем значение

// ===== Единица трансляции 1:
#include "header.hpp"

int main()
{
    val<long> = 42;
    print();
}

// ===== Единица трансляции 2:

#include "header.hpp"
```

⁴ В английском языке следует быть внимательным и не путать шаблон переменной (variable template) и вариативный шаблон (variadic template). Хотя термины и похожи, означают они совершенно разные вещи.

```
void print()
{
    std::cout << val<long> << '\n'; // OK: выводит 42
}
```

Шаблоны переменных могут иметь аргументы шаблона по умолчанию:

```
template<typename T = long double>
constexpr T pi = T{3.1415926535897932385};
```

Можно использовать значение по умолчанию или значение любого иного типа:

```
std::cout << pi<> << '\n'; // Выводит long double
std::cout << pi<float> << '\n'; // Выводит float
```

Обратите, однако, внимание на то, что вы всегда должны использовать угловые скобки. Использование `pi` без таковых является ошибкой:

```
std::cout << pi << '\n'; // Ошибка
```

Шаблоны переменных могут быть параметризованы параметрами, не являющимися типами, которые могут также использоваться для параметризации инициализатора. Например:

```
#include <iostream>
#include <array>

template<int N> // Массив с N элементами,
    std::array<int, N> arr{}; // инициализированными нулем

template<auto N> // Тип dval зависит от
constexpr decltype(N) dval = N; // переданного значения

int main()
{
    std::cout << dval<'c'> << '\n'; // N имеет значение 'c' типа char
    arr<10>[0] = 42; // Первый элемент глобального arr

    // Цикл использует значения из arr
    for (std::size_t i = 0; i < arr<10>.size(); ++i)
    {
        std::cout << arr<10>[i] << '\n';
    }
}
```

И вновь обратите внимание: даже когда инициализация и итерирование `arr` происходит в различных единицах трансляции, используется одна и та же переменная `std::array<int, 10> arr` из глобальной области видимости.

Шаблоны переменных для данных-членов

Одним из полезных применений шаблонов переменных является определение переменных, которые являются членами шаблона класса. Например, если шаблон класса определяется следующим образом

```
template<typename T>
class MyClass
{
```

```
public:
    static constexpr int max = 1000;
};
```

который позволяет определить различные значения для разных специализаций `MyClass<>`, то вы можете определить

```
template<typename T>
int myMax = MyClass<T>::max;
```

так что прикладные программисты могут просто писать

```
auto i = myMax<std::string>;
```

вместо

```
auto i = MyClass<std::string>::max;
```

Это означает, что для стандартного класса, такого как

```
namespace std
{
    template<typename T> class numeric_limits
    {
        public:
            ...
            static constexpr bool is_signed = false;
            ...
    };
}
```

можно определить

```
template<typename T>
constexpr bool isSigned = std::numeric_limits<T>::is_signed;
```

чтобы иметь возможность писать

```
isSigned<char>
```

вместо

```
std::numeric_limits<char>::is_signed
```

Суффикс свойств `_v`

Начиная с C++17, стандартная библиотека использует шаблоны переменных для определения сокращений для всех свойств типов в стандартной библиотеке, которые дают (булево) значение. Например, чтобы иметь возможность писать

```
std::is_const_v<T> // Начиная с C++17
```

вместо

```
std::is_const<T>::value // Начиная с C++11
```

стандартная библиотека определяет

```
namespace std
{
    template<typename T>
    constexpr bool is_const_v = is_const<T>::value;
}
```

5.7. Шаблонные параметры шаблонов

Было бы полезно иметь возможность использовать шаблон класса в качестве параметра шаблона. В качестве примера мы вновь воспользуемся нашим шаблоном класса стека.

При использовании в стеках различных внутренних контейнеров прикладной программист вынужден указывать тип элементов стека дважды: чтобы указать тип внутреннего контейнера, необходимо указать как тип этого контейнера, так и (еще раз) тип его элементов.

```
Stack<int, std::vector<int>> vStack; // Стек, использующий вектор
```

Шаблонные параметры шаблонов позволяют объявлять шаблон класса `Stack` путем указания типа контейнера без повторного указания типа его элементов:

```
Stack<int, std::vector> vStack; // Стек, использующий вектор
```

Для этого нужно указать второй параметр шаблона как шаблонный параметр шаблона. В принципе это будет выглядеть следующим образом⁵:

basics/stack8decl.hpp

```
template<typename T,
         template<typename Elemt> class Cont = std::deque>
class Stack
{
private:
    Cont<T> elems;           // Элементы
public:
    void push(T const&);    // Добавление элементов в стек
    void pop();              // Снятие со стека
    T const& top() const;   // Возврат верхнего элемента
    bool empty() const;     // Проверка пустоты стека
{
    return elems.empty();
}
...
};
```

Отличие заключается в том, что второй параметр шаблона объявляется как шаблон класса:

```
template<typename Elemt> class Cont
```

Значение по умолчанию изменяется и вместо `std::deque<T>` становится `std::deque`. Параметр представляет собой шаблон класса, инстанцируемый для типа, передаваемого в качестве первого параметра шаблона:

```
Cont<T> elems;
```

⁵ У данного кода до версии C++17 имеется одна проблема, которую мы сейчас рассмотрим. Однако она проявляется только для значения по умолчанию `std::deque`. Поэтому данный пример допустимо использовать в качестве иллюстрации общих возможностей шаблонных параметров шаблонов с этим значением по умолчанию перед тем, как мы перейдем к рассмотрению, как работать с ним в версиях, предшествующих C++17.

То, что в приведенном выше коде первый параметр шаблона использован для инстанцирования второго параметра шаблона, — особенность данного примера, но отнюдь не правило. В общем случае можно инстанцировать шаблонный параметр шаблона любым типом из шаблона класса.

Как обычно, вместо ключевого слова `template` для параметров шаблона можно использовать ключевое слово `class`. До C++11 имя `Cont` могло заменяться именем шаблона класса

```
template<typename T,
         template<class Elec> class Cont = std::deque>
class Stack // OK
{
    ...
};
```

Начиная с C++11, можно также заменять `Cont` именем шаблона псевдонима, но до C++17 в стандарт не было внесено соответствующее изменение, разрешающее использование ключевого слова `typename` вместо `class` для объявления шаблонного параметра шаблона:

```
template<typename T,
         template<typename Elec> typename Cont = std::deque>
class Stack // Ошибка до C++17
{
    ...
};
```

Эти два варианта означают в точности одно и то же: использование `class` вместо `typename` не мешает использовать шаблон псевдонима в качестве аргумента, соответствующего параметру `Cont`.

Поскольку параметр шаблона шаблонного параметра шаблона не используется, можно просто опустить его имя (если только оно не обеспечивает полезное самодокументирование исходного текста):

```
template<typename T, template<typename> class Cont = std::deque>
class Stack
{
    ...
};
```

Соответственно должны быть модифицированы и функции-члены. Так, вы должны указать второй параметр шаблона как шаблонный параметр шаблона. То же самое относится и к реализации функций-членов. Функция-член `push()`, например, реализуется следующим образом:

```
template<typename T, template<typename> class Cont>
void Stack<T, Cont>::push(T const& elem)
{
    elems.push_back(elem); // Добавление копии переданного элемента
}
```

Обратите внимание: в то время как шаблонные параметры шаблона представляют собой заместители для шаблона класса или псевдонима, соответствующего заместителя для шаблонов функции или переменных не существует.

Соответствие шаблонных аргументов шаблонов

Если попытаться использовать новую версию `Stack`, то будет выдано сообщение об ошибке, информирующее, что значение по умолчанию `std::deque` несогласно с шаблонным параметром шаблона `Cont`. Проблема заключается в том, что до C++ 17 шаблонный аргумент шаблона должен был быть шаблоном с параметрами, точно соответствующими параметрам шаблонного параметра шаблона, который этот аргумент замещает, с некоторыми исключениями, относящимися к параметрам вариативных шаблонов (см. раздел 12.3.4). Аргументы шаблона по умолчанию для шаблонных аргументов шаблона не рассматриваются, так что добиться точного соответствия, просто опустив аргументы, которые имеют значения по умолчанию, нельзя (в C++17 значения аргументов по умолчанию *рассматриваются*).

Имеющаяся до C++17 проблема в этом примере связана с тем, что шаблон `std::deque` стандартной библиотеки имеет более чем один параметр шаблона: второй параметр (который описывает *распределитель памяти*) имеет значение по умолчанию, но до C++17 оно не учитывалось при сопоставлении `std::deque` параметру `Cont`.

Обходной путь заключается в том, чтобы переписать объявление класса так, чтобы параметр `Cont` ожидал контейнеры с двумя параметрами шаблона:

```
template<typename T,
         template<typename Elemt,
                  typename Alloc = std::allocator<Elemt>>
         class Cont = std::deque>
class Stack
{
private:
    Cont<T> elems; // Элементы
    ***
};
```

И вновь можно опустить имя `Alloc`, поскольку оно не используется.

Итак, последняя версия нашего шаблона `Stack` (включая шаблоны-члены для присваиваний стеков с отличающимися типами элементов) теперь выглядит следующим образом:

basics/stack 9.hpp

```
#include <deque>
#include <cassert>
#include <memory>

template<typename T,
         template<typename Elemt,
```

```
        typename = std::allocator<Elem>>
    class Cont = std::deque>
class Stack
{
private:
    Cont<T> elems;           // Элементы
public:
    void push(T const&);   // Добавление элементов в стек
    void pop();             // Снятие со стека
    T const& top() const;  // Возврат верхнего элемента
    bool empty() const     // Проверка пустоты стека
    {
        return elems.empty();
    }

    // Присваивание стека с элементами типа T2
    template<typename T2,
              template<typename Elemt2,
                        typename = std::allocator<Elemt2>
                      >class Cont2>
    Stack<T, Cont2>& operator= (Stack<T2, Cont2> const&);

    // Для доступа к закрытым членам Stack<T2> для любого типа T2:
    template<typename, template<typename, typename>class>
    friend class Stack;
};

template<typename T, template<typename, typename> class Cont>
void Stack<T, Cont>::push(T const& elem)
{
    elems.push_back(elem); // Добавление копии переданного элемента
}

template<typename T, template<typename, typename> class Cont>
void Stack<T, Cont>::pop()
{
    assert(!elems.empty());
    elems.pop_back();      // Удаление последнего элемента
}

template<typename T, template<typename, typename> class Cont>
T const& Stack<T, Cont>::top() const
{
    assert(!elems.empty());
    return elems.back();   // Возврат последнего элемента
}
template<typename T, template<typename, typename> class Cont>
template<typename T2, template<typename, typename> class Cont2>
Stack<T, Cont2>&
Stack<T, Cont>::operator= (Stack<T2, Cont2> const& op2)
{
    elems.clear();         // Удаление существующих элементов
    elems.insert(elems.begin(),          // Вставка в начало
                op2.elems.begin(), // всех элементов из op2
                op2.elems.end());
    return *this;
}
```

Обратите внимание на то, что для получения доступа ко всем членам определения мы объявляем, что все прочие экземпляры стека являются друзьями (опуская имена параметров шаблонов):

```
template<typename, template<typename, typename> class>
friend class Stack;
```

Тем не менее *не все* шаблоны стандартных контейнеров могут использоватьсь в качестве параметра `Cont`. Например, `std::array` работать не будет, потому что он включает в качестве длины массива параметр шаблона, не являющийся типом, который не имеет соответствия в нашем объявлении шаблонного параметра шаблона.

Приведенная ниже программа использует все возможности этой окончательной версии стека:

`basics/stack9test.cpp`

```
#include "stack9.hpp"
#include <iostream>
#include <vector>
int main()
{
    Stack<int> iStack; // Стек элементов типа int
    Stack<float> fStack; // Стек элементов типа float

    // Работа со стеком элементов типа int
    iStack.push(1);
    iStack.push(2);
    std::cout << "iStack.top(): " << iStack.top() << '\n';

    // Работа со стеком элементов типа float
    fStack.push(3.3);
    std::cout << "fStack.top(): " << fStack.top() << '\n';

    // Присваивание стека с отличающимся типом
    // элементов и дальнейшая работа
    fStack = iStack;
    fStack.push(4.4);
    std::cout << "fStack.top(): " << fStack.top() << '\n';

    // Стек для элементов типа double с использованием
    // вектора в качестве внутреннего контейнера
    Stack<double, std::vector> vStack;
    vStack.push(5.5);
    vStack.push(6.6);
    std::cout << "vStack.top(): " << vStack.top() << '\n';

    vStack = fStack;
    std::cout << "vStack: ";

    while(!vStack.empty())
    {
        std::cout << vStack.top() << ' ';
    }
}
```

```
        vStack.pop();
    }

    std::cout << '\n';
}
```

Вывод программы имеет следующий вид:

```
iStack.top(): 2
fStack.top(): 3.3
fStack.top(): 4.4
vStack.top(): 6.6
vStack: 4.4 2 1
```

Продолжение обсуждения данной темы и примеры шаблонных параметров шаблонов вы найдете в разделах 12.2.3, 12.3.4 и 19.2.2.

5.8. Резюме

- Для обращения к имени типа, которое зависит от параметра шаблона, его следует предварить ключевым словом `typename`.
- Для доступа к членам базового класса, которые зависят от параметров шаблона, следует квалифицировать обращение с помощью `this->` или имени их класса.
- Вложенные классы и функции-члены также могут быть шаблонами. Одним из применений этой возможности является реализация обобщенных операций с преобразованиями внутренних типов.
- Шаблонные версии конструкторов или операторов присваивания не заменяют предопределенные конструкторы или операторы присваивания.
- С помощью инициализации с фигурными скобками или явного вызова конструктора по умолчанию можно гарантировать, что переменные и члены шаблонов инициализируются со значением по умолчанию, даже если они инстанцируются со встроенными типами.
- Можно предоставить отдельные шаблоны для массивов, которые могут применяться также и для строковых литералов.
- При передаче массивов или строковых литералов во время вывода аргументов типы аргументов низводятся (выполняется преобразование массива в указатель) только в том случае, когда параметр не является ссылкой.
- Начиная с C++14, можно использовать *шаблоны переменных*.
- Шаблоны классов можно использовать в качестве параметров шаблонов – так называемых *шаблонных параметров шаблонов*.
- Для шаблонных параметров шаблонов должно выполняться точное соответствие.

Глава 6

Семантика перемещения и `enable_if<>`

Одной из наиболее значимых возможностей, введенных в C++11, была семантика перемещения. Можно использовать ее для оптимизации копирований и присваиваний, перемещая (“воруя”) внутренние ресурсы из исходного объекта в объект назначения вместо копирования этого содержимого. Это может быть выполнено при условии, что исходный объект больше не нуждается в своем внутреннем значении или состоянии (потому что все равно будет отброшен).

Семантика перемещения оказывает значительное влияние на дизайн шаблонов, и для ее поддержки в обобщенном коде были введены специальные правила. Эта глава знакомит вас с указанными возможностями.

6.1. Прямая передача

Предположим, что нужно написать обобщенный код, который передает фундаментальные свойства передаваемых аргументов.

- Модифицируемый объект должен оставаться модифицируемым.
- Константный объект должен быть передан как объект, предназначенный только для чтения.
- Перемещаемый объект (из которого можно “украсть” его внутреннее содержимое за ненужностью) должен передаваться как перемещаемый объект.

Для достижения этой функциональности без шаблонов мы должны запрограммировать все три случая. Например, чтобы передать аргумент из вызова `f()` соответствующей функции `g()`, мы пишем:

`basics/move1.cpp`

```
#include <utility>
#include <iostream>

class X
{
    ***
};

void g(X&)
{
    std::cout << "g() для переменной\n";
}
void g(X const&)
{
    std::cout << "g() для константы\n";
}
```

```

void g(X&&)
{
    std::cout << "g() для перемещаемого объекта\n";
}

// Передача функцией f() аргумента val в g():
void f(X& val)
{
    g(val); // val - неконстантное lvalue => вызов g(X&)
}
void f(X const& val)
{
    g(val); // val - константное lvalue => вызов g(X const&)
}
void f(X&& val)
{
    g(std::move(val)); // val - неконстантное lvalue =>
                        // необходим std::move() для вызова g(X&&)
}

int main()
{
    X v;           // Создание переменной
    X const c; // Создание константы
    f(v);         // Для неконстанты вызов f(X&) => g(X&)
    f(c);         // Для константы вызов f(X const&) => g(X const&)
    f(X());       // Для временного объекта вызов f(X&&) => g(X&&)
    f(std::move(v)); // Для перемещаемой переменной
                      // вызов f(X&&) => g(X&&)
}

```

Здесь мы видим три различные реализации `f()`, передающие аргументы в `g()`:

```

void f(X& val)
{
    g(val); // val - неконстантное l-значение => вызов g(X&)
}
void f(X const& val)
{
    g(val); // val - константное l-значение => вызов g(X const&)
}
void f(X&& val)
{
    g(std::move(val)); // val - неконстантное l-значение =>
                        // необходим std::move() для вызова g(X&&)
}

```

Обратите внимание на то, что код для перемещаемых объектов (работающий со *ссылками на r-значение* (или просто *r-ссылками*)) отличается от прочего кода: он должен использовать `std::move()`, потому что согласно правилам языка семантика перемещения не передается¹. Хотя `val` в третьей `f()` объявляется как *r-ссылка*, категория его значения при использовании в качестве выражения — не-

¹ То, что семантика перемещения не передается автоматически, сделано преднамеренно и имеет важное значение. Если бы это не было сделано, мы бы теряли значение перемещаемого объекта при первом его использовании в функции.

константное l-значение (см. приложение Б, “Категории значений”), и оно ведет себя так же, как и val в первой f(). Без применения move() для неконстантных значений вместо g(X&&) будет вызываться g(X&).

Если мы попытаемся объединить все три случая в обобщенном коде, то столкнемся с проблемой:

```
template<typename T>
void f(T& val)
{
    g(val);
}
```

Этот код работает для двух первых случаев, но не для третьего, когда передается перемещаемый объект.

По этой причине C++11 вводит специальные правила для *прямой передачи* (идеальной передачи, perfect forwarding) параметров. Идиоматическая схема кода имеет следующий вид:

```
template<typename T>
void f(T&& val)
{
    g(std::forward<T>(val)); // Прямая передача val в g()
}
```

Учтите, что std::move() не имеет шаблонного параметра и “запускает” семантику перемещения для передаваемого аргумента, в то время как std::forward<>() “передает” потенциальную семантику перемещения в зависимости от переданного аргумента шаблона.

Не думайте, что T&& для параметра шаблона T ведет себя так же, как X&& для конкретного типа X. Нет, здесь применяются разные правила! Однако синтаксически они выглядят одинаково.

- X&& для конкретного типа X объявляет параметр как r-ссылку. Она может быть связана только с перемещаемым объектом (r-g-значением, таким как временный объект, и x-значением, таким как объект, переданный с использованием std::move(); подробности см. в приложении Б, “Категории значений”). Она всегда изменяма, и вы всегда можете “украсть” ее значение².
- T&& для параметра шаблона T объявляет *передаваемую ссылку* (forwarding reference), именуемую также *универсальной ссылкой* (universal reference)³.

² Тип наподобие X const&& корректен, но на практике не распространен, поскольку “кража” внутреннего представления перемещаемого объекта требует его изменения. Однако он может использоваться для обеспечения передачи только временных значений или объектов с примененным к ним std::move() без возможности их модифицировать.

³ Термин *универсальная ссылка* был придуман Скоттом Мейерсом (Scott Meyers) как общий термин, который может означать и l-ссылку, и r-ссылку. Из-за слишком большой универсальности термина “универсальный” стандарт C++17 вводит термин *передаваемая ссылка* (forwarding reference), так как основной причиной использования такой ссылки является передача объектов. Однако обратите внимание на то, что передача не выполняется автоматически. Термин описывает не что такое, а для чего оно обычно используется.

Она может быть связана с изменяемым, неизменяемым (`const`) или перемещаемым объектом. В определении функции параметр может быть изменяемым, неизменяемым или указывать на объект, у которого можно “украсть” внутреннее содержимое.

Обратите внимание на то, что `T` в действительности должно быть именем параметра шаблона. Зависимости от параметра шаблона недостаточно. Для параметра шаблона `T` объявление наподобие `typename T::iterator&&` представляет собой просто г-ссылку, а не передаваемую ссылку.

Таким образом, полностью программа для прямой передачи аргументов выглядит следующим образом:

basics/move2.cpp

```
#include <utility>
#include <iostream>

class X
{
    ...
};

void g(X&)
{
    std::cout << "g() для переменной\n";
}
void g(X const&)
{
    std::cout << "g() для константы\n";
}
void g(X&&)
{
    std::cout << "g() для перемещаемого объекта\n";
}

// f() выполняет прямую передачу аргумента val в g():
template<typename T>
void f(T&& val)
{
    g(std::forward<T>(val)); // Вызов "правильной" g() для
                            // любого аргумента val
}

int main()
{
    X v;           // Создание переменной
    X const c; // Создание константы
    f(v);         // Для переменной вызывается f(X&) => g(X&)
    f(c);         // Для константы вызывается f(X const&) => g(X const&)
    f(X());       // Для временного значения вызывается f(X&&) => g(X&&)
    f(std::move( v )); // Для перемещаемой переменной
                        // вызывается f(X&&) => g(X&&)
}
```

Конечно же, прямая передача может также использоваться с вариативными шаблонами (см. примеры в разделе 4.3). Подробнее прямая передача рассматривается в разделе 15.6.3.

6.2. Шаблоны специальных функций-членов

Шаблоны функций-членов могут использоваться и для специальных функций-членов, включая конструкторы, что, однако, может привести к удивительному поведению.

Рассмотрим следующий пример.

basics/specialmemtmp11.cpp

```

Person p3(p1);           // Копирование Person =>
                         // вызов копирующего конструктора
Person p4(std::move(p1)); // Перемещение Person =>
                         // вызов перемещающего конструктора
}

```

Здесь представлен класс Person с членом-строкой name, для которого предлагаются инициализирующие конструкторы. Для поддержки семантики перемещения мы перегружаем конструктор, принимающий std::string.

- Мы предоставляем версию для строкового объекта, в которой name инициализируется копией переданного аргумента:

```

Person(std::string const& n) : name(n)
{
    std::cout << "Конструктор, копирующий строку "
          << name << "'\n";
}

```

- Мы предоставляем версию для перемещаемого строкового объекта, в которой для “кражи” значения используется std::move():

```

Person(std::string&& n) : name(std::move(n))
{
    std::cout << "Конструктор, перемещающий строку "
          << name << "'\n";
}

```

Как и ожидается, первый конструктор вызывается для строковых объектов (l-значений), а второй – для перемещаемых объектов (r-значений):

```

std::string s = "sname";
Person p1(s);           // Инициализация строкой =>
                         // копирующий строку конструктор
Person p2("tmp");       // Инициализация строковым литералом =>
                         // перемещающий строку конструктор

```

Помимо этих конструкторов в примере имеются реализации копирующего конструктора и перемещающего конструктора для случаев копирования/перемещения Person в целом:

```

Person p3(p1);           // Копирование Person =>
                         // вызов копирующего конструктора
Person p4(std::move(p1)); // Перемещение Person =>
                         // вызов перемещающего конструктора

```

Давайте теперь заменим два строковых конструктора одним обобщенным конструктором с использованием прямой передачи аргумента члену name.

basics/specialmemtmp12.hpp

```

#include <utility>
#include <string>
#include <iostream>

```

```

class Person
{
private:
    std::string name;
public:
    // Обобщенный конструктор для передачи начального имени:
    template<typename STR>
    explicit Person(STR&& n) : name(std::forward<STR>(n))
    {
        std::cout << "Шаблонный конструктор для '" << name << "'\n";
    }
    // Копирующий и перемещающий конструкторы:
    Person(Person const& p) : name(p.name)
    {
        std::cout << "Копирующий конструктор Person '" << name << "'\n";
    }
    Person(Person&& p) : name(std::move(p.name))
    {
        std::cout << "Перемещающий конструктор Person '" << name << "'\n";
    }
};

```

Создание объектов при передаче строк работает корректно, как и ожидалось:

```

std::string s = "sname";
Person p1(s);      // Инициализация строкой =>
                   // вызов шаблонного конструктора
Person p2("tmp"); // Инициализация строковым литералом =>
                   // вызов шаблонного конструктора

```

Обратите внимание, что при создании p2 временная строка не создается: параметр STR выводится как тип `char const[4]`. Применение `std::forward<STR>` к параметру-указателю конструктора не оказывает особого действия, и член `name` таким образом создается из С-строки с завершающим нулевым символом.

Но если мы попытаемся вызвать копирующий конструктор, то получим ошибку:

```
Person p3(p1); // Ошибка
```

в то время как инициализация нового объекта `Person` перемещаемым объектом работает корректно:

```
Person p4(std::move(p1)); // OK: перемещаемый объект Person =>
                         // вызов перемещающего конструктора
```

Обратите внимание на то, что копирование константного объекта `Person` также отлично работает:

```
Person const p2c("ctmp"); // Инициализация константного объекта
                          // строковым литералом
Person p3c(p2c);          // OK: копирование константного Person =>
                          // вызов копирующего конструктора
```

Проблема заключается в том, что согласно правилам разрешения перегрузки языка программирования C++ (см. раздел 16.2.4) для неконстантного l-значения `Person p` шаблон члена

```
template<typename STR>
Person(STR&& n)
```

оказывается лучшим соответствием, чем (обычно предопределенный) копирующий конструктор:

```
Person(Person const& p)
```

STR просто заменяется типом Person&, в то время как для копирующего конструктора необходимо преобразование в const.

Можно было бы попытаться решить эту проблему, предостав员я дополнительно неконстантный копирующий конструктор:

```
Person(Person& p)
```

Однако это лишь частичное решение, потому что для объектов производного класса шаблон члена остается лучшим соответствием. Что нам действительно нужно — это отключить шаблон члена для случая, когда переданный аргумент представляет собой Person или выражение, которое может быть преобразовано в Person. Это может быть сделано с помощью конструкции std::enable_if<>, о которой мы расскажем в следующем разделе.

6.3. Отключение шаблонов с помощью `enable_if<>`

Начиная с C++11, в стандартной библиотеке C++ имеется вспомогательный шаблон std::enable_if<>, позволяющий игнорировать шаблоны функций при определенных условиях времени компиляции.

Например, если шаблон функции foo<>() определен следующим образом:

```
template<typename T>
typename std::enable_if < (sizeof(T) > 4) ::type
foo()
{
}
```

то это определение foo<>() игнорируется, если условие `sizeof(T)>4` дает false⁴. Если `sizeof(T)>4` дает true, то шаблон функции раскрывается до

```
void foo()
{
}
```

То есть std::enable_if<> представляет собой свойство типа, которое вычисляет заданное выражение времени компиляции, переданное в качестве его (первого) аргумента шаблона, и ведет себя следующим образом.

- Если выражение вычисляется как true, член-тип type дает тип:
 - void, если второй аргумент не передан;
 - в противном случае — второй аргумент шаблона.

⁴ Не забывайте помечать условия в круглые скобки, так как в противном случае символ > в условии будет рассматриваться компилятором как завершение списка аргументов шаблона.

- Если выражение вычисляется как `false`, член-тип `type` не определен. Благодаря возможности шаблонов, которая называется SFINAE (substitution failure is not an error — ошибка подстановки ошибкой не является), о которой мы поговорим позже (см. раздел 8.4), шаблон функции с выражением `enable_if` в этом случае игнорируется.

Как и для всех свойств типов, которые представляют собой тип, начиная с C++14, имеется соответствующий шаблон псевдонима `std::enable_if_t<>`, который позволяет опустить `typename` и `::type` (см. подробности в разделе 2.8). Таким образом, начиная с C++14, можно написать

```
template<typename T>
std::enable_if_t <(sizeof(T) > 4) >
foo()
{
}
```

Если передать в `enable_if<>` или `enable_if_t<>` второй аргумент:

```
template<typename T>
std::enable_if_t <(sizeof(T) > 4), T >
foo()
{
    return T();
}
```

то конструкция `enable_if` раскроется до этого второго аргумента, если значение выражения будет вычислено как `true`. Так что, если `MyType` представляет собой конкретный тип, переданный или выведенный как `T`, и его размер окажется больше 4, результатом будет

```
MyType foo();
```

Обратите внимание на то, что выражение `enable_if` в середине объявления является довольно неуклюжей конструкцией. По этой причине наиболее распространенным способом использования `std::enable_if<>` является использование дополнительного аргумента шаблона функции со значением по умолчанию:

```
template<typename T,
         typename = std::enable_if_t<(sizeof(T) > 4) >>
void foo()
```

Такая запись при `sizeof(T) > 4` раскрывается в

```
template<typename T,
         typename = void>
void foo()
```

Если и это все еще выглядит слишком неуклюжим, и вы хотите сделать требование/ограничение более явно выраженным, с помощью шаблона псевдонима можно определить для него собственное имя⁵:

⁵ Мы благодарим Стивена Дьюхарста (Stephen C. Dewhurst) за то, что он обратил наше внимание на эту возможность..

```
template<typename T>
using EnableIfSizeGreater4 = std::enable_if_t < (sizeof(T) > 4) >;
template<typename T,
         typename = EnableIfSizeGreater4<T>>
void foo()
{
}
```

Вопросы реализации `std::enable_if` рассмотрены в разделе 20.3.

6.4. Использование `enable_if`

Мы можем использовать `enable_if` для решения нашей проблемы с шаблоном конструктора из раздела 6.2.

Задача, которую мы должны решить, — это отключение объявления шаблонного конструктора

```
template<typename STR>
Person(STR&& n);
```

если переданный аргумент `STR` имеет правильный тип (является `std::string` или типом, преобразуемым в `std::string`).

Для этого мы воспользуемся еще одним стандартным свойством типа `std::is_convertible<FROM, TO>`. Начиная с C++17, соответствующее объявление имеет следующий вид:

```
template<typename STR,
        typename = std::enable_if_t<
            std::is_convertible_v<STR, std::string>>>
Person(STR&& n);
```

Если тип `STR` может быть преобразован в тип `std::string`, полное объявление раскрывается в

```
template<typename STR,
        typename = void>
Person(STR&& n);
```

Если тип `STR` не может быть преобразован в `std::string`, весь шаблон функции игнорируется⁶.

И вновь для ограничения с помощью шаблона псевдонима можно определить собственное имя:

```
template<typename T>
using EnableIfString = std::enable_if_t <
    std::is_convertible_v<T, std::string>>;
***
```

⁶ Если вам интересно, почему мы вместо этого не проверяем, не преобразуется ли `STR` в `Person`, учтите: мы определяем функцию, которая могла бы позволить нам преобразовать `string` в `Person`. Таким образом, конструктор должен знать, разрешен ли он, а это зависит от того, может ли быть выполнено преобразование типов, что зависит от того, включен ли конструктор... и так далее. Никогда не используйте `enable_if` в местах, которые влияют на условие, используемое в `enable_if`. Это логическая ошибка, которую компиляторы могут не обнаружить.

```
template<typename STR, typename = EnableIfString<STR>>
Person(STR && n);
```

Таким образом, весь класс Person должен иметь следующий вид:

basics/specialmemtmp13.hpp

```
#include <utility>
#include <string>
#include <iostream>
#include <type_traits>

template<typename T>
using EnableIfString = std::enable_if_t<
    std::is_convertible_v<T, std::string>>;
class Person
{
private:
    std::string name;
public:
    // Обобщенный конструктор для передачи начального имени:
    template<typename STR, typename = EnableIfString<STR>>
    explicit Person(STR && n)
        : name(std::forward<STR>(n))
    {
        std::cout << "Шаблонный конструктор для '"
            << name << "'\n";
    }
    // Копирующий и перемещающий конструкторы:
    Person(Person const& p) : name(p.name)
    {
        std::cout << "Копирующий конструктор Person '"
            << name << "'\n";
    }
    Person(Person&& p) : name(std::move(p.name))
    {
        std::cout << "Перемещающий конструктор Person '"
            << name << "'\n";
    }
};
```

Теперь все ведет себя так, как и ожидалось:

basics/specialmemtmp13.cpp

```
#include "specialmemtmp13.hpp"

int main()
{
    std::string s = "sname";
    Person p1(s);           // Инициализация строковым объектом =>
                           // вызов шаблонного конструктора
    Person p2("tmp");       // Инициализация строковым литералом =>
                           // вызов шаблонного конструктора
    Person p3(p1);          // OK, вызов копирующего конструктора
    Person p4(std::move(p1)); // OK, вызов перемещающего конструктора
}
```

И вновь заметим, что в C++14 из-за отсутствия версии `_v` для свойств типов, возвращающих значения, мы должны объявить шаблон псевдонима следующим образом:

```
template<typename T>
using EnableIfString = std::enable_if_t <
    std::is_convertible<T, std::string>::value >;
```

А в C++11 из-за отсутствия версии `_t` для свойств типов, возвращающих типы, мы должны объявить шаблон специальной функции следующим образом:

```
template<typename T>
using EnableIfString
= typename std::enable_if<std::is_convertible<T, std::string>::value
>::type;
```

Но все это теперь скрыто в определении `EnableIfString<>`.

Заметим также, что имеется альтернатива использованию `std::is_convertible<>`, потому эта конструкция требует, чтобы типы были неявно преобразуемы. С помощью `std::is_constructible<>` мы также позволяем использовать для инициализации явные преобразования. Однако в этом случае порядок аргументов является противоположным:

```
template<typename T>
using EnableIfString = std::enable_if_t <
    std::is_constructible_v<std::string, T >>;
```

Подробную информацию о `std::is_constructible<>` можно найти в разделе Г.3.2, а в разделе Г.3.3 – информацию о `std::is_convertible<>`. Подробная информация о шаблоне `enable_if<>` и его применении в вариативных шаблонах содержится в разделе Г.6.

Отключение специальных функций-членов

Обратите внимание: как правило, мы не можем использовать `enable_if<>` для отключения стандартных копирующих/перемещающих конструкторов и операторов присваивания. Причина заключается в том, что шаблоны функций-членов никогда не учитываются как специальные функции-члены и игнорируются, когда, например, требуется копирующий конструктор. Таким образом, при наличии объявления

```
class C
{
public:
    template<typename T>
    C(T const&)
    {
        std::cout << "Шаблонный копирующий конструктор\n";
    }
    ...
};
```

при необходимости получения копии `C` используется предопределенный копирующий конструктор:

```
C x;
C y{x}; // Используется предопределенный копирующий конструктор
        // (не шаблон функции-члена)
```

(В действительности нет способа использования шаблона члена, потому что нет никакого способа указать или вывести его параметр шаблона Т.)

Удаление предопределенного копирующего конструктора решением не является, потому что попытка скопировать С приводит к ошибке.

Тем не менее имеется одно хитрое решение⁷. Можно объявить копирующий конструктор для аргументов const volatile и пометить его как удаленный (т.е. определить его с =delete). Это предотвращает неявное объявление другого копирующего конструктора. При этом можно определить шаблон конструктора, который для типов, не являющихся volatile, будет предпочтительнее (удаленного) копирующего конструктора:

```
class C
{
public:
    ...
    // Определяем предопределенный копирующий конструктор как
    // удаленный (с преобразованием в volatile для того, чтобы
    // обеспечить лучшее совпадение)
    C(C const volatile&) = delete;

    // Реализация шаблона копирующего конструктора
    // с лучшим соответствием:
    template<typename T>
    C(T const&)
    {
        std::cout << "Шаблонный копирующий конструктор\n";
    }
    ...
};
```

Теперь шаблонный конструктор используется даже для “нормального” копирования:

```
C x;
C y{x}; // Используется шаблон члена
```

В таком шаблонном конструкторе можно использовать дополнительные ограничения с помощью enable_if<>. Например, чтобы предотвратить возможность копирования объектов шаблона класса C<>, если параметр шаблона — целочисленный тип, можно написать следующий исходный текст:

```
template<typename T>
class C
{
public:
    ...
    // Реализация шаблона копирующего конструктора
    // с лучшим соответствием:
    C(C const volatile&) = delete;
```

⁷Мы благодарим Петера Димова (Peter Dimov) за то, что он указал нам на этот способ.

```
// Если T - не целочисленный тип, предоставляем шаблон
// копирующего конструктора с лучшим соответствием:
template<typename U,
         typename = std::enable_if_t<!std::is_integral<U>::value>>
C(C<U> const&)
{
    ...
}
...
};
```

6.5. Применение концептов для упрощения выражений `enable_if`

Даже при использовании шаблонов псевдонимов синтаксис `enable_if` довольно неуклюж, поскольку использует обходной путь: для получения желаемого результата мы добавляем дополнительный параметр шаблона и “ злоупотребляем” этим параметром для предоставления определенных требований к шаблонам функций, чтобы последние были в принципе доступны. Код, подобный получающемуся, трудно читать, и он делает остальные части шаблона функции трудными для понимания.

В принципе нам нужна просто возможность языка, которая бы позволяла формулировать требования или ограничения для функции таким образом, чтобы функция игнорировалась, если она не соответствует указанным требованиям/ограничениям.

Этот вопрос должен был быть решен долгожданной возможностью языка под названием *концепты* (concepts), которая позволяет формулировать требования/условия для шаблонов с использованием собственного простого синтаксиса. К сожалению, несмотря на долгое обсуждение, концепты пока еще не стали частью стандарта C++17. Однако некоторые компиляторы предоставляют экспериментальную поддержку такой возможности, и, вероятно, концепты станут частью очередного, следующего после C++17, стандарта.

При наличии концептов в том виде, в котором они были предложены, нам было бы достаточно просто написать:

```
template<typename STR>
requires std::is_convertible_v<STR, std::string>
Person(STR&& n) : name(std::forward<STR>(n))
{
    ...
}
```

Можно даже указать требование как обобщенный концепт

```
template<typename T>
concept ConvertibleToString = std::is_convertible_v<T, std::string>;
```

и сформулировать этот концепт в виде требования:

```
template<typename STR>
requires ConvertibleToString<STR>
```

```
Person(STR&& n) : name(std::forward<STR>(n))  
{  
    ...  
}
```

Оно может быть сформулировано и следующим образом:

```
template<ConvertibleToString STR>  
Person(STR&& n) : name(std::forward<STR>(n))  
{  
    ...  
}
```

Более подробно концепты C++ рассматриваются в приложении Д, “Концепты”.

6.6. Резюме

- В шаблонах можно “идеально” передать параметры, объявив их как *передаваемые ссылки* (forwarding references) (объявляются с типом, образованным именем параметра шаблона, за которым следует &&) и использовав std::forward() в соответствующем вызове.
- При использовании шаблонов функций-членов с прямой передачей они могут давать лучшее соответствие, чем предопределенные специальные функции-члены для копирования или перемещения объектов.
- Применение std::enable_if<> позволяет отключить шаблон функции, если не выполнено условие времени компиляции (существование шаблона в таком случае игнорируется).
- Используя std::enable_if<>, вы можете избежать проблемы, когда шаблоны конструкторов или операторов присваивания, которые могут быть вызваны для одного аргумента, обеспечивают лучшее соответствие, чем неявно генерируемые специальные функции-члены.
- Можно сделать шаблонной (и применить enable_if<>) специальную функцию-член, удаляя предопределенную специальную функцию-член для const volatile.
- Концепты позволяют нам применять более интуитивно понятный синтаксис для выражения требований к шаблонам функций.

Глава 7

По значению или по ссылке?

Поскольку C++ изначально предоставляет возможность передачи аргумента в функцию и по значению, и по ссылке, не всегда просто решить, какой из методов выбрать: обычно передача по ссылке дешевле для нетривиальных объектов, но более сложна. C++11 добавил еще и семантику перемещения, так что теперь у нас есть различные способы передачи по ссылке¹.

1. **x const&** (константная l-ссылка):

параметр ссылается на переданный объект, который невозможно модифицировать.

2. **x&** (неконстантная l-ссылка):

параметр ссылается на переданный объект, который можно модифицировать.

3. **x&&** (r-ссылка):

параметр ссылается на переданный объект с семантикой перемещения, что означает, что его можно модифицировать или “украсть” его значение.

Выбор способа объявления параметров с известными конкретными типами является достаточно сложной задачей. В шаблонах типы не известны, и поэтому становится еще труднее решить, какой механизм передачи аргументов использовать в том или ином случае.

Тем не менее в разделе 1.6.1 мы рекомендовали передавать параметры в шаблоны функций по значению, если только нет веских причин поступать иначе, например, следующих:

- невозможность копирования²;
- параметры используются для возврата данных;
- шаблоны просто передают параметры куда-то еще, сохраняя все свойства исходных аргументов;
- имеется серьезное улучшение производительности.

В этой главе обсуждаются различные подходы к объявлению параметров в шаблонах, причины общей рекомендации передавать аргументы по значению, а также аргументы в пользу отказа от этого решения. Кроме того, здесь

¹ Константная r-ссылка X const&& также возможна, но не имеет установленного семантического значения.

² Заметим, что, начиная с C++17, можно передавать временные сущности (r-значения) по значению, даже если копирующий конструктор или перемещающий конструктор отсутствуют (см. раздел B.2.1). Поэтому, начиная с C++17, существует дополнительное ограничение — невозможность копирования l-значений.

рассматриваются сложные проблемы, с которыми вы столкнетесь при работе со строковыми литералами и другими видами массивов.

При чтении этой главы было бы полезно ознакомиться с терминологией, связанной с категориями значений (*l*-значения, *r*-значения, *pr*-значения, *x*-значения и т.д.), которая описана в приложении Б, “Категории значений”.

7.1. Передача по значению

При передаче аргументов по значению каждый аргумент в принципе должен быть скопирован. Таким образом, каждый параметр становится копией переданного аргумента. В случае классов объект, создаваемый как копия, обычно инициализируется с помощью копирующего конструктора.

Вызов копирующего конструктора может быть достаточно дорогим. Однако существуют различные пути избежать дорогостоящего копирования даже при передаче параметров по значению: компиляторы при оптимизации зачастую могут полностью удалить операции копирования и с помощью семантики перемещения сделать копирование объектов дешевым даже для сложных объектов.

Например, рассмотрим простой шаблон функции, реализованный с передачей аргумента по значению:

```
template<typename T>
void printV(T arg)
{
    ...
}
```

При вызове этого шаблона функции для целочисленного значения получающийся в результате код имеет вид

```
void printV(int arg)
{
    ...
}
```

Параметр *arg* становится копией любого переданного аргумента, является он объектом, литералом или значением, возвращаемым вызовом функции.

Если мы определим объект типа `std::string` и вызовем для него наш шаблон функции:

```
std::string s = "hi";
printV(s);
```

то параметр шаблона *T* будет инстанцирован как `std::string`, так что мы получим

```
void printV(std::string arg)
{
    ...
}
```

И вновь при передаче строки *arg* становится копией *s*. На этот раз копия создается копирующим конструктором класса `string`, который является потенциально

дорогостоящей операцией, потому что в принципе эта операция копирования выполняет глубокое копирование, так что для копии выделяется собственная память для хранения значения³.

Однако потенциальный копирующий конструктор вызывается не всегда. Рассмотрим следующий код:

```
std::string returnString();
std::string s = "hi";
printV(s);           // Копирующий конструктор
printV(std::string("hi")); // Обычно копирование устраниется (или
                         // использует перемещающий конструктор)
printV(returnString()); // Обычно копирование устраниется (или
                         // использует перемещающий конструктор)
printV(std::move(s)); // Перемещающий конструктор
```

В первом вызове передается *l-значение*, что означает, что используется копирующий конструктор. Однако во втором и третьем вызовах, когда шаблон функции вызывается непосредственно для *pr-значений* (временные объекты, создаваемые на лету или возвращаемые другой функцией; см. приложение Б, “Категории значений”), компиляторы обычно выполняют оптимизацию, передавая аргумент так, что копирующий конструктор не вызывается вовсе. Обратите внимание на то, что, начиная с C++17, эта оптимизация является обязательной. До C++17 компилятор, который не отбрасывал копирование при оптимизации, должен был как минимум попытаться использовать семантику перемещения, которая обычно удешевляет копирование. В последнем вызове при передаче *x-значения* (существующий неконстантный объект с `std::move()`) мы принудительно вызываем перемещающий конструктор, указывая, что больше не нуждаемся в значении *s*.

Таким образом, вызов реализации `printV()`, которая объявлена с передачей параметра по значению, обычно оказывается дороже только при передаче *l-значения* (объекта, созданного ранее и обычно используемого после вызова, раз уж мы не использовали `std::move()` при его передаче). К сожалению, это довольно распространенный случай. Одной из причин является распространенность практики создания объектов заранее для передачи их другим функциям позднее (после некоторых изменений).

Низведение при передаче по значению

У передачи по значению есть еще одно свойство, о котором мы должны упомянуть: при передаче аргументов по значению тип *низводится* (decay). Это означает,

³ Сама реализация класса `string` может включать некоторые оптимизации, удешевляющие копирование. Одной из них является оптимизация небольших строк (*small string optimization — SSO*), которая использует для хранения значения без выделения памяти некоторое количество памяти непосредственно внутри объекта, пока это значение не слишком длинное. Другой оптимизацией является оптимизация копирования при записи (*copy-on-write — COW*), которая создает копию, используя ту же память, что и источник, пока ни источник, ни копия не изменяются. Однако оптимизация копирования при записи имеет существенные недостатки в многопоточном коде и по этой причине она запрещена для стандартных строк, начиная со стандарта C++11.

что обычные массивы преобразуются в указатели, и что квалификаторы, например `const` и `volatile`, удаляются (так же, как и при использовании значения в качестве инициализатора для объекта, объявленного с помощью `auto`)⁴:

```
template<typename T>
void printV(T arg)
{
    ...
}
std::string const c = "hi";
printV(c); // с низводится, так что arg имеет тип std::string
printV("hi"); // Низводится до указателя,
               // так что arg имеет тип char const*
int arr[4];
printV(arr); // Низводится до указателя,
              // так что arg имеет тип int*
```

Таким образом, при передаче строкового литерала "hi" его тип `char const [3]` низводится до `char const*`, так что именно таков выведенный тип `T`. Итак, шаблон создается следующим образом:

```
void printV(char const* arg)
{
    ...
}
```

Это поведение является производным от языка программирования С и имеет свои преимущества и недостатки. Часто оно упрощает обработку передаваемых строковых литералов, но его недостатком является то, что внутри `printV()` мы не можем различить передачу указателя на один элемент и передачу массива. Как бороться со строковыми литералами и другими массивами, мы обсудим в разделе 7.4.

7.2. Передача по ссылке

Давайте теперь рассмотрим различные “за” передачу по ссылке. Во всех случаях копия не создается (поскольку параметр просто ссылается на передаваемый аргумент). Кроме того, при этом не происходит низведения типа. Однако иногда такая передача невозможна, а если и возможна, то бывают ситуации, когда результатирующий тип параметра может вызвать проблемы.

7.2.1. Передача с помощью константной ссылки

Чтобы избежать любого (ненужного) копирования при передаче объектов, не являющихся временными, можно использовать константные ссылки. Например:

```
template<typename T>
void printR(T const& arg)
{
    ...
}
```

⁴ Термин *низведение* (decay) пришел из языка С и применим также к преобразованию типа функции в указатель на функцию (см. раздел 11.1.1).

При использовании данного объявления передача объекта никогда не приводит к созданию копии (независимо от ее дороговизны):

```
std::string returnString();
std::string s = "hi";
printR(s);           // Копия не создается
printR(std::string("hi")); // Копия не создается
printR(returnString()); // Копия не создается
printR(std::move(s)); // Копия не создается
```

По ссылке передается даже `int` (что несколько контрпродуктивно, хотя и не имеет особого значения). Таким образом:

```
int i = 42;
printR(i); // Передача ссылки вместо копирования i
```

приводит к следующему инстанцированию `printR()`:

```
void printR(int const& arg)
{
    ...
}
```

За сценой передача аргумента по ссылке осуществляется с помощью передачи адреса аргумента. Адреса кодируются компактно, так что передача адреса из вызывающей функции в вызываемую сама по себе эффективна. Однако при передаче адреса для компилятора создается некоторая неопределенность: что вызываемая функция делает с этим адресом? В теории вызываемый код может изменять все значения, до которых удается “добраться” с использованием этого адреса. Это означает, что компилятор должен считать, что все значения, которые он может кешировать (обычно в регистрах процессора), после вызова являются недействительными. Перезагрузка всех этих значений может оказаться довольно дорогой. Вы можете возразить, что мы передаем *константную* ссылку: не может ли компилятор на этом основании определить, что никакие изменения не могут произойти? К сожалению, это не так, потому что вызывающий код может изменить объект через свою собственную, неконстантную ссылку⁵.

Эти плохие новости смягчаются применением встраивания: если компилятор может развернуть вызов как *встроенный*, он может увидеть вызывающий и вызываемый код *вместе* и во многих случаях “понять”, что адрес не используется ни для чего, кроме передачи указанного им значения. Шаблоны функций часто очень короткие и потому являются вероятными кандидатами для встраивания. Однако, если шаблон инкапсулирует более сложный алгоритм, встраивание вряд ли произойдет.

Передача по ссылке не приводит к низведению

При передаче аргументов по ссылке *низведение* их типов не выполняется. Это означает, что массивы не преобразуются к указателям, а квалификаторы наподобие `const` и `volatile` не удаляются. Однако, поскольку параметр *вызыва*-*ется* как `T const&`, сам параметр *шаблона* `T` как `const` не выводится. Например:

⁵ Еще одним, более явным способом изменения объекта является применение `const_cast`.

```
template<typename T>
void printR(T const& arg)
{
    ***
}
std::string const c = "hi";
printR(c); // Т выводится как std::string, arg - std::string const&
printR("hi"); // Т выводится как char[3], arg - char const(&)[3]
int arr[4];
printR(arr); // Т выводится как int[4], arg - int const(&)[4]
```

Таким образом, *локальные* объекты `printR()`, объявленные с типом `T`, константными не являются.

7.2.2. Передача с помощью неконстантной ссылки

Когда вы хотите возвращать значения с помощью переданных аргументов (то есть когда вы хотите использовать выходные параметры), вы должны использовать неконстантные ссылки (если только не предпочитаете передавать их через указатели). И вновь это означает, что при передаче аргументов не создается копия. Параметры шаблона вызываемой функции просто получают непосредственный доступ к переданному аргументу.

Рассмотрим следующий код:

```
template<typename T>
void outR(T& arg)
{
    ***
}
```

Обратите внимание на то, что вызов `outR()` для временного объекта (`pr-значение`) или существующего объекта, переданного с использованием `std::move()` (`x-значение`), обычно не разрешен:

```
std::string returnString();
std::string s = "hi";
outR(s); // OK: Т выводится как std::string,
          // arg - std::string&
outR(std::string("hi")); // Ошибка: нельзя передавать временный
                       // объект (pr-значение)
outR(returnString()); // Ошибка: нельзя передавать временный
                      // объект (pr-значение)
outR(std::move(s)); // Ошибка: нельзя передавать x-значение
```

Можно передавать массивы неконстантных типов (над которыми здесь так же, как и в предыдущем разделе, не будет выполняться низведение типов):

```
int arr[4];
outR(arr); // OK: Т выводится как int[4], arg - int(&)[4]
```

Таким образом, можно изменять элементы, а также, например, работать с размером массива:

```
template<typename T>
void outR(T& arg)
```

```

    {
        if (std::is_array<T>::value)
        {
            std::cout << "Массив из " << std::extent<T>::value
                << " элементов\n";
        }
    }

    ...
}

```

Однако шаблоны здесь с хитростью. Если вы передадите `const`-аргумент, вывод может привести к тому, что `arg` станет объявлением константной ссылки, что означает, что становится возможным передавать `l`-значение там, где, казалось бы, ожидается `l`-значение:

```

std::string const c = "hi";
outR(c);                                // OK: выводится как std::string const
outR(returnConstString()); // OK: то же, если returnConstString()
                           // возвращает const string
outR(std::move(c));                      // OK: T выводится как std::string const6
outR("hi");                             // OK: T выводится как char const[3]

```

Конечно, любая попытка изменить переданный аргумент внутри шаблона функции в такой ситуации является ошибкой. Передача константного объекта возможна в самом выражении вызова, но, когда функция полностью инстанцирована (что может произойти позже в процессе компиляции), любая попытка изменить значение вызовет ошибку (которая, однако, может произойти глубоко внутри вызванного шаблона; см. раздел 9.4).

Если вы хотите запретить передачу константных объектов по неконстантным ссылкам, то можно сделать следующее:

- использовать `static_assert` для генерации ошибки времени компиляции:

```

template<typename T>
void outR(T& arg)
{
    static_assert(!std::is_const<T>::value,
                 "Выходной параметр foo<T>(T&) - константа");

    ...
}

```

- отключить шаблон для этого случая совсем с помощью `std::enable_if<>` (см. раздел 6.3):

```

template <typename T,
          typename = std::enable_if_t<!std::is_const<T>::value>
void outR(T& arg)

{
    ...
}

```

⁶ При передаче `std::move(c)`, сначала `std::move()` преобразует `c` в `std::string const&&`, что приводит к выводу `T` как `std::string const`.

или концептов (если таковые поддерживаются; см. раздел 6.5 и приложение Д, “Концепты”):

```
template<typename T>
requires !std::is_const_v<T>
void outR(T& arg)
{
    ***
}
```

7.2.3. Передача с помощью передаваемой ссылки

Одной из причин использования передачи аргументов по ссылке является возможность прямой передачи параметра (см. раздел 6.1). Но помните, что при использовании передаваемой ссылки, которая определяется как г-ссылка параметра шаблона, применяются специальные правила. Рассмотрим следующий код:

```
template<typename T>
void passR(T& arg)      // arg объявлен как передаваемая ссылка
{
    ***
}
```

Передаваемой ссылке можно передать что угодно, и, как всегда при передаче аргумента по ссылке, копия не создается:

```
std::string s = "hi";
passR(s);                  // OK: Т выводится как std::string&
                            // (а также как тип arg)
passR(std::string("hi"));  // OK: Т выводится как std::string,
                            // arg - std::string&&
passR(returnString());    // OK: Т выводится как std::string,
                            // arg - std::string&&
passR(std::move(s));      // OK: Т выводится как std::string,
                            // arg - std::string&&
passR(arr);               // OK: Т выводится как int(&)[4]
                            // (а также как тип arg)
```

Однако специальные правила вывода типов могут несколько удивить:

```
std::string const c = "hi";
passR(c);      // OK: Т выводится как std::string const&
passR("hi");   // OK: Т выводится как char const(&)[3]
                // (а также как тип arg)
int arr[4];
passR("hi");   // OK: Т выводится как int (&)[4]
                // (а также как тип arg)
```

В каждом из этих случаев внутри `passR()` параметр `arg` имеет тип, который знает, передали мы г-значение (чтобы использовать семантику перемещения) или константное/неконстантное l-значение. Это единственный способ передачи аргумента, который можно использовать для того, чтобы обеспечить различное поведение для каждого из этих трех случаев.

Это создает впечатление, что объявление параметра как передаваемой ссылки — почти идеальное решение. Но будьте осторожны, бесплатных обедов не бывает.

Например, это единственный случай, когда параметр T шаблона может неявно стать ссылочным типом. Как следствие, может возникнуть ошибка при использовании T при объявлении локального объекта без инициализации:

```
template<typename T>
void passR(T&& arg) // arg – передаваемая ссылка
{
    T x; // Для переданных 1-значений x представляет
          // собой ссылку, которая требует инициализатор
    ...
}
passR(42); // OK: T выводится как int
int i;
passR(i); // Ошибка: T выводится как int&, что делает
           // объявление x в passR() некорректным
```

Как поступить в этой ситуации, рассказывается в разделе 15.6.2.

7.3. Использование `std::ref()` и `std::cref()`

Начиная с C++11, можно позволить вызывающему коду решить, должен ли аргумент шаблона функции передаваться по значению или по ссылке. Когда шаблон объявлен как принимающий аргументы по значению, вызывающий код может использовать `std::cref()` и `std::ref()`, объявленные в заголовочном файле `<functional>`, для передачи аргумента по ссылке. Например:

```
template<typename T>
void printT(T arg)
{
    ...
}

std::string s = "hello";
printT(s); // Передача s по значению
printT(std::cref(s)); // Передача s "как будто по ссылке"
```

Однако обратите внимание на то, что `std::cref()` не изменяет обработку параметра в шаблоне. Вместо этого данная конструкция использует хитрый трюк: он заворачивает переданный аргумент s в объект, который действует как ссылка. Фактически она создает объект типа `std::reference_wrapper<>`, ссылающийся на исходный аргумент, и передает этот объект по значению. Эта обертка более или менее поддерживает только одну операцию: неявное преобразование типа обратно в исходный тип, давая при этом исходный объект⁷. Так, всякий раз, когда у вас есть корректный оператор для переданного объекта, вместо последнего можно использовать оболочку. Например:

⁷ Можно также вызывать функцию-член `get()` этого объекта-оболочки или использовать его как функциональный объект.

basics/cref.cpp

```
#include <functional>      // Для std::cref()
#include <string>
#include <iostream>

void printString(std::string const& s)
{
    std::cout << s << '\n';
}

template<typename T>
void printT(T arg)
{
    printString(arg);      // Может преобразовывать arg
}                                // обратно в std::string

int main()

{
    std::string s = "hello";
    printT(s);            // Вывод s, переданного по значению
    printT(std::cref(s)); // Вывод s, переданного "как будто по ссылке"
}
```

Последний вызов передает по значению объект типа `std::reference_wrapper<string const>` параметру `arg`, который затем передается далее и преобразуется при этом в базовый тип `std::string`.

Обратите внимание на то, что компилятор должен знать о необходимости неявного преобразования обратно в исходный тип. По этой причине `std::ref()` и `std::cref()` обычно хорошо работают только тогда, когда вы передаете объекты с помощью обобщенного кода обобщенным функциям. Например, попытки непосредственно вывести переданный объект обобщенного типа `T` будут неуспешны, поскольку нет оператора вывода, определенного для `std::reference_wrapper<>`:

```
template<typename T>
void printV(T arg)
{
    std::cout << arg << '\n';
}

***  

std::string s = "hello";
printV(s);          // OK
printV(std::cref(s)); // Ошибка: для оболочки нет оператора <<
```

Кроме того, следующий код не будет работать, потому что объект оболочки нельзя сравнивать с `char const*` или `std::string`:

```
template<typename T1, typename T2>
bool isless(T1 arg1, T2 arg2)
{
    return arg1 < arg2;
}

***  

std::string s = "hello";
```

```
if (isless(std::cref(s), "world")) ... // Ошибка
if (isless(std::cref(s), std::string("world"))) ... // Ошибка
```

Не поможет также попытка использовать общий тип T для arg1 и arg2:

```
template<typename T>
bool isless(T arg1, T arg2)
{
    return arg1 < arg2;
}
```

поскольку в этом случае компилятор получит конфликтующие типы при попытке вывести T для arg1 и arg2.

Таким образом, эффект применения класса `std::reference_wrapper` заключается в возможности использовать ссылку как “первоклассный объект”, который можно копировать и потому передавать по значению в шаблоны функций. Его также можно использовать в классах, например для хранения в контейнерах ссылок на объекты. Но в конечном итоге всегда требуется обратное преобразование к базовому типу.

7.4. Работа со строковыми литералами и массивами

До сих пор мы видели различные результаты для параметров шаблонов при использовании строковых литералов и обычных С-массивов:

- передача по значению низводит данные типы, так что они становятся указателями на тип элемента;
- при любой разновидности передачи по ссылке низведение не выполняется, так что аргументы, ставшие ссылками, ссылаются на массивы.

И то, и другое и хорошо, и плохо. При низведении массива к указателю вы теряете возможность различать обработку указателей на элементы и обработку переданных массивов. С другой стороны, при работе с параметрами, в которые могут быть переданы строковые литералы, отсутствие низведения может стать проблемой, поскольку строковые литералы разного размера в результате оказываются имеющими различные типы. Например:

```
template<typename T>
void foo(T const& arg1, T const& arg2)
{
    ...
}
foo("hi", "guy"); // Ошибка
```

Здесь вызов `foo ("hi", "guy")` не будет компилироваться, поскольку "hi" имеет тип `char const[3]`, в то время как "guy" имеет тип `char const[4]`; шаблон же требует, чтобы они были одного и того же типа T. Этот код успешно компилируется только тогда, когда строковые литералы в нем имеют одинаковую длину. По этой причине настоятельно рекомендуется при тестировании использовать строковые литералы разной длины.

При объявлении шаблона функции `foo()` с передачей передачи аргументов по значению упомянутый вызов вполне возможен:

```
template<typename T>
void foo(T arg1, T arg2)
{
    ...
}
foo("hi", "guy"); // Компилируется, но...
```

Но это не значит, что все проблемы исчезли. Все становится даже хуже — проблемы времени компиляции могут стать проблемами времени выполнения. Рассмотрим следующий код, где мы сравниваем переданные аргументы с помощью оператора `==`:

```
template<typename T>
void foo(T arg1, T arg2)
{
    if (arg1 == arg2) // Ой: сравнение адресов
    {
        // или переданных массивов
        ...
    }
}
foo("hi", "guy"); // Компилируется, но...
```

Как было сказано, вы должны знать, что следует интерпретировать переданные указатели на символы как строки. Но, вероятно, это необходимо в любом случае, потому что шаблон должен также работать с аргументами, получаемыми из уже низведенных строковых литералов (например, передаваемых из другой функции, куда они были переданы по значению, или присвоенных объекту, объявленному с ключевым словом `auto`).

Тем не менее во многих случаях низведение полезно, особенно для проверки того, имеют ли два объекта (оба переданные как аргументы или один переданный как аргумент, а второй — ожидающий аргумента) один и тот же тип или являются конвертируемыми в один и тот же тип. Одним из типичных применений является прямая передача. Но если вы хотите использовать прямую передачу, то нужно объявить параметры как передаваемые ссылки. При этом можно выполнить явное низведение аргументов с использованием свойства типа `std::decay<>()` (см. конкретный пример в рассказе о `std::make_pair()` в разделе 7.6).

Обратите внимание на то, что другие свойства типов иногда также неявно выполняют низведение, как, например, `std::common_type<>`, который дает общий тип для двух переданных в качестве аргументов типов (см. раздел 1.3.3 и раздел Г.5).

7.4.1. Специальные реализации для строковых литералов и обычных массивов

Возможно, вам придется использовать различные реализации в зависимости от того, передан указатель или массив. Конечно, при этом требуется, чтобы переданный массив не был низведен к указателю.

Чтобы различать эти случаи, необходимо выяснить, не передан ли массив. В принципе, имеется два варианта.

- Можно объявить параметры шаблона так, чтобы они были корректны только для массивов:

```
template<typename T, std::size_t L1, std::size_t L2>
void foo(T(&arg1)[L1], T(&arg2)[L2])
{
    T* pa = arg1; // Низведение arg1
    T* pb = arg2; // Низведение arg2

    if (compareArrays(pa, L1, pb, L2))
    {
        ...
    }
}
```

Здесь `arg1` и `arg2` должны быть обычными массивами элементов одного и того же типа `T`, но с разными размерами `L1` и `L2`. Заметим, однако, что может потребоваться несколько реализаций для поддержки массивов различных видов (см. раздел 5.4).

- Можно использовать свойства типов для выяснения, передан массив (или указатель):

```
template<typename T,
         typename = std::enable_if_t<std::is_array_v<T>>>
void foo(T && arg1, T && arg2)
{
    ...
}
```

Благодаря специальной обработке зачастую наилучший способ работы с массивами различными способами заключается в том, чтобы просто использовать различные имена функций. Еще лучше, конечно, чтобы вызывающий шаблон код использовал `std::vector` или `std::array`. Но, пока строковые литералы будут представлять собой обычные массивы, мы всегда должны учитывать их существование.

7.5. Работа с возвращаемыми значениями

Что касается возвращаемых значений, то здесь вы также можете выбирать между возвратом значений или ссылок. Однако возврат ссылки — потенциальный источник неприятностей, поскольку вы ссылаетесь на нечто, что находится вне вашего контроля. Есть несколько случаев, когда возврат ссылки представляет собой обычную практику.

- Возврат элементов контейнеров или строк (например, с помощью `operator[]` или `front()`).
- Предоставление возможности записи члена класса.

- Возврат объектов для цепочек вызовов (операторы `operator<<` и `operator>>` для потоков и `operator=` в общем случае для объектов классов).

Кроме того, распространено предоставление доступа к члену для чтения путем возврата константной ссылки.

Обратите внимание на то, что во всех этих случаях ненадлежащее применение ссылок может вызывать проблемы. Например:

```
std::string* s = new std::string("whatever");
auto& c = (*s)[0];
delete s;
std::cout << c;      // Ошибка времени выполнения
```

Здесь мы получаем ссылку на элемент строки, но к тому времени, когда используется эта ссылка, основная строка больше не существует (т.е. имеется *висячая ссылка* (*dangling reference*)), так что мы сталкиваемся с неопределенным поведением. Это несколько надуманный пример (опытный программист сразу заметит проблему), но такие вещи могут легко стать менее очевидными. Например:

```
auto s = std::make_shared<std::string>("whatever");
auto& c = (*s)[0];
s.reset();
std::cout << c;      // Ошибка времени выполнения
```

Таким образом, мы должны гарантировать, что шаблоны функции возвращают их результат по значению. Однако, как говорилось в данной главе, применение параметра шаблона `T` не гарантирует, что это не будет ссылкой, потому что `T` иногда может неявно выводиться как ссылка:

```
template<typename T>
T retR(T& p)          // p - передаваемая ссылка
{
    return T{...}; // Ой: возврат по ссылке для 1-значений
}
```

Даже когда `T` представляет собой параметр шаблона, выведенный из вызова с передачей по значению, она может стать ссылочным типом при явном указании, что параметр шаблона является ссылкой:

```
template<typename T>
T retV(T p)          // Примечание: T может стать ссылкой
{
    return T{...}; // Ой: возвращает ссылку, если T - ссылка
}

int x;
retV<int&>(x);      // retT() инстанцирована для T как int&
```

Для достижения безопасности возможны два варианта действий:

- Использовать свойство типа `std::remove_reference<>` (см. раздел Г.4) для преобразования `T` в тип, не являющийся ссылочным:

```
template<typename T>
typename std::remove_reference<T>::type retV(T p)
```

```
{
    return T{...}; // Всегда возврат значения
}
```

Могут оказаться полезными и другие свойства, такие как `std::decay<>` (см. раздел Г.4), которые также неявно удаляют ссылочность.

- Позволить компилятору выводить тип возвращаемого значения, просто объявляя возвращаемый тип как `auto` (начиная с C++14; см. раздел 1.3.2), поскольку при использовании `auto` всегда выполняется низведение типа:

```
template<typename T>
auto retV(T p)      // Выведенный компилятором тип,
{                  // возвращаемый по значению
    return T{...}; // Всегда возврат значения
}
```

7.6. Рекомендуемые объявления параметров шаблона

Как мы узнали в предыдущих разделах, существуют разные способы объявления параметров, которые зависят от параметров шаблона.

- Объявление аргументов как передаваемых **по значению**.

Этот подход прост, он низводит строковые литералы и обычные массивы до указателей, но не обеспечивает наивысшую производительность для больших объектов. Вызывающий код может по-прежнему выполнить передачу по ссылке с помощью `std::cref()` и `std::ref()`, но должен быть осторожен и убедиться, что такая передача является допустимой.

- Объявление аргументов как передаваемых **по ссылке**.

Этот подход часто обеспечивает лучшую производительность для достаточно крупных объектов, особенно при передаче

- существующих объектов (*l*-значений) *l*-ссылкам;
- временных объектов (*rg*-значений) или объектов, помеченных как перемещаемые (*x*-значений) *g*-ссылкам;
- обоих видов значений передаваемым ссылкам.

Поскольку во всех этих случаях над аргументами не выполняется низведение типов, может понадобиться специальная обработка при передаче строковых литералов и других массивов. Для передаваемых ссылок следует также остерегаться неявного вывода ссылочных типов.

Общие рекомендации

С учетом сказанного для шаблонов функций мы рекомендуем следующее.

1. По умолчанию объявляйте параметры **передаваемыми по значению**. Это просто и обычно работает даже со строковыми литералами. Для небольших аргументов и для временных или перемещаемых объектов производительность оказывается достаточной. Чтобы избежать дорогостоящих

копирований,зывающий код может иногда использовать `std::ref()` и `std::cref()` при передаче существующих крупных объектов (l-значений).

2. При наличии важных причин поступайте **иначе**.

- Если вам нужны *выходные* параметры, которые возвращают новые объекты или позволяют вызывающему коду модифицировать аргументы, передавайте аргумент как неконстантную ссылку (если только вы не предпочитаете передать указатель). Однако можно рассмотреть запрет случайногоятия константных объектов, как показано в разделе 7.2.2.
 - Если шаблон предназначен для *передачи* аргумента, используйте прямую передачу, т.е. объявляйте параметры как передаваемые ссылки и используйте в соответствующих местах `std::forward<>()`. Подумайте о применении `std::decay<>` или `std::common_type<>` для “согласования” различных типов строковых литералов и массивов.
 - Если ключевой характеристикой является *производительность* и ожидается, что копирование аргументов будет дорогим, используйте константные ссылки. Этот способ, конечно, не применяется, если вам в любом случае нужна локальная копия.
3. Если вы знаете ситуацию более точно, не следуйте этим рекомендациям. Однако ни в коем случае не следует делать интуитивных предположений о производительности. Здесь постоянно ошибаются даже эксперты. Обязательно выполняйте измерения!

Не переборщите с обобщенностью

Заметим, что на практике шаблоны функций часто не предназначены для произвольных типов аргументов и используют некоторые ограничения. Например, вы можете знать, что будут передаваться только векторы некоторого типа. В этом случае лучше не объявлять такую функцию слишком обобщенно, потому что, как уже говорилось, это может привести к удивительным побочным эффектам. Вместо этого используйте следующее объявление:

```
template<typename T>
void printVector(std::vector<T> const& v)
{
    ***
}
```

При таком объявлении параметра `v` в `printVector()` мы можем быть уверены, что передаваемый тип `T` не может быть ссылкой, поскольку использовать ссылки как типы элементов вектора нельзя. Кроме того, достаточно очевидно, что передача вектора по значению почти всегда дорогостоящая, так как копирующий конструктор `std::vector<>` создает копии всех элементов. По этой причине, вероятно, никогда не следует передавать вектор по значению. Если мы

объявим параметр *v* просто как имеющий тип *T*, то принятие решения о выборе между передачей по значению и по ссылке становится менее очевидным.

Пример `std::make_pair()`

Хорошим примером, демонстрирующим ловушки при принятии решения о механизме передачи параметров, является `std::make_pair<>()`. Это обычный шаблон функции из стандартной библиотеки C++ для создания объектов `std::pair<>` с использованием вывода типа. Его объявление менялось от версии к версии стандарта C++.

- В первом стандарте C++ (C++98) шаблон `make_pair<>()` был объявлен в пространстве имен `std` с использованием передачи по ссылке во избежание излишнего копирования:

```
template<typename T1, typename T2>
pair<T1, T2> make_pair(T1 const& a, T2 const& b)
{
    return pair<T1, T2>(a, b);
}
```

Однако это почти немедленно привело к значительным проблемам при использовании пар строковых литералов или массивов разных размеров.

- В результате, в C++03 определение функции было изменено на вызов по значению:

```
template<typename T1, typename T2>
pair<T1, T2> make_pair(T1 a, T2 b)
{
    return pair<T1, T2>(a, b);
}
```

Как можно прочесть в обосновании этого решения, “*оказалось, что это гораздо меньшее изменение стандарта, чем два других предложения, и все проблемы эффективности были более чем компенсированы преимуществами данного решения*”.

- Однако в C++11 функция `make_pair()` должна поддерживать семантику перемещения, так что ее аргументы должны были стать передаваемыми ссылками. По этой причине определение функции в очередной раз изменено, теперь примерно следующим образом:

```
template<typename T1, typename T2>
constexpr pair<typename decay<T1>::type,
              typename decay<T2>::type>
make_pair(T1&& a, T2&& b)
{
    return pair<typename decay<T1>::type,
                typename decay<T2>::type>(
                    forward<T1>(a),
                    forward<T2>(b));
}
```

Полная реализация более сложная: для поддержки `std::ref()` и `std:: cref()` функция также выполняет разворачивание экземпляров `std::reference_wrapper` в реальные ссылки.

Стандартная библиотека C++ теперь во множестве мест прибегает к аналогичной прямой передаче аргументов, часто в сочетании с использованием `std::decay<>`.

7.7. Резюме

- При тестировании шаблонов следует использовать строки различных длин.
- Для переданных по значению параметров шаблона выполняется низведение типов, в то время как при передаче по ссылке низведение не происходит.
- Свойство типа `std::decay<>` позволяет выполнить низведение параметров шаблонов, переданных по ссылке.
- В некоторых случаях `std::cref()` и `std::ref()` позволяют передать аргументы по ссылке в шаблон функции, в котором они объявлены как передаваемые по значению.
- Передача параметров шаблонов по значению проста, но может не обеспечивать наивысшую производительность.
- Передавайте параметры в шаблоны функций по значению, если только у вас нет веских причин поступать иначе.
- Убедитесь, что возвращаемые значения обычно передаются по значению (что может означать, что параметры шаблона не могут быть указаны непосредственно как возвращаемый тип).
- Всегда, когда важна производительность, измеряйте ее. Не полагайтесь на интуицию — она обычно врет.

Глава 8

Программирование времени компиляции

C++ всегда включал несколько простых способов вычисления значений во время компиляции. Шаблоны значительно расширяют возможности в этой области, а дальнейшая эволюция языка только добавляет новые инструменты в этот набор.

В простом случае можно решить, следует использовать тот или иной код либо выбрать один из нескольких вариантов шаблонного кода. Но компилятор во время компиляции способен даже вычислить итоговый результат всей работы потока управления при условии, что все необходимые входные данные ему доступны.

Фактически C++ имеет несколько вариантов поддержки программирования времени компиляции.

- Еще до C++98 шаблоны предоставляли возможность вычисления во время компиляции, включая применение циклов и выбор пути выполнения. (Однако некоторые рассматривают эти действия как “ злоупотребления” возможностями шаблонов, например потому, что это требует применения неинтуитивного синтаксиса.)
- С помощью частичной специализации можно выполнять выбор времени компиляции между различными реализациями шаблонов классов, в зависимости от конкретных ограничений или требований.
- Принцип SFINAE обеспечивает возможность выбора между различными реализациями шаблона функции для разных типов или разных ограничений.
- В C++11 и C++14 вычисления времени компиляции получили большую поддержку с использованием возможностей `constexpr`, с применением “интуитивного” пути выполнения и, начиная с C++14, с использованием большинства разновидностей инструкций (в том числе цикла `for`, инструкции `switch` и т.п.).
- В C++17 вводится “`if` времени компиляции” для отбрасывания инструкций, зависящих от условий или ограничений времени компиляции. Эта возможность применима даже вне шаблонов.

Все упомянутые возможности рассматриваются в данной главе с особым вниманием к роли и контексту шаблонов.

8.1. Шаблонное метапрограммирование

Шаблоны инстанцируются во время компиляции (в отличие от динамических языков, где обобщенность обрабатывается во время выполнения). Оказывается, что некоторые из возможностей шаблонов C++ могут быть объединены с процессом

инстанцирования для создания своего рода примитивного рекурсивного “языка программирования” в рамках самого языка программирования C++. По этой причине шаблоны могут быть использованы для “вычисления программы”. В главе 23, “Метапрограммирование”, этот вопрос будет рассматриваться существенно подробнее, а пока продемонстрируем на коротком примере, что это действительно возможно.

Приведенный ниже код во время компиляции выясняет, является ли данное число простым.

basics/isprime.hpp

```
template<unsigned p,      // p: проверяемое значение,
         unsigned d>    // d: текущий делитель
struct DoIsPrime
{
    static constexpr bool value = (p % d != 0) &&
                                  DoIsPrime < p, d - 1 >::value;
};

template<unsigned p>      // Завершение рекурсии при делителе 2
struct DoIsPrime<p, 2>
{
    static constexpr bool value = (p%2 != 0);
};

template<unsigned p>      // Шаблон простого числа
struct IsPrime
{
    // Начинаем рекурсию с делителя p/2:
    static constexpr bool value = DoIsPrime<p,p/2>::value;
};

// Частные случаи (для недопустимости бесконечной рекурсии):
template<>
struct IsPrime<0> { static constexpr bool value = false; };
template<>
struct IsPrime<1> { static constexpr bool value = false; };
template<>
struct IsPrime<2> { static constexpr bool value = true;  };
template<>
struct IsPrime<3> { static constexpr bool value = true;  };
```

Шаблон `IsPrime<>` возвращает в члене `value` значение, указывающее, является ли переданный шаблону параметр `p` простым числом. Для этой цели инстанцируется `DoIsPrime<>`, который рекурсивно развертывает выражение проверки для каждого делителя `d` от `p/2` до 2, делит ли этот делитель `p` без остатка.

Например, выражение

`IsPrime<9>::value`

¹ Фактически первым, обнаружившим это, был Эрвин Анрух (Erwin Unruh), который представил программу вычисления простых чисел во время компиляции (см. раздел 23.7).

разворачивается в

```
DoIsPrime<9, 4>::value
```

которое разворачивается в

```
9%4!=0 && DoIsPrime<9, 3>::value
```

которое разворачивается в

```
9%4!=0 && 9%3!=0 && DoIsPrime<9, 2>::value
```

которое разворачивается в

```
9%4!=0 && 9%3!=0 && 9%2!=0
```

вычисление которого дает значение `false`, поскольку $9\%3$ равно 0.

Как показывает эта цепочка инстанцирований:

- мы используем рекурсивное разворачивание `DoIsPrime<>` для выполнения итераций по всем делителям от $p/2$ до 2 для выяснения, нет ли такого делителя, на который исходное число делится нацело (без остатка);
- частичная специализация `DoIsPrime<>` для d , равного 2, служит в качестве критерия завершения рекурсии.

Обратите внимание на то, что все это делается во время компиляции, так что

```
IsPrime<9>::value
```

становится равным `false` во время компиляции.

Возможно, синтаксис шаблонов и неуклюж, но код, подобный показанному, работает, начиная с C++98 (и даже ранее), и оказался достаточно полезным при разработке ряда библиотек².

Более подробно эти вопросы рассматриваются в главе 23, “Метапрограммирование”.

8.2. Вычисления с использованием `constexpr`

В C++11 в язык введена новая функциональная возможность — `constexpr`, которая значительно упрощает различные виды вычислений времени компиляции. В частности, при наличии корректных входных данных `constexpr`-функции могут быть вычислены во время компиляции. В то время как в C++11 на `constexpr`-функции были наложены строгие ограничения (например, каждое определение `constexpr`-функции, по существу, могло состоять только из оператора `return`), большинство этих ограничений были удалены в C++14. Конечно, успешное вычисление `constexpr`-функции по-прежнему требует, чтобы все вычислительные действия были возможны и корректны во время компиляции, что

² До C++11 было принято объявлять члены-значения как константы перечислений, а не статические члены данных, чтобы избежать необходимости определений статических членов данных вне класса (подробнее см. в разделе 23.6). Например:

```
enum { value = (p%d != 0) && DoIsPrime<p, d-1>::value };
```

исключает такие вещи, как, например, динамическое выделение памяти или генерацию исключений.

Наш пример с проверкой числа на простоту в C++11 мог бы быть реализован следующим образом:

basics/isprime11.hpp

```
constexpr bool
doIsPrime(unsigned p, // p: проверяемое значение,
          unsigned d) // d: текущий делитель
{
    return d!=2 ? (p%d!=0) // Проверка текущего
                 && doIsPrime(p,d-1) // и меньшего делителей
                 : (p % 2 != 0); // Завершение рекурсии при 2
}

constexpr bool isPrime(unsigned p)
{
    return p<4 ? !(p<2) // Обработка особых случаев
                : doIsPrime(p,p/2); // Начало рекурсии от p/2
}
```

Из-за ограничения на одну инструкцию мы можем использовать как механизм отбора только условный оператор, и нам все еще нужна рекурсия для итерации по элементам. Но синтаксис обычной C++-функции делает этот код более доступным и понятным, чем первая версия, основанная на инстанцировании шаблона.

Начиная с C++14, `constexpr`-функции могут использовать большинство управляющих структур, доступных в общем коде C++. Так, вместо написания громоздкого кода с шаблонами или загадочного одностороннего кода, теперь можно просто воспользоваться обычным циклом `for`:

basics/isprime14.hpp

```
constexpr bool isPrime(unsigned int p)
{
    for (unsigned int d = 2; d <= p / 2; ++d)
    {
        if (p % d == 0)
        {
            return false; // Найден делитель без остатка
        }
    }
    return p > 1; // Такого делителя нет
}
```

Как в версии C++11, так и в версии C++14 нашей реализации `constexpr isPrime()`, мы можем просто вызвать

`isPrime(9)`

чтобы узнать, является ли простым число 9. Обратите внимание: это может быть сделано во время компиляции, но из этого не следует, что это обязательно будет

сделано во время компиляции. В контексте, который требует значение времени компиляции (например, длина массива или аргумент шаблона, не являющийся типом), компилятор попытается вычислить значение `constexpr`-функции во время компиляции, и, если это не представляется возможным (поскольку в конечном итоге должна быть получена константа), мы получим сообщение об ошибке. В других контекстах компилятор может попытаться выполнить вычисление во время компиляции³, но если это невозможно, то сообщения об ошибке не будет, а вызов функции останется вызовом времени выполнения.

Например, код

```
constexpr bool b1 = isPrime(9); // Вычисляется во время компиляции
```

приведет к вычислению во время компиляции. То же самое верно и для кода

```
const bool b2 = isPrime(9);      // Вычисляется во время компиляции  
                                // для области видимости пространства имен
```

если `b2` определена глобально или в пространстве имен. В области видимости блока компилятор может сам решить, следует выполнять вычисления во время компиляции или во время выполнения⁴. То же самое верно и для кода

```
bool fiftySevenIsPrime()  
{  
    return isPrime(57); // Вычисляется во время компиляции  
}                      // или во время выполнения
```

где компилятор может вычислить значение, возвращаемое `isPrime`, во время компиляции, но может этого и не делать.

С другой стороны, исходный текст

```
int x;  
***  
std::cout << isPrime(x); // Вычисляется во время выполнения
```

будет генерировать код, который будет проверять простоту значения `x` во время выполнения.

8.3. Выбор пути выполнения с помощью частичной специализации

Интересным применением проверок времени компиляции, таких как `isPrime()`, является использование частичной специализации для выбора различных реализаций во время компиляции.

Например, мы можем выбирать ту или иную реализацию в зависимости от того, является ли аргумент шаблона простым числом:

³ На момент написания книги в 2017 году большинство компиляторов пытались выполнять вычисления времени компиляции, даже если это не было строго необходимо.

⁴ Теоретически даже при наличии `constexpr` компилятор может принять решение о вычислении начального значения `b` во время выполнения. Компилятор обязан только проверить, возможно ли такое вычисление во время компиляции.

```

// Вспомогательный шаблон:
template<int SZ, bool = isPrime(SZ)>
struct Helper;

// Реализация для составного SZ:
template<int SZ>
struct Helper<SZ, false>
{
    ...
};

// Реализация для простого SZ:
template<int SZ>
struct Helper<SZ, true>
{
    ...
};

template<typename T, std::size_t SZ>
long foo(std::array<T, SZ> const& coll)
{
    Helper<SZ> h; // Реализация зависит от того, простое
    ...             // ли число элементов в массиве
}

```

Здесь, в зависимости от того, является ли аргумент размера `std::array<...>` простым числом, мы используем две различные реализации класса `Helper<>`. Этот способ применения частичной специализации широко применяется для выбора между различными реализациями шаблона функции в зависимости от свойств аргументов, для которых он вызывается.

Выше мы использовали две частичные специализации для реализации двух возможных альтернатив. Вместо этого можно также использовать первичный шаблон для одной из альтернатив (по умолчанию) и частичные специализации для любого частного случая:

```

// Первичный вспомогательный шаблон:
template<int SZ, bool = isPrime(SZ)>
struct Helper
{
    ...
};

// Специальная реализация для простого SZ:
template<int SZ>
struct Helper<SZ, true>
{
    ...
};

```

Поскольку шаблоны функций не поддерживают частичную специализацию, вы должны использовать другие механизмы для изменения реализации функции на основе определенных ограничений. Наши варианты включают следующее:

- использование классов со статическими функциями;
- использование `std::enable_if` (см. раздел 6.3);

- использование *SFINAE* (об этом будет рассказано далее);
- применение конструкции *if* времени компиляции, доступной начиная с C++17 (см. раздел 8.5).

В главе 20 рассматриваются методы выбора реализации функций, основанные на ограничениях.

8.4. SFINAE

В C++ перегрузка функций для различных типов аргументов — распространенная практика. Когда компилятор видит вызов перегруженной функции, он должен рассмотреть каждого кандидата отдельно, оценивая аргументы вызова и выбирая кандидата, который наилучшим образом соответствует передаваемым аргументам (см. дополнительную информацию в приложении В, “Разрешение перегрузки”).

В тех случаях, когда набор кандидатов для вызова включает шаблоны функций, компилятор сначала должен определить, какие аргументы шаблона должны использоваться для этого кандидата, затем подставить эти аргументы в список параметров функции и возвращаемый тип, а потом оценить, насколько хорошо она соответствует аргументам (так же, как это делается в случае обычной функции). Однако процесс подстановки может столкнуться с проблемами: он может давать конструкции, которые не имеют никакого смысла. Вместо того, чтобы считать такую бессмысленную подстановку ошибкой, правила языка требуют просто игнорировать кандидатов с такими проблемами.

Этот принцип именуется *SFINAE*, что представляет собой аббревиатуру для “*substitution failure is not an error*” (ошибка подстановки ошибкой не является).

Обратите внимание на то, что описанный здесь процесс подстановки отличается от процесса инстанцирования по требованию (см. раздел 2.2): подстановка может быть выполнена даже для потенциальных инстанцирований, которые на самом деле не нужны (так что компилятор может выяснить, действительно ли они не нужны). Эта подстановка выполняется непосредственно в объявлении функции (но не в ее теле).

Рассмотрим следующий пример:

basics/len1.hpp

```
// Количество элементов в массиве:
template<typename T, unsigned N>
std::size_t len(T(&) [N])
{
    return N;
}

// Количество элементов для типа, имеющего член size_type:
template<typename T>
typename T::size_type len(T const& t)
{
    return t.size();
}
```

Здесь мы определяем два шаблона функций `len()`, получающих один обобщенный аргумент⁵.

1. Первый шаблон функции объявляет параметр как `T(&)[N]`, что означает, что параметр должен быть массивом из `N` элементов типа `T`.
2. Шаблон второй функции объявляет параметр просто как `T`, что снимает все ограничения на сам параметр, но имеет возвращаемый тип `T::size_type`, что требует от передаваемого типа аргумента наличия соответствующего члена `size_type`.

При передаче простого массива или строкового литерала соответствие наблюдается только в случае шаблона функции для простых массивов:

```
int a[10];
std::cout << len(a); // OK: соответствует только len() для массива
std::cout << len("Hi"); // OK: соответствует только len() для массива
```

Согласно сигнатуре для второго шаблона функции также наблюдается соответствие при замене `T` соответственно на `int[10]` и `char const[3]`, но эти подстановки ведут к потенциальным ошибкам в возвращаемом типе `T::size_type`. Поэтому второй шаблон при рассмотрении этих вызовов игнорируется.

При передаче `std::vector<>` соответствие наблюдается только для второго шаблона:

```
std::vector<int> v;
std::cout << len(v); // OK: годится только len() с типом size_type
```

При передаче простого указателя не подходит ни один из шаблонов функций (без сообщения об ошибке). В результате компилятор сообщает о том, что не может найти ни одной функции `len()`, соответствующей переданным аргументам:

```
int* p;
std::cout << len(p); // Ошибка: подходящая len() не найдена
```

Обратите внимание: эта ситуация отличается от передачи объекта с типом, имеющим тип-член `size_type`, но не имеющим функции-члена `size()`, как, например, для `std::allocator<>`:

```
std::allocator<int> x;
std::cout << len(x); // Ошибка: len() найдена, но нет члена size()
```

При передаче объекта такого типа компилятор находит второй шаблон функции как соответствующий вызову. Поэтому вместо сообщения об ошибке, гласящего, что не найдена соответствующая функция `len()`, мы получаем сообщение об ошибке времени компиляции, говорящей о том, что невозможен вызов `size()` для `std::allocator<int>`. На этот раз второй шаблон функции не игнорируется.

Игнорирование кандидата при бессмысленности подстановки возвращаемого типа может заставить компилятор выбрать другого кандидата, параметры которого подходят хуже. Например:

⁵ Мы не именуем эту функцию `size()`, поскольку хотим избежать конфликта имен со стандартной библиотекой C++, которая определяет стандартный шаблон функции `std::size()`, начиная с C++17.

basics/len2.hpp

```
// Количество элементов в простом массиве:
template<typename T, unsigned N>
std::size_t len(T (&) [N])
{
    return N;
}

// Количество элементов для типа, имеющего тип-член size_type:
template<typename T>
typename T::size_type len(T const& t)
{
    return t.size();
}

// Запасной вариант для всех прочих типов:
std::size_t len(...)
{
    return 0;
}
```

Здесь мы также предоставляем общую функцию `len()`, которая соответствует всегда, но имеет наихудшее соответствие (с многоточием (...) в разрешении перегрузки (см. раздел B.2).

Итак, для обычных массивов и векторов у нас есть два варианта, причем лучше подходит вариант с конкретными типами. Для указателей годится только запасной вариант, так что компилятор больше не жалуется на отсутствие `len()` для этого вызова⁶. Но для распределителя памяти подходят второй и третий варианты функции, при этом второй шаблон обеспечивает лучшее соответствие. Так что это все еще приводит к сообщению об ошибке о том, что функция-член `size()` не может быть вызвана:

```
int a[10];
std::cout << len(a);      // OK: лучше подходит len() для массива
std::cout << len("Hi");   // OK: лучше подходит len() для массива

std::vector<int> v;
std::cout << len(v);     // OK: лучше подходит len() для типа с
                        // типом-членом size_type

int* p;
std::cout << len(p);     // OK: годится только запасной вариант

std::allocator<int> x;
std::cout << len(x);     // Ошибка: вторая функция len() подходит
                        // лучше, но не может вызвать size() для x
```

Подробнее SFINAE рассматривается в разделе 15.7, а некоторые приложения SFINAE – в разделе 19.4.

⁶ На практике такая запасная функция обычно предоставляет более полезное значение по умолчанию, генерирует исключение или содержит `static_assert` для вывода информативного сообщения об ошибке.

SFINAE и разрешение перегрузки

Со временем принцип SFINAE стал столь важным и настолько превалирующим среди разработчиков шаблонов, что эта аббревиатура стала глаголом. Иногда говорят “мы *SFINAE’им функцию*”, имея в виду применение механизма SFINAE для обеспечения игнорирования шаблонов функции для определенных ограничений; код шаблонов при этих ограничениях становится недопустимым. Всякий раз, когда вы читаете в стандарте C++, что “*шаблон функции не должен участвовать в разрешении перегрузки, если...*”, это означает, что для того, чтобы шаблон функции игнорировался для определенных случаев, используется SFINAE.

Например, `std::thread` объявляет конструктор:

```
namespace std
{
    class thread
    {
        public:
            /**
             * template<typename F, typename... Args>
             *     explicit thread(F&& f, Args&& ... args);
            */
    };
}
```

со следующим примечанием:

Примечание: этот конструктор не должен участвовать в разрешении перегрузки, если `decay_t<F>` представляет собой тот же тип, что и `std::thread`.

Это означает, что шаблонный конструктор игнорируется, если он вызывается с `std::thread` в качестве первого и единственного аргумента. Причина в том, что в противном случае шаблон-член наподобие показанного иногда может демонстрировать лучшее соответствие, чем любой предопределенный копирующий или перемещающий конструктор (см. подробности в разделах 6.2 и 16.2.4). Применяя принцип SFINAE для игнорирования шаблона конструктора при вызове для потока, мы гарантируем, что, если поток строится из другого потока, всегда используется предопределенный копирующий или перемещающий конструктор⁷.

Применение такой методики на индивидуальной основе может оказаться громоздким. К счастью, стандартная библиотека предоставляет инструменты для более легкого отключения шаблонов. Наиболее известным средством является шаблон `std::enable_if<>`, который был представлен в разделе 6.3. Он позволяет отключить шаблон, просто заменив тип конструкцией с условием отключения шаблона.

В результате реальное объявление `std::thread` обычно имеет следующий вид:

```
namespace std
{
```

⁷ Поскольку копирующий конструктор для класса `thread` удален, это также гарантирует, что копирование `thread` оказывается запрещенным.

```

class thread
{
public:
    ...
    template<typename F, typename... Args,
              typename = enable_if_t<!is_same_v<
                decay_t<F>, thread>>>
        explicit thread(F&& f, Args&&... args);
    ...
};

}

```

Подробности реализации `std::enable_if` и описание применения частичной специализации и SFINAE приведены в разделе 20.3.

8.4.1. Выражение SFINAE с `decltype`

Не всегда легко найти и сформулировать правильное выражение для того, чтобы воспользоваться принципом SFINAE для обеспечения игнорирования шаблона функции при определенных условиях.

Предположим, что мы хотим гарантировать, что шаблон функции `len()` игнорируется для аргументов типа, который имеет тип-член `size_type`, но не имеет функции-члена `size()`. Без каких-либо требований к функции-члену `size()` в объявлении функции шаблон будет выбран, после чего его институцирование приведет к ошибке:

```

template<typename T>
typename T::size_type len(T const& t)
{
    return t.size();
}

std::allocator<int> x;
std::cout << len(x) // Ошибка: выбрана len(), но x
      << '\n'; // не имеет функции-члена size()

```

Вот как выглядят обычные действия в такой ситуации.

- Указываем возвращаемый тип с помощью *синтаксиса завершающего возвращаемого типа* (trailing return type syntax), с использованием `auto` в начале объявления и `->` перед возвращаемым типом в конце.
- Определяем возвращаемый тип с использованием `decltype` и оператора “запятая”.
- Формулируем все выражения, которые должны быть корректны, в начале оператора “запятая” (преобразуя в `void`, если оператор “запятая” перегружен).
- Определяем объект реального возвращаемого типа в конце оператора запятой.

Например:

```
template<typename T>
auto len(T const& t) -> decltype((void)(t.size()), T::size_type())
```

```
{
    return t.size();
}
```

Здесь возвращаемый тип задается выражением

```
decltype((void)(t.size()), T::size_type())
```

Операнд конструкции `decltype` представляет собой список выражений, разделенных запятыми, так что последнее выражение `T::size_type()` дает значение желаемого типа возвращаемого значения (который конструкция `decltype` использует для преобразования в возвращаемый тип). Перед (последней) запятой находится выражение, которые должно быть корректным (и которое в данном случае представляет собой просто `t.size()`). Приведение выражения к `void` используется для того, чтобы избежать возможной перегрузки оператора запятой для типа данного выражения.

Обратите внимание на то, что аргумент `decltype` является *невычисляемым операндом*, следовательно, например, можно создать “фиктивные объекты” без вызова конструкторов (см. раздел 11.2.3).

8.5. Инструкция `if` времени компиляции

Частичная специализация, SFINAE и `std::enable_if` позволяют включать или отключать шаблоны в целом. C++17 дополнительно вводит инструкцию `if` времени компиляции, которая позволяет включать или отключать конкретные инструкции на основе условий времени компиляции. Используя синтаксис `if constexpr(...)`, компилятор использует выражения времени компиляции для принятия решения о применении части *then* или части *else* (если таковая имеется).

В качестве первого примера рассмотрим вариативный шаблон функции `print()`, представленный в разделе 4.1.1. Он выводит свои аргументы (произвольных типов) с использованием рекурсии. Вместо предоставления отдельной функции для прекращения рекурсии `if constexpr` позволяет нам решать, продолжать ли рекурсию, локально⁸:

```
template<typename T, typename... Types>
void print(T const& firstArg, Types const& ... args)
{
    std::cout << firstArg << '\n';

    if constexpr(sizeof...(args) > 0)
    {
        print(args...); // Код доступен только при условии
                      // sizeof... (args)>0 (начиная с C++17)
    }
}
```

⁸ Несмотря на то что в коде данная инструкция выглядит как `if constexpr`, в литературе (по историческим причинам и поскольку ее можно рассматривать как `constexpr`-версию `if`), ее часто упоминают как `constexpr if`.

Здесь, если `print()` вызывается только с одним аргументом, `args` становится пустым пакетом параметров, так что `sizeof...(args)` становится равным 0. В результате такой рекурсивный вызов `print()` становится завершающим рекурсию, так как код вызова в нем не создается, и рекурсия заканчивается.

Тот факт, что код не инстанцируется, означает, что выполняется только первый этап трансляции (*время определения*), во время которого проверяется правильность синтаксиса и имен, которые не зависят от параметров шаблона (см. раздел 1.1.3). Например:

```
template<typename T>
void foo(T t)
{
    if constexpr(std::is_integral_v<T>)
    {
        if (t > 0)
        {
            foo(t - 1); // OK
        }
    }
    else
    {
        undeclared(t); // Ошибка, если не объявлена и не отброшена
                        // (T не является целочисленным)
        undeclared(); // Ошибка, если не объявлена
                      // (даже если отброшена)
        static_assert(false, "no integral"); // Всегда работает
                                            // (даже если отброшена)
        static_assert(!std::is_integral_v<T>, "no integral"); // OK
    }
}
```

Обратите внимание: `if constexpr` может использоваться в любой функции, не только в шаблонах. Требуется только выражение времени компиляции, возвращающее логическое значение. Например:

```
int main()
{
    if constexpr(std::numeric_limits<char>::is_signed)
    {
        foo(42); // OK
    }
    else
    {
        undeclared(42); // Ошибка, если undeclared() не объявлена
        static_assert(false, "unsigned"); // Всегда работает
                                         // (даже если отброшена)
        static_assert(!std::numeric_limits<char>::is_signed,
                     "char is unsigned"); // OK
    }
}
```

С помощью этой возможности допустимо, например, использовать нашу функцию времени компиляции `isPrime()`, представленную в разделе 8.2, для выполнения дополнительного кода, если заданный размера не является простым числом:

```

template<typename T, std::size_t SZ>
void foo(std::array<T, SZ> const& coll)
{
    if constexpr(!isPrime(SZ))
    {
        ... // Специальная дополнительная обработка,
        // если размер массива - составное число
    }

    ...
}

```

Подробности см. в разделе 14.6.

8.6. Резюме

- Шаблоны обеспечивают возможность вычисления во время компиляции (с использованием рекурсии для итерации и частичной специализации или оператора ?: для выбора).
- `constexpr`-функции позволяют заменить большинство вычислений времени компиляции “обычными функциями”, вызываемыми в контексте времени компиляции.
- Частичная специализация позволяет осуществлять выбор между различными реализациями шаблонов класса на основе некоторых ограничений времени компиляции.
- Шаблоны используются только при необходимости и когда подстановка в объявление шаблона функции не приводит к некорректному коду. Этот принцип называется SFINAЕ (substitution failure is not an error — ошибка подстановки ошибкой не является).
- SFINAЕ может использоваться для предоставления шаблонов функций только для определенных типов и/или ограничений.
- Начиная с C++17, `if` времени компиляции позволяет включать или отбрасывать инструкции в соответствии с условиями времени компиляции (даже вне шаблонов).

Глава 9

Применение шаблонов на практике

Код шаблона немного отличается от обычного кода. В некотором смысле шаблоны лежат где-то между макросами и обычными, нешаблонными объявлениями. Хотя это может быть и чрезмерным упрощением, оно имеет следствия не только для способа написания алгоритмов и структур данных с использованием шаблонов, но и для повседневной логистики выражения и анализа программ с применением шаблонов.

В этой главе мы рассмотрим некоторые практические вопросы без углубления в технические детали, лежащие в их основе. Многие из этих деталей рассматриваются в главе 14, “Инстанцирование”. Чтобы упростить обсуждение, мы предполагаем, что наши системы компиляции C++ состоят из традиционных компиляторов и компоновщиков (системы, не попадающие в эту категорию, крайне редки).

9.1. Модель включения

Существует несколько способов организации исходных текстов шаблонов. В этом разделе представлен наиболее популярный подход: модель включения.

9.1.1. Ошибки компоновщика

Большинство программистов на C и C++ организуют свой нешаблонный код в основном следующим образом.

- Классы и прочие объявления типов полностью размещаются в *заголовочных файлах*. Обычно эти файлы имеют расширение .hpp (или .h, .h, .hh, ..hxx).
- Что касается глобальных (невстраиваемых) переменных и (невстраиваемых) функций, в заголовочный файл помещаются только объявления, а определения размещаются в файлах, компилируемых как отдельные единицы трансляции. Такие *CPP-файлы* обычно имеют расширение .cpp (или .c, .c, .cc, .cxx).

Этот способ вполне работоспособен: он делает определение необходимого типа легко доступным во всей программе и позволяет избежать ошибок компоновки, связанных с дублированием определений переменных и функций.

С учетом данных соглашений распространенная ошибка, на которую часто жалуются начинающие программисты, может быть проиллюстрирована следующей (неверной) небольшой программой. Как и для “обычного кода”, шаблон объявлен в заголовочном файле:

basics/myfirst.hpp

```
#ifndef MYFIRST_HPP
#define MYFIRST_HPP

// Объявление шаблона
template<typename T>
void printTypeof(T const&);

#endif // MYFIRST_HPP
```

`printTypeof()` представляет собой объявление простой вспомогательной функции, которая выводит некоторую информацию о типе. Реализация этой функции помещается в CPP-файл:

basics/myfirst.cpp

```
#include <iostream>
#include <typeinfo>
#include "myfirst.hpp"

// Реализация/определение шаблона
template<typename T>
void printTypeof(T const& x)
{
    std::cout << typeid(x).name() << '\n';
}
```

В этом примере использован оператор `typeid`, позволяющий вывести строку, описывающую тип переданного ему выражения. Он возвращает `l`-значение статического типа `std::type_info`, который предоставляет функцию-член `name()`, возвращающую строковое представление типа. На самом деле стандарт C++ не требует от `name()` возврата чего-то значимого, но в хороших реализациях C++ вы должны получить строку, которая представляет собой хорошее описание типа выражения, переданного `typeid`!

Наконец, мы используем шаблон в другом CPP-файле, в который объявление шаблона включено с использованием директивы препроцессора `#include`:

basics/myfirstmain.cpp

```
#include "myfirst.hpp"

// Использование шаблона
int main()
{
    double ice = 3.0;
    printTypeof(ice); // Вызов шаблона функции для double
}
```

¹ В некоторых реализациях эта строка *декорирована* (закодирована с использованием типов аргументов и имен охватывающих областей видимости для отличия от других имен), но в таких реализациях для преобразования ее в удобочитаемый текст имеются соответствующие утилиты.

Компилятор C++, вероятно, примет эту программу без каких-либо проблем, но компоновщик, скорее всего, сообщит об ошибке отсутствия определения функции `printTypeof()`.

Причиной этой ошибки является то, что определение шаблона функции `printTypeof()` не инстанцировано. Чтобы шаблон был инстанцирован, компилятор должен знать, какие определения должны быть инстанцированы и для каких именно аргументов шаблона. К сожалению, в предыдущем примере эти две части информации находятся в файлах, компилируемых по отдельности. Таким образом, когда наш компилятор видит вызов `printTypeof()`, но не имеет определения шаблона, чтобы инстанцировать его для `double`, он просто предполагает, что такое определение представлено в другом месте, и создает для этого определения соответствующую ссылку (разрешаемую компоновщиком). С другой стороны, когда компилятор обрабатывает файл `myfirst.cpp`, он не имеет никаких указаний о том, что он должен инстанцировать определение содержащегося в нем шаблона для некоторых конкретных аргументов.

9.1.2. Шаблоны в заголовочных файлах

Обычное решение описанной проблемы заключается в использовании того же подхода, что и для макросов или встраиваемых функций: мы включаем определения шаблона в заголовочный файл, объявляющий этот шаблон.

Таким образом, вместо предоставления файла `myfirst.cpp`, мы переписываем `myfirst.hpp` так, чтобы он содержал как объявление, так и определение шаблона.

basics/myfirst2.hpp

```
#ifndef MYFIRST_HPP
#define MYFIRST_HPP
#include <iostream>
#include <typeinfo>

// Объявление шаблона
template<typename T>
void printTypeof(T const&);

// Реализация/определение шаблона
template<typename T>
void printTypeof(T const& x)
{
    std::cout << typeid(x).name() << '\n';
}
#endif // MYFIRST_HPP
```

Этот способ организации шаблонов называется *моделью включения* (*inclusion model*). При ее использовании наша программа теперь корректно компилируется, компонуется и выполняется.

Следует сделать несколько замечаний. Наиболее важным является то, что этот подход значительно увеличивает стоимость включения заголовочного

файла `myfirst.hpp`. В нашем примере стоимость является не результатом размера определения самого шаблона, а результатом того факта, что мы должны также включать заголовочные файлы, используемые в определении нашего шаблона — в данном случае `<iostream>` и `<typeinfo>`. Вы можете обнаружить, что это приводит к включению десятков тысяч строк кода, потому что такие заголовочные файлы, как `<iostream>`, содержат множество собственных определений шаблонов.

На практике это становится реальной проблемой, потому что значительно увеличивает время, необходимое компилятору для компиляции больших программ. Поэтому мы рассмотрим некоторые возможные пути решения этой проблемы, включая предкомпилированные заголовочные файлы (см. раздел 9.3) и использование явного инстанцирования шаблонов (см. раздел 14.5).

Несмотря на рассматриваемую проблему времени построения, мы рекомендуем по возможности использовать модель включения для организации ваших шаблонов, пока не будет доступен более совершенный механизм. На момент написания этой книги в 2017 году такой механизм находится в работе: это *модули*, которые представлены в разделе 17.11. Они представляют собой механизм языка, который позволяет программисту более логично организовывать код — так, чтобы компилятор мог отдельно компилировать все объявления, а затем эффективно и выборочно импортировать обработанные объявления всякий раз, когда они необходимы.

Еще одно (более тонкое) замечание о модели включения заключается в том, что невстраиваемые шаблоны функций отличаются от встраиваемых функций и макросов важным моментом: они не встраиваются в точке вызова. Вместо этого при их создании генерируется новая копия функции. Поскольку это автоматический процесс, компилятор может в конечном итоге создать две копии функции в двух разных файлах, а некоторые компоновщики могут сообщать об ошибках, встречая два различных определения одной и той же функции. Теоретически это не должно быть нашей проблемой: это проблема для системы компиляции C++. На практике все хорошо работает большую часть времени, и нам не приходится сталкиваться с этой проблемой. Однако для больших проектов, которые создают свои собственные библиотеки кода, эти проблемы иногда возникают. Обсуждение схем инстанцирования в главе 14, “Инстанцирование”, и тщательное изучение документации, поставляемой с компилятором C++, должно помочь решить эти проблемы.

Наконец, мы должны указать на то, что вопросы, относящиеся в нашем примере к шаблонам обычных функций, применимы и к функциям-членам и статическим членам-данным шаблонов классов, а также к шаблонам функций-членов.

9.2. Шаблоны и `inline`

Объявление функций встраиваемыми является распространенным способом улучшить время работы программы. Спецификатор `inline` задумывался как подсказка реализации о том, что встраивание тела функции в точку вызова предпочтительнее, чем механизм вызова обычной функции.

Однако реализация может игнорировать эту подсказку. Следовательно, единственным гарантированным эффектом от `inline` является возможность много-кратного определения функции в программе (обычно из-за того, что оно находится в заголовочном файле, который включается в нескольких местах).

Как и встраиваемые функции, шаблоны функций могут быть определены в нескольких единицах трансляции. Обычно это достигается путем размещения определения в заголовочном файле, который включается в несколько СРР-файлов.

Это, однако, не означает, что шаблоны функции используют встраивание по умолчанию. Это полностью зависит от компилятора и того, предпочтительнее ли встраивание тела шаблона функции в точку вызова, чем механизм вызова обычной функции. Возможно, это покажется удивительным, но компиляторы куда лучше, чем программисты, оценивают возможность встраивания, и выясняют, приведет оно к повышению производительности или нет. В результате точная стратегия компилятора в отношении к `inline` зависит от компилятора и даже от параметров компилятора, выбранных для конкретной компиляции.

Тем не менее, используя соответствующие средства наблюдения за производительностью программы, программист может иметь более полную информацию, чем компилятор, и поэтому может пожелать переопределить решение компилятора (например, при настройке программного обеспечения для конкретных платформ, таких как мобильные телефоны, или для особых входных данных). Иногда это возможно только с помощью специальных нестандартных атрибутов компилятора, например `noinline` или `always_inline`.

Стоит отметить, что полные специализации шаблонов функций в этом отношении действуют как обычные функции: их определение может появляться лишь однократно, если только они не определены как `inline` (см. раздел 16.3). Более подробный обзор этой темы представлен в приложении А, “Правило одного определения”.

9.3. Предкомпилированные заголовочные файлы

Даже без шаблонов заголовочные файлы C++ могут стать очень большими и требовать много времени для обработки при компиляции. Шаблоны вносят свой вклад в эту тенденцию, и возмущения программистов заставили поставщиков компиляторов реализовать схему, обычно известную как *предкомпилированные заголовочные файлы* (precompiled headers – РЧН). Эта схема работает вне области применения стандарта и основана на специфике конкретного производителя. Подробности о том, как создавать и использовать файлы предкомпилированных заголовочных файлов, вы найдете в документации по соответствующей системе компиляции C++, а здесь вы получите только некоторое общее представление о том, как работает этот механизм.

Когда компилятор транслирует файл, он начинает с самого начала файла и читает его до конца. По мере обработки каждого токена в файле (которые могут находиться во включаемых с помощью директивы `#include` файлах) компилятор

изменяет свое внутреннее состояние, включая выполнение таких действий, как добавление записей в таблицу символов, в которой позже может выполняться поиск. При этом компилятор может также генерировать код в объектных файлах.

Схема предкомпилированных заголовочных файлов опирается на тот факт, что код может быть организован таким образом, что многие файлы начинаются с одних и тех же строк кода. Давайте предположим, что каждый компилируемый файл начинается с одних и тех же N строк кода. Мы могли бы скомпилировать эти N строк и сохранить текущее состояние компилятора в предкомпилиированном заголовочном файле. Затем для каждого файла в нашей программе можно просто загрузить сохраненное состояние и начать компиляцию с $N+1$ -й строки. Здесь следует заметить, что загрузка сохраненного состояния является операцией, которая может быть на порядки быстрее, чем реальная компиляция первых N строк. Однако при первом выполнении сохранение состояния обычно дороже, чем простая компиляция N строк. Увеличение стоимости колеблется примерно от 20 до 200 процентов.

Ключом к эффективному использованию предкомпилированных заголовочных файлов является обеспечение как можно большего количества одинаковых строк кода в начале каждого файла. На практике это означает, что файлы должны начинаться с одних и тех же директив `#include` в одном и том же порядке, которые (как упоминалось ранее) потребляют значительную часть времени построения программы. Следовательно, может оказаться очень выгодным обратить внимание на порядок, в котором включаются заголовочные файлы. Например, следующие два файла:

```
#include <iostream>
#include <vector>
#include <list>
...
и
#include <list>
#include <vector>
...*
```

мешают использованию предкомпилированных заголовочных файлов, потому что у них нет общего начального состояния.

Некоторые программисты считают, что лучше включить в файл некоторые дополнительные ненужные заголовочные файлы, но обеспечить возможность ускорить трансляцию за счет использования предкомпилиированного заголовочного файла. Такое решение может существенно облегчить управление стратегией включения. Например, обычно относительно просто создать заголовочный файл с именем `std.hpp`, который включает в себя все стандартные заголовки²:

```
#include <iostream>
#include <string>
#include <vector>
```

² Теоретически стандартные заголовочные файлы не обязаны соответствовать физическим файлам. Однако на практике это всегда физические файлы, причем эти файлы весьма большого размера.

```
#include <deque>
#include <list>
***
```

Затем этот файл может быть предкомпилирован, и каждый файл программы, использующий стандартную библиотеку, просто начинается следующим образом:

```
#include "std.hpp"
```

Обычно это требует некоторого времени для предкомпиляции, но в системе с достаточным объемом памяти схема с предкомпилированным заголовочным файлом позволяет работать значительно быстрее, чем требуется почти любому стандартному заголовочному файлу без предварительной компиляции. Стандартные заголовочные файлы особенно удобно использовать таким образом, потому что они редко изменяются, а следовательно, предкомпилированный заголовочный файл для файла `std.hpp` может быть построен один раз. В противном случае предкомпилированные заголовочные файлы обычно являются частью конфигурации зависимостей проекта (например, обновляются по мере необходимости популярной программой `make` или соответствующим инструментарием интегрированной среды разработки (IDE)).

Одним из привлекательных подходов к управлению предкомпилированными заголовочными файлами является создание *слоев* предкомпилированных заголовочных файлов, которые образуются из наиболее широко используемых и стабильных заголовочных файлов (как, например, наш заголовочный файл `std.hpp`), для которых не ожидается внесение изменений, а потому имеет смысл их предварительная компиляция. Однако, если заголовочные файлы находятся в постоянном развитии, их предкомпиляция может потребовать больше времени, чем даст экономия при повторном их использовании. Ключевой концепцией этого подхода является то, что предкомпилированные заголовочные файлы более стабильного слоя могут быть повторно использованы для улучшения времени компиляции менее стабильных заголовочных файлов. Предположим, например, что в дополнение к нашему заголовочному файлу `std.hpp` (который мы предварительно скомпилировали), мы также определили заголовочный файл `core.hpp`, который включает дополнительные возможности, специфичные для нашего проекта, но тем не менее достигшие определенного уровня стабильности:

```
#include "std.hpp"
#include "core_data.hpp"
#include "core_algos.hpp"
***
```

Поскольку этот файл начинается с `#include "std.hpp"`, компилятор может загрузить соответствующий предкомпилированный заголовок и продолжить работу со следующими строками без перекомпиляции всех стандартных заголовков. Когда файл будет полностью обработан, может быть создан новый предкомпилированный заголовочный файл. Затем приложения могут использовать `#include "core.hpp"` для обеспечения быстрого доступа к большому количеству функциональности, потому что компилятор может загрузить последний предкомпилированный заголовочный файл.

9.4. Расшифровка романов об ошибках

Обычные ошибки компиляции, как правило, довольно кратки и говорят по существу. Например, когда компилятор говорит, что "класс X не имеет члена 'fun'", обычно не слишком трудно понять, что за неприятность случилась в нашем коде (например, мы могли ошибиться и набрать `fun` вместо `run`). Но шаблоны — совсем другое дело. Давайте рассмотрим несколько примеров.

Простое несоответствие типов

Рассмотрим следующий относительно простой пример с использованием стандартной библиотеки C++:

`basics/errornovell.cpp`

```
#include <string>
#include <map>
#include <algorithm>

int main()
{
    std::map<std::string, double> coll;
    ...
    // Поиск первой непустой строки в coll:
    auto pos = std::find_if(coll.begin(), coll.end(),
                           [] (std::string const & s)
    {
        return s != "";
    });
}
```

В нем содержится небольшая ошибка: в лямбда-выражении, используемом для поиска первой совпадающей строки в коллекции, мы выполняем сравнение с заданной строкой. Однако элементами в отображении являются пары ключ/значение, так что здесь ожидается тип `std::pair<std::string const, double>`.

Одна из версий популярного компилятора GNU C++ сообщает об ошибке следующим образом:

```
1 In file included from /cygdrive/p/gcc/gcc61/include/bits/
                  stl_algobase.h:71:0,
2                     from /cygdrive/p/gcc/gcc61/include/bits/
                  char_traits.h:39,
3                     from /cygdrive/p/gcc/gcc61/include/string:40,
4                     from errornovell.cpp:1:
5 /cygdrive/p/gcc/gcc61/include/bits/predefined_ops.h:
In instantiation of 'bool __gnu_cxx::__ops::__Iter_pred
<_Predicate>::operator()(_Iterator) [with _Iterator =
std::__Rb_tree_iterator<std::pair<const std::__cxx11::basic_
string<char>, double> >; _Predicate = main()
::<lambda(const string&)>]':
6 /cygdrive/p/gcc/gcc61/include/bits/stl_algo.h:104:42: required
from '__InputIterator std::__find_if(__InputIterator,
__InputIterator, _Predicate, std::__input_iterator_tag)
[with __InputIterator = std::__Rb_tree_iterator<std::pair<const
```

```

    std::__cxx11::basic_string<char>, double> >; _Predicate =
    __gnu_cxx::__ops::__Iter_pred<main()::<lambda(const string&)> >]'  

7 /cygdrive/p/gcc/gcc61-include/bits/stl_algo.h:161:23: required
   from '__Iterator std::__find_if(__Iterator, __Iterator, __Predicate)
[with __Iterator = std::Rb_tree_iterator<std::pair<const std::__cxx11::basic_string<char>, double> >; __Predicate = __gnu_cxx::__ops::__Iter_pred<main()::<lambda(const string&)> >]'  

8 /cygdrive/p/gcc/gcc61-include/bits/stl_algo.h:3824:28: required
   from '__IIter std::find_if(__IIter, __IIter, __Predicate) [with
   __IIter = std::Rb_tree_iterator<std::pair<const std::__cxx11::basic_string<char>, double> >; __Predicate = main()::<lambda(const string&)>]'  

9 errornovel1.cpp:13:29: required from here  

10 /cygdrive/p/gcc/gcc61-include/bits/predefined_ops.h:234:11:
   error: no match for call to '(main()::<lambda(const string&)>)
   (std::pair<const std::__cxx11::basic_string<char>, double>&)'  

11     { return bool(_M_pred(*__it)); }  

12     ^~~~~~  

13 /cygdrive/p/gcc/gcc61-include/bits/predefined_ops.h:234:11:
   note: candidate: bool (*) (const string&) {aka bool (*) (const std::__cxx11::basic_string<char>&)} <conversion>  

14 /cygdrive/p/gcc/gcc61-include/bits/predefined_ops.h:234:11:
   note: candidate expects 2 arguments, 2 provided  

15 errornovel1.cpp:11:52: note: candidate: main()::<lambda(
   const string&)>
16                               [] (std::string const& s) {  

17  

18 errornovel1.cpp:11:52: note: no known conversion for argument
   1 from 'std::pair<const std::__cxx11::basic_string<char>,
   double>' to 'const string& {aka const std::__cxx11::basic_string<char>&}'

```

Такое сообщение больше напоминает неудачный роман, чем диагностику. Его размеры и непонятность подавляют начинающих пользователей шаблонов, отталкивая их от этой возможности языка программирования. Однако при некоторой практике подобные сообщения становятся воспринимаемыми, и ошибки находятся легко — по крайней мере, относительно легко.

Первая часть этого сообщения об ошибке говорит, что ошибка произошла в экземпляре шаблона функции глубоко внутри заголовочного файла `predefined_ops.h`, включаемого в `errornovel1.cpp` через различные другие заголовочные файлы. Здесь и в следующих строках компилятор сообщает, что именно было инстанцировано и с какими аргументами. В данном случае все началось с инструкции, заканчивающейся в строке 13 файла `errornovel1.cpp`, которая представляет собой следующее:

```

auto pos = std::find_if(coll.begin(), coll.end(),
                        [] (std::string const& s)
{
    return s != "";
});

```

Это приводит к инстанцированию шаблона `find_if` в строке 115 заголовочного файла `stl_algo.h`, где код

```
_IIter std::find_if(_IIter, _IIter, __Predicate)
```

инстанцируется с

```
_Iter = std::_Rb_tree_iterator<std::pair<const
                           std::basic_string<char>, double>>
_Predicate = main()::<lambda(const string&)>
```

Компилятор сообщает всю эту цепочку инстанцирований, что позволяет нам определить цепь событий, которые привели к данному инстанцированию, вызвавшему ошибку. Зачастую программист просто не ожидает такой длинной цепочки инстанцирований.

Однако в нашем примере мы готовы поверить, что все эти виды шаблонов должны быть созданы, и нас просто интересует, почему же он не работает. Эта информация находится в последней части сообщения. Часть, в которой говорится “no match for call”, подразумевает, что вызов функции не может быть разрешен, потому что типы аргументов и типы параметров не совпадают. В ней указано, что именно вызывается

```
(main()::<lambda(const string&)>)(
    std::pair<const std::basic_string<char>,
              double>&)
```

и код, который приводит к этому вызову:

```
{ return bool(_M_pred(*_it)); }
```

Кроме того, сразу после этого строки, содержащая текст “note: candidate:”, поясняет, что единственный тип-кандидат ожидает `const string&`, что этот кандидат определен в строке 11 файла `errornovell.cpp` как лямбда-выражение `[] (std::string const& s)`, и почему этот возможный кандидат не годится:

```
no known conversion for argument 1
from 'std::pair<const std::basic_string<char>, double>'  
to 'const string& {aka const std::basic_string<char>&}'
```

описывая имеющуюся у нас проблему невозможности преобразования типов один в другой.

Несомненно, что сообщение об ошибке могло бы быть получше. Реальную проблему стоило бы выводить до истории инстанцирований, а вместо того чтобы использовать для экземпляра полное расширенное имя шаблона, как, например, `std::basic_string<char>`, было бы достаточно использовать только `std::string`. Однако также верно и то, что вся эта диагностическая информация в ряде ситуаций может быть очень полезной. Поэтому не удивительно, что другие компиляторы в основном предоставляют аналогичную информацию. Так, вывод Visual C++ выглядит примерно следующим образом (показан вывод русской локализации Visual C++ 2015):

```
1 C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\INCLUDE\algorithm(87): error C2664: "bool main::<lambda_8df7a0afc6b1183  
ce0c8e081d1832054>::operator ()(const std::string &) const":  
невозможно преобразовать аргумент 1 из "std::pair<const _Kty,  
_Ty>" в "const std::string &"  
2           with  
3           [
```

```

4           _Kty=std::string,
5           _Ty=double
6       ]
7 C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\INCLUDE\
algorithm(87): note: Причина: невозможно преобразовать "std::
pair<const _Kty,_Ty>" в "const std::string"
8       with
9       [
10          _Kty=std::string,
11          _Ty=double
12      ]
13 C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\INCLUDE\
algorithm(87): note: Для выполнения данного преобразования нет
доступного оператора преобразования, определенного пользователем,
или вызов оператора невозможен
14 C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\INCLUDE\
algorithm(98): note: см. ссылку на создание экземпляров функции
шаблон при компиляции "_Init std::find_if<std::
(Tree_unchecked_iterator<_Mytree>,_Pr>(_Init,_Init,_Pr &)"
15       with
16       [
17          _Init=std::Tree_unchecked_iterator<std::
              _Tree_val<std::Tree_simple_types<std::pair<
                  const std::string,double>>>,
18          _Mytree=std::Tree_val<std::Tree_simple_types<
              std::pair<const std::string,double>>>,
19          _Pr=main::<lambda_8df7a0afc6b1183ce0c8e081d1832054>
20      ]
21 errorNovell111.cpp(14): note: см. ссылку на создание экземпляров
функции шаблон при компиляции "_Init std::find_if<std::
(Tree_iterator<std::Tree_val<std::Tree_simple_types<std::
pair<const _Kty,_Ty>>>,>,main::
<lambda_8df7a0afc6b1183ce0c8e081d1832054>>(_Init,_Init,_Pr)"
22       with
23       [
24          _Init=std::Tree_iterator<std::Tree_val<
              std::Tree_simple_types<std::pair<const
                  std::string,double>>>,
25          _Kty=std::string,
26          _Ty=double,
27          _Pr=main::<lambda_8df7a0afc6b1183ce0c8e081d1832054>
28      }

```

Здесь вновь предоставлена цепочка инстанцирования с информацией, которая говорит нам о том, что именно было инстанцировано, с какими аргументами и в каком месте кода, и мы видим, что

```

невозможно преобразовать "std::pair<const _Kty,_Ty>"  

                                в "const std::string"  

       with  

       [  

          _Kty=std::string,  

          _Ty=double  

      ]

```

Отсутствие `const` у некоторых компиляторов

К сожалению, иногда случается, что обобщенный код представляет собой проблему только для некоторых компиляторов. Рассмотрим следующий пример:

basics/errornovel2.cpp

```
#include <string>
#include <unordered_set>
class Customer
{
private:
    std::string name;
public:
    Customer(std::string const& n)
        : name(n)
    {
    }
    std::string getName() const
    {
        return name;
    }
};
int main()
{
    // Пользовательская хеш-функция:
    struct MyCustomerHash
    {
        // Примечание: отсутствие const является ошибкой только
        // для g++ и clang:
        std::size_t operator()(Customer const& c)
        {
            return std::hash<std::string>()(c.getName());
        }
    };
    // Используем пользовательскую хеш-функцию для хеш-таблицы
    // с элементами типа Customer:
    std::unordered_set<Customer, MyCustomerHash> coll;
    ***
}
```

При использовании Visual Studio 2013 или 2015 этот код компилируется, как и ожидалось. Однако в случае g++ или clang код приводит к сообщениям об ошибке. Например, в g++ 6.1 первое сообщение об ошибке имеет следующий вид:

```
1 In file included from /cygdrive/p/gcc/gcc61/include/
   bits/hashtable.h:35:0,
2           from /cygdrive/p/gcc/gcc61/include/
   unordered_set:47,
3           from errornovel2.cpp:2:
4 /cygdrive/p/gcc/gcc61/include/bits/hashtable_policy.h:
 In instantiation of 'struct std::__detail::__is_noexcept_hash
<Customer, main()::MyCustomerHash>':
5 /cygdrive/p/gcc/gcc61/include/type_traits:143:12: required from
 'struct std::__and_<std::__is_fast_hash<main()::MyCustomerHash>,
 std::__detail::__is_noexcept_hash<Customer,
```

```

main()::MyCustomerHash> >
6 /cygdrive/p/gcc/gcc61-include/type_traits:154:38: required from
'struct std::__not<std::__and<std::__is_fast_hash<main()>::
MyCustomerHash>, std::__detail::__is_noexcept_hash<Customer,
main()::MyCustomerHash> > >'  

7 /cygdrive/p/gcc/gcc61-include/bits/unordered_set.h:95:63:
required from 'class std::unordered_set<Customer, main()::
MyCustomerHash>'  

8 error novel2.cpp:28:47: required from here  

9 /cygdrive/p/gcc/gcc61-include/bits/hashtable_policy.h:85:34:
error: no match for call to '(const main()::MyCustomerHash)
(const Customer&)'  

10    noexcept(declval<const _Hash&>())(declval<const _Key&>())>  

11    ~~~~~^~~~~~  

12 error novel2.cpp:22:17: note: candidate: std::size_t
main()::MyCustomerHash::operator()(const Customer&)
<near match>  

13    std::size_t operator()(const Customer& c) {  

14    ^~~~~~  

15 error novel2.cpp:22:17: note: passing 'const main()::
MyCustomerHash*' as 'this' argument discards qualifiers

```

закоторым следуют более 20 других сообщений об ошибке:

```

16 In file included from /cygdrive/p/gcc/gcc61-include/
      bits/move.h:57:0,
17      from /cygdrive/p/gcc/gcc61-include/bits/
      stl_pair.h:59,
18      from /cygdrive/p/gcc/gcc61-include/bits/
      stl_algobase.h:64,
19      from /cygdrive/p/gcc/gcc61-include/bits/
      char_traits.h:39,
20      from /cygdrive/p/gcc/gcc61-include/string:40,
21      from error novel2.cpp:1:  

22 /cygdrive/p/gcc/gcc61-include/type_traits: In instantiation
of 'struct std::__not<std::__and<std::__is_fast_hash<main()::
MyCustomerHash>, std::__detail::__is_noexcept_hash<Customer,
main()::MyCustomerHash> > >':  

23 /cygdrive/p/gcc/gcc61-include/bits/unordered_set.h:95:63:
required from 'class std::unordered_set<Customer, main()::
MyCustomerHash>'  

24 error novel2.cpp:28:47: required from here  

25 /cygdrive/p/gcc/gcc61-include/type_traits:154:38: error: 'value'
is not a member of 'std::__and<std::__is_fast_hash<main()::
MyCustomerHash>, std::__detail::__is_noexcept_hash<Customer,
main()::MyCustomerHash> >'  

26      : public integral_constant<bool, !_Pp::value>
27      ^~~~  

28 In file included from /cygdrive/p/gcc/gcc61-include/
      unordered_set:48:0,
29      from error novel2.cpp:2:  

30 /cygdrive/p/gcc/gcc61-include/bits/unordered_set.h: In
instantiation of 'class std::unordered_set<Customer, main()::
MyCustomerHash>':  

31 error novel2.cpp:28:47: required from here  

32 /cygdrive/p/gcc/gcc61-include/bits/unordered_set.h:95:63: error:
'value' is not a member of 'std::__not<std::__and<std::__

```

Это сообщение об ошибке так же трудно прочесть, как и предыдущие (даже просто найти начало и конец каждого сообщения — работа непростая). Проблема заключается в том, что глубоко в заголовочном файле `hashtable_policy.h` в инстанцировании `std::unordered_set<>`, инициированном

```
std::unordered_set<Customer, MyCustomerHash> coll;
```

не находится соответствие для вызова

```
const main()::MyCustomerHash (const Customer&)
```

в инстанцировании

```
noexcept(declval<const _Hash&>() (declval<const _Key&>()))>
```

(`declval<const _Hash&>()`) представляет собой выражение типа `main()::MyCustomerHash`). Возможный “близкий” кандидат представляет собой

```
std::size_t main()::MyCustomerHash::operator()(const Customer&)
```

который объявлен как

```
std::size_t operator() (const Customer& c) {  
    ^~~~~~
```

и последнее примечание информирует о проблеме:

passing 'const main()::MyCustomerHash*' as 'this' argument discards qualifiers

Вы можете понять, в чем суть проблемы? Данная реализация шаблона класса `std::unordered_set` требует, чтобы оператор вызова объекта хеша был константной функцией-членом (см. также раздел 11.1.1). И когда это не так, где-то глубоко “в ливере” алгоритма возникает ошибка.

Все прочие сообщения об ошибках просто вытекают из первого и исчезают, если добавить квалификатор `const` к оператору хеш-функции:

```
std::size_t operator() (const Customer& c) const {  
    ...  
}
```

Подсказка clang 3.9 немного лучше, так как в конце первого сообщения об ошибке ясно говорит, что `operator()` функтора хеширования не помечен как `const`:

```
***  
errornovel2.cpp:28:47: note: in instantiation of template class  
'std::unordered_set<Customer, MyCustomerHash, std::equal_to<Customer>,  
std::allocator<Customer> >' requested here  
std::unordered_set<Customer,MyCustomerHash> coll;  
^  
  
errornovel2.cpp:22:17: note: candidate function not viable: 'this'  
argument has type 'const MyCustomerHash', but method is not marked  
const  
std::size_t operator() (const Customer& c) {  
^
```

Обратите внимание на то, что clang упоминает здесь параметры шаблона по умолчанию, такие как `std::allocator<Customer>`, в то время как gcc их опускает.

Как можно видеть, часто полезно иметь несколько компиляторов для тестирования кода. Это не только поможет написать более переносимый код, но и, когда один компилятор выдаст особенно непостижимое сообщение об ошибке, другой компилятор может помочь вам его понять.

9.5. Некоторые замечания

Организация исходного кода в заголовочных файлах и СРР-файлах является практическим следствием различных воплощений *правила одного определения* (one-definition rule – ODR). Подробное обсуждение этого правила содержится в приложении А, “Правило одного определения”.

Модель включения является прагматичным ответом, во многом продиктованным существующей практикой реализации компиляторов C++. Однако первая реализация C++ была иной: подразумевалось включение определений шаблона, что создало определенную иллюзию *разделения* (см. подробности этой модели в главе 14, “Инстанцирование”).

Первый стандарт C++ ([25]) предоставил явную поддержку разделения *модели разделения* компиляции шаблонов посредством *экспортированных шаблонов*. Модель разделения позволяла маркировать объявления шаблонов как `export`, чтобы объявлять их в заголовочных файлах, в то время как соответствующие определения помещались в СРР-файлы, так же, как объявления и определения нешаблонного кода. В отличие от модели включения эта модель была теоретической, не основанной на какой-либо существующей реализации, и сама по себе оказалась гораздо более сложной, чем предполагал Комитет по стандартизации C++. Потребовалось более пяти лет, чтобы увидеть ее первую реализацию опубликованной (в мае 2002 года), и с тех пор никаких других реализаций не появлялось. Чтобы стандарт C++ лучше соответствовал существующей практике, Комитет по стандартизации C++ удалил из C++11 экспортируемые шаблоны. Читатели, желающие более подробно узнать о модели разделения (и ее ловушках), могут обратиться к разделам 6.3 и 10.3 первого издания этой книги ([71]).

Иногда бывает заманчиво представить способы расширения концепции предкомпилированных заголовочных файлов, чтобы для одиночной компиляции

можно было загрузить более одного заголовка. В принципе это обеспечило бы более тонкий, мелкозернистый подход к предварительной компиляции. Главным препятствием здесь является препроцессор: макросы в одном заголовочном файле могут полностью изменить смысл последующих заголовочных файлов. Однако после того, как файл будет предкомпилирован, а макрообработка завершена, задача изменения предкомпилированного заголовочного файла с учетом эффектов, вызванных другими заголовками, окажется слишком сложной с практической точки зрения. Ожидается, что для решения этой проблемы в C++ в не слишком отдаленном будущем будет добавлена новая возможность языка, известная как *модули* (см. раздел 17.11) (макроопределения не смогут попасть в интерфейсы модулей).

9.6. Резюме

- Модель включения является моделью, наиболее применимой для организации шаблонного кода. Ее альтернативы рассматриваются в главе 14, “Инстанцирование”.
- При определении в заголовочных файлах вне классов и структур `inline` необходим только для полных специализаций шаблонов функций.
- Чтобы воспользоваться преимуществами предкомпилированных заголовочных файлов, не забудьте сохранять один и тот же порядок директив `#include` в разных файлах.
- Отладка кода с шаблонами может оказаться весьма сложной задачей.

Глава 10

Основные термины в области шаблонов

Предыдущие главы книги были посвящены знакомству с основами концепции шаблонов в C++. Теперь, прежде чем перейти к более подробному рассмотрению шаблонов, хотелось бы уделить внимание терминам, которые используются при изложении материала. В этом есть необходимость, поскольку в сообществе C++ (и даже в ранних версиях стандарта) иногда отсутствует четкое понимание концепций и терминологии.

10.1. “Шаблон класса” или “шаблонный класс”

В C++ структуры, классы и объединения имеют общее название *типы классов*, или *классовые типы* (class types). Без дополнительного уточнения слово “класс” обычно служит для обозначения типов класса, заданных с помощью ключевых слов `class` или `struct`¹. Особо следует отметить, что понятие “тип класса” включает объединения, а “класс” — нет.

Имеется определенная некоторая путаница в отношении того, как следует именовать класс, являющийся шаблоном.

- Термин *шаблон класса* (class template) означает, что класс является шаблоном. Другими словами, это параметризованное описание семейства классов.
- С другой стороны, термин *шаблонный класс* (template class) используется
 - как синоним для шаблона класса;
 - для обозначения классов, сгенерированных из шаблона;
 - для обозначения классов с именем, которое является *идентификатором шаблона* (комбинацией имени шаблона, за которым следуют аргументы шаблона между < и >).

Разница между вторым и третьим значениями весьма незначительна и в остальной части книги не играет сколько-нибудь заметной роли.

Из-за упомянутой неточности в данной книге мы старались избегать термина *шаблонный класс*.

¹ В C++ единственное различие между `class` и `struct` заключается в том, что доступ по умолчанию для класса является закрытым (`private`), в то время как доступ по умолчанию к членам структуры — открытым (`public`). Однако мы предпочитаем использовать `class` для типов, в которых применяются новые возможности C++, а `struct` — для обычных структур C, которые могут использоваться как “простые старые данные” (`plain old data` — POD).

Аналогично мы используем термины *шаблон функции* (function template), *шаблон члена* (member template), *шаблон функции-члена* (member function template) и *шаблон переменной* (variable template), но стараемся избегать терминов *шаблонная функция* (template function), *шаблонный член* (template member), *шаблонная функция-член* (template member function) и *шаблонная переменная* (template variable).

10.2. Подстановка, инстанцирование и специализация

При обработке исходного текста, в котором используются шаблоны, компилятор C++ должен в разные моменты времени выполнять *подстановку* (substitute) конкретных аргументов шаблона вместо параметров шаблона в шаблоне. Иногда эта замена только предварительная: компилятору может потребоваться проверить, является ли допустимым такое замещение (см. разделы 8.4 и 15.7).

Процесс создания *определения* (definition) для обычного класса, псевдонима типа, функции, функции-члена или переменной из шаблона путем подстановки конкретных аргументов в качестве параметров шаблона называется *инстанцированием шаблона* (template instantiation).

Удивительно, но в настоящее время нет стандартного или хотя бы в целом согласованного термина для обозначения процесса создания *объявления* (declaration), которое не является определением, путем подстановки параметров шаблона. Мы встречались с такими терминами, используемыми разными группами, как *частичное инстанцирование* (partial instantiation) или *инстанцирование объявления* (instantiation of a declaration), но они ни в коем случае не являются универсальными. Возможно, более интуитивным является термин *неполное инстанцирование* (incomplete instantiation) (которое в случае шаблона класса производит неполный класс).

Сущности, появляющиеся в результате инстанцирования или неполного инстанцирования (т.е. класс, функция, функция-член или переменная) в общем случае называются *специализациями* (specialization).

Однако в C++ процесс инстанцирования является не единственным способом получить специализацию. Существуют альтернативные механизмы, позволяющие программисту явно задавать объявление, привязанное к определенной подстановке параметров шаблона. Как упоминалось в разделе 2.5, такая специализация вводится с помощью префикса `template<>`:

```
template<typename T1, typename T2> // Первичный шаблон класса
class MyClass
{
    ...
};

template<> // Явная специализация
class MyClass<std::string, float>
{
    ...
};
```

Строго говоря, это так называемая *явная специализация* (explicit specialization) (в отличие от *инстанцируемой*, или *генерируемой специализации* (instantiated specialization, generated specialization)).

Как отмечалось в разделе 2.6, специализации, в которых остаются параметры шаблона, называются *частичными специализациями* (partial specialization).

```
template<typename T>      // Частичная специализация
class MyClass<T, T>
{
    ...
};

template<typename T>      // Частичная специализация
class MyClass<bool, T>
{
    ...
};
```

Если речь идет о специализации (явной или частичной), то общий шаблон называется *первичным шаблоном* (primary template).

10.3. Объявления и определения

До сих пор понятия *объявление* (declaration) и *определение* (definition) встречались не слишком часто. Однако оба понятия достаточно точно определены в стандарте C++, и именно эти значения данных понятий используются в нашей книге.

Объявление (declaration) является конструкцией C++, которая вводит или повторно задает имя в области видимости C++. Такое задание всегда включает частичную классификацию имени, но для корректности объявления указание всех деталей не требуется, например:

```
class C;          // Объявление С как класса
void f(int p);   // Объявление f() как функции,
                  // а p - как именованного параметра
extern int v;    // Объявление v как переменной
```

Отметим, что макроопределения и метки *goto*, несмотря на то, что они имеют “имена”, в C++ объявлениями не считаются.

Объявления становятся *определениями* (definition), когда делается известной информация об их структуре или, в случае переменных, когда для них должна быть выделена память. Для определений типов классов это означает, что должно быть предоставлено заключенное в фигурные скобки тело. Для определений функций это означает, что должно быть предоставлено заключенное в фигурные скобки тело (в общем случае) или функция должна быть определена как `=default`² или `=delete`. В случае переменных для того, чтобы объявление

² Функции по умолчанию являются специальными функциями-членами, получающими реализации по умолчанию, генерируемые компилятором (такие как копирующий конструктор по умолчанию).

стало определением, достаточно инициализации или отсутствия спецификатора `extern`. Далее приведены примеры, которые дополняют представленные выше объявления, не являющиеся определениями.

```
class C {};           // Определение (и объявление) класса C
void f(int p)        // Определение (и объявление) функции f()
{
    std::cout << p << '\n';
}
extern int v = 1;     // Инициализатор делает этот код определением v
int w;               // Объявления глобальных переменных, не
                     // предшествуемые ключевым словом extern,
                     // также являются определениями
```

Распространение этого принципа на шаблоны приводит к тому, что объявление шаблона класса или шаблона функции называется определением, если оно имеет тело. Следовательно,

```
template<typename T>
void func(T);
```

является объявлением, но не определением, в то время как

```
template<typename T>
class S {};
```

фактически является определением.

10.3.1. Полные и неполные типы

Типы могут быть *полными* (*complete*) или *неполными* (*incomplete*); это понятие тесно связано с различиями между *объявлением* и *определением*. Некоторые языковые конструкции требуют *полные типы* (*complete types*), тогда как другие являются корректными и при использовании *неполных типов* (*incomplete types*).

Неполный тип представляет собой одно из следующего списка:

- объявленный, но не определенный тип класса;
- тип массива с не определенными границами;
- тип массива с элементами неполного типа;
- `void`;
- тип перечисления, пока не определен его базовый тип или значения перечислителей;
- любой из перечисленных выше типов, к которому применен спецификатор `const` и/или `volatile`.

Все прочие типы являются *полными*, например:

```
class C;           // C – неполный тип
C const* cp;      // cp – указатель на неполный тип
extern C elems[10]; // elems имеет неполный тип
extern int arr[]; // arr имеет неполный тип
```

```
class C { };           // С теперь является полным типом (а
                      // следовательно, ср и elems больше
                      // не ссылаются на неполные типы)
int arr[10];          // arr имеет полный тип
```

Подсказки по работе с неполными типами в шаблонах представлены в разделе 11.5.

10.4. Правило одного определения

В языке C++ на повторные объявления различных сущностей накладываются определенные ограничения. Вся совокупность этих ограничений известна как *правило одного определения* (one definition rule – ODR). Детали этого правила очень сложны и охватывают огромное множество ситуаций. В последующих главах различные аспекты правила одного определения будут проиллюстрированы для каждого рассматриваемого случая, а полное его описание читатель найдет в приложении А, “Правило одного определения”. Пока что достаточно знать лишь основные положения этого правила.

- Обычные, т.е. невстраиваемые функции и функции-члены, так же как и (невстраиваемые) глобальные переменные и статические данные-члены, должны определяться однократно в рамках *программы* в целом³.
- Типы классов (включая структуры и объединения), шаблоны (включая частичные, но не полные специализации) и встраиваемые функции и переменные должны быть определены не более одного раза в пределах *единицы трансляции* (translation unit), и все эти определения должны быть идентичными.

Единица трансляции представляет собой то, что получается в результате обработки исходного файла процессором; другими словами, она включает содержимое, заданное директивами включения файлов `#include` и полученное в результате раскрытия макроопределений.

В оставшейся части книги *связываемый объект* (linkable entity) будет означать одно из следующего списка: функция или функция-член, глобальная переменная или статические данные-члены (включая любой из перечисленных объектов, сгенерированный из шаблона), видимые компоновщику.

10.5. Аргументы и параметры шаблонов

Сравним шаблон класса

```
template<typename T, int N>
class ArrayInClass
```

³ Глобальные и статические переменные и данные-члены могут быть определены как `inline`, начиная с C++17. Эта возможность снимает требование о том, что они должны быть определены только в одной единице трансляции.

```
{
public:
    T array[N];
};
```

с аналогичным обычным классом:

```
class DoubleArrayInClass
{
public:
    double array[10];
};
```

Последний становится, по сути, эквивалентен первому, если заменить параметры T и N значениями `double` и `10` соответственно. В C++ эта подстановка обозначается как

```
ArrayInClass<double,10>
```

Заметим, что за именем шаблона следуют так называемые *аргументы шаблона* в угловых скобках.

Зависят ли эти аргументы от параметров шаблона или нет, комбинация имени шаблона, за которым следуют аргументы в угловых скобках, называется *идентификатором шаблона* (*template-id*).

Это имя может использоваться почти так же, как и соответствующие нешаблонные объекты, например:

```
int main()
{
    ArrayInClass<double, 10> ad;
    ad.array[0] = 1.0;
}
```

Важно различать *параметры шаблона* и *аргументы шаблона*. Коротко говоря, можно сказать, что “*параметры* инициализируются *аргументами*”⁴. Или, если быть более точным:

- *параметрами шаблона* являются те имена, которые перечислены после ключевого слова `template` в объявлении или определении шаблона (в нашем примере — T и N);
- *аргументы шаблона* являются элементами, которые подставляются вместо параметров шаблона (`double` и `10` в нашем примере). В отличие от параметров шаблона, аргументы шаблона могут представлять собой нечто большее, чем просто “имена”.

Подстановка аргументов шаблона вместо параметров шаблона выполняется явно посредством идентификатора шаблона, однако есть различные ситуации, когда подстановка выполняется неявно (например, если вместо параметров подставляются их аргументы по умолчанию).

⁴ В академических кругах “аргументы” иногда называются *фактическими параметрами* (actual parameters), а “параметры” — *формальными параметрами* (formal parameters).

Фундаментальный принцип заключается в том, что любой аргумент шаблона должен быть величиной или значением, которое определимо во время компиляции. Как станет понятно позже, это сулит огромные выгоды в плане стоимости времени выполнения шаблонных объектов. Поскольку параметры шаблона в конечном счете заменяются значениями времени компиляции, они сами могут быть использованы для образования выражений, вычисляемых во время компиляции. Эта возможность используется в шаблоне `ArrayInClass` для задания размера члена массива `array`. Размер массива должен быть так называемым *константным выражением* (constant-expression), и параметр шаблона `N` является именно таким.

Поскольку параметры шаблона — это объекты времени компиляции, их также можно использовать для создания корректных аргументов шаблонов. Приведем пример:

```
template<typename T>
class Dozen
{
public:
    ArrayInClass<T, 12> contents;
};
```

Обратите внимание на то, что в данном примере имя `T` является как параметром шаблона, так и аргументом шаблона. Таким образом обеспечивается механизм конструирования более сложных шаблонов из более простых. Разумеется, этот механизм не имеет фундаментальных отличий от механизмов, которые позволяют связывать типы и функции.

10.6. Резюме

- Термины *шаблон класса*, *шаблон функции* и *шаблон переменной* используются соответственно для классов, функций и переменных, которые являются шаблонами.
- *Инстанцирование шаблона* представляет собой процесс создания обычных классов или функций путем замены *параметров шаблонов конкретными аргументами*. Результатирующая сущность представляет собой *специализацию*.
- Типы могут быть полными и неполными.
- Согласно правилу одного определения невстраиваемые функции, функции-члены, глобальные переменные и статические данные-члены должны определяться однократно в рамках программы в целом.

Глава 11

Обобщенные библиотеки

До сих пор наше обсуждение шаблонов было сосредоточено на их специфических особенностях, возможностях и ограничениях, с направленностью на непосредственные задачи и приложения (то, с чем в первую очередь сталкиваются прикладные программисты). Однако наиболее эффективными шаблоны оказываются тогда, когда используются для написания обобщенных библиотек и каркасов, где наши проекты должны рассматривать потенциальные применения априори максимально широко. Хотя к таким конструкциям может быть применим почти весь материал данной книги, здесь рассматриваются некоторые общие вопросы, возникающие при написании переносимых компонентов, которые планируется использовать для типов, не существующих пока что даже в проекте.

Перечень поднятых здесь вопросов не завершен в любом смысле, но он кратко резюмирует некоторые из возможностей, представленных в книге до этого момента, вводит некоторые дополнительные функциональные возможности и ссылается на некоторые особенности шаблонов и языка, рассматриваемые в этой книге позже. Мы надеемся, что данный материал еще сильнее мотивирует читателей на чтение всех последующих глав этой книги.

11.1. Вызываемые объекты

Многие библиотеки включают интерфейсы, которым клиентский код передает некоторые сущности, которые должны быть “вызваны”. Примеры включают операции, планируемые для выполнения в другом потоке; функции, описывающие, как хеш-значения должны храниться в хеш-таблице; объекты, описывающие порядок, в котором следует сортировать элементы в коллекции; и обобщенная оболочка, обеспечивающая некоторые значения аргументов по умолчанию. Стандартная библиотека не является исключением: она определяет многие компоненты, которые принимают такие вызываемые сущности.

Одним из терминов, используемых в этом контексте, является понятие *обратного вызова* (callback). Традиционно этот термин зарезервирован для сущностей, которые передаются в качестве аргументов вызова функции (в отличие от, например, аргументов шаблона), и мы придерживаемся этой традиции. Например, функция сортировки может включать параметр обратного вызова как “критерий сортировки”, который вызывается, чтобы определить, находится ли один элемент по отношению к другому в требуемом порядке сортировки.

В C++ имеется ряд типов, хорошо подходящих для обратных вызовов, потому что они могут как быть переданы в качестве аргументов функции, так и вызываться непосредственно с синтаксисом вызова `f (...)`.

- Типы указателей на функции.
- Типы классов с перегруженным оператором `operator()` (которые иногда называются *функторами* (functors)), включая лямбда-выражения.
- Типы классов с функциями преобразования, дающими указатель на функцию или ссылку на функцию.

Вместе все эти типы называются *типами функциональных объектов*, а значение такого типа называется функциональным объектом.

Стандартная библиотека C++ вводит несколько более широкое понятие *вызываемого типа* (callable type), который представляет собой либо тип функционального объекта, либо указатель на член. Объект вызываемого типа называется *вызываемым объектом* (callable object).

Обобщенный код часто извлекает выгоду от возможности принять вызываемый объект любого вида, и шаблоны позволяют легко это сделать.

11.1.1. Поддержка функциональных объектов

Давайте рассмотрим, как реализован алгоритм `for_each()` стандартной библиотеки (воспользуемся для этого именем “`foreach`”, чтобы избежать возможных конфликтов имен, и для упрощения не будем ничего возвращать из этой функции):

basics/foreach.hpp

```
template<typename Iter, typename Callable>
void foreach (Iter current, Iter end, Callable op)
{
    while (current != end) // Пока не достигнут конец
    {
        op(*current);      // Вызов переданного оператора
                            // для текущего элемента
        ++current;         // и перемещение итератора к
                            // следующему элементу
    }
}
```

Приведенная ниже программа демонстрирует применение этого шаблона с различными функциональными объектами:

basics/foreach.cpp

```
#include <iostream>
#include <vector>
#include "foreach.hpp"

// Вызываемая функция:
void func(int i)
{
    std::cout << "func() вызывается для: " << i << '\n';
}
```

```

// Тип функционального объекта (для объектов, которые могут
// быть использованы как функции):
class FuncObj
{
public:
    void operator()(int i) const // Примечание: константная
    {                           // функция-член
        std::cout << "FuncObj::op() вызывается для: " << i << '\n';
    }
};

int main()
{
    std::vector<int> primes = { 2, 3, 5, 7, 11, 13, 17, 19 };

    foreach(primes.begin(), primes.end(), // Диапазон
            func); // Функция (название к указателю)

    foreach(primes.begin(), primes.end(), // Диапазон
            &func); // Указатель на функцию

    foreach(primes.begin(), primes.end(), // Диапазон
            FuncObj()); // Функциональный объект

    foreach(primes.begin(), primes.end(), // Диапазон
            [](int i){ // Лямбда-выражение
                std::cout << "Лямбда-выражение вызвано для: "
                << i << '\n';
            });
}
}

```

Давайте подробнее рассмотрим каждый случай.

- Когда мы передаем имя *функции* в качестве аргумента функции, мы в действительности передаем не саму функцию, а указатель или ссылку на нее. Как и в случае массивов (см. раздел 7.4), аргумент-функция низводится до указателя при передаче по значению, а в случае параметра, тип которого является параметром шаблона, будет выведен тип указателя на функцию. Функции, так же как и массивы, могут передаваться по ссылке без низведения типа. Однако типы функций в действительности не могут квалифицироваться с помощью `const`. Если мы объявим последний параметр `foreach()` с типом `Callable const&`, этот `const` будет просто проигнорирован. (Вообще говоря, в коде C++ редко используются ссылки на функции.)
- Наш второй вызов явно принимает *указатель на функцию* с использованием адреса имени функции. Это эквивалентно первому вызову (в котором имя функции неявно низводилось до значения указателя), но, возможно, выглядит немного яснее.
- При передаче *функционального объекта* мы передаем объект типа класса в качестве вызываемого объекта. Вызов через тип класса обычно сводится к вызову его `operator()`. Поэтому вызов

```
op(*current);
```

обычно трансформируется в

```
op.operator()(*current); // Вызов operator() с параметром
// *current для объекта op
```

Обратите внимание на то, что при определении `operator()` вы обычно должны определять как константную функцию-член. В противном случае может произойти тонкая ошибка, когда каркасы или библиотеки ожидают, что такой вызов не изменяет состояние переданного объекта (подробнее см. раздел 9.4).

Возможно также, что объект типа класса будет неявно преобразовываться в указатель или ссылку на *функцию суррогатного вызова* (*surrogate call function*) (обсуждается в разделе B.3.5). В таком случае вызов

```
op(*current);
```

будет преобразован в

```
(op.operator F())(*current);
```

где `F` представляет собой тип указателя или ссылки на функцию, в которую может быть преобразован объект типа класса. Это относительно нераспространенный подход.

- *Лямбда-выражения* создают функторы (именуемые *замыканиями* (*closure*)), и поэтому данный случай не слишком отличается от случая использования функтора. Однако лямбда-выражения являются очень удобной сокращенной записью для представления функторов, потому они часто появляются в коде, начиная с C++11.

Интересно, что лямбда-выражения, которые начинаются с `[` (без захвата) производят оператор преобразования в тип указателя на функцию. Однако он никогда не будет выбран как *функция суррогатного вызова*, потому что его соответствие всегда хуже, чем у обычного `operator()` замыкания.

11.1.2. Работа с функциями-членами и дополнительными аргументами

Одна возможная сущность для вызова в предыдущем примере не использовалась — это функция-член. Дело в том, что вызов нестатической функции-члена обычно включает указание объекта, для которого применяется вызов, с использованием синтаксиса наподобие `object.memfunc(...)` или `ptr->memfunc(...)`, не соответствующий обычной схеме *функциональный_объект(...)*.

К счастью, начиная с C++17, стандартная библиотека C++ предоставляет утилиту `std::invoke()`, которая удобно унифицирует этот случай со случаями с синтаксисом вызова обычных функций, тем самым позволяя вызывать любой вызываемый объект с помощью одной формы вызова. Следующая реализация нашего шаблона `foreach()` использует `std::invoke()`:

basics/foreachinvoke.hpp

```
#include <utility>
#include <functional>

template<typename Iter, typename Callable, typename... Args>

void foreach (Iter current, Iter end, Callable op,
             Args const& ... args)
{
    while (current != end)      // Пока не достигнут конец
    {
        std::invoke(op,          // Вызов переданного вызываемого
                    args...,     // объекта с любыми аргументами
                    args
                    *current); // и текущим элементом current
        ++current;
    }
}
```

Здесь кроме параметра вызываемого объекта мы также принимаем произвольное количество дополнительных параметров. Затем шаблон `foreach()` вызывает `std::invoke()` с данным вызываемым объектом, за которым следуют дополнительные переданные параметры вместе со ссылкой на элемент. `std::invoke()` обрабатывает этот код следующим образом.

- Если вызываемый объект представляет собой указатель на член, он использует первый дополнительный аргумент как этот объект. Все остальные дополнительные параметры просто передаются как аргументы вызываемому объекту.
- В противном случае все дополнительные параметры просто передаются как аргументы вызываемого объекта.

Обратите внимание на то, что мы не можем использовать здесь прямую передачу для вызываемого объекта или дополнительных параметров: первый вызов может “украсть” их значения, что приведет к неожиданному поведению ор в последующих итерациях.

Эта реализация позволяет по-прежнему компилировать все наши исходные вызовы `foreach()`, показанные выше. Теперь, кроме того, мы можем также передавать дополнительные аргументы вызываемым объектам, а сам вызываемый объект может быть функцией-членом¹. Следующий клиентский код иллюстрирует это:

basics/foreachinvoke.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include "foreachinvoke.hpp"
```

¹ `std::invoke()` позволяет также использовать в качестве типа обратного вызова указатель на член-данное. Вместо вызова функции шаблон возвращает значение соответствующего члена-данного в объекте, на который ссылается дополнительный аргумент.

```

// Класс с функцией-членом, который должен быть вызван
class MyClass
{
public:
    void memfunc(int i) const
    {
        std::cout << "MyClass::memfunc() вызван для: " << i << '\n';
    }
};

int main()
{
    std::vector<int> primes = { 2, 3, 5, 7, 11, 13, 17, 19 };

    // Передача лямбда-выражения в качестве вызываемого
    // объекта и дополнительный аргумент:
    foreach (primes.begin(),
              primes.end(), // Элементы второго аргумента
              // лямбда-выражения
              [] (std::string const& prefix, int i) // Вызываемое
              { std::cout << prefix << i << '\n'; }, // лямбда-выражение
              "- значение: "); // Первый аргумент лямбда-выражения

    // Вызов obj.memfunc() для/с каждого элемента primes,
    // передаваемого в качестве аргумента
    MyClass obj;

    foreach (primes.begin(), // Элементы, используемые
            primes.end(), // в качестве аргументов
            &MyClass::memfunc, // Вызываемая функция-член
            obj); // Объект вызова memfunc()
}

```

Первый вызов `foreach()` получает четвертый аргумент (строковый литерал `"- значение: "`), используемый в качестве первого параметра лямбда-выражения, в то время как текущий элемент вектора связывается со вторым параметром лямбда-выражения. Второй вызов передает функцию-член `memfunc()` в качестве третьего аргумента вызова `obj`, передаваемого как четвертый аргумент.

Свойства типов, которые проверяют, может ли вызываемый объект быть использован `std::invoke()`, описываются в разделе Г.3.1.

11.1.3. Оборачивание вызовов функций

Типичное применение `std::invoke()` заключается в том, чтобы получить "обертку" для вызова одной функции (например, для выполнения записи о вызовах в журнал, измерения их продолжительности или подготовки некоторого контекста, как, например, запуск для них новых потоков). Теперь мы можем обеспечить поддержку семантики перемещения с помощью прямой передачи как вызываемого объекта, так и всех передаваемых аргументов:

basics/Invoke.hpp

```
#include <utility>      // Для std::invoke()
#include <functional> // Для std::forward()

template<typename Callable, typename... Args>
decltype(auto) call(Callable&& op, Args&& ... args)
{
    return std::invoke(
        std::forward<Callable>(op), // Передача вызываемого объекта
        std::forward<Args>(args)...); // с дополнительными аргументами
}
```

Еще одним интересным аспектом является работа с возвращаемым значением вызываемой функции для его “прямой передачи” назад вызывающему коду. Для поддержки возвращения ссылок (как, например, `std::ostream&`) следует использовать `decltype(auto)` вместо просто `auto`:

```
template<typename Callable, typename... Args>
decltype(auto) call(Callable&& op, Args&& ... args)
```

`decltype(auto)` (доступно начиная с C++14) является *типов-заместителем* (placeholder type), который определяет тип переменной, возвращаемый тип или аргумент шаблона из типа связанного выражения (инициализатор, возвращаемое значение или аргумент шаблона). Подробности его использования можно найти в разделе 15.10.3.

Если вы хотите временно хранить значение, возвращаемое `std::invoke()`, в переменной, чтобы вернуть его после того, как будет сделано что-то иное (например, выполнение некоторых действий с возвращаемым значением или журнальная запись о конце вызова), вы также должны объявить эту временную переменную с использованием `decltype(auto)`:

```
decltype(auto) ret(std::invoke(std::forward<Callable>(op),
                               std::forward<Args>(args)...));
***
```

```
return ret;
```

Обратите внимание на то, что объявлять `ret` с помощью `auto&&` некорректно. В качестве ссылки `auto&&` продлевает время жизни возвращаемого значения до конца ее области видимости (см. раздел 11.3), но не за пределы инструкции `return` вызывающему коду.

Однако имеется проблема и при использовании `decltype(auto)`: если вызываемый объект имеет возвращаемый тип `void`, инициализация `ret` как `decltype(auto)` не разрешена, так как `void` является неполным типом. У вас есть следующие варианты.

- Объявить в строке до инструкции `object`, деструктор которого выполняет наблюдаемое поведение, которое вы хотите реализовать. Например²:

²Выражаем благодарность Даниэлю Крюглеру (Daniel Krügler) за подсказку этого метода.

```

struct cleanup
{
    ~cleanup()
    {
        ... // Код, выполняемый при возврате
    }
} dummy;
return std::invoke(std::forward<Callable>(op),
                  std::forward<Args>(args)...);

```

- По-разному реализовать случаи для void и для не-void:

basics/invokeret.hpp

```

#include <utility>      // Для std::invoke()
#include <functional>    // Для std::forward()
#include <type_traits>   // Для std::is_same<> и invoke_result<>

template<typename Callable, typename... Args>
decltype(auto) call(Callable&& op, Args&& ... args)
{
    if constexpr(std::is_same_v<std::invoke_result_t<
                           Callable, Args...>,
                           void>)
    {
        // Возвращаемый тип - void:
        std::invoke(std::forward<Callable>(op),
                    std::forward<Args>(args)...);
        ...
        return;
    }
    else
    {
        // Возвращаемый тип - не void:
        decltype(auto) ret{std::invoke(std::forward<Callable>(op),
                                      std::forward<Args>(args))};
        ...
        return ret;
    }
}

```

Имея

```
if constexpr(std::is_same_v<std::invoke_result_t<Callable, Args...>,
            void>)
```

мы выполняем проверку времени компиляции, является ли void возвращаемым типом вызываемого объекта callable с Args... . Детальное описание std::invoke_result<> см. в разделе Г.3.1³.

Будем надеяться, что будущие версии C++ смогут избежать необходимости такой специальной обработки void (см. раздел 17.7).

³ std::invoke_result<> доступен, начиная с C++17. Для получения возвращаемого типа, начиная с C++11, можно использовать код typename std::result_of<Callable(Args...)>::type.

11.2. Другие утилиты для реализации обобщенных библиотек

`std::invoke()` — лишь один из примеров полезных утилит, предоставляемых стандартной библиотекой C++ для реализации обобщенных библиотек. Ниже мы бегло рассмотрим некоторые другие важные утилиты.

11.2.1. Свойства типов

Стандартная библиотека предоставляет различные утилиты, именуемые *свойствами типов* (type traits), которые позволяют нам вычислять и изменять типы. Они обеспечивают поддержку различных случаев, когда обобщенный код должен адаптироваться к различным типам, или реагировать на возможности типов, для которых он инстанцируется. Например:

```
#include <type_traits>
template<typename T>
class C
{
    // Гарантируем, что T не является void
    // (игнорируя const или volatile):
    static_assert(!std::is_same_v<std::remove_cv_t<T>, void>,
                  "Неверное инстанцирование класса C для void");
public:
    template<typename V>
    void f(V&& v)
    {
        if constexpr(std::is_reference_v<T>)
        {
            ... // Специальный код, если T – ссылочный тип
        }

        if constexpr(std::is_convertible_v<std::decay_t<V>, T>)
        {
            ... // Специальный код, если V преобразуемый в T
        }

        if constexpr(std::has_virtual_destructor_v<V>)
        {
            ... // Специальный код, если у V виртуальный деструктор
        }
    }
};
```

Как показывает этот пример, путем проверки некоторых условий мы можем выбирать между различными реализациями шаблона. Здесь мы используем функциональную возможность `if` времени компиляции, которая появилась в C++17 (см. раздел 8.5), но вместо нее могли бы использовать `std::enable_if`, частичную специализацию или SFINAE для включения или отключения вспомогательных шаблонов (подробную информацию см. в главе 8, “Программирование времени компиляции”).

Обратите, однако, внимание на то, что свойства типов должны использоваться с особой осторожностью: они могут вести себя иначе, чем может ожидать простодушный программист. Например:

```
std::remove_const_t<int const&> // Дает int const&
```

Здесь, поскольку ссылка не является `const` (хотя вы не можете ее изменять), вызов не действует и дает переданный тип.

Как следствие, порядок удаления ссылочности и `const` имеет значение:

```
std::remove_const_t<std::remove_reference_t<int const&>> // int
std::remove_reference_t<std::remove_const_t<int const&>> // int const
```

Вместо этого можно просто вызвать

```
std::decay_t<int const&> // Дает int
```

Однако этот код будет также преобразовывать массивы и функции в соответствующие типы указателей.

Кроме того, есть случаи, когда свойства типов имеют определенные требования, невыполнение которых ведет к неопределенному поведению⁴. Например:

```
make_unsigned_t<int> // unsigned int
make_unsigned_t<int const&> // Неопределенное поведение
                           // (надеемся, ошибка)
```

Иногда результат может оказаться удивительным. Например:

```
add_rvalue_reference_t<int> // int&
add_rvalue_reference_t<int const> // int const&
add_rvalue_reference_t<int const&> // int const& (l-ссылка
                                    // остается l-ссылкой)
```

Здесь мы могли бы ожидать, что `add_rvalue_reference` всегда дает в результате r-ссылки, но правила свертывания ссылок C++ (см. раздел 15.6.1) приводят к тому, что сочетание l-ссылки и r-ссылки дает l-ссылку.

В качестве еще одного примера:

```
is_copy_assignable_v<int> // Дает true (в общем случае можно
                           // присваивать int типу int)
isAssignable_v<int, int> // Дает false (нельзя вызвать 42 = 42)
```

В то время как `is_copy_assignable` просто проверяет в общем случае, можно ли присвоить один `int` другому (проверка операции для l-значений), `isAssignable` принимает во внимание категорию значения (см. приложение Б, “Категории значений”) (здесь проверяется, можно ли присвоить pr-значение rg-значению). То есть первое выражение эквивалентно

```
isAssignable_v<int&, int&> // Дает true
```

⁴При рассмотрении C++17 было внесено предложение требовать, чтобы нарушение предусловий свойств типа всегда приводило к ошибке времени компиляции. Однако это изменение было отложено, поскольку некоторые свойства типов имеют чрезмерно ограничивающие требования, как, например, требование всегда обеспечивать полноту типов.

По той же причине:

```
is_swappable_v<int>           // Дает true (подразумевая 1-значения)
is_swappable_v<int&, int&>    // Дает true (эквивалент
                                // предыдущей проверки)
is_swappable_with_v<int, int> // Дает false (учитывает
                                // категорию значения)
```

По всем этим причинам тщательно изучайте точное определение свойств типов. Стандартные свойства типов подробно описаны в приложении Г, “Стандартные утилиты для работы с типами”.

11.2.2. `std::addressof()`

Шаблон функции `std::addressof()` дает фактический адрес объекта или функции. Он работает, даже если тип объекта имеет перегруженный оператор `&`. Несмотря на то что последнее случается очень редко, тем не менее такое возможно (например, в интеллектуальных указателях). Таким образом, рекомендуется использовать `addressof()`, если вы нуждаетесь в адресе произвольного типа:

```
template<typename T>
void f(T&& x)
{
    auto p = &x;                      // Может не работать при
                                      // перегруженном операторе &
    auto q = std::addressof(x); // Работает даже при
                                // перегруженном операторе &
    ...
}
```

11.2.3. `std::declval()`

Шаблон функции `std::declval()` может использоваться в качестве заместителя для ссылки на объект определенного типа. Функция не имеет определения и поэтому не может быть вызвана (и не создает объект). Следовательно, она может использоваться только в невычисляемых операндах (таких как конструкции `decltype` и `sizeof`). Так что вместо попыток создать объект можно считать, что у вас имеется объект соответствующего типа.

Например, следующее объявление выводит возвращаемый тип по умолчанию `RT` из переданных шаблону параметров `T1` и `T2`:

`basics/maxdefaultdeclval.hpp`

```
#include <utility>
template < typename T1, typename T2,
          typename RT = std::decay_t<
              decltype(true ? std::declval<T1>()
                           : std::declval<T2>()) >>
RT max(T1 a, T2 b)
{
    return b < a ? a : b;
}
```

Чтобы избежать необходимости вызовов конструкторов (по умолчанию) для типов T1 и T2 в вызове оператора ?: в выражении инициализации RT, мы используем std::declval, “используя” объекты соответствующего типа без их создания. Конечно, это возможно только в невычислимом контексте decltype.

Не забывайте использовать свойство типа std::decay<>, чтобы гарантировать, что тип возвращаемого значения по умолчанию не может быть ссылкой, так как сам шаблон std::declval() дает г-ссылку. В противном случае вызовы наподобие max(1, 2) будут давать возвращаемое значение типа int&&⁵. Подробности см. в разделе 19.3.4.

11.3. Прямая передача временных значений

Как было показано в разделе 6.1, можно использовать *передаваемые ссылки* и std::forward<> для “прямой передачи” обобщенных параметров:

```
template<typename T>
void f(T&& t) // t - передаваемая ссылка
{
    g(std::forward<T>(t)); // Прямая передача переданного
} // аргумента в g()
```

Однако иногда нам нужно выполнить в обобщенном коде прямую передачу данных, полученных не через параметр. В этом случае можно использовать auto&& для создания переменной, которая может быть передана. Например, предположим, что мы связываем в цепочку вызовы функций get() и set(), где возвращаемое значение метода get() должно быть прямо передано в set():

```
template<typename T>
void foo(T x)
{
    set(get(x));
}
```

Предположим далее, что нам нужно обновить наш код для выполнения некоторой операции над промежуточным значением, полученным путем вызова get(). Мы делаем это, сохраняя значение в переменной, объявленной с помощью auto&&:

```
template<typename T>
void foo(T x)
{
    auto&& val = get(x);
    /**
     * Прямая передача значения, возвращенного
     * методом get(), методу set():
     */
    set(std::forward<decltype(val)>(val));
}
```

Это позволяет избежать излишнего копирования промежуточного значения.

⁵ Выражаем благодарность Даниэлю Крюглеру за указание на этот факт.

11.4. Ссылки в качестве параметров шаблонов

Хотя это и не слишком распространенная практика, но параметры типов шаблонов могут быть ссылочными типами. Например:

basics/tmp1paramref.cpp

```
#include <iostream>

template<typename T>
void tmplParamIsReference(T)
{
    std::cout << "T является ссылкой: "
        << std::is_reference_v<T> << '\n';
}

int main()
{
    std::cout << std::boolalpha;
    int i;
    int& r = i;
    tmplParamIsReference(i);          // false
    tmplParamIsReference(r);          // false
    tmplParamIsReference<int&>(i);   // true
    tmplParamIsReference<int&>(r);   // true
}
```

Даже если передать в `tmplParamIsReference()` ссылочную переменную, параметр шаблона `T` будет выведен как тип, на который она ссылается (поскольку для ссылочной переменной `v` выражение `v` имеет тип, на который ссылается `v`; тип выражения никогда не является ссылкой). Однако можно обеспечить ссылку, явно указывая тип `T`:

```
tmplParamIsReference<int&>(r);
tmplParamIsReference<int&>(i);
```

Это действие может коренным образом изменить поведение шаблона, а поскольку, скорее всего, шаблон разрабатывался без учета такой возможности, это может привести к ошибочному или непредвиденному поведению. Рассмотрим следующий пример:

basics/referror1.cpp

```
template < typename T, T Z = T{} >
class RefMem
{
private:
    T zero;
public:
    RefMem() : zero{Z}
    {
    }
};
```

```

int null = 0;

int main()
{
    RefMem<int> rml, rm2;
    rml = rm2;           // OK
    RefMem<int&> rm3; // Ошибка: неверное значение по умолчанию для N
    RefMem<int&,0>rm4;// Ошибка: неверное значение по умолчанию для N
    extern int null;
    RefMem<int&, null> rm5, rm6;
    rm5 = rm6;          // Ошибка: operator= удален из-за члена-ссылки
}

```

Здесь у нас есть класс с членом, имеющим тип параметра шаблона *T*, который инициализируется нетиповым параметром шаблона *Z*, который по умолчанию имеет нулевое значение. Создание экземпляра класса с типом *int* работает, как и ожидалось. Однако при попытке инстанцировать его со ссылкой ситуация усложняется:

- инициализация по умолчанию больше не работает;
- вы больше не можете просто передать 0 в качестве инициализатора для *int*;
- и, что, вероятно, самое удивительное, оператор присваивания больше недоступен, поскольку классы с нестатическими членами-ссылками по умолчанию имеют удаленный оператор присваивания.

Кроме того, использование ссылочных типов для параметров шаблонов, не являющихся типами, оказывается сложным и может быть опасным. Рассмотрим следующий пример:

basics/referror2.cpp

```

#include <vector>
#include <iostream>

template<typename T, int& SZ> // Примечание: размер является ссылкой
class Arr
{
private:
    std::vector<T> elems;
public:
    Arr() : elems(SZ)           // SZ - начальный размер вектора
    {
    }
    void print() const
    {
        for (int i = 0; i < SZ; ++i) // Цикл по SZ элементам
        {
            std::cout << elems[i] << ' ';
        }
    }
};

```

```

int size = 10;

int main()
{
    Arr<int&, size> y; // Ошибка времени компиляции глубоко
                         // в коде класса std::vector<>
    Arr<int, size> x;  // Инициализация внутреннего вектора
                         // с 10 элементами
    x.print();          // OK
    size += 100;        // Ой: изменение SZ в Arr<>
    x.print();          // Ошибка времени выполнения: некорректное
                         // обращение к памяти: цикл по 110 элементам
}

```

Здесь попытка инстанцирования `Arr` для элементов ссылочного типа приводит к ошибке глубоко в коде класса `std::vector<>`, потому что он не может быть создан с ссылками в качестве элементов:

```
Arr<int&, size> y; // Ошибка времени компиляции глубоко
                     // в коде класса std::vector<>
```

Такая ошибка часто приводит к “романам”, описанным в разделе 9.4, в которых компилятор выводит всю историю инстанцирования шаблона от исходного шаблона до фактического шаблона, в котором была обнаружена ошибка.

Возможно, еще хуже — ошибка времени выполнения, связанная с тем, что параметр размера является ссылкой: он позволяет изменяться записанному значению размера без уведомления об этом контейнера (т.е. значение размера может стать некорректным). Таким образом, операции с использованием размера (такие как член `print()`) будут демонстрировать неопределенное поведение (приводя программу к аварийному завершению или к чему похуже):

```
Arr<int, size> x; // Инициализация внутреннего вектора
                     // с 10 элементами
x.print();          // OK
size += 100;        // Ой: изменение SZ в Arr<>
x.print();          // Ошибка времени выполнения: некорректное
                     // обращение к памяти: цикл по 110 элементам
```

Обратите внимание на то, что изменение параметра шаблона `SZ` на `int const&` проблему не решает, потому что сам `size` остается изменяемым.

Возможно, этот пример надуманный. Однако такие вопросы реально возникают в более сложных ситуациях. Кроме того, в C++17 параметры, не являющиеся типами, можно выводить, например:

```
template<typename T, decltype(auto) SZ>
class Arr;
```

С помощью `decltype(auto)` можно легко создать ссылочные типы и, таким образом, их следует вообще избегать в данном контексте (по умолчанию используя `auto`). Подробности представлены в разделе 15.10.3.

По этой причине стандартная библиотека C++ содержит некоторые удивительные спецификации и ограничения, как, например, приведенные далее.

- Для того чтобы оператор присваивания существовал, даже если параметры шаблона инстанцируются для ссылок, классы `std::pair<>` и `std::tuple<>` реализуют операторы присваивания вместо использования поведения по умолчанию. Например:

```
namespace std
{
    template<typename T1, typename T2>
    struct pair
    {
        T1 first;
        T2 second;

        /**
         * Копирующий/перемещающий конструкторы
         * normally работают со ссылками...
         pair(pair const&) = default;
         pair(pair&&) = default;

        /**
         * ...но оператор присваивания должен быть
         * определен и для работы со ссылками:
         pair& operator=(pair const& p);
         pair& operator=(pair&& p) noexcept(...);

        ...
    };
}
```

- Из-за сложности возможных побочных эффектов инстанцирование шаблонов классов стандартной библиотеки C++17 `std::optional<>` и `std::variant<>` для ссылочных типов является некорректным действием (по крайней мере в C++17).

Для запрета ссылок достаточно простого статического утверждения:

```
template<typename T>
class optional
{
    static_assert(!std::is_reference<T>::value,
                 "Некорректное инстанцирование "
                 "optional<T> для ссылки");
    ...
};
```

Ссылочные типы в целом довольно сильно отличаются от других типов, и к ним применяются несколько уникальных языковых правил. Они воздействуют, например, на объявления параметров вызова (см. раздел 7), а также на способы определения свойств типов (см. раздел 19.6.1).

11.5. Откладывание вычислений

При реализации шаблонов иногда возникает вопрос, может ли код справиться с неполными типами (см. раздел 10.3.1). Рассмотрим следующий шаблон класса:

```
template<typename T>
class Cont
```

```
{
    private:
        T* elems;
    public:
        ...
};
```

До сих пор этот класс мог использоваться с неполными типами. Это полезно, например, в случае классов, которые ссылаются на элементы своего собственного типа:

```
struct Node
{
    std::string value;
    Cont<Node> next; // Возможно только если Cont
}; // принимает неполные типы
```

Однако, например, простого использования некоторых свойств может быть достаточно, чтобы потерять возможность работы с неполными типами. Например:

```
template<typename T>
class Cont
{
    private:
        T* elems;
    public:
        ...
        typename std::conditional<std::is_move_constructible<T>::value,
                                T&&,
                                T&
                            >::type
        foo();
};
```

Здесь мы используем свойство `std::conditional` (см. раздел Г.5) для выяснения, является ли возвращаемым типом функции-члена `foo()` тип `T&&` или `T&`. Решение зависит от того, поддерживает ли тип параметра шаблона `T` семантику перемещения.

Проблема заключается в том, что свойство `std::is_move_constructible` требует, чтобы ее аргумент был полным типом (а также не типом `void` и не массивом с неизвестными границами; см. раздел Г.3.2). Поэтому при таком объявлении `foo()` объявление `struct node` является некорректным⁶.

Мы можем решить эту проблему, заменив `foo()` шаблоном члена так, что вычисление `std::is_move_constructible` откладывается до точки инстанцирования `foo()`:

```
template<typename T>
class Cont
{
    private:
```

⁶ Не все компиляторы сообщают об ошибке, если `std::is_move_constructible` не является неполным типом. Это разрешено, потому что для такого рода ошибок диагностика не требуется. Таким образом, это как минимум является проблемой переносимости.

```

T* elems;
public:
    template<typename D = T>
    typename std::conditional<std::is_move_constructible<D>::value,
                                T&&, T& >::type
        foo();
};

```

Теперь свойства зависят от параметра шаблона D (по умолчанию равного T, значению, которое нам и нужно в конечном итоге), и компилятор должен ждать до тех пор, пока функция foo() не будет вызвана для конкретного типа наподобие Node перед вычислением свойств (тогда Node представляет собой полный тип; он является неполным только во время определения).

11.6. О чем следует подумать при написании обобщенных библиотек

Перечислим некоторые вещи, о которых следует помнить при реализации обобщенных библиотек (обратите внимание на то, что о некоторых из них может быть рассказано позже в этой книге).

- Используйте передаваемые ссылки для передачи значений в шаблоны (см. раздел 6.1). Если значения не зависят от параметров шаблона, используйте auto&& (см. раздел 11.3).
- Когда параметры объявлены как передаваемые ссылки, будьте готовы к тому, что параметр шаблона будет иметь ссылочный тип при передаче l-значений (см. раздел 15.6.2).
- Используйте std::addressof(), когда необходимо указать адрес объекта, зависящего от параметра шаблона, чтобы избежать сюрпризов, когда он связан с типом с перегруженным оператором operator& (см. раздел 11.2.2).
- Используя шаблоны функций-членов, убедитесь, что их соответствие не лучше, чем предопределенные копирующий/перемещающий конструктор или оператор присваивания (раздел 6.4).
- Подумайте об использовании std::decay, когда параметры шаблона могут быть строковыми литералами, не передающимися по значению (раздел 7.4 и раздел Г.4).
- При наличии *выходных* параметров, зависящих от параметров шаблона, будьте готовы справиться с ситуацией, когда могут быть указаны константные аргументы шаблона (см., например, раздел 7.2.2).
- Будьте готовы справиться с побочными эффектами параметров шаблона, которые представляют собой ссылки (см. подробности в разделе 11.4 и пример в разделе 19.6.1). В частности, может потребоваться гарантировать, что возвращаемый тип не может стать ссылкой (см. раздел 7.5).

- Будьте готовы к работе с неполными типами для поддержки, например, рекурсивных структур данных (см. раздел 11.5).
- Выполните перегрузки для всех типов массивов, а не только $T[SZ]$ (см. раздел 5.4).

11.7. Резюме

- Шаблоны позволяют передавать в качестве *вызываемых объектов* функции, указатели на функции, функциональные объекты, функторы и лямбда-выражения.
- При определении классов с перегруженным оператором `operator()` объявит его как `const` (если только вызов не изменяет состояние объекта).
- С использованием `std::invoke()` можно реализовать код, который сможет обрабатывать все вызываемые объекты, включая функции-члены.
- Используйте `decltype(auto)` для прямой передачи возвращаемого значения.
- Свойства типов представляют собой функции, проверяющие свойства и возможности типов.
- Используйте `std::addressof()`, когда вам нужен адрес объекта в шаблоне.
- Используйте `std::declval()` для создания значений определенных типов в невычисляемых выражениях.
- Используйте `auto&&` для прямой передачи объектов в обобщенный код, если их тип не зависит от параметров шаблонов.
- Будьте готовы иметь дело с побочными эффектами параметров шаблонов, являющихся ссылками.
- Шаблоны можно использовать для откладывания вычислений выражений (например, для поддержки использования неполных типов в шаблонах класса).

ЧАСТЬ II

Углубленное изучение шаблонов

Первая часть данной книги представляет собой учебное пособие по основным концепциям языка, на которых базируются шаблоны C++. Содержащегося в ней материала вполне достаточно для ответа на большинство вопросов, возникающих при обычном практическом программировании на C++. Вторая часть книги организована в виде справочника — в ней содержатся ответы на менее типичные вопросы, которые могут возникнуть при использовании расширенных средств языка для достижения более сложных и интересных эффектов при программировании. По желанию, при первом чтении книги эту часть можно пропустить, возвращаясь к определенным темам по ссылкам в ходе изучения следующих глав или при поиске терминов в предметном указателе.

Наша цель — сделать материал книги более понятным и полным, сохраняя при этом сжатый характер его изложения. Поэтому приведенные в ней примеры являются короткими и зачастую до известной степени искусственными. Это сделано для того, чтобы не уклоняться в сторону от рассматриваемой темы и не затрагивать вопросов, которые к ней не относятся.

Кроме того, здесь освещены будущие изменения и расширения языка шаблонов в C++.

Данная часть книги включает перечисленные ниже темы.

- Базовые вопросы, касающиеся объявлений шаблонов.
- Значение имен в шаблонах.
- Механизм инстанцирования шаблонов C++.
- Правила вывода аргументов шаблонов.
- Специализация и перегрузка.
- Будущие возможности.

Глава 12

Вглубь шаблонов

В этой главе дается *более глубокий* обзор основных понятий из области шаблонов, с которыми читатель познакомился в первой части книги. Речь идет об объявлениях шаблонов, ограничениях, накладываемых на параметры и аргументы шаблонов и т.п.

12.1. Параметризованные объявления

В настоящее время в C++ поддерживаются четыре основных типа шаблонов — шаблоны классов, шаблоны функций, шаблоны переменных и шаблоны псевдонимов. Шаблоны каждой из этих разновидностей могут находиться как в области видимости пространства имен, так и в области видимости класса. В области видимости класса они становятся вложенными шаблонами классов, шаблонами функций-членов, шаблонами статических данных-членов и шаблонами псевдонимов-членов. Такие шаблоны объявляются почти так же, как и обычные классы, функции, переменные и псевдонимы типов (или их двойники — члены классов), за исключением того, что для шаблонов указывается *выражение параметризации* (parameterization clause) вида

```
template<Параметры>
```

Обратите внимание на наличие в C++17 еще одной конструкции, которая вводится с таким же выражением параметризации: *правила вывода* (deduction guides) (см. раздел 2.9 и раздел 15.12.1). В данной книге мы не называем их *шаблонами* (например, они не инстанцируются), но их синтаксис напоминает шаблоны функций.

К объявлениям фактических параметров мы вернемся в последующих разделах, а сейчас рассмотрим несколько примеров, в которых проиллюстрированы четыре указанные разновидности шаблонов. Они могут находиться в *области видимости пространства имен* (глобальной или пространства имен), как показано ниже.

```
details/definitions1.hpp
```

```
template<typename T> // Шаблон класса
class Data
{
public:
    static constexpr bool copyable = true;
    ...
};

template<typename T> // Шаблон функции
void log(T x)
```

```

{
    ...
}

template<typename T> // Шаблон переменной (начиная с C++14)
T zero = 0;

template<typename T> // Шаблон переменной (начиная с C++14)
bool DataCopyable = Data<T>::copyable;

template<typename T> // Шаблон псевдонима
using DataList = Data<T*>;

```

Обратите внимание на то, что в этом примере статический член `Data<T>::copyable` не является шаблоном переменной, несмотря на то, что он параметризован косвенно через параметризацию шаблона класса `Data`. Однако шаблон переменной может находиться в области видимости класса (как показывает следующий пример), и в этом случае это шаблон статического члена-данного.

В следующем примере показаны четыре разновидности шаблонов в виде членов класса, определенных в своем родительском классе:

```

details/definitions2.hpp
class Collection
{
public:
    template<typename T> // Определение шаблона класса-члена
    class Node           // в пределах класса
    {
        ...
    };

    template<typename T> // Определение шаблона функции-члена в
    T* alloc()          // пределах класса (и потому неявно
    {                  // являющейся inline)
        ...
    }

    template<typename T> // Шаблон переменной-члена (с C++14)
    static T zero = 0;

    template<typename T> // Шаблон псевдонима-члена
    using NodePtr = Node<T>* ;
};

```

Обратите внимание, что в C++17 переменные (включая статические члены-данные) и шаблоны переменных могут быть “встраиваемыми” (`inline`), что означает, что их определение может быть повторено в нескольких единицах трансляции. Это избыточно для шаблонов переменных, которые всегда могут быть определены в нескольких единицах трансляции. Однако в отличие от функций-членов, определение статического члена-данных в его классе не делает его встраиваемым: ключевое слово `inline` должно быть указано во всех случаях.

Наконец, в следующем коде показано, как шаблоны членов, которые не являются шаблонами псевдонимов, могут быть определены вне класса.

details/definitions3.hpp

```
template<typename T> // Шаблон класса области
class List // видимости пространства имен
{
public:
    List() = default; // Поскольку определен
                       // шаблонный конструктор

    template<typename U> // Еще один шаблон класса,
    class Handle; // без определения

    template<typename U> // Шаблон функции-члена
    List(List<U> const&); // (конструктор)

    template<typename U> // Шаблон переменной-члена (с C++14)
    static U zero;
};

template<typename T> // Определение шаблона класса-члена вне класса
template<typename U>
class List<T>::Handle
{
    ...
};

template<typename T> // Определение шаблона функции-члена вне класса
template<typename T2>
List<T>::List(List<T2> const& b)
{
    ...
}

template<typename T> // Определение шаблона члена-данных вне класса
template<typename U>
U List<T>::zero = 0;
```

Шаблоны-члены класса, определенные вне пределов охватывающего их класса, могут требовать несколько конструкций параметризации `template<...>`: одну для самого шаблона и по одной для каждого охватывающего шаблона класса. Конструкции перечисляются, начиная с самого внешнего шаблона класса.

Обратите также внимание на то, что шаблон конструктора (особый вид шаблона функции-члена) отключает неявное объявление конструктора по умолчанию (потому что он неявно объявляется только тогда, когда никакой иной конструктор не объявлен). Добавление объявления по умолчанию

`List() = default;`

гарантирует, что экземпляр `List<T>` конструируется по умолчанию с семантикой неявно объявленного конструктора.

Шаблоны объединений

Шаблоны объединений (*union templates*) также возможны (они трактуются как разновидность шаблона класса):

```
template<typename T>
union AllocChunk
{
    T object;
    unsigned char bytes[sizeof(T)];
};
```

Аргументы вызова по умолчанию

Шаблоны функций могут иметь аргументы вызова по умолчанию, как и обычные объявления функций:

```
template<typename T>
void report_top(Stack<T> const&, int number = 10);

template<typename T>
void fill(Array<T>&, T const& = T{}); // T{} для встроенных
                                         // типов равно нулю
```

Последнее объявление показывает, что аргумент вызова по умолчанию может зависеть от параметров шаблона. Он также может быть определен как (единственный доступный до C++11 способ показан в разделе 5.2)

```
template<typename T>
void fill(Array<T>&, T const& = T()); // T() для встроенных
                                         // типов равно нулю
```

При вызове функции `fill()` аргумент по умолчанию не инстанцируется, если указан второй аргумент вызова функции. Это гарантирует, что сообщение об ошибке не будет выдано, если аргумент вызова по умолчанию не может быть инстанцирован для конкретного `T`. Например:

```
class Value
{
public:
    explicit Value(int); // Нет конструктора по умолчанию
};

void init(Array<Value>& array)
{
    Value zero(0);
    fill(array, zero); // OK: конструктор по
                       // умолчанию не используется
    fill(array);       // Ошибка: использован не определенный
                       // конструктор по умолчанию для Value
}
```

Нешаблонные члены шаблонов классов

Помимо четырех основных типов шаблонов, объявленных внутри класса, могут также иметься обычные члены класса, параметризованные в силу того, что

они являются частью шаблона класса. Иногда их (ошибочно) также называют *шаблонами членов*. Хотя они могут быть параметризованы, такие определения не являются “шаблонами первого класса”. Их параметры полностью определяются шаблоном, членами которого они являются. Например:

```
template<int I>
class CupBoard
{
    class Shelf; // Обычный класс в шаблоне класса
    void open(); // Обычная функция в шаблоне класса
    enum Wood : unsigned char; // Обычное перечисление
                                // в шаблоне класса
    static double totalWeight; // Обычный статический
                                // член-данные в шаблоне класса
};
```

Соответствующие определения указывают параметризацию только для шаблонов родительского класса, но не для самого члена, поскольку он шаблоном не является (то есть с его именем после последнего `::` не связана никакая конструкция параметризации):

```
template<int I> // Определение обычного класса в шаблоне класса
class CupBoard<I>::Shelf
{
    ...
};

template<int I> // Определение обычной функции в шаблоне класса
void CupBoard<I>::open()
{
    ...
}

template<int I>           // Определение обычного перечисления
enum CupBoard<I>::Wood   // в шаблоне класса
{
    Maple, Cherry, Oak
};

template<int I>           // Определение обычного статического
double CupBoard<I>::totalWeight = 0.0; // члена в шаблоне класса
```

Начиная с C++17, статический член `totalWeight` может быть инициализирован внутри шаблона класса с использованием `inline`:

```
template<int I>
class CupBoard
{
    ...
    inline static double totalWeight = 0.0;
};
```

Хотя такие параметризованные определения обычно называются *шаблонами*, это не совсем верный термин. Для таких сущностей был предложен термин *шаблоид* (*temploid*). Начиная с C++17, стандарт C++ определяет понятие *шаблонной сущности* (*templated entity*), которая включает шаблоны и шаблоиды, а также

рекурсивно все сущности, определенные или созданные в шаблонных сущностях (что включает в себя, например, дружественную функцию, определенную внутри шаблона класса (см. раздел 2.4) или тип замыкания лямбда-выражения, содержащегося в шаблоне). Пока что эти термины не получили широкого распространения, но они могут быть полезны в будущем для более точного обмена информацией о шаблонах в C++.

12.1.1. Виртуальные функции-члены

Шаблоны функций-членов не могут быть объявлены как виртуальные. Это ограничение накладывается потому, что в обычной реализации механизма вызова виртуальных функций используется таблица фиксированного размера, одна строка которой соответствует одной виртуальной функции. Однако число инстанцированных шаблонов функции-члена не является фиксированным, пока не завершится трансляция всей программы. Следовательно, для того, чтобы поддержка шаблонов виртуальных членов-функций стала возможной, требуется реализация радикально нового вида механизма позднего связывания в компиляторах и компоновщиках C++.

В отличие от функций-членов, обычные члены шаблонов классов могут быть виртуальными, поскольку их количество при инстанцировании класса фиксировано.

```
template<typename T>
class Dynamic
{
public:
    virtual ~Dynamic();      // OK: один деструктор на
                            // экземпляр Dynamic<T>
    template<typename T2>
    virtual void copy(T2 const&); // Ошибка: неизвестное количество экземпляров copy()
                                // у одного экземпляра Dynamic<T>
};
```

12.1.2. Связывание шаблонов

Каждый шаблон должен иметь имя, и это имя должно быть уникальным в пределах его области видимости, за исключением шаблонов функций, которые могут быть перегружены (см. главу 16, “Специализация и перегрузка”). Особо отметим, что, в отличие от типов классов, для шаблонов классов не допускается использование имен, совпадающих с именами объектов других сущностей:

```
int C;
...
class C; // OK: имена классов и не классов в разных "пространствах"

int X;
...
template<typename T>
class X; // Ошибка: конфликт с именем переменной X
```

```
struct S;
...
template<typename T>
class S; // Ошибка: конфликт с именем структуры S
```

Имена шаблонов имеют связывание, но они не могут иметь *связывание C*. Нестандартные связывания могут иметь значения, зависящие от реализации (однако нам неизвестна реализация, которая поддерживала бы нестандартные правила связывания имен для шаблонов).

```
extern "C++" template<typename T>
void normal(); // По умолчанию спецификация
               // связывания может быть опущена

extern "C" template<typename T>
void invalid(); // Ошибка: шаблоны не могут иметь связывание C

extern "Java" template<typename T>
void javaLink(); // Нестандартно, но, возможно, когда-то появится
                 // какой-то компилятор, поддерживающий связывание,
                 // совместимое с дженериками Java
```

Шаблоны обычно имеют внешнее связывание. Исключением являются шаблоны функций в области видимости пространства имен, описанные как `static`, шаблоны, которые являются прямыми или косвенными членами безымянного пространства имен (которое имеет внутреннее связывание) и шаблоны членов безымянных классов (которые не имеют связывания). Например:

```
template<typename T> // Ссылается на ту же сущность, что и
void external(); // объявление с тем же именем (и областью
                  // видимости) в другом файле

template<typename T> // Не связано с шаблоном с тем же именем
static void internal(); // в другом файле

template<typename T> // Повторное объявление предыдущего шаблона
static void internal();

namespace
{
    template<typename> // Также не связано с шаблоном с тем же
    void otherInternal(); // именем в другом файле, даже с тем,
} // который находится в аналогичном
   // безымянном пространстве имен

namespace
{
    template<typename> // Повторное объявление
    void otherInternal(); // предыдущего шаблона
}

struct
{
    template<typename T> // Связывания нет:
```

```
void f(T) {} // не может быть повторно объявлен
} x;
```

Обратите внимание: поскольку последний шаблон члена не имеет связывания, он должен быть определен в безымянном классе, поскольку нет никакого способа обеспечить его определение вне класса.

В настоящее время шаблоны не могут быть объявлены в области видимости функции или локального класса, но обобщенные лямбда-выражения (см. раздел 15.10.6), которые имеют связанные типы замыкания, содержащие шаблоны функций-членов, могут находиться в локальных областях видимости, что подразумевает наличие разновидности локальных шаблонов функций-членов.

Связывание экземпляра шаблона такое же, что и у шаблона. Например, функция `internal<void>()`, инстанцированная из объявленного выше шаблона `internal`, будет иметь внутреннее связывание. Это имеет интересные следствия в случае шаблонов переменных. Рассмотрим следующий пример:

```
template<typename T> T zero = T{};
```

Все инстанцирования `zero` имеют внешнее связывание, даже такие как `zero <int const>`. Это может показаться парадоксальным, учитывая, что

```
int const zero_int = int{};
```

дает внутреннее связывание `zero_int`, поскольку она объявлена с константным типом. Аналогично все инстанцирования шаблона

```
template<typename T> int const max_volume = 11;
```

имеют внешнее связывание, несмотря на то, что все они также имеют тип `int const`.

12.1.3. Первичные шаблоны

С помощью обычных конструкций объявлений шаблонов объявляются так называемые *первичные шаблоны* (primary templates). В таких объявлениях после имени отсутствуют аргументы шаблона в угловых скобках:

```
// OK: первичный шаблон
template<typename T> class Box;

// Ошибка: не специализируется
template<typename T> class Box<T>;

// OK: первичный шаблон
template<typename T> void translate(T);

// Ошибка: не разрешено для функций
template<typename T> void translate<T>(T);

// OK: первичный шаблон
template<typename T> constexpr T zero = T{};

// Ошибка: не специализируется
template<typename T> constexpr T zero<T> = T{};
```

Вторичные (не первичные) шаблоны классов получаются при объявлении *частичных специализаций* шаблонов классов или переменных, которые рассматриваются в главе 16, “Специализация и перегрузка”. Шаблоны функций всегда должны быть первичными (см. раздел 17.3, где рассмотрены возможные изменения в этой области в будущем).

12.2. Параметры шаблонов

Существует три основных вида параметров шаблонов.

1. Параметры типа (типовые параметры) (они на сегодняшний день используются наиболее часто).
2. Параметры, не являющиеся типами (нетиповые параметры).
3. Шаблонные параметры шаблонов.

Любой из этих основных видов параметров шаблона может быть использован в качестве основы *пакета параметров шаблона* (template parameter pack) (см. раздел 12.2.4).

Параметры шаблона объявлены в начальном операторе параметризации объявления шаблона¹. Такие объявления не обязательно должны быть именованными:

```
template<typename, int>
class X; // X<> параметризован типом и целочисленным значением
```

Имя параметра, конечно же, необходимо, если на этот параметр имеется ссылка позже в шаблоне. Обратите также внимание, что на имя параметра шаблона можно ссылаться в объявлении последующего параметра (но не до самого рассматриваемого параметра):

```
template<typename T,           // Первый параметр используется
         T Root,             // в объявлениях второго и
         template<T> class Buf> // третьего параметров
class Structure;
```

12.2.1. Параметры типа

Параметры типа (типовые параметры) вводятся с помощью ключевых слов *typename* либо *class*; оба варианта эквивалентны². За ключевым словом должен следовать простой идентификатор, за которым идет запятая, означающая начало следующего объявления параметра, закрывающая угловая скобка (*>*) для обозначения конца параметризованного выражения или знак равенства (=) для обозначения начала заданного по умолчанию аргумента шаблона.

¹ Начиная с C++14 исключением являются неявные параметры типа шаблона для обобщенных лямбда-выражений; см. раздел 15.10.6.

² Ключевое слово *class* не означает, что подставляемый параметр должен иметь тип класса. Это может быть любой доступный тип.

В пределах объявления шаблона параметр типа ведет себя подобно *псевдониму типа* (см. раздел 2.8). Например, нельзя использовать имя вида `class T`, где `T` является параметром шаблона, даже если вместо `T` подставляется тип класса.

```
template<typename Allocator>
class List
{
    // Ошибка: использование "Allocator* allocptr":
    class Allocator* allocptr;

    // Ошибка: использование "friend Allocator"
    friend class Allocator;

    ...
};
```

12.2.2. Параметры, не являющиеся типами

Не являющиеся типами параметры (нетиповые параметры) — это константные значения, которые могут быть определены во время компиляции или при компоновке³. Тип такого параметра (другими словами, тип значения, которое он обозначает) должен быть одним из следующих:

- целочисленный тип или тип перечисления;
- тип указателя⁴;
- тип указателя на член;
- тип l-ссылки (разрешены как ссылка на объект, так и ссылка на функцию);
- `std::nullptr_t`;
- тип, содержащий `auto` или `decltype(auto)` (только начиная с C++17; см. раздел 15.10.1).

В данный момент все прочие типы исключены (хотя в будущем возможно добавление типов с плавающей точкой; см. раздел 17.2).

Возможно, это покажется несколько неожиданным, но объявление параметра шаблона, не являющегося типом, в некоторых случаях также может начинаться с ключевого слова `typename`:

```
template<typename T,
         typename T::Allocator* Allocator> // Параметр типа
class List; // Параметр, не
// являющийся типом
```

или с ключевого слова `class`:

³ Шаблонные параметры шаблона также не обозначают типы, однако они не рассматриваются в качестве параметров, не являющихся типами. Эта странность вызвана историческими причинами: шаблонные параметры шаблонов были добавлены в язык после типовых и нетиповых параметров.

⁴ На момент написания книги разрешались только типы “указатель на объект” и “указатель на функцию”, что исключает применение типов наподобие `void*`. Однако все компиляторы принимают также и `void*`.

```
template<class X*> // Параметр, не являющийся типом,
class Y;           // и имеющий тип указателя
```

Разницу здесь увидеть легко: в первом случае за ключевым словом следует простой идентификатор, а затем один из небольшого набора лексем (“=” для аргумента по умолчанию, “,” для указания того, что далее следует другой параметр шаблона, или закрывающая угловая скобка “>”, завершающая список параметров шаблона). В разделах 5.1 и 13.3.2 объясняется необходимость ключевого слова `typename` в первом параметре, не являющемся типом.

Возможно использование типов функций и массивов, но они неявно низводятся к типу указателя:

```
template<int buf[5]> class Lexer;      // В действительности int*
template<int* buf> class Lexer;       // OK: повторное объявление
template<int fun()> struct FuncWrap; // fun представляет собой
                                      // тип указателя на функцию
template<int (*)()> struct FuncWrap; // OK: повторное объявление
```

Параметры, не являющиеся типами, объявляются почти так же, как и переменные, но они не могут включать спецификаторы, не являющиеся типами, такие как `static`, `mutable` и т.д. Возможно использование модификаторов `const` или `volatile`, но если такой модификатор появляется у нетиповых параметров внешнего уровня вложенности, он попросту игнорируется:

```
template<int const length> class Buffer; // const здесь бесполезен
template<int length> class Buffer;       // То же самое объявление
```

И наконец, параметры, не являющиеся типами, всегда являются *пр-значениями*⁵. Их адрес нельзя получить, и им нельзя ничего присвоить. С другой стороны, параметр, не являющийся типом и имеющий тип l-ссылки, может использоваться для описания l-значения:

```
template<int& Counter>
struct LocalIncrement
{
    LocalIncrement()
    {
        Counter = Counter+1;    // OK: Ссылка на int
    }
    ~LocalIncrement()
    {
        Counter = Counter-1;
    }
};
```

R-ссылка в качестве параметра не разрешена.

12.2.3. Шаблонные параметры шаблонов

Шаблонные параметры шаблонов являются символами-заместителями для шаблонов классов или псевдонимов. Они объявляются во многом подобно

⁵Обсуждение категорий значений, таких как r-значения или l-значения, приведено в приложении Б, “Категории значений”.

шаблонам классов, однако при этом нельзя использовать ключевые слова `struct` и `union`:

```
template<template<typename X> class C> // OK
void f(C<int>* p);

template<template<typename X> struct C> // Ошибка: struct
void f(C<int>* p);

template<template<typename X> union C> // Ошибка: union
void f(C<int>* p);
```

C++17 позволяет использовать ключевое слово `typename` вместо `class`: это изменение было мотивировано тем, что шаблонные параметры шаблона можно заменять не только шаблонами класса, но и шаблонами псевдонимов (которые инстанцируются для произвольных типов). Так, в C++17 приведенный выше пример может быть записан как

```
template<template<typename X> typename C> // OK начиная с C++17
void f(C<int>* p);
```

В области видимости их объявлений шаблонные параметры шаблонов используются так же, как и другие шаблоны классов или псевдонимов.

Параметры шаблонных параметров шаблонов могут иметь аргументы, заданные по умолчанию. Эти аргументы применяются в том случае, когда при использовании шаблонного параметра шаблона соответствующие параметры не указаны:

```
template<template<typename T,
              typename A = MyAllocator> class Container>
class Adaptation
{
    Container<int> storage; // Неявно эквивалентно
                           // Container<int,MyAllocator>
    ...
};
```

`T` и `A` являются именами параметров шаблона шаблонного параметра шаблона `Container`. Эти имена используются только в объявлении других параметров этого шаблонного параметра шаблона. Следующий надуманный шаблон иллюстрирует данную концепцию:

```
template<template<typename T, T*> class Buf> // OK
class Lexer
{
    static T* storage; // Ошибка: здесь не может быть использован
                       // параметр шаблонного параметра шаблона
    ...
};
```

Однако обычно имена параметров шаблона шаблонного параметра шаблона не используются, поэтому им зачастую вообще не присваиваются никакие имена. Например, рассмотренный выше шаблон `Adaptation` можно объявить следующим образом:

```
template<template<typename,
              typename = MyAllocator> class Container>
class Adaptation
{
    Container<int> storage; // Неявный эквивалент
    ...                      // Container<int,MyAllocator>
};
```

12.2.4. Пакеты параметров шаблонов

Начиная с C++11, параметр шаблона любого вида может быть превращен в *пакет параметров шаблона* путем добавления многоточия (...) перед именем параметра шаблона или, если параметр шаблона безымянный, там, где должно находиться имя параметра шаблона:

```
template<typename... Types> // Объявление пакета параметров
class Tuple;                // шаблона с именем Types
```

Пакет параметров шаблона ведет себя как базовый параметр шаблона, но с важным отличием: в то время как обычный параметр шаблона соответствует ровно одному аргументу шаблона, пакет параметров шаблона может соответствовать любому количеству аргументов шаблона. Это означает, что шаблон класса Tuple, объявленный выше, принимает любое количество (возможно различных) типов в качестве аргументов шаблона:

```
using IntTuple      = Tuple<int>;           // OK: один аргумент шаблона
using IntCharTuple = Tuple<int,char>;        // OK: два аргумента шаблона
using IntTriple    = Tuple<int,int,int>;       // OK: три аргумента шаблона
using EmptyTuple   = Tuple<>;                 // OK: нет аргументов шаблона
```

Аналогично пакеты параметров шаблонов, не являющиеся типами, и шаблонные параметры шаблонов также могут принимать любое количество соответствующих аргументов шаблона:

```
// OK: объявление пакета параметров шаблонов, не являющихся типами:
template<typename T, unsigned... Dimensions>
class MultiArray;

// OK: матрица размера 3x3
using TransformMatrix = MultiArray<double, 3, 3>;

// OK: объявление пакета шаблонных параметров шаблонов
template<typename T, template<typename, typename>... Containers>
void testContainers();
```

Пример MultiArray требует, чтобы все аргументы шаблона, не являющиеся типами, были одного и того же типа unsigned. Начиная с C++17, вводится возможность вывода аргументов шаблона, не являющихся типами, которая позволяет нам в определенной мере обойти это ограничение (подробнее об этом см. раздел 15.10.1).

Первичные шаблоны классов, шаблоны переменных и шаблоны псевдонимов могут иметь не более одного пакета параметров шаблона, и, если он присутствует,

он должен быть последним параметром шаблона. Шаблоны функции имеют более слабые ограничения: разрешены множественные пакеты параметров шаблонов, лишь бы каждый параметр шаблона после пакета параметров шаблона либо имел значение по умолчанию (см. следующий раздел), либо мог быть выведен (см. главу 15, “Вывод аргументов шаблона”):

```
// Ошибка: пакет параметров шаблона не является последним
template<typename... Types, typename Last>
class LastType;

// OK: за пакетом параметров шаблона следует
// выводимый параметр шаблона
template<typename... TestTypes, typename T>
void runTests(T value);

template<unsigned...> struct Tensor;
template<unsigned... Dims1, unsigned... Dims2>
auto compose(Tensor<Dims1...>, Tensor<Dims2...>);
// OK: размерности тензора могут быть выведены
```

Последний пример представляет собой объявление функции с выводимым возвращаемым типом (возможность, введенная в C++14), см. также раздел 15.10.1.

Объявления частичных специализаций шаблонов классов и переменных (см. главу 16, “Специализация и перегрузка”) *могут* иметь несколько пакетов параметров, в отличие от их прототипа — первичного шаблона. Дело в том, что частичная специализация выбирается с помощью процесса выведения, который практически идентичен используемому для шаблонов функций.

```
template<typename...> Typelist;
template<typename X, typename Y> struct Zip;
template<typename... Xs, typename... Ys>
struct Zip<Typelist<Xs...>, Typelist<Ys...>>;
    // OK: частичная специализация использует вывод
    // для определения подстановок X и Y
```

Вероятно, не будет сюрпризом, что пакет параметров типов не может быть раскрыт в своем же выражении списка параметров. Например:

```
template<typename... Ts, Ts... vals> struct StaticValues {};
// Ошибка: Ts не может использоваться в собственном списке параметров
```

Однако вложенные шаблоны могут создавать корректные подобные ситуации:

```
template<typename... Ts> struct ArgList
{
    template<Ts... vals> struct Vals {};
};
ArgList<int, char, char>::Vals<3, 'x', 'y'> tada;
```

Шаблон, содержащий пакет параметров шаблона, называется *вариативным шаблоном* (variadic template), потому что он принимает переменное количество аргументов шаблона. Применение вариативных шаблонов описывается в главе 4, “Вариативные шаблоны”, и разделе 12.4.

12.2.5. Аргументы шаблона по умолчанию

Параметры шаблона любого вида, не являющиеся пакетом параметров шаблона, могут быть снабжены аргументами по умолчанию, но при этом аргумент по умолчанию должен соответствовать разновидности параметра (например, параметр типа не может иметь аргумент по умолчанию, не являющийся типом). Аргумент, заданный по умолчанию, не может зависеть от собственного параметра, поскольку имя параметра становится доступно только после значения по умолчанию. Однако он может зависеть от предшествующих ему параметров:

```
template<typename T, typename Allocator = allocator<T>>
class List;
```

Параметры шаблона для шаблона класса, переменной или псевдонима могут иметь аргументы по умолчанию только в случае, когда аргументами по умолчанию снабжены также и все последующие параметры. (Аналогичное ограничение имеется для значений аргументов вызова функции по умолчанию.) Последующие значения по умолчанию обычно указываются в том же объявлении шаблона, но они могут также быть объявлены и в предыдущих объявлениях этого шаблона. Сказанное поясняет приведенный ниже пример.

```
template<typename T1, typename T2, typename T3,
         typename T4 = char, typename T5 = char>
class Quintuple; // OK

template<typename T1, typename T2, typename T3 = char,
         typename T4, typename T5>
class Quintuple; // OK: T4 и T5 уже имеют значения по умолчанию

template<typename T1 = char, typename T2, typename T3,
         typename T4, typename T5>
class Quintuple; // Ошибка: T1 не может иметь значение по умолчанию,
                 // так как T2 значения по умолчанию не имеет
```

Аргументы шаблона по умолчанию для параметров шаблонов функций не требуют, чтобы последующие параметры шаблона имели аргументы шаблона по умолчанию⁶:

```
template<typename R = void, typename T>
R * addressof(T& value); // OK: если R не указан явно,
                         // тип R представляет собой void
```

Аргументы шаблона по умолчанию не могут повторяться:

```
template<typename T = void>
class Value;

template<typename T = void>
class Value; // Ошибка: повторяющийся аргумент по умолчанию
```

Ряд контекстов не допускают аргументов шаблонов по умолчанию.

⁶ Аргументы шаблона для последующих параметров шаблона могут быть определены с помощью вывода аргументов шаблонов; см. главу 15, “Вывод аргументов шаблона”.

- Частичные специализации:

```
template<typename T>
class C;

...
template<typename T = int>
class C<T*>; // Ошибка
```

- Пакеты параметров:

```
template<typename... Ts = int> struct X; // Ошибка
```

- Определения членов шаблона класса вне класса:

```
template<typename T> struct X
{
    T f();
};

template<typename T = int> T X<T>::f() // Ошибка
{
    ...
}
```

- Объявление дружественного шаблона класса:

```
struct S
{
    template<typename = void> friend struct F;
};
```

- Объявление дружественного шаблона функции, если только это не определение, и нет его объявления в другом месте единицы трансляции:

```
struct S
{
    // Ошибка: не определение:
    template<typename = void> friend void f();

    // Пока что OK
    template<typename = void> friend void g()
    {
    }
};

template<typename> void g(); // Ошибка: шаблон g()
// получил аргумент шаблона по умолчанию при
// определении; другое его объявление находится
// здесь не может
```

12.3. Аргументы шаблонов

Аргументы шаблонов — это значения, которые подставляются вместо параметров шаблона при инстанцировании последнего. Такие значения можно определять с использованием нескольких различных механизмов.

- Явные аргументы шаблона: за именем шаблона могут следовать явно указанные значения аргументов шаблона, заключенные в угловые скоб-

ки. Полученное в результате имя называется *идентификатором шаблона* (*template-id*).

- Внедренное имя класса: в области видимости шаблона класса X с параметрами шаблона P1, P2, ... имя этого шаблона (X) может быть эквивалентно идентификатору шаблона X<P1, P2, ...>. Более подробно это разъясняется в разделе 13.2.3.
- Аргументы шаблона по умолчанию: при наличии таких аргументов явно указанные аргументы шаблона в экземплярах шаблонов классов могут быть опущены. Однако для шаблона класса или шаблона псевдонима, даже если все параметры шаблона имеют значения по умолчанию, все равно должны быть указаны (возможно, пустые) угловые скобки.
- Вывод аргументов: аргументы шаблонов функций, не указанные явно, могут быть получены путем вывода из типов аргументов вызова функции в ее вызове. Более подробно это описано в главе 15, “Вывод аргументов шаблона”. Вывод также осуществляется и в некоторых других ситуациях. Если все аргументы шаблона могут быть получены путем вывода, указывать угловые скобки после имени шаблона функции не требуется. В стандарте C++17 появилась также возможность вывести аргументы шаблона класса из инициализатора объявления переменной или функциональной записи преобразования типа; см. обсуждение этого вопроса в разделе 15.12.

12.3.1. Аргументы шаблонов функций

Аргументы шаблона функции могут быть заданы явно, получены путем вывода на основе способа использования шаблона или предоставлены в виде аргументов шаблона по умолчанию. Например:

details/max.cpp

```
template<typename T>
T max(T a, T b)
{
    return b < a ? a : b;
}

int main()
{
    ::max<double>(1.0,-3.0); // Явное указание аргумента
    ::max(1.0,-3.0);         // Неявный вывод аргумента как double
    ::max<int>(1.0, 3.0);    // Явное указание <int> подавляет
                           // вывод; результат имеет тип int
}
```

Некоторые аргументы шаблонов не могут быть выведены потому, что соответствующие им параметры шаблона отсутствуют в типах параметров функции или по некоторой иной причине (см. раздел 15.2). Соответствующие параметры обычно помещаются в начале списка параметров шаблона, так что они могут быть указаны явно, позволяя при этом выводить прочие аргументы. Например:

details/implicit.cpp

```
template<typename DstT, typename SrcT>
DstT implicit_cast(SrcT const& x) // SrcT можно вывести, DstT - нет
{
    return x;
}

int main()
{
    double value = implicit_cast<double>(-1);
}
```

Если обратить порядок параметров шаблона в приведенном примере (другими словами, если записать `template<typename SrcT, typename DstT>`), то вызов `implicit_cast` будет требовать явного указания обоих аргументов шаблона.

Кроме того, такие параметры бесполезно размещать после пакета параметров шаблона или в частичной специализации, поскольку при этом нет никакой возможности явно их указывать или выводить.

```
template<typename ... Ts, int N>
void f(double (&)[N + 1], Ts ... ps); // Бесполезное объявление,
                                         // так как N не может быть
                                         // определено или выведено
```

Поскольку шаблоны функций могут быть перегружены, явного указания всех аргументов шаблона функции может оказаться недостаточно для идентификации конкретной функции: в некоторых случаях таким образом задается *множество* функций. В приведенном ниже примере иллюстрируется следствие из этого обстоятельства.

```
template<typename Func, typename T>
void apply(Func funcPtr, T x)
{
    funcPtr(x);
}

template<typename T> void single(T);

template<typename T> void multi(T);
template<typename T> void multi(T*);

int main()
{
    apply(&single<int>, 3); // OK
    apply(&multi<int>, 7); // Ошибка: нет единственной multi<int>
}
```

В этом примере первый вызов `apply()` корректен, поскольку тип выражения `&single<int>` является недвусмысленным. В результате значение аргумента шаблона для параметра `Func` легко получается путем вывода. Однако во втором вызове `&multi<int>` тип может быть одним из двух разных типов, а следовательно, в данном случае `Func` вывести невозможно.

Более того, явное указание аргументов шаблона функции может привести к попытке сконструировать неверный тип или выражение C++. Рассмотрим следующий перегруженный шаблон функции (RT1 и RT2 являются неопределенными типами):

```
template<typename T> RT1 test(typename T::X const*);  
template<typename T> RT2 test(...);
```

Выражение `test<int>` для первого из двух шаблонов функций не имеет смысла, поскольку у типа `int` нет члена-типа `X`. Однако для второго шаблона такая проблема отсутствует. Следовательно, выражение `&test<int>` идентифицирует адрес единственной функции. Тот факт, что подстановка `int` в первом шаблоне невозможна, не делает это выражение некорректным.

Принцип SFINAE представляет собой важную составную часть практического применения перегрузки шаблонов функций и рассматривается в разделах 8.4 и 15.7.

12.3.2. Аргументы типов

Аргументы типов (типовые аргументы) шаблона являются “значениями”, которые указываются для параметров типов шаблона. В качестве аргументов шаблона в общем случае могут выступать любые типы (включая `void`, типы функций, ссылок и т.д.) но их подстановка вместо параметров шаблонов должна приводить к корректным конструкциям:

```
template<typename T>  
void clear(T p)  
{  
    *p = 0; // Требует применимости к T унарного *  
}  
  
int main()  
{  
    int a;  
    clear(a); // Ошибка: int не поддерживает унарный *  
}
```

12.3.3. Аргументы, не являющиеся типами

Не являющиеся типами (нетиповые) аргументы шаблона представляют собой значения, которые подставляются вместо параметров, не являющихся типами. Такая величина может быть одной из перечисленных ниже.

- Другой параметр шаблона, не являющийся типом и имеющий верный тип.
- Константа времени компиляции с целочисленным типом или типом перечисления. Это допустимо только в случае, когда параметр имеет тип, соответствующий типу этого значения (или типу, к которому оно может быть неявно преобразовано без сужения. Например, тип `char` допускается для параметра с типом `int`, но значение `500` не годится для параметра, который представляет собой 8-битный `char`).

- Имя внешней переменной или функции, которой предшествует встроенный унарный оператор & (получение адреса). Для переменных функций и массивов & можно опускать. Такие аргументы шаблона соответствуют не являющимся типом параметрам с типом указателя. C++17 ослабляет это требование, допуская любые константные выражения, генерирующие указатель на функцию или переменную.
 - Аргументы того же вида, но не предваряемые оператором &, являются корректными аргументами для не являющихся типом параметров ссылочного типа. Здесь C++17 также ослабляет ограничение, допуская любое константное выражение, дающее gl-значение для функции или переменной.
 - Константный указатель на член класса, другими словами, выражение вида &C::m, где C — тип класса, а m — нестатический член класса (данные или функция). Такие значения соответствуют только не являющимся типом параметрам с типом указателей на член класса. И вновь в C++17 ограничения ослаблены: разрешено любое константное выражение, вычисляемое в константу, соответствующую указателю на член.
 - Константа нулевого указателя является корректным аргументом для не являющегося типом параметра с типом указателя или указателя на член.

Целочисленные параметры шаблонов, пожалуй, наиболее распространенная разновидность параметров шаблонов, не являющихся типами, могут использовать неявное приведение к типу параметра. С введением `constexpr` функций преобразования в C++11 это означает, что аргумент перед преобразованием может иметь тип класса.

До C++ 17 при проверке соответствия аргумента параметру, который является указателем или ссылкой, *пользовательские преобразования типа* (конструкторы от одного аргумента и операторы приведения типов) и преобразования производных классов в базовые не рассматривались, хотя в других случаях они оказывались корректными неявными преобразованиями. Неявные преобразования, которые добавляют к аргументу `const` или более `volatile`, вполне допустимы.

Вот несколько корректных примеров аргументов шаблонов, не являющихся типами:

```

template<typename T> void templ_func();
C<void(), &templ_func<double>>* c4; // Инстанцирования шаблона
                                         // функции являются функциями

struct X
{
    static bool b;
    int n;
    constexpr operator int() const
    {
        return 42;
    }
};

C<bool&, X::b>* c5;      // Статические члены класса являются
                           // приемлемыми именами переменных/функций

C<int X::*>* c6; // Пример константного указателя на член

C < long, X{} >* c7; // OK: X сначала преобразуется в int с
                       // constexpr преобразования, а затем в
                       // long с помощью стандартного
                       // целочисленного преобразования

```

Общим ограничением для аргументов шаблона является следующее: компилятор или компоновщик должны быть способны точно определить их значения при создании исполняемого файла. Значения, которые не известны до начала выполнения программы (например, адреса локальных переменных), не отвечают требованию, состоящему в том, что шаблоны должны быть инстанцированы к моменту завершения построения программы.

Но даже при выполнении данного ограничения существует несколько константных значений, которые (возможно, это покажется странным) в настоящее время некорректны:

- числа с плавающей точкой;
- строковые литералы.

(До C++11 были не разрешены также константные нулевые указатели.)

Одна из проблем со строковыми литералами состоит в том, что два идентичных литерала могут храниться по двум разным адресам. Существует альтернативный (но громоздкий) способ определения шаблонов, инстанцирование которых осуществляется через константные строки, включая введение дополнительной переменной для хранения строки:

```

template<char const* str>
class Message
{
    ...
};

extern char const hello[] = "Hello World!";
char const hello11[]      = "Hello World!";

```

```
void foo()
{
    static char const hello17[] = "Hello World!";

    Message<hello> msg03; // OK во всех версиях
    Message<hello11> msg11; // OK начиная с C++11
    Message<hello17> msg17; // OK начиная с C++17
}
```

Требование заключается в том, что параметр шаблона, не являющийся типом, и объявленный как ссылка или указатель, может быть *константным выражением* с внешним связыванием во всех версиях C++; с внутренним связыванием, начиная с C++11; или с любым связыванием, начиная с C++17.

В разделе 17.2 обсуждаются возможные будущие изменения в этой области.

Вот несколько других (менее неожиданных) некорректных примеров:

```
template<typename T, T nontypeParam>
class C;

struct Base
{
    int i;
} base;

struct Derived : public Base
{
} derived;

C<Base*, &derived>* err1; // Ошибка: преобразование производного
                         // класса в базовый не рассматривается
C<int&, base.i>* err2; // Ошибка: поля переменных в качестве
                         // переменных не рассматриваются
int a[10];
C<int*, &a[0]>* err3; // Ошибка: адреса элементов массива
                         // также не допускаются
```

12.3.4. Шаблонные аргументы шаблонов

Шаблонный аргумент шаблона должен в общем случае быть шаблоном класса или псевдонимом с параметрами, которые *точно* соответствуют параметрам шаблонного параметра шаблона, вместо которого он подставляется. До C++17 аргументы шаблона, заданные по умолчанию для шаблонного *аргумента* шаблона, игнорируются (но если шаблонный *параметр* шаблона имеет аргументы по умолчанию, они учитываются при инстанцировании). C++17 ослабляет правило соответствия, требуя только лишь, чтобы шаблонный параметр шаблона был как минимум специализирован (см. раздел 16.2.2), как и соответствующий шаблонный аргумент шаблона.

Таким образом, приведенный ниже пример некорректен до C++17:

```
#include <list>

// Объявление в пространстве имен std:
// template<typename T, typename Allocator = allocator<T>>
// class list;
```

```

template<typename T1, typename T2,
         template<typename> class Cont> // Cont ожидает один параметр
class Rel
{
    ...
};

Rel<int, double, std::list> rel; // Ошибка до C++17: std::list имеет
                                // больше одного параметра шаблона

```

Проблема в этом примере заключается в том, что шаблон `std::list` стандартной библиотеки имеет более одного параметра. Второй параметр (который описывает так называемый *распределитель памяти*) имеет значение по умолчанию, но до C++17 оно не учитывается при установлении соответствия `std::list` параметру `Cont`.

Вариативные шаблонные параметры шаблона являются исключением из описанного выше правила точного совпадения, действовавшего до C++17, и предлагаю решение для этого ограничения: они обеспечивают более общее соответствие шаблонным аргументам шаблона. Пакет шаблонных параметров шаблона может соответствовать нулю или большему количеству параметров шаблона того же вида в шаблонном аргументе шаблона:

```

#include <list>
template<typename T1, typename T2,
         template<typename...> class Cont> // Cont ожидает любое
class Rel                               // количество параметров типа
{
    ...
};

Rel<int, double, std::list> rel;          // OK: std::list имеет два
// параметра шаблона, но может использоваться с одним параметром

```

Пакеты параметров шаблона могут соответствовать только аргументам шаблона того же вида. Например, следующий шаблон класса может быть инстанцирован с любым шаблоном класса или псевдонима, имеющим только типовые параметры шаблона, потому что пакет типовых параметров шаблона, переданный как `TT`, можно сопоставить нулю или большему количеству типовых параметров шаблона:

```

#include <list>
#include <map>
// Объявление в пространстве имен std:
// template<typename Key, typename T,
//           typename Compare = less<Key>,
//           typename Allocator = allocator<pair<Key const, T>>>
// class map;
#include <array>
// Объявление в пространстве имен std:
// template<typename T, size_t N>
// class array;

template<template<typename...> class TT>
class AlmostAnyTmpl
{
};

```

```
AlmostAnyTmpl<std::vector> withVector; // Два параметра типа
AlmostAnyTmpl<std::map> withMap; // Четыре параметра типа
AlmostAnyTmpl<std::array> withArray; // Ошибка: пакет типовых
// параметров шаблона не соответствует
// параметру шаблона, не являющемуся типом
```

Тот факт, что до C++ 17 для объявления шаблонного параметра шаблона могло использоваться только ключевое слово `class`, не говорит о том, что в качестве подставляемых аргументов разрешалось использовать только шаблоны класса, объявленные с помощью ключевого слова `class`. В действительности в качестве шаблонных параметров шаблона могут использоваться также `struct`, `union` и шаблоны псевдонимов (шаблоны псевдонимов — начиная с C++11, в котором они были введены). Это аналогично наблюдению, что в качестве аргумента для параметра типа шаблона, объявленного с помощью ключевого слова `class`, может использоваться любой тип.

12.3.5. Эквивалентность

Два набора аргументов шаблона являются эквивалентными, если значения аргументов попарно идентичны друг другу. Для аргументов типа шаблоны типа не имеют значения — в конечном счете сравниваются типы, лежащие в основе объявлений псевдонимов. Для целочисленных аргументов, не являющихся типами, сравнивается значение аргумента; способ получения этого значения роли не играет. Сказанное выше иллюстрируется следующим примером:

```
template<typename T, int I>
class Mix;

using Int = int;

Mix<int, 3*3>* p1;
Mix<Int, 4+5>* p2; // p2 имеет тот же тип, что и p1
```

(Как видно из этого примера, для установления эквивалентности списков аргументов шаблонов определение шаблона не требуется.)

Однако в контекстах, зависимых от шаблонов, “значение” аргумента шаблона не всегда может быть установлено определенно, и правила эквивалентности становятся немного более сложными. Рассмотрим следующий пример:

```
template<int N> struct I {};

template<int M, int N> void f(I<M+N>); // #1
template<int N, int M> void f(I<N+M>); // #2

template<int M, int N> void f(I<N+M>); // #3 Ошибка
```

Внимательно изучая объявления 1 и 2, вы заметите, что, просто переименовав `M` и `N` в соответственно `N` и `M`, вы получаете то же объявление. Поэтому эти два объявления эквивалентны и объявляют один и тот же шаблон функции `f`. Выражения `M+N` и `N+M` в этих двух объявлениях называются **эквивалентными**.

Однако объявление 3 немного отличается: в нем порядок операндов оказывается обратным. Это делает выражение $N+M$ в объявлении 3 не эквивалентным ни одному из этих двух выражений. Однако, поскольку выражение будет давать один и тот же результат для любых значений параметров шаблона, участвующих в нем, эти выражения называются *функционально эквивалентными*. Объявлять шаблоны способами, которые отличаются только потому, что объявления включают функционально эквивалентные выражения, в действительности не являющиеся эквивалентными, считается ошибкой. Однако такая ошибка не должна быть диагностируема компилятором. Дело в том, что некоторые компиляторы могут, например, внутренне представлять $N+1+1$ точно так же, как $N+2$, в то время как другие компиляторы этого не могут. Вместо того чтобы навязывать конкретный вариант реализации, стандарт позволяет любой из вариантов и требует от программистов проявлять осторожность в этой области.

Функция, сгенерированная из шаблона функции, никогда не эквивалентна обычной функции, даже если обе они имеют один и тот же тип и одно и то же имя. Отсюда вытекают два важных следствия для членов классов.

1. Функция, сгенерированная из шаблона функции-члена, никогда не может перекрывать виртуальную функцию.
2. Конструктор, сгенерированный из шаблона конструктора, никогда не может быть копирующим или перемещающим конструктором⁷. Аналогично оператор присваивания, сгенерированный из шаблона присваивания, никогда не является оператором копирующего или перемещающего присваивания (однако это гораздо меньшая проблема, поскольку неявные вызовы копирующего или перемещающего присваивания гораздо менее распространены).

Это может быть и хорошо, и плохо. Подробнее эти вопросы рассматривались в разделах 6.2 и 6.4.

10

12.4. Вариативные шаблоны

10

Вариативные шаблоны, представленные в разделе 4.1, представляют собой шаблоны, которые содержат по крайней мере один пакет параметров шаблона (см. раздел 12.2.4)⁸. Вариативные шаблоны полезны, когда поведение шаблона может быть обобщено на произвольное количество аргументов. Шаблон класса `Tuple`, представленный в разделе 12.2.4, является одним из таких типов, потому что кортеж может иметь любое количество элементов, которые обрабатываются одним и тем же образом. Мы также можем представить простую функцию

⁷ Однако шаблон конструктора может быть конструктором по умолчанию.

⁸ Термин *вариативный* (*variadic*) позаимствован у языка программирования C, в котором вариативные функции могут принимать переменное количество аргументов. Вариативные шаблоны переняли у вариативных функций троеточие для обозначения нуля или большего количества аргументов и предназначались для безопасной с точки зрения типов замены вариативных функций C в некоторых приложениях.

`print()`, которая принимает произвольное количество аргументов и последовательно выводит каждый из них.

Когда для вариативного шаблона определяются аргументы шаблона, каждый пакет параметров шаблона соответствует последовательности из нуля или большего количества аргументов. Мы называем такую последовательность аргументов шаблона *пакетом аргументов*. В следующем примере показано, как пакет параметров шаблона `Tuple` соответствует различным пакетам аргументов в зависимости от аргументов шаблона, переданных `Tuple`:

```
template<typename... Types>
class Tuple
{
    // Операции над списком типов в
};

int main()
{
    Tuple<> t0;           // Types содержит пустой список
    Tuple<int> t1;        // Types содержит int
    Tuple<int, float> t2; // Types содержит int и float
}
```

Поскольку пакет параметров шаблона представляет собой список аргументов шаблона, а не единственный аргумент, он должен использоваться в контексте, где одна и та же языковая конструкция применяется ко всем аргументам пакета. Одной из таких конструкций является операция `sizeof...`, которая подсчитывает количество аргументов в пакете:

```
template<typename... Types>
class Tuple
{
public:
    static constexpr std::size_t length = sizeof...(Types);
};

// Массив из одного целого числа:
int a1[Tuple<int>::length];
// Массив из трех целых чисел:
int a3[Tuple<short, int, long>::length];
```

12.4.1. Раскрытие пакета

Выражение `sizeof...` является примером *раскрытия*⁹ (expansion) пакета. Раскрытие пакета — это конструкция, которая разделяет пакет аргументов на отдельные аргументы. В то время как `sizeof...` просто подсчитывает количество отдельных аргументов, другие виды пакетов параметров могут раскрываться в несколько элементов в этом списке. Такое расширение пакета идентифицируется многоточием (`...`) справа от элемента в списке. Вот простой пример, где мы

⁹ Устоявшегося перевода в русскоязычной литературе пока нет. Могут использоваться такие варианты, как расширение, разложение, развертывание. — *Примеч. пер.*

создаем новый шаблон класса `MyTuple`, производный от `Tuple`, с передачей его аргументов:

```
template<typename... Types>
class MyTuple : public Tuple<Types...>
{
    // Дополнительные операции, предоставляемые MyTuple
};

MyTuple<int, float> t2; // Наследует Tuple<int, float>
```

Аргумент шаблона `Types...` представляет собой раскрытие пакета, которое создает последовательность аргументов шаблона, по одному для каждого аргумента в пакете аргументов, подставляемом вместо `Types`. Как показано в этом примере, инстанцирование типа `MyTuple<int, float>` подставляет пакет аргументов `int, float` вместо пакета параметров типов шаблона `Types`. Когда это происходит при раскрытии пакета `Types...` мы получаем один аргумент шаблона для `int` и один для `float`, так что `MyTuple<int, float>` наследует `Tuple<int, float>`.

Чтобы интуитивно понять раскрытие пакета, рассматривайте его в терминах синтаксического раскрытия, когда пакет параметров шаблона заменяется точным количеством параметров (не пакетов) шаблона, и раскрытие пакета записывается как отдельные аргументы, по одному разу для каждого из параметров шаблона. Например, вот как будет выглядеть `MyTuple`, если выполнить раскрытие для двух параметров¹⁰:

```
template<typename T1, typename T2>
class MyTuple : public Tuple<T1, T2>
{
    // Дополнительные операции, предоставляемые MyTuple
};
```

и для трех параметров:

```
template<typename T1, typename T2, typename T3>
class MyTuple : public Tuple<T1, T2, T3>
{
    // Дополнительные операции, предоставляемые MyTuple
};
```

Обратите, однако, внимание на то, что доступа к отдельным элементам пакета параметров непосредственно по имени нет, потому что такие имена, как `T1`, `T2` и так далее, не определены в вариативном шаблоне. Если вам нужны типы, единственное, что вы можете сделать — это (рекурсивно) передать их другому классу или функции.

Каждое раскрытие пакета имеет свою *схему* (pattern), которая представляет собой тип или выражение, которое будет повторяться для каждого аргумента

¹⁰ Такое синтаксическое понимание раскрытия пакета является полезным инструментом, но не работает, когда пакет параметров шаблона имеет нулевую длину. В разделе 12.4.5 предоставлена более подробная информация об интерпретации раскрытия пакета нулевой длины.

в пакете аргументов, и обычно располагается перед многоточием, обозначающим раскрытие пакета. Наши предыдущие примеры имели только тривиальные схемы — имя пакета параметров, но эти схемы могут быть произвольно сложными. Например, можно определить новый тип `PtrTuple`, производный от `Tuple` указателей на типы аргументов:

```
template<typename... Types>
class PtrTuple : public Tuple<Types* ...>
{
    // Дополнительные операции, предоставляемые PtrTuple
};

PtrTuple<int, float> t3; // Наследует Tuple<int*, float*>
```

Схема для раскрытия пакета `Types* ...` в приведенном выше примере — это `Types*`. Повторные подстановки в эту схему создают последовательность аргументов типа шаблона, причем все они являются указателями на типы из пакета аргументов, подставляемые вместо `Types`. При синтаксической интерпретации раскрытия пакета `PtrTuple` для раскрытия трех параметров будет иметь следующий вид:

```
template<typename T1, typename T2, typename T3>
class PtrTuple : public Tuple<T1*, T2*, T3*>
{
    // Дополнительные операции, предоставляемые PtrTuple
};
```

12.4.2. Где может происходить раскрытие пакета

Наши примеры до настоящего времени были сосредоточены на использовании раскрытия пакета для создания последовательности аргументов шаблона. Раскрытие пакета фактически может использоваться в языке практически везде, где грамматика допускает список, разделенный запятыми, например:

- в списке базовых классов;
- в списке инициализаторов базовых классов в конструкторе;
- в списке аргументов вызова (схема представляет собой выражение аргумента);
- в списке инициализаторов (например, в списке инициализации в фигурных скобках);
- в списке параметров шаблонов в шаблоне класса, функции или псевдонима;
- в списке исключений, которые могут генерироваться функцией (не рекомендован в C++11 и C++14 и запрещен в C++17);
- в атрибуте, если сам атрибут поддерживает раскрытие пакета (хотя такого атрибута, определенного в стандарте C++, нет);
- при указании выравнивания в объявлении;
- при указании списка захвата лямбда-выражения;

- в списке параметров типа функции;
- в объявлении `using` (начиная с C++17; см. раздел 4.4.5).

Мы уже упоминали о `sizeof...` как о механизме раскрытия пакета, который на самом деле список не генерирует. В C++17 также добавлены *выражения свертки* (fold expressions), которые являются еще одним механизмом, не генерирующим разделенного запятыми списка (см. раздел 12.4.6).

Некоторые из этих контекстов раскрытия пакетов включены лишь для полноты картины, так что мы сосредоточим свое внимание только на тех контекстах раскрытия пакетов, которые могут быть полезны на практике. Поскольку раскрытие пакета во всех контекстах следует одним и тем же принципам и синтаксису, вы должны суметь экстраполировать приведенные здесь примеры на более экзотические контексты раскрытия пакетов, если в них появится необходимость.

Раскрытие пакета в списке базовых классов приводит к некоторому количеству непосредственных базовых классов. Такое раскрытие может быть полезным для агрегации внешних данных и функциональности через объекты “миксины” (`mixin`), которые представляют собой классы, предназначенные для смешения (`mixed in`) в одну иерархию классов для предоставления новых вариантов поведения. Например, показанный ниже класс `Point` использует раскрытие пакета в нескольких различных контекстах, чтобы разрешить произвольные миксины¹¹:

```
template<typename... Mixins>
class Point : public Mixins... // Раскрытие пакета базовых классов
{
    double x, y, z;
public:
    Point() : Mixins()... { } // Раскрытие пакета инициализаторов
                            // базового класса
    template<typename Visitor>
    void visitMixins(Visitor visitor)
    {
        // Раскрытие пакета аргументов вызова:
        visitor(static_cast<Mixins&>(*this)...);
    }
};
struct Color
{
    char red, green, blue;
};
struct Label
{
    std::string name;
};
Point<Color, Label> p; // Наследует как Color, так и Label
```

Класс `Point` использует раскрытие пакета для получения каждого из предоставленных миксинов и развертывания его в открытый базовый класс. Затем конструктор по умолчанию класса `Point` применяет раскрытие пакета в списке инициализаторов базовых классов для инициализации каждого из базовых классов, добавленных с помощью механизма миксинов.

¹¹ Эти объекты более подробно рассматриваются в разделе 21.3.

Функция-член шаблона `visitMixins` является наиболее интересной в том отношении, что она использует результат раскрытия пакета в качестве аргументов вызова. Путем приведения `*this` к каждому из типов миксина раскрытие пакета создает аргументы вызова, которые относятся к каждому из базовых классов. Вопросы написания функции-посетителя для использования с `visitMixins`, которая может использовать произвольное количество аргументов вызова функции, рассматриваются в разделе 12.4.3.

Раскрытие пакета может также использоваться внутри списка параметров шаблона для создания пакетов шаблонных параметров шаблонов или параметров шаблонов, не являющихся типами:

```
template<typename... Ts>
struct Values
{
    template<Ts... Vs>
    struct Holder
    {
    };
};

int i;
Values<char, int, int*>::Holder<'a', 17, &i> valueHolder;
```

Обратите внимание на то, что, когда аргументы типов для `Values<...>` указаны, список аргументов, не являющихся типами, для `Values<...>::Holder` имеет фиксированную длину; таким образом, пакет параметров `Vs` не является пакетом параметров переменной длины.

`Values` представляет собой пакет параметров шаблона, не являющихся типами, в котором каждый из фактических аргументов шаблона может иметь свой тип, предоставляемый пакетом параметров типа шаблона `Types`. Обратите внимание на двойную роль многоточий в объявлении `Values` — для объявления параметра шаблона как пакета параметров и для объявления типа этого пакета параметров шаблона как раскрытия пакета. Хотя такие пакеты параметров шаблонов на практике встречаются редко, тот же принцип применяется в гораздо более распространенном контексте параметров функций.

12.4.3. Пакеты параметров функций

Пакет параметров функции является параметром функции, который соответствует нулю или большему количеству аргументов вызова функции. Как и пакет параметров шаблона, пакет параметров функции вводится с помощью многоточия (...) до (или вместо) имени параметра функции, и, так же как пакет параметров шаблона, пакет параметров функции должен при использовании раскрываться с помощью процесса раскрытия пакета. Пакеты параметров шаблона и пакеты параметров функции вместе именуются *пакетами параметров*.

В отличие от пакета параметров шаблонов, пакеты параметров функций всегда являются раскрытиями параметров, так что их объявленные типы должны включать по меньшей мере один пакет параметров. В следующем примере мы вводим новый конструктор `Point`, который выполняет копирующую инициализацию каждого из миксинов из предоставленных аргументов конструктора:

```

template<typename... Mixins>
class Point : public Mixins...
{
    double x, y, z;
public:
    // Конструктор по умолчанию, функция-посетитель и т.п. опущены
    Point(Mixins... mixins)      // mixins - пакет параметров функции
        : Mixins(mixins)... { } // Инициализация каждого объекта
    };                         // базового класса предоставленным
                                // значением

struct Color
{
    char red, green, blue;
};

struct Label
{
    std::string name;
};

Point<Color, Label> p({0x7F,0,0x7F}, {"center"});

```

Пакет параметров функции для шаблона функции может зависеть от пакетов параметров шаблонов, объявленных в этом шаблоне, что позволяет шаблону функции принимать произвольное количество аргументов вызова без потери информации о типе:

```

template<typename... Types>
void print(Types... values);

int main
{
    std::string welcome("Welcome to ");
    print(welcome, "C++ ", 2011, '\n'); // Вызов print<std::string,
}                                         //     char const*,int,char>

```

При вызове шаблона функции `print()` с некоторым количеством аргументов типы этих аргументов будут помещены в пакет аргументов для замены пакета параметров типа шаблона `Types`, в то время как фактические значения аргументов будут помещены в пакет аргументов для замены значений пакета параметров функции `values`. Процесс определения аргументов из вызова функции подробно рассматривается в главе 15, “Вывод аргументов шаблона”. Пока что достаточно отметить, что *i*-й тип в `Types` представляет собой тип *i*-го значения в `values` и что оба эти пакета параметров доступны в теле шаблона функции `print()`.

Фактическая реализация `print()` использует рекурсивное инстанцирование шаблона, метод шаблонного метапрограммирования, описанный в разделе 8.1 и главе 23, “Метапрограммирование”.

Между безымянным пакетом параметров функции в конце списка параметров и аргументами в стиле “*vararg*” языка С имеется синтаксическая неоднозначность. Например:

```

template<typename T> void c_style(int, T...);
template<typename... T> void pack(int, T...);

```

В первом случае “`T...`” рассматривается как “`T, ...`”: неименованный параметр типа `T`, за которым следует параметр `vararg` в стиле С. Во втором случае

конструкция “T...” рассматривается как пакет параметра функции, потому что T является допустимой схемой раскрытия. Неоднозначность можно устранить, добавив запятую перед многоточием (которая заставляет рассматривать многоточие как параметр `vargarg` в стиле C), или добавляя после многоточия ... идентификатор, который делает его именованным пакетом параметров функции. Обратите внимание на то, что в обобщенном лямбда-выражении завершающее многоточие ... будет рассматриваться как обозначающее пакет параметров, если тип, который непосредственно ему предшествует (без промежуточной запятой), содержит `auto`.

12.4.4. Множественные и вложенные раскрытия пакетов

Схема раскрытия пакета может быть произвольно сложной и включать множественные, различные пакеты параметров. При инстанцировании раскрытия пакета, содержащего несколько пакетов параметров, все пакеты параметров должны иметь одинаковую длину. Результирующая последовательность типов или значений будет создаваться поэлементно, подстановкой первого аргумента каждого пакета параметров в схему, за которым будет следовать второй аргумент каждого пакета параметров, и так далее. Например, следующая функция копирует все свои аргументы перед передачей их функциональному объекту f:

```
template<typename F, typename... Types>
void forwardCopy(F f, Types const&... values)
{
    f(Types(values)...);
}
```

Раскрытие пакета аргумента вызова именует два пакета параметров, `Types` и `values`. При инстанцировании этого шаблона поэлементное раскрытие пакетов параметров `Types` и `values` выполняет ряд конструирований объектов, при котором копия *i*-го значения в `values` создается путем приведения его к *i*-му типу в `Types`. С использованием синтаксической интерпретации раскрытия пакета трехаргументный шаблон `forwardCopy` будет выглядеть следующим образом:

```
template<typename F, typename T1, typename T2, typename T3>
void forwardCopy(F f, T1 const& v1, T2 const& v2, T3 const& v3)
{
    f(T1(v1), T2(v2), T3(v3));
}
```

Раскрытия пакетов сами могут быть вложенными. В таких случаях каждый встреченный пакет параметров “раскрывается” ближайшим охватывающим раскрытием пакета (и только им). Приведенный далее пример иллюстрирует вложенное раскрытие пакета, включающее три различных пакета параметров:

```
template<typename... OuterTypes>
class Nested
{
    template<typename... InnerTypes>
    void f(InnerTypes const& ... innerValues)
    {
```

```

        g(OuterTypes(InnerTypes(innerValues)...)...);
    }
};

```

В вызове `g()` раскрытие пакета со схемой `InnerTypes(innerValues)` является наиболее глубоко вложенным раскрытием пакета, которое раскрывает `InnerTypes` и `innerValues` и создает последовательность аргументов вызова функции для инициализации объекта, обозначаемого `OuterTypes`. Схема внешнего раскрытия пакета включает внутреннее раскрытие, производит набор аргументов вызова для функции `g()`, созданный на основании инициализации каждого из типов в `OuterTypes` из последовательности аргументов вызова функции, сгенерированных внутренним раскрытием. В синтаксической интерпретации этого раскрытия пакета, где `OuterTypes` имеет два аргумента, а `InnerTypes` и `innerValues` — по три аргумента, вложенность становится более очевидной:

```

template<typename O1, typename O2>
class Nested
{
    template<typename I1, typename I2, typename I3>
    void f(I1 const& iv1, I2 const& iv2, I3 const& iv3)
    {
        g(O1(I1(iv1), I2(iv2), I3(iv3)),
          O2(I1(iv1), I2(iv2), I3(iv3)));
    }
};

```

Множественные и вложенные раскрытия пакетов являются весьма мощным инструментарием (см., например, раздел 26.2).

12.4.5. Раскрытия пакетов нулевой длины

Синтаксическая интерпретация раскрытия пакета может быть полезным инструментом для понимания того, как будет себя вести инстанцирование вариативного шаблона с различным количеством аргументов. Однако синтаксические интерпретации часто не работают при наличии пакетов аргументов нулевой длины. Чтобы проиллюстрировать это, рассмотрим шаблон класса `Point` из раздела 12.4.2 с синтаксической подстановкой нуля аргументов:

```

template<>
class Point :
{
    Point() : { }
};

```

Записанный код некорректен, поскольку список параметров шаблона пуст, так что пустые списки базовых классов и их инициализаторов вырождаются в паразитные двоеточия.

Пакет расширения на самом деле является семантической конструкцией, и подстановка пакета аргументов любого размера не влияет на то, как выполняется синтаксический анализ раскрытия пакета (или охватывающего его вариативного шаблона). Когда раскрытие пакета приводит к пустому списку,

программа (семантически) ведет себя так, как если бы этот список отсутствовал. Инстанцирование `Point<>` выполняется с отсутствием базовых классов, а его конструктор по умолчанию не имеет инициализаторов базовых классов и является корректно сформированным. Эти семантические правила выполняются даже тогда, когда синтаксическая интерпретация раскрытия пакета нулевой длины дает точно определенный (но иной) код. Например:

```
template<typename T, typename... Types>
void g(Types... values)
{
    T v(values...);
}
```

Вариативный шаблон функции `g()` создает значение `v`, которое инициализируется непосредственно заданной последовательностью значений. Если эта последовательность значений пуста, объявление `v` синтаксически выглядит как `T v()`. Однако, поскольку подстановка в раскрытие пакета является семантической и не влияет на тип сущности, получающейся при синтаксическом анализе, `v` инициализируется с нулевым количеством аргументов, то есть является инициализацией значения¹².

12.4.6. Выражения свертки

Рекуррентная схема в программировании именуется *сверткой* (fold) операции над последовательностью значений. Например, *правая свертка* (right fold) функции `fn` над последовательностью `x[1], x[2], ..., x[n-1], x[n]` задается выражением

```
fn(x[1], fn(x[2], fn(..., fn(x[n-1], x[n])...)))
```

При изучении новых возможностей языка Комитет по стандартизации C++ столкнулся с необходимостью работы с такими конструкциями для частного случая логического бинарного оператора (например, `&&` или `||`), примененного к раскрытию пакета. Без такой дополнительной возможности мы могли бы написать следующий код для получения нужной функциональности для оператора `&&` следующим образом:

```
bool and_all()
{
    return true;
}
template<typename T>
bool and_all(T cond)
{
    return cond;
}
```

¹² Имеются аналогичные ограничения на члены шаблонов классов и вложенных классов в рамках шаблонов классов: если член объявлен с типом, который не является типом функции, но после инстанцирования типа этого члена является типом функции, то программа считается некорректно сформированной, потому что семантическая интерпретация члена изменяется от члена-данных до функции-члена.

```
template<typename T, typename... Ts>
bool and_all(T cond, Ts...conds)
{
    return cond && and_all(conds...);
}
```

В C++17 была добавлена новая возможность, именуемая *выражениями свертки* (fold expressions) (см. краткое описание в разделе 4.2). Она применима ко всем бинарным операторам за исключением `,`, `->` и `[]`.

Для данной нераскрытой схемы выражения `pack` и выражения `value`, не являющегося схемой, C++17 позволяет записать для любого такого оператора `op` либо

`(pack op ... op value)`

для правой свертки оператора (именуемой *бинарной правой сверткой*), либо

`(value op ... op pack)`

для левой свертки (именуемой *бинарной левой сверткой*). Обратите внимание на то, что в данном случае наличие скобок обязательно. Некоторые соответствующие примеры имеются в разделе 4.2.

Наличие такой возможности позволяет записать код, вызывающий свойство для каждого переданного типа `T`, наподобие

```
template<typename... T> bool g()
{
    return and_all(trait<T>()...);
```

(где `and_all` определено выше) как

```
template<typename... T> bool g()
{
    return (trait<T>() && ... && true);
```

Как и следует ожидать, выражения свертки являются раскрытиями пакета. Обратите внимание: если пакет пуст, то тип выражения свертки по-прежнему может быть определен из операнда, не являющегося пакетом (`value` в приведенном псевдокоде).

Однако разработчики данной возможности хотели позволить обходиться без операнда `value`. Поэтому в C++17 доступны две другие формы: *унарная правая свертка*

`(pack op ...)`

и *унарная левая свертка*

`(... op pack)`

Здесь также необходимо наличие скобок. Очевидно, что в данном случае мы сталкиваемся с проблемой пустого раскрытия: как определить его тип и значение? Ответ заключается в том, что пустое раскрытие унарной свертки в общем случае является ошибкой, с тремя исключениями.

- Пустое раскрытие унарной свертки `&&` дает значение `true`.
- Пустое раскрытие унарной свертки `||` дает значение `false`.
- Пустое раскрытие унарной свертки оператора запятой `(,)` дает выражение типа `void`.

Обратите внимание, что если вы перегрузите один из этих специальных операторов необычным способом, возможны сюрпризы. Например:

```
struct BooleanSymbol
{
    ...
};

BooleanSymbol operator||(BooleanSymbol, BooleanSymbol);

template<typename... BTs> void symbolic(BTs... ps)
{
    BooleanSymbol result = (ps || ...);
    ...
}
```

Предположим, что мы вызываем `symbolic` с типами, производными от `BooleanSymbol`. Для всех раскрытий результат будет производить значение `BooleanSymbol`, за исключением пустого раскрытия, которое будет давать значение типа `bool`¹³. Поэтому обычно рекомендуется не использовать *унарные выражения свертки*, прибегая вместо них к бинарным выражениям свертки (с явно указанным значением для пустого раскрытия).

12.5. Друзья

Основная идея объявления друзей проста: определить классы или функции, которые имеют привилегированную связь с классом, в котором находится `friend`-объявление. Однако все осложняется следующими двумя фактами.

1. Объявление дружбы может быть только объявлением сущности.
2. Объявление дружественной функции может быть определением.

12.5.1. Дружественные классы шаблонов классов

Объявления дружественных классов не могут быть определениями, а потому редко проблематичны. В контексте шаблонов единственным новым аспектом объявления дружественного класса является возможность объявить другом конкретный экземпляр шаблона класса:

¹³ Поскольку перегрузка этих специальных операторов достаточно необычна, данная проблема, к счастью, возникает редко (но она очень тонка). Кстати, первоначальное предложение для выражений свертки включало также значения для пустых раскрытий для более распространенных операторов, таких как `+` и `*`, что привело бы к гораздо более серьезным проблемам.

```
template<typename T>
class Node;
template<typename T>
class Tree
{
    friend class Node<T>;
    ...
};
```

Обратите внимание на то, что шаблон класса должен быть видимым в точке, где один из его экземпляров становится другом класса или шаблона класса. В случае обычного класса такое требование не предусмотрено:

```
template<typename T>
class Tree
{
    friend class Factory; // OK, даже если это первое
                          // объявление Factory
    friend class Node<T>; // Ошибка, если Node
};                      // не является видимым
```

Более подробно этот вопрос рассматривается в разделе 13.2.2.

Одно из приложений, представленное в разделе 5.5, является объявлением в качестве дружественных инстанцирований другого шаблона класса:

```
template<typename T>
class Stack
{
public:
    ...
    // Присваивание стека элементов типа T2
    template<typename T2>
    Stack<T2>& operator= (Stack<T2> const&);
    // Для получения доступа к закрытым членам
    // Stack<T2> для любого типа T2:
    template<typename> friend class Stack;
    ...
};
```

Стандарт C++11 добавляет синтаксис, позволяющий сделать другом параметр шаблона:

```
template<typename T>
class Wrap
{
    friend T;
    ...
};
```

Этот код корректен для любого типа T, но игнорируется, если T на самом деле не является типом класса¹⁴.

¹⁴ Это было самым первым расширением, добавленным в C++11 (предложено Уильямом Миллером (William M. "Mike" Miller)).

12.5.2. Дружественные функции шаблонов классов

Экземпляр шаблона функции можно сделать другом, добавив после имени функции-друга угловые скобки. Угловые скобки могут содержать аргументы шаблона, но если аргументы могут быть выведены, то угловые скобки можно оставить пустыми:

```
template<typename T1, typename T2>
void combine(T1, T2);
class Mixer
{
    friend void combine<>(int&, int&); // OK: T1 = int&, T2 = int&
    friend void combine<int, int>(int, int); // OK: T1 = int, T2 = int
    friend void combine<char>(char, int); // OK: T1 = char T2 = int
    friend void combine<char>(char&, int); // Ошибка: нет соответствующего шаблона combine()
    friend void combine<>(long, long)
    {
        ***
    }
    // Ошибка: определение не разрешено!
};
```

Обратите внимание на то, что мы не можем *определить* экземпляр шаблона (в лучшем случае мы можем определить специализацию), а следовательно, объявление друга, именующее экземпляр, не может быть определением.

Если после имени нет угловых скобок, то у нас есть две возможности.

1. Если имя *не* квалифицированное (иными словами, не содержит `::`), оно никогда не ссылается на экземпляр шаблона. Если нет подходящей видимой нешаблонной функции при объявлении друга, это объявление является первым объявлением данной функции. Это объявление может быть также и определением.
2. Если имя *является* квалифицированным (содержит `::`), это имя должно ссылаться на ранее объявленную функцию или шаблон функции. Функции с соответствием аргументов предпочтительнее, чем соответствующий аргументам шаблон функции. Однако такое объявление друга не может быть определением.

Приведенный ниже пример может пояснить разные возможности.

```
void multiply(void*); // Обычная функция

template<typename T>
void multiply(T); // Шаблон функции
class Comrades
{
    friend void multiply(int) { }
    // Определение новой функции ::multiply(int)
```

```

friend void ::multiply(void*);
// Ссылка на приведенную выше обычную функцию,
// но не на экземпляр multiply<void*>

friend void ::multiply(int);
// Ссылка на экземпляр шаблона

friend void ::multiply<double*>(double*);
// Квалифицированные имена также могут иметь
// угловые скобки, но шаблон должен быть видимым

friend void ::error() { }
// Ошибка: квалифицированный друг не может быть определением
};


```

В наших предыдущих примерах мы объявляли дружественные функции в обычном классе. Те же правила применяются и когда мы объявляем их в шаблонах классов, но в определении того, какая функция является другом, могут участвовать параметры шаблона:

```

template<typename T>
class Node
{
    Node<T>* allocate();
    ...
};

template<typename T>
class List
{
    friend Node<T>* Node<T>::allocate();
    ...
};


```

Дружественная функция может быть также *определенна* в шаблоне класса; в этом случае она инстанцируется только когда на самом деле используется. Это обычно требует от дружественной функции, чтобы она использовала сам шаблон класса в типе дружественной функции, что позволяет легче выражать функции в шаблоне класса, которые могут быть вызваны, как если бы они были видимы в области видимости пространства имен:

```

template<typename T>
class Creator
{
    friend void feed(Creator<T>) // Каждый T инстанцирует
    {                           // другую функцию ::feed()
        ...
    };
};

int main()
{
    Creator<void> one;
    feed(one); // Инстанцирует ::feed(Creator<void>)
    Creator<double> two;
    feed(two); // Инстанцирует ::feed(Creator<double>)
}


```

В этом примере каждое инстанцирование `Creator` создает другую функцию. Обратите внимание: даже несмотря на то, что эти функции генерируются как часть инстанцирования шаблона, сами функции являются обычными функциями, а не экземплярами шаблона. Однако они рассматриваются как *шаблонные сущности* (см. раздел 12.1), и их определение инстанцируется только при использовании. Учтите также, что, поскольку тела этих функций определяются внутри определения класса, они неявно являются встраиваемыми. Следовательно, не является ошибкой генерация одной и той же функции в двух разных единицах трансляции. Более подробно эта тема рассматривается в разделах 13.2.2 и 21.2.1.

12.5.3. Дружественные шаблоны

Обычно при объявлении друга, который является экземпляром шаблона функции или класса, мы можем точно указать, какая именно сущность должна быть другом. Иногда, тем не менее, полезно указать, что все экземпляры шаблона являются друзьями класса. Для этого нужен *дружественный шаблон*. Например:

```
class Manager
{
    template<typename T>
    friend class Task;

    template<typename T>
    friend void Schedule<T>::dispatch(Task<T>*);

    template<typename T>
    friend int ticket()
    {
        return ++Manager::counter;
    }
    static int counter;
};
```

Так же, как и в случае объявлений обычных друзей, дружественный шаблон может быть определением, только если он использует неквалифицированное имя функции, не сопровождающееся угловыми скобками.

Дружественный шаблон может объявлять только первичные шаблоны и их члены. Любые частичные и явные специализации, связанные с первичным шаблоном, также автоматически рассматриваются как друзья.

12.6. Заключительные замечания

Общие концепции и синтаксис шаблонов C++ оставались относительно стабильными с момента их создания в конце 1980-х годов. Шаблоны классов и функций являются частью первоначальных шаблонных возможностей языка. То же относится к параметрам шаблонов, являющимся и не являющимся типами.

Однако в оригинальный дизайн вошли некоторые значительные добавления, в основном продиктованные потребностями стандартной библиотеки C++. Шаблоны членов можно рассматривать как наиболее фундаментальные из этих

дополнений. Любопытно, что официальное голосование ввело в стандарт C++ только шаблоны *функций*-членов; шаблоны *классов*-членов стали частью стандарта по недосмотру редакции.

Дружественные шаблоны, аргументы шаблона по умолчанию и шаблонные параметры шаблонов пришли в язык позже в ходе стандартизации C++98. Способность объявлять шаблонные параметры шаблонов иногда называют *обобщенностью более высокого порядка*. Первоначально они были введены для поддержки определенной модели распределителей памяти в стандартной библиотеке C++, но позже эта модель распределителей была заменена другой, не использующей шаблонные параметры шаблонов. Позднее шаблонные параметры шаблонов были близки к удалению из языка, потому что их спецификация оставалась неполной почти до конца процесса стандартизации C++98. В конце концов большинство членов Комитета проголосовали за то, чтобы оставить их в стандарте, и их спецификации были завершены.

Шаблоны псевдонимов были введены как часть стандарта 2011 года. Шаблоны псевдонимов служат тем же целям, что и часто запрашиваемая возможность “`typedef` для шаблонов”, обеспечивающая легкое написание шаблона, который является просто другим названием существующего шаблона класса. Спецификация N2258, внесшая их в стандарт, принадлежит Габриэлю Дос Райсу (Gabriel Dos Reis) и Бъярне Страуструпу (Bjarne Stroustrup); свой вклад в некоторые из ранних проектов этого предложения внес Мэт Маркус (Mat Marcus). Габриэль также проработал подробности предложения шаблона переменных для C++14 (N3651). Первоначально это предложение предназначалось только для поддержки `constexpr`-переменных, но это ограничение было отменено ко времени принятия предложения в проект стандарта.

Вариативные шаблоны были обусловлены потребностями стандартной библиотеки C++11 и библиотеки Boost [10], где библиотеки шаблонов C++ использовали все более сложные и запутанные методы для работы с шаблонами с произвольным количеством аргументов шаблона. Первоначальные спецификации стандарта (N2242) предоставили Дуг Грегор (Doug Gregor), Яакко Ярви (Jaakko Järví), Гэри Паулл (Gary Powell), Джэнс Маурер (Jens Maurer) и Джейсон Меррилл (Jason Merrill). Дуг также разработал исходную реализацию этой возможности (в GNU GCC) во время работы над спецификацией, что значительно помогло использованию этой возможности в стандартной библиотеке.

Выражения свертки были разработаны Эндрю Саттоном (Andrew Sutton) и Ричардом Смитом (Richard Smith) – они были добавлены в C++17 на основании их статьи N4191.

Глава 13

Имена в шаблонах

Имена в большинстве языков программирования представляют собой фундаментальную концепцию. Они являются средством, с помощью которого программист может обращаться к ранее созданным объектам. Когда компилятор C++ встречает имя, он должен выполнить его “поиск”, чтобы определить, на какой объект ссылается это имя. С точки зрения реализации C++ в этом отношении является сложным языком. Рассмотрим, например, выражение C++ `x*y;`. Если `x` и `y` — имена переменных, данное выражение является умножением, но если `x` является именем типа, то это не что иное, как объявление `y` как указателя на объект типа `x`.

Из этого небольшого примера видно, что C++ (как и C) является так называемым *контекстно-зависимым языком программирования*. Другими словами, конструкцию языка не всегда можно распознать без знания ее более широкого контекста. Естественно задать вопрос: а какое отношение это имеет к шаблонам? Шаблоны являются конструкциями, которые имеют дело с несколькими контекстами: 1) контекст, в котором шаблон появляется, 2) контекст, в котором шаблон инстанцируется, и 3) контекст, связанный с аргументами шаблона, для которых происходит инстанцирование. Следовательно, теперь вас не должно удивить то, что имена в C++ требуют к себе особого внимания.

13.1. Систематизация имен

Имена в C++ классифицируются разными способами, причем этих способов существует огромное количество. Чтобы помочь справиться с этим изобилием терминологии, все способы классификации имен сведены в табл. 13.1. К счастью, многие вопросы, касающиеся шаблонов C++, станут гораздо понятнее, если ознакомиться с основными концепциями именования.

1. Имя является *полным, или квалифицированным именем* (qualified name), если область видимости, которой оно принадлежит, явно указывается либо с помощью оператора разрешения области видимости (`::`), либо с помощью оператора доступа к членам класса (`.` или `->`). Например, `this->count` — квалифицированное имя, а `count` — нет (даже если само по себе `count` в действительности является ссылкой на член класса).
2. Имя является *зависимым именем* (dependent name), если оно каким-либо образом зависит от параметра шаблона. Например, `std::vector<T>::iterator` — зависимое имя, если `T` — параметр шаблона, и независимое, если `T` является известным псевдонимом типа (таким как `T` из `using T = int`).

Таблица 13.1. Систематизация имен¹

Классификация	Пояснения и примечания
Идентификатор	Имя, которое содержит только неразрывные последовательности букв, знаков подчеркивания (<u>) и цифр. Идентификатор не может начинаться с цифры; кроме того, некоторые идентификаторы зарезервированы в реализации языка: их нельзя самостоятельно вводить в программы (используйте эмпирическое правило: избегайте идентификаторов, начинающихся с подчеркиваний и двойных подчеркиваний). Понятие “буква” интерпретируется расширенно: сюда включаются специальные <i>универсальные имена символов (universal character names – UCN)</i>, с помощью которых кодируются знаки из неалфавитных языков</u>
Идентификатор функции оператора	Ключевое слово <code>operator</code> , за которым следует символ, обозначающий оператор, например <code>operator new</code> или <code>operator[]</code> ¹
Идентификатор функции преобразования типа	Используется для обозначения определенного пользователем неявного оператора преобразования, например <code>operator int&</code> , который может также быть представлен как <code>operator int bitand</code>
Идентификатор оператора литерала	Используется для обозначения пользовательского оператора литерала — например, <code>operator ""_km</code> , который будет использоваться при написании литерала наподобие <code>100_km</code> (введен в C++11)
Идентификатор шаблона	Имя шаблона, за которым следуют аргументы шаблона, заключенные в угловые скобки, например <code>List<T, int, 0></code> . Идентификатором шаблона может также быть идентификатор функции оператора или идентификатор оператора литерала, за которым следуют аргументы шаблона, заключенные в угловые скобки; например <code>operator+<X<int>></code>
Неквалифицированный идентификатор	Обобщение идентификатора. Неквалифицированный идентификатор может быть любым из приведенных выше видов идентификаторов (идентификатор, идентификатор функции оператора, идентификатор функции преобразования типа, идентификатор оператора литерала или идентификатор шаблона), а также “имя деструктора” (например, записи наподобие <code>~Data</code> или <code>~List<T, T, N></code>)
Квалифицированный идентификатор	Неквалифицированный идентификатор, который квалифицирован именем класса, перечисления или пространства имён, или с помощью оператора разрешения глобальной области видимости. Заметим, что такое имя само по себе может быть квалифицированным. Примерами являются <code>::x</code> , <code>S::x</code> , <code>Array<T>::y</code> и <code>::N::A<T>::z</code>

¹Многие операторы имеют альтернативные представления. Например, `operator &` можно записать как `operator bitand`, даже если он обозначает унарный *оператор получения адреса*.

Окончание табл. 13.1

Классификация	Пояснения и примечания
Квалифицированное имя	Этот термин в стандарте не определен, но мы используем его для обозначения имен, которые подвергаются так называемому <i>квалифицированному поиску</i> (<i>qualified lookup</i>). В частности, это могут быть квалифицированные или неквалифицированные идентификаторы, которые используются после явного оператора доступа (. или ->). Примерами являются <code>S::x</code> , <code>this->f</code> и <code>p->A::m</code> . Однако просто <code>class_mem</code> в контексте, когда он неявно эквивалентен <code>this->class_mem</code> , не является квалифицированным именем: доступ к члену класса должен быть явным
Неквалифицированное имя	Неквалифицированный идентификатор, который не является квалифицированным именем. Это не стандартный термин, но он соответствует именам, которые подвергаются тому, что в стандарте именуется <i>неквалифицированным поиском</i>
Имя	Квалифицированное или неквалифицированное имя
Зависимое имя	Имя, которое каким-либо образом зависит от параметра шаблона. Обычно квалифицированное или неквалифицированное имя, которое явно содержит параметр шаблона, является зависимым. Более того, квалифицированное имя, которое включает оператор доступа к члену класса (. или ->), обычно является зависимым, если тип выражения в левой части оператора является <i>типозависимым</i> (<i>type-dependent</i> , концепция, рассматриваемая в разделе 13.3.6). В частности, <code>b</code> в <code>this->b</code> в общем случае является зависимым именем при присутствии в шаблоне. И наконец, имя, являющееся субъектом поиска, зависящего от аргумента (описан в разделе 13.2), такое как <code>ident</code> в вызове вида <code>ident(x, y)</code> или + в выражении <code>x+y</code> , является зависимым именем тогда и только тогда, когда любое из выражений аргументов является типозависимым
Независимое имя	Имя, которое не является зависимым согласно данному выше определению

С этой таблицей полезно ознакомиться хотя бы для того, чтобы получить некоторое представление о терминах, которые иногда используются при описании тем, касающихся шаблонов C++. Однако запоминать точное значение каждого термина вовсе не обязательно. Если возникнет необходимость, всегда можно вернуться к данной таблице.

13.2. Поиск имен

Существует много незначительных деталей, касающихся поиска имен в C++, но здесь мы остановимся только на нескольких основных концепциях. Подробностям будем уделять внимание только в случаях, когда 1) нужно

убедиться в правильности интуитивной трактовки, и 2) в “патологических” случаях, которые тем или иным образом описаны в стандарте.

Поиск квалифицированных имен проводится в области видимости, вытекающей из квалифицирующей конструкции. Если эта область видимости является классом, то поиск также проводится и в базовых классах. Однако при поиске квалифицированных имен не рассматриваются области видимости, охватывающие данную. Основной принцип такого поиска иллюстрируется приведенным ниже кодом:

```
int x;
class B
{
public:
    int i;
};

class D : public B
{
};

void f(D* pd)
{
    pd->i = 3; // Находит B::i
    D::x = 2;   // Ошибка: не находит ::x
}           // в охватывающей области видимости
```

Поиск же неквалифицированных имен, напротив, выполняется в последовательно расширяющихся областях видимости, охватывающих данную (однако в определениях функций-членов сначала проводится поиск в области видимости класса и его базовых классов, а уже затем — в охватывающих областях видимости). Такая разновидность поиска называется *обычным поиском* (ordinary lookup). Приведенный ниже пример иллюстрирует главную идею, лежащую в основе обычного поиска:

```
extern int count;           // #1
int lookup_example(int count) // #2
{
    if (count < 0)
    {
        int count = 1;      // #3
        lookup_example(count); // Неквалифицированное count из #3
    }

    return count + ::count; // Первое, неквалифицированное count из
}                           // #2; второе, квалифицированное - из #1
```

Современные методы поиска неквалифицированных имен в дополнение к обычному поиску могут включать так называемый *поиск, зависящий от аргументов* (argument-dependent lookup — ADL)². Прежде чем перейти к подробному

² Этот поиск в C++98 и C++03 назывался также поиском Кёнига (или расширенным поиском Кёнига) в честь Эндрю Кёнига (Andrew Koenig), который впервые предложил вариант данного механизма.

рассмотрению ADL, рассмотрим механизм этого поиска на нашем вечном примере шаблона `max()`:

```
template<typename T>
T max(T a, T b)
{
    return b < a ? a : b;
}
```

Предположим, что нам необходимо применить этот шаблон к типу, определенному в другом пространстве имен:

```
namespace BigMath
{
    class BigNumber
    {
        ...
    };
    bool operator < (BigNumber const&, BigNumber const&);
    ...
}

using BigMath::BigNumber;

void g(BigNumber const& a, BigNumber const& b)
{
    ...
    BigNumber x = ::max(a, b);
    ...
}
```

Проблема заключается в том, что шаблону `max()` ничего не известно о пространстве имен `BigMath`, и с помощью обычного поиска не будет найден оператор `<`, применимый к значениям типа `BigNumber`. Если не ввести некоторые специальные правила, такие ситуации в значительной степени сокращают применимость шаблонов в контексте пространств имен C++. Поиск ADL является ответом C++ на необходимость введения таких “специальных правил”.

13.2.1. Поиск, зависящий от аргументов

ADL относится главным образом к неквалифицированным именам, которые выглядят как если бы они именовали функции, не являющиеся членами, в вызовах функций или применениях операторов. ADL не будет выполняться, если обычный поиск находит

- имя функции-члена;
- имя переменной;
- имя типа или
- имя объявления функции с областью видимости блока.

ADL также запрещен, если имя вызываемой функции заключено в круглые скобки.

В противном случае, если после имени следует заключенный в круглые скобки список выражений аргументов, ADL выполняется путем поиска имени в пространствах имен и классах, “связанных” или “ассоциированных” с типами аргументов вызова. Точное определение этих *связанных пространств имен и связанных классов* будет дано позже, но интуитивно их можно рассматривать как все пространства имен и классы, которые очевидным образом непосредственно имеют отношение к данному типу. Например, если тип является указателем на класс X, то связанные классы и пространство имен будут включать X, а также все пространства имен и классы, к которым принадлежит X.

Точное определение множества *связанных пространств имен и связанных классов* для данного типа регламентируется приведенными далее правилами.

- Для встроенных типов это пустое множество.
- Для типов указателей и массивов множество связанных пространств имен и классов – это пространства имен и классы лежащего в основе типа (на который указывает указатель или который является типом элемента массива).
- Для перечислимых типов связанным пространством имен является пространство имен, в котором объявлено перечисление.
- Для членов классов связанным классом является охватывающий класс.
- Для типов классов (включая объединения) множеством связанных классов является сам тип класса, его охватывающий класс, а также все непосредственные или опосредованные базовые классы. Множество связанных пространств имен представляет собой пространства имен, в которых объявлены связанные классы. Если класс является экземпляром шаблона класса, то сюда включаются и типы аргументов типов шаблона, а также классы и пространства имен, в которых объявлены шаблонные аргументы шаблона.
- Для типов функций множества связанных пространств имен и классов включают пространства имен и классы, связанные со всеми типами параметров, а также связанные с типами возвращаемых значений.
- Для указателей на члены класса X множества связанных пространств имен и классов включают пространства имен и классы, связанные с X в дополнение к связанным с типом члена класса (если это тип указателя на функцию-член, то учитываются также типы параметров и возвращаемых значений этой функции-члена).

При применении ADL осуществляется последовательный поиск имени во всех связанных пространствах имен так, как если бы это имя было квалифицировано поочередно с помощью каждого из этих пространств имен (директивы `using` при этом игнорируются). Этот механизм иллюстрируется приведенным ниже примером:

details/adl.cpp

```
#include <iostream>
namespace X
{
    template<typename T> void f(T);
}
namespace N
{
    using namespace X;
    enum E { e1 };
    void f(E)
    {
        std::cout << "Вызов N::f(N::E)\n";
    }
}

void f(int)
{
    std::cout << "Вызов ::f(int)\n";
}

int main()
{
    ::f(N::e1); // Квалифицированное имя функции: без ADL
    f(N::e1);   // Обычный поиск находит ::f(), ADL находит
}                                // N::f(), последний предпочтительнее
```

Заметим, что в данном примере директива `using` в пространстве имен `N` при выполнении ADL игнорируется. Следовательно, `X::f()` никогда даже не будет рассматриваться как кандидат для вызова в `main()`.

13.2.2. ADL объявлений друзей

Объявление дружественной функции может быть первым объявлением функции-кандидата при поиске. В этом случае считается, что функция объявлена в области видимости ближайшего пространства имен (которым может быть глобальное пространство имен), охватывающего класс, содержащий объявление дружественной функции. Однако такое объявление друга не является непосредственно видимым в данной области видимости. Рассмотрим следующий пример:

```
template<typename T>
class C
{
    ***
    friend void f();
    friend void f(C<T> const&);

    ***
};

void g(C<int>* p)
{
    f();    // Видима ли здесь f()?
    f(*p); // Видима ли здесь f(C<int> const&)?
}
```

Если объявления дружественных конструкций видимы в охватывающем пространстве имен, то инстанцирование шаблона класса может сделать видимыми объявления обычных функций. Это может привести к удивительному поведению: вызов `f()` ведет к ошибке компиляции, если только инстанцирование класса `C` не произошло ранее в программе!

С другой стороны, может быть полезным объявить (и определить) функцию только в объявлении друга (см. в разделе 21.2.1 методику, основанную на таком поведении). Такая функция может быть найдена, когда класс, для которого она является другом, находится среди связанных классов, рассматриваемых ADL.

Рассмотрим наш последний пример. Вызов `f()` не имеет связанных классов или пространств имен, поскольку не имеет аргументов: это некорректный вызов в нашем примере. Однако вызов `f(*p)` имеет связанный класс `C<int>` (поскольку это тип `*p`), и с ним также связано глобальное пространство имен (поскольку это пространство имен, в котором объявлен тип `*p`). Следовательно, объявление второй дружественной функции может быть найдено, если класс `C<int>` в действительности полностью инстанцирован до этого вызова. Чтобы обеспечить выполнение данного условия, предполагается, что вызов, инициирующий поиск дружественных конструкций в связанных классах, фактически вызывает инстанцирование класса (если оно еще не выполнено)³.

Возможность ADL найти объявления и определения друзей иногда называют *внесением*, или *инъекцией имени друга* (*friend name injection*). Однако этот термин несколько вводит в заблуждение, потому что это название возможности C++, имевшейся еще до разработки стандарта, которая на самом деле “вносит” имена объявлений друзей в охватывающую область видимости, делая их видимыми для обычного поиска имен. В приведенном выше примере это означает, что оба вызова корректны. В заключительных примечаниях к главе история внесения имен друзей рассматривается более подробно.

13.2.3. Внесение имен классов

Имя класса “внесено” в область видимости самого этого класса и, следовательно, является доступным в данной области видимости как неквалифицированное имя (однако оно недоступно в качестве квалифицированного имени, поскольку это запись, используемая для обозначения конструкторов). Например:

`details/inject.cpp`

```
#include <iostream>
int C;
class C
{
    private:
        int i[2];
    public:
```

³Хотя это очевидным образом входило в намерения тех, кто писал стандарт C++, из самого стандарта это не ясно.

```

static int f()
{
    return sizeof(C);
};

int f()
{
    return sizeof(C);
}

int main()
{
    std::cout << "C::f() = " << C::f() << ','
    << " ::f() = " << ::f() << '\n';
}

```

Функция-член `C::f()` возвращает размер типа `C`, в то время как функция `::f()` возвращает размер переменной `C` (другими словами, размер объекта типа `int`).

Шаблоны классов также имеют внесенные имена классов. Однако они еще более непривычны, чем обычные внесенные имена классов: за ними могут идти аргументы шаблона (в этом случае они являются внесенными именами *шаблона класса*), но если за ними не следуют аргументы шаблона, то они представляют класс с использованием параметров шаблонов в качестве аргументов (или при частичной специализации с использованием аргументов специализации), если контекст ожидает тип, или шаблон, если контекст ожидает шаблон. Это поясняет следующую ситуацию:

```

template<template<typename> class TT> class X
{
};

template<typename T> class C
{
    C* a;           // OK: то же, что и "C<T>* a;" 
    C<void>& b;     // OK
    X<C> c;        // OK: С без списка аргументов шаблонов
                    // обозначает шаблон С
    X<::C> d;      // OK: ::C не является внесенным именем класса,
                    // а потому всегда обозначает шаблон
};

```

Обратите внимание на то, как неквалифицированное имя ссылается на внесенное имя, и на то, что имя шаблона не рассматривается, если за ним не следует список аргументов. Для компенсации можно заставить имя шаблона быть найденным, используя квалификатор области видимости файла `::`.

Внесенное имя класса для вариативного шаблона имеет свои хитрости: если внесенное имя класса было сформировано непосредственно с использованием вариативных шаблонных параметров шаблона в качестве аргументов шаблона, то внесенное имя будет содержать пакеты параметров шаблона, которые не были раскрыты (о раскрытии пакетов см. раздел 12.4.1). Таким образом, при

формировании внесенного имени класса для вариативного шаблона аргумент шаблона, соответствующий пакету параметров шаблона, представляет собой раскрытие пакета, схемой которого является этот пакет параметров шаблона:

```
template<int I, typename... T> class V
{
    V* a;           // OK: то же, что и "V<I, T...>* a;"
    V<0, void> b; // OK
};
```

13.2.4. Текущие инстанцирования

Внесенное имя класса для класса или шаблона класса, по сути, является псевдонимом определяемого типа. Для класса, не являющегося шаблоном, это свойство очевидно, потому что сам класс является единственным типом с указанным именем в этой области видимости. Однако внутри шаблона класса или вложенного класса в шаблоне класса каждое инстанцирование шаблона производит другой тип. Это свойство особенно интересно в данном контексте, потому что означает, что внесенное имя класса относится к тому же инстанцированию шаблона класса, а не к некоторой иной специализации этого шаблона класса (то же самое справедливо и для вложенных классов шаблонов классов).

В шаблоне класса внесенное имя класса или любой тип, эквивалентный внесенному имени класса (включая просмотр объявлений псевдонимов типов) любого охватывающего класса или класса шаблона называется *ссылающимся на текущее инстанцирование*. Типы, которые зависят от параметра шаблона (то есть *зависимые типы*), но не ссылающиеся на текущее инстанцирование, называются *ссылающимися на неизвестную специализацию*, которая может быть инстанцирована из того же самого шаблона класса или некоторого совершенно иного шаблона класса. Следующий пример иллюстрирует это различие:

```
template<typename T> class Node
{
    using Type = T;
    Node* next;           // Node ссылается на текущее
                          // инстанцирование
    Node<Type>* previous; // Node<Type> ссылается на
                          // текущее инстанцирование
    Node<T*>* parent;   // Node<T*> ссылается на
                          // неизвестную специализацию
};
```

Определение, является ли тип ссылающимся на текущее инстанцирование, может быть особенно запутанным при наличии вложенных классов и шаблонов классов. Внесенные имена классов *охватывающих* классов и шаблонов классов (или эквивалентные им типы) ссылаются на текущее инстанцирование, в то время как имена других вложенных классов или шаблонов класса — нет:

```
template<typename T> class C
{
    using Type = T;
    struct I
    {
```

```

C* c;           // С ссылается на текущее инстанцирование
C<Type>* c2; // C<Type> ссылается на текущее инстанцирование
I* i;           // I ссылается на текущее инстанцирование
};

struct J
{
    C* c;           // С ссылается на текущее инстанцирование
    C<Type>* c2; // C<Type> ссылается на текущее инстанцирование
    I* i;           // I ссылается на неизвестную специализацию,
                    // поскольку I не охватывает J
    J* j;           // J ссылается на текущее инстанцирование
};

```

Когда тип ссылается на текущее инстанцирование, содержимое этого инстанцированного класса гарантированно инстанцируется из шаблона класса или его вложенного класса, который в настоящее время определяется. Это имеет последствия для поиска имен при синтаксическом анализе шаблонов (теме нашего следующего раздела), но это также приводит к альтернативному, больше похожему на игру способу определения, ссылается тип X в определении шаблона класса на текущее инстанцирование или на неизвестную специализацию: если другой программист может написать такую явную специализацию (описываемую подробно в главе 16, “Специализация и перегрузка”), что X ссылается на нее, то X ссылается на неизвестную специализацию. Рассмотрим, например, инстанцирование типа `C<int>::J` в контексте приведенного выше примера: мы знаем, что определение `C<T>::J` используется для инстанцирования конкретного типа (так как мы инстанцируем этот тип). Кроме того, поскольку явная специализация не может специализировать шаблон или член шаблона без одновременной специализации всех охватывающих шаблонов или членов, `C<int>` будет инстанцироваться из определения охватывающего класса. Следовательно, ссылки на `J` и `C<int>` (где Type представляет собой `int`), внутри `J` ссылаются на текущее инстанцирование. С другой стороны, можно было бы написать явную специализацию для `C<int>::I` следующим образом:

```

template<> struct C<int>::I
{
    // Определение специализации
};

```

Здесь специализация `C<int>::I` предоставляет совершенно иное определение, чем видимое из определения `C<T>::J`, так что `I` внутри определения `C<T>::J` ссылается на неизвестную специализацию.

13.3. Синтаксический анализ шаблонов

В большинстве случаев компилятор выполняет два фундаментальных действия — *токенизацию*, именуемую также лексическим анализом, и синтаксический анализ текста программы. При лексическом анализе исходный текст программы рассматривается как последовательность символов, из которой генерируется последовательность токенов. Например, если компилятор встречает последовательность символов `int* p = 0;`, лексический анализатор разделяет ее

на отдельные токены — ключевое слово `int`, символ/оператор `*`, идентификатор `p`, символ/оператор `=`, целочисленный литерал `0` и символ/оператор `;`.

После лексического анализа в дело вступает синтаксический анализ, который находит в последовательности токенов известные разрешенные языковые конструкции путем рекурсивной свертки токенов или обнаруженных ранее конструкций в конструкции более высокого уровня⁴. Например, токен `0` является корректным *выражением*, комбинация символа `*`, за которым следует идентификатор `p`, является корректным *декларатором*; а декларатор, за которым следует знак `=`, сопровождаемый выражением `"0"`, в свою очередь является корректным *инициализирующим декларатором*. И наконец, ключевое слово `int` является известным именем типа; когда за ним следует объявление инициализирующий декларатор `*p = 0`, мы получаем инициализирующее объявление переменной `p`.

13.3.1. Зависимость от контекста в нешаблонных конструкциях

Как вы, вероятно, знаете или предполагаете, лексический анализ осуществляется легче, чем синтаксический. К счастью, синтаксический анализ достаточно хорошо проработан теоретически, так что использование теории синтаксического анализа позволяет довольно легко разрабатывать синтаксические анализаторы для множества различных языков программирования. Лучше всего теория синтаксического анализа разработана для так называемых *контекстно-свободных языков программирования*, в то время как C++ является контекстно-зависимым языком программирования. В связи с этим компилятор C++ использует таблицы символов и при лексическом, и при синтаксическом анализе. Когда проводится анализ объявления, оно вносится в таблицу символов. Когда при лексическом анализе обнаруживается идентификатор, выполняется поиск в таблице символов, и если выясняется, что это тип, выполняется соответствующее аннотирование токена.

Например, если компилятор C++ встречает

`x*`

то лексический анализатор обнаруживает `x`. Если это тип, то анализатор видит идентификатор, тип, `x`
символ, `*`

и делает вывод, что начинается объявление. Однако если `x` не является типом, то синтаксический анализатор получает от лексического анализатора

идентификатор, не тип, `x`
символ, `*`

и корректный анализ данной конструкции возможен только как выражения умножения. Детали этих принципов зависят от конкретной реализации стратегии, но суть из этого примера должна быть ясна.

⁴ Подробнее о процессах лексического и синтаксического анализа вы можете прочесть в книге Ахо А., Лам М., Сети Р., Ульман Д. *Компиляторы: принципы, технологии и инструментарий*. — М.: Издательский дом “Вильямс”, 2008. — Примеч. ред.

Еще один пример контекстной зависимости иллюстрируется следующим выражением:

```
X<1>(0)
```

Если X является именем шаблона класса, то в предыдущем выражении целое 0 приводится к типу X<1>, сгенерированному из этого шаблона. Если X не является шаблоном, то предыдущее выражение эквивалентно следующему:

```
(X<1>)0
```

Другими словами, X сравнивается с 1, а результат этого сравнения — “истина” или “ложь” (которые в данном случае неявно преобразуются в 1 или 0) — сравнивается с 0. Хотя код, подобный приведенному, используется редко, он является корректным кодом C++ (и, кстати, корректным кодом С). Следовательно, синтаксический анализатор C++ будет проводить поиск имен, находящихся перед <, и интерпретировать < как угловую скобку, только если имя является именем шаблона; в противном случае < рассматривается как обычный символ оператора “меньше чем”.

Такая форма контекстной чувствительности — одно из неудачных последствий выбора для ограничения списка аргументов шаблона угловых скобок. Ниже приведен пример еще одного такого следствия:

```
template<bool B>
class Invert
{
public:
    static bool const result = !B;
};
void g()
{
    bool test = Invert<(1>0>>::result; // Необходимы круглые скобки!
}
```

Если опустить круглые скобки в выражении Invert<(1>0>), то символ “больше чем” будет ошибочно воспринят в качестве закрывающего список аргументов шаблона. Это сделало бы код неверным, поскольку компилятор воспринял его как эквивалент ((Invert<1>))0>::result⁵.

Лексический анализатор также не лишен проблем, связанных с угловыми скобками. Например, в

```
List<List<int>> a;
// ^-- пробела между правыми угловыми скобками нет
```

два символа > объединяются в токен правого сдвига >>, и, следовательно, *лексическим анализатором* как два отдельных токена не рассматривается. Это следствие принципа *максимального поглощения* (*maximum munch*) при лексическом

⁵ Отметим, что двойные скобки, которые используются для того, чтобы избежать синтаксического анализа выражения (Invert<1>)0 как оператора приведения, — еще один источник синтаксической неопределенности.

анализе: реализация C++ должна собирать в токен как можно больше идущих подряд символов⁶.

Как уже упоминалось в разделе 2.2, начиная с C++11, стандарт C++ выделяет этот случай, когда вложенные идентификаторы шаблонов закрываются с помощью токена сдвига вправо `>>`, а синтаксический анализатор рассматривает сдвиг вправо как эквивалент двух отдельных правых угловых скобок `>` и `>`, чтобы можно было завершать вложенные идентификаторы шаблонов без дополнительного пробела⁷. Интересно отметить, что это изменение автоматически меняет смысл некоторых (следует признать, весьма надуманных) программ. Рассмотрим следующий пример:

names/anglebrackethack.cpp

```
#include <iostream>
template<int I> struct X
{
    static int const c = 2;
};
template<> struct X<0>
{
    typedef int c;
};
template<typename T> struct Y
{
    static int const c = 3;
};
static int const c = 4;
int main()
{
    std::cout << (Y<X<1> >::c >::c >::c) << ' ';
    std::cout << (Y<X< 1>>::c >::c >::c) << '\n';
}
```

Эта корректная с точки зрения стандарта C++98 программа выводит

0 3

Но эта же программа (являющаяся корректной программой с точки зрения стандарта C++11) при рассмотрении двух закрывающих угловых скобок как завершения вложенного шаблона в C++11⁸ дает вывод

0 0

⁶ В этот принцип были добавлены некоторые исключения, необходимые для решения описываемых в данном разделе проблем.

⁷ Версии стандарта C++ 1998 и 2003 гг. не поддерживали такое закрытие вложенных шаблонов. Однако необходимость введения пробела между двумя идущими подряд правыми угловыми скобками оказалась такой неприятной для программистов, работающих с шаблонами, что Комитет по стандартизации решил внести соответствующее исправление в стандарт 2011 г.

⁸ Некоторые компиляторы, предоставляющие возможность выбора работы в режимах C++98 или C++03, сохраняют в этих режимах поведение C++11 и дают два нуля даже при формальной компиляции в режиме C++98/C++03.

Имеется аналогичная проблема, связанная с применением диграфа < : в качестве альтернативы для символа [(который недоступен на некоторых традиционных клавиатурах). Рассмотрим следующий пример:

```
template<typename T> struct G {};
struct S;
G<::S> gs; // Корректный код C++11; ошибка в более ранних версиях
```

До C++11 последняя строка кода была эквивалентна явно недопустимой записи G[: S> gs;. Для решения этой проблемы был добавлен еще один “лексический хак”: когда компилятор видит символы < : :, за которыми не сразу следуют : или >, ведущая пара знаков < : не рассматривается как токен диграфа, эквивалентный [⁹. Этот лексический хак может сделать ранее корректный (конечно, задуманный) код неверным¹⁰:

```
#define F(X) X ## :
int a[] = { 1, 2, 3 }, i = 1;
int n = a F(<:::)i]; // Корректно в C++98/C++03, но не в C++11
```

Чтобы понять, что происходит, обратите внимание на то, что “лексический хак” применяется к *токенам препроцессора*, которые представляют собой токены, приемлемые для препроцессора (они могут стать неприемлемыми после обработки препроцессором), и они обрабатываются до завершения раскрытия макроса. С учетом сказанного, C++98/C++03 безоговорочно превращает < : в [в вызове макроса F(< :: :), так что определение n раскрывается в

```
int n = a [ :: i];
```

что вполне корректно. Однако C++11 не выполняет преобразования диграфа, потому что до раскрытия макроса за последовательностью < :: : не следуют ни :, ни >, а). Без преобразования диграфа оператор конкатенации ## должен попытаться собрать :: и : в новый токен препроцессора, но это не работает, потому что :: : не является допустимым токеном. Стандарт объявляет это неопределенным поведением, которое позволяет компилятору делать все, что угодно. Некоторые компиляторы будут диагностировать эту проблему, в то время как другие не будут и просто разделят два токена препроцессора, что приведет к синтаксической ошибке, так как в этом случае будет получено такое определение n:

```
int n = a < :: : i];
```

13.3.2. Зависимые имена типов

Проблемы с именами в шаблонах не всегда удается полностью классифицировать. В частности, один шаблон не может заглянуть в другой шаблон, поскольку содержимое последнего может оказаться некорректным в силу явной специализации. Ниже приведен несколько искусственный пример, иллюстрирующий данное утверждение:

⁹ Таким образом, это еще одно исключение из упоминавшегося выше принципа *максимального поглощения*.

¹⁰ Благодарим Ричарда Смита (Richard Smith) за данный пример.

```

template<typename T>
class Trap
{
public:
    enum { x };           // #1 Здесь x не является типом
};

template<typename T>
class Victim
{
public:
    int y;
    void poof()
    {
        Trap<T>::x* y; // #2 Объявление или умножение?
    }
};

template<>
class Trap<void>           // Специализация!
{
public:
    using x = int;        // #3 Здесь x является типом
};

void boom(Victim<void>& bomb)
{
    bomb.poof();
}

```

Когда компилятор выполняет синтаксический анализ строки #2, он должен решить, с какой конструкцией он имеет дело — с объявлением или умножением. Это решение, в свою очередь, зависит от того, является ли зависимое квалифицированное имя `Trap<T>::x` именем типа. Неплохо бы, конечно, заглянуть в этот момент в шаблон `Trap`, и обнаружить, что согласно строке #1 `Trap<T>::x` не является типом, поэтому для строки #2 остается только умножение. Однако несколько позже все портит имеющийся код перекрытия `Trap<T>::x` для случая, когда `T` является `void`. В этом случае `Trap<T>::x` на самом деле представляет собой тип `int`.

В этом примере тип `Trap<T>` является *зависимым типом*, потому что тип зависит от параметра `T` шаблона. Кроме того, `Trap<T>` ссылается на неизвестную специализацию (описана в разделе 13.2.4), так что компилятор не может безопасно заглянуть внутрь шаблона, чтобы определить, является ли имя `Trap<T>` типом или нет. Должен ли тип, предваряемый `::`, ссылаться на текущее инстанцирование — например, с `Victim<T>::y` — компилятор мог бы выяснить из определения шаблона, поскольку понятно, что здесь не может вклиниваться никакая иная специализация. Таким образом, когда тип, предваренный `::`, ссылается на текущее инстанцирование, поиск квалифицированных имен в шаблоне ведет себя очень похоже на поиск квалифицированных имен для несвязанных типов.

Однако, как иллюстрирует пример, поиск имен в неизвестной специализации по-прежнему остается проблемой. Определение языка разрешает эту проблему,

указывая, что в общем случае зависимое квалифицированное имя *не* обозначает тип, если только это имя не предваряется ключевым словом `typename`. Если после подстановки аргументов шаблона оказывается, что имя не является именем типа, программа считается некорректной, и компилятор C++ должен пожаловаться вам на это во время инстанцирования. Обратите внимание на то, что использование `typename` отличается от использования этого ключевого слова для обозначения параметров типа шаблона. В отличие от параметров типа нельзя заменить ключевое слово `typename` ключевым словом `class`.

Префикс `typename` для имени *необходим*, когда имя удовлетворяет всем следующим условиям¹¹:

1. Оно квалифицировано, но за ним не следует `::` для образования более квалифицированного имени.
2. Оно не является частью *уточненного спецификатора типа* (*elaborated type specifier*) (т.е. имени типа, которое начинается с одного из ключевых слов `class`, `struct`, `union` или `enum`).
3. Оно не используется в списке спецификаций базовых классов или в списке инициализаторов членов, вводимых определением конструктора¹².
4. Оно зависит от параметра шаблона.
5. Оно является *членом неизвестной специализации*, что означает, что тип, именованный квалификатором, ссылается на неизвестную специализацию.

Кроме того, префикс `typename` *не разрешен*, если только не выполнены как минимум два первых условия. Для иллюстрации рассмотрим следующий ошибочный пример¹³:

```
template<typename, T>
struct S : typename, X<T>::Base
{
    S() : typename, X<T>::Base(typename, X<T>::Base(0))
    {
    }
    typename, X<T> f()
    {
        typename, X<T>::C* p; // Объявление указателя p
        X<T>::D* q;          // Умножение!
    }
    typename, X<int>::C* s;
    using Type = T;
    using OtherType = typename, S<T>::Type;
};
```

¹¹ Заметим, что C++20, вероятно, удалит необходимость `typename` в большинстве случаев (подробнее об этом см. раздел 17.1).

¹² Синтаксически в этих контекстах разрешены только имена типов, так что квалифицированное имя всегда предполагается именем типа.

¹³ Адаптировано из [72], доказав раз и навсегда, что C++ способствует повторному использованию кода.

Каждое `typename` — корректное либо нет — для удобства указания помечено подстрочным номером. Первое `typename1`, означает параметр шаблона. К этому первому использованию `typename` приведенные выше правила не относятся. Второе и третье включение `typename` не разрешается согласно второму правилу. Именам базовых классов в этих двух контекстах не может предшествовать `typename`. Однако `typename4` должно быть указано. Здесь имя базового класса используется не для обозначения того, что должно инициализироваться или порождаться из чего-либо, а является частью выражения для создания временного объекта `X<T> : : Base` из аргумента 0 (если угодно — это можно рассматривать как разновидность преобразования типов). Пятое `typename` запрещено, поскольку имя, которое за ним следует (`X<T>`), не является квалифицированным именем. Шестое вхождение требуется, если это выражение предназначено для объявления указателя. В следующей строке ключевое слово `typename` отсутствует, поэтому она интерпретируется компилятором как умножение. Седьмое `typename` необязательно, поскольку оно удовлетворяет первым двум правилам, но не удовлетворяет двум последним. И наконец `typename8` не обязательное, поскольку ссылается на член текущего инстанцирования (а потому не удовлетворяет последнему правилу).

Последнее из правил определения, требуется ли префикс `typename`, иногда может оказаться сложно вычислимым, потому что оно зависит от правил определения, ссылается ли тип на текущее инстанцирование или на неизвестную специализацию. В таких случаях безопаснее добавить ключевое слово `typename`, чтобы указать, что ваше намерение в том, чтобы следующее далее квалифицированное имя было типом. Ключевое слово `typename`, даже будучи необязательным, будет документировать ваши намерения.

13.3.3. Зависимые имена шаблонов

Проблема, во многом подобная той, с которой мы столкнулись в предыдущем разделе, возникает и в случае, когда имя шаблона является зависимым. В общем случае от компилятора C++ требуется, чтобы он рассматривал знак `<`, следующий за именем шаблона, как начало списка аргументов шаблона; в противном случае это оператор “меньше чем”. Как и в случае с именами типов, компилятор должен предполагать, что зависимое имя не ссылается на шаблон, если только программист не обеспечивает дополнительную информацию с помощью ключевого слова `template`:

```
template<typename T>
class Shell
{
public:
    template<int N>
    class In
    {
public:
    template<int M>
    class Deep
    {
public:
```

```

        virtual void f();
    };
};

template<typename T, int N>
class Weird
{
public:
    void case1(typename Shell<T>::template In<N>::template Deep<N>*&p)
    {
        p->template Deep<N>::f(); // Запрет виртуального вызова
    }
    void case2(typename Shell<T>::template In<N>::template Deep<N>*&p)
    {
        p.template Deep<N>::f(); // Запрет виртуального вызова
    }
};

```

В этом несколько запутанном примере показано, как для всех операторов, могущих квалифицировать имя `(::, -> и .)`, может потребоваться использовать ключевое слово `template`. В частности, оно требуется всякий раз, когда тип имени или выражения, предшествующего квалифицирующему оператору, зависит от параметра шаблона и ссылается на неизвестную специализацию, а имя, которое следует за оператором, является идентификатором шаблона (другими словами, имя шаблона, за которым следуют аргументы шаблона в угловых скобках). Например, в выражении

`p.template Deep<N>::f()`

тип `p` зависит от параметра шаблона `T`. Следовательно, компилятор C++ не может проводить поиск `Deep` для выяснения, является ли это имя шаблоном, и необходимо явно указать это с помощью предшествующего данному имени ключевого слова `template`. Без этого предваряющего ключевого слова синтаксический анализ `p.Deep<N>::f()` проводится следующим образом: `((p.Deep)<N>)f()`. Заметим также, что может потребоваться использовать ключевое слово `template` несколько раз в пределах одного квалифицированного имени, поскольку квалификаторы сами по себе могут быть квалифицированы с помощью зависимого квалификатора (что иллюстрируют объявления параметров `case1` и `case2` в предыдущем примере).

Если опустить ключевое слово `template` в таких ситуациях, как показанная, то открывающая и закрывающая угловые скобки анализируются как операторы “меньше чем” и “больше чем”. Как и в случае ключевого слова `typename`, можно безопасно добавить префикс `template`, чтобы указать, что следующее имя является идентификатором шаблона, даже если в данном конкретном случае префикс `template` не является строго необходимым.

13.3.4. Зависимые имена в объявлениях `using`

Объявления `using` могут быть привнесены в имена из двух мест — пространств имен и классов. Случай пространств имен в данном контексте нас не интересует,

поскольку не существует *шаблонов пространств имен*. Что касается классов, то в действительности объявления `using` могут привноситься только из базового класса в порожденный. Такие объявления `using` в порожденном классе ведут себя как “символические связи” или “ярлыки”, направленные из производного класса к базовому, тем самым позволяя членам производного класса обращаться к соответствующему имени базового класса, как если бы оно было объявлено в производном классе. Краткий, не содержащий шаблонов, пример проиллюстрирует сказанное лучше, чем множество слов.

```
class BX
{
public:
    void f(int);
    void f(char const*);
    void g();
};

class DX : private BX
{
public:
    using BX::f;
};
```

Объявление `using` привносит имя `f` из базового класса `BX` в порожденный класс `DX`. В данном случае это имя связано с двумя разными объявлениями; таким образом подчеркивается, что мы имеем дело с механизмом для имен, а не с отдельными объявлениями таких имен. Заметим также, что такой вид `using`-объявления может сделать доступным член класса, который в противном случае был бы недоступен. Базовый класс `BX` (и, соответственно, его члены) является закрытым по отношению к классу `DX`, за исключением функций `BX::f`, которые введены в открытый интерфейсе `DX` и являются, следовательно, доступными для клиентов `DX`.

Теперь вы, вероятно, можете осознать проблему, когда объявление `using` привносит имя из зависимого класса. Хотя мы знаем об имени, мы не знаем, является ли это имя типом, шаблоном или чем-то иным:

```
template<typename T>
class BXT
{
public:
    using Mystery = T;
    template<typename U>
    struct Magic;
};

template<typename T>
class DXTT : private BXT<T>
{
public:
    using typename BXT<T>::Mystery;
    Mystery* p; // Без ранее использованного typename
                // было бы синтаксической ошибкой
};
```

И вновь, если мы хотим, чтобы зависимое имя было внесено объявлениями `using` для обозначения типа, мы должны явно сказать об этом, использовав ключевое слово `typename`. Как ни странно, стандарт C++ не предусматривает аналогичный механизм для обозначения таких зависимых имен, как шаблоны. Следующий фрагмент кода иллюстрирует данную проблему:

```
template<typename T>
class DXTM : private BXT<T>
{
public:
    using BXT<T>::template Magic; // Ошибка: не стандарт
    Magic<T>* plink;           // Синтаксическая ошибка: Magic
};                                // не является известным шаблоном
```

Комитет по стандартизации не склонен решать эту проблему. Однако шаблоны псевдонимов C++ предоставляют частичный обходной путь:

```
template<typename T>
class DXTM : private BXT<T>
{
public:
    template<typename U>
    using Magic = typename
        BXT<T>::template Magic<T>; // Шаблон псевдонима
    Magic<T>* plink;           // OK
};
```

Это несколько громоздкое решение, но оно достигает желаемого эффекта в случае шаблонов классов. В случае шаблонов функций (пожалуй, менее распространенному) проблема, к сожалению, остается нерешенной.

13.3.5. ADL и явные аргументы шаблонов

Рассмотрим следующий пример:

```
namespace N
{
    class X
    {
        ...
    };

    template<int I> void select(X* );
}

void g(N::X* xp)
{
    select<3>(xp); // Ошибка: ADL не выполняется!
}
```

В этом примере можно предположить, что в вызове `select<3>(xp)` шаблон `select()` отыскивается с помощью ADL. Однако это не так, поскольку компилятор не может принять решение о том, что `xp` является аргументом вызова функции, пока не будет решено, что `<3>` является списком аргументов шаблона. И наоборот, невозможно решить, что `<3>` является списком аргументов шаблона, пока

не выяснится, что `select()` представляет собой шаблон. Поскольку эту проблему курицы и яйца разрешить невозможно, выражение анализируется как `(select<3>)(xp)`, что не имеет смысла.

Этот пример может создать впечатление, что ADL для идентификаторов шаблонов не работает, но это не так. Приведенный код может быть исправлен путем введения шаблона функции с именем `select`, который виден при вызове:

```
template<typename T> void select();
```

Несмотря на то что это не имеет никакого смысла для вызова `select<3>(xp)`, наличие этого шаблона функции гарантирует, что `select<3>` будет при анализе рассматриваться как идентификатор шаблона. Затем ADL находит шаблон функции `N::select`, и вызов будет успешным.

13.3.6. Зависимые выражения

Подобно именам, выражения сами по себе могут быть зависимыми от параметров шаблона. Выражение, которое зависит от параметра шаблона, может вести себя по-разному от одного инстанцирования к другому — например, выбирая другую перегруженную функцию или производя другой тип или значение константы. Выражения, которые не зависят от параметра шаблона, напротив, обеспечивают одинаковое поведение во всех инстанцированиях.

Выражение может быть зависимым от параметра шаблона несколькими различными способами. Наиболее распространенной формой зависимых выражений является *выражение, зависящее от типа*, где тип самого выражения может варьироваться от одного инстанцирования к другому — например, выражение, которое ссылается на параметр функции, типом которого является тип параметра шаблона:

```
template<typename T> void typeDependent1(T x)
{
    x; // Выражение зависит от типа, поскольку
} // тип x может варьироваться
```

Выражения, которые имеют зависимые от типа подвыражения, в общем случае сами являются зависимыми от типа — например, вызов функции `f()` с аргументом `x`:

```
template<typename T> void typeDependent2(T x)
{
    f(x); // Выражение зависит от типа, поскольку
} // от типа зависит x
```

Обратите внимание на то, что здесь тип `f(x)` может варьироваться от одного инстанцирования к другому, как потому что вызов `f` может быть разрешен в шаблон, тип результата которого зависит от типа аргумента, так и из-за того, что двухфазный поиск (который рассматривается в разделе 14.3.1) в различных инстанцированиях может найти совершенно разные функции с именем `f`.

Не все выражения, которые включают параметры шаблона, зависимы от типа. Например, выражение, включающее параметры шаблона, может выдавать разные

константные *значения* от инстанцирования к инстанцированию. Такие выражения называются *зависящими от значения*, и простейшими из них являются те, которые ссылаются на нетиповой параметр шаблона независимого типа. Например:

```
template<int N> void valueDependent1()
{
    N; // Это выражение зависито от значения, но не от типа,
        // поскольку N имеет фиксированный тип, но варьируемое
} // константное значение
```

Подобно зависимым от типа выражениям, выражения зависят от значений в общем случае тогда, когда они составлены из других подвыражений, зависящих от значения, поэтому $N+N$ или $f(N)$ также являются зависящими от значения выражениями.

Интересно, что некоторые операции, такие как `sizeof`, имеют известный тип результата, поэтому они могут превратить зависящий от типа операнд в зависящее от значения выражение, которое не зависит от типа. Например:

```
template<typename T> void valueDependent2(T x)
{
    sizeof(x); // Выражение зависит от значения, но не от типа
}
```

Операция `sizeof` всегда дает значение типа `std::size_t`, независимо от ее входных данных, так что выражение `sizeof` никогда не является зависящим от типа, даже если — как в данном случае — его подвыражение является зависимым от типа. Однако результатирующее значение константы будет варьироваться от одного инстанцирования к другому, так что `sizeof(x)` является выражением, зависящим от значения.

Что если мы применим `sizeof` к выражению, зависящему от значения?

```
template<typename T> void maybeDependent(T const& x)
{
    sizeof(sizeof(x));
}
```

Здесь внутреннее выражение `sizeof` является зависимым от значения, как отмечалось выше. Однако внешнее выражение `sizeof` всегда вычисляет размер типа `std::size_t`, так что и тип, и значение получаемой константы одинаково при любых инстанцированиях этого шаблона, несмотря на то, что наиболее глубоко вложенное выражение (`x`) является зависимым от типа. Любое выражение, включающее параметр шаблона, является *выражением, зависящим от инстанцирования*¹⁴, даже если и его тип, и константное выражение являются инвариант-

¹⁴ Термины *выражение, зависящее от типа* и *выражение, зависящее от значения*, использовались в стандарте C++ для описания семантики шаблонов и влияли на некоторые аспекты инстанцирования шаблонов (глава 14, “Инстанцирование”). С другой стороны, термин *выражение, зависящее от инстанцирования*, используется в основном авторами компиляторов C++. Наше определение выражений, зависящих от инстанцирования, взято из *Itanium C++ ABI* [43], где представлена основа для бинарной совместимости ряда различных компиляторов C++.

том для различных корректных инстанцирований. Однако выражения, зависящие от инстанцирования, могут оказаться некорректны при инстанцировании. Например, инстанцирование `maybeDependent()` с неполным типом класса приведет к ошибке, так как `sizeof` не может быть применено к таким типам.

Зависимость от типа, значения и инстанцирования можно рассматривать как ряд все более всеобъемлющих классификаций выражений. Любое выражение, зависящее от типа, также рассматривается как зависящее от значения, поскольку выражение, тип которого варьируется от одного инстанцирования к другому, естественно, будет иметь постоянно меняющееся значение в разных инстанцированиях. Аналогично выражение, тип или значение которого варьируется от одного инстанцирования к другому, некоторым образом зависит от параметра шаблона, поэтому как выражения, зависящие от типа, так и выражения, зависящие от значения, зависят от инстанцирования. Это отношение включения иллюстрируется на рис. 13.1.

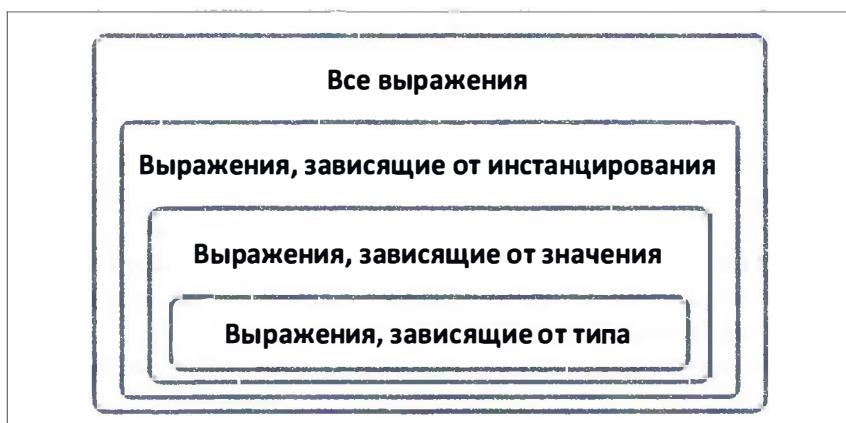


Рис. 13.1. Отношение включения среди выражений, зависящих от типа, значения и инстанцирования

По мере передвижения от наиболее глубоко вложенного внутреннего контекста (выражения, зависимые от типа) ко внешнему, поведение шаблона определяется при синтаксическом анализе во все большей степени, и поэтому не может варьироваться от одного инстанцирования к другому. Рассмотрим, например, вызов `f(x)`: если `x` зависит от типа, то `f` является зависимым именем, которое является субъектом двухфазного поиска (раздел 14.3.1), тогда как если `x` является зависимым от значения, но не от типа, `f` является независимым именем, для которого поиск имени может быть полностью выполнен во время синтаксического анализа шаблона.

13.3.7. Ошибки компиляции

Компилятор C++ может (но не обязан!) диагностировать ошибку во время синтаксического анализа шаблона, когда все инстанцирования шаблона приводят

к этой ошибке. Давайте расширим пример `f(x)` из предыдущего раздела для более подробного изучения:

```
void f() { }
template<int x>
void nondependentCall()
{
    f(x); // x зависит от значения, так что имя f независимое;
           // этот вызов не может быть успешным
}
```

Здесь вызов `f(x)` ведет к ошибке в каждом инстанцировании, потому что `f` является независимым именем, а единственная видимая `f` принимает нуль аргументов, а не один. Компилятор C++ может сообщить об ошибке при синтаксическом анализе шаблона, а может ждать первого инстанцирования шаблона. Самые распространенные компиляторы ведут себя по-разному даже в этом простом примере. Можно создать подобные примеры с выражениями, зависящими от инстанцирования, но не зависящими от значения:

```
template<int N>
void instantiationDependentBound()
{
    constexpr int x = sizeof(N);
    constexpr int y = sizeof(N) + 1;
    int array[x-y]; // Массив имеет отрицательный размер
}                   // во всех инстанцированиях
```

13.4. Наследование и шаблоны классов

Шаблоны классов могут как порождать производные классы, так и сами быть производными классами. В большинстве случаев особой разницы между сценариями с использованием шаблонов и без них нет; однако есть один важный тонкий момент при порождении шаблона класса из базового класса, обращение к которому выполняется с помощью зависимого имени. Давайте сначала рассмотрим более простой случай независимых базовых классов.

13.4.1. Независимые базовые классы

В шаблоне класса независимый базовый класс является классом с полным типом, который может быть определен без знания аргументов шаблона. Другими словами, для обозначения этого класса используется независимое имя.

```
template<typename X>
class Base
{
public:
    int basefield;
    using T = int;
};

class D1: public Base<Base<void>> // В действительности
{                                     // это не шаблон
```

```

public:
    void f()
    {
        basefield = 3;           // Обычный доступ к
    }                           // унаследованному члену
};

template<typename T>
class D2 : public Base<double>      // Независимый базовый класс
{
public:
    void f()
    {
        basefield = 7;           // Обычный доступ к
    }                           // унаследованному члену
    T strange;                 // T не параметр шаблона -
};                                // это Base<double>::T!

```

Поведение независимых базовых классов в шаблонах очень похоже на поведение базовых классов в обычных нешаблонных классах, однако здесь имеет место некоторая досадная неожиданность: когда поиск неквалифицированного имени выполняется в производном шаблоне, независимые базовые классы рассматриваются до списка параметров шаблона. Это означает, что в предыдущем примере член класса `strange` шаблона класса `D2` всегда имеет тип `T`, соответствующий `Base<double>::T` (другими словами, `int`). Например, следующая функция (при использовании предыдущих объявлений) с точки зрения C++ некорректна:

```

void g(D2<int*>& d2, int* p)
{
    d2.strange = p; // Ошибка: несоответствие типов!
}

```

Такое поведение далеко не интуитивно и требует от разработчика порожденного шаблона внимания по отношению к именам в независимых базовых классах, от которых он порождается, даже когда это порождение является непрямым или имена являются закрытыми. Вероятно, было бы предпочтительнее разместить параметры шаблона в области видимости “шаблонизируемой” сущности.

13.4.2. Зависимые базовые классы

В предыдущем примере базовый класс был полностью определенным и не зависел от параметра шаблона. Это означает, что компилятор C++ может искать независимые имена в тех базовых классах, где видимо определение шаблона. Альтернатива (не разрешенная стандартом C++) могла бы заключаться в отсрочке поиска таких имен, пока шаблон не будет инстанцирован. Недостаток этого подхода состоит в том, что до инстанцирования откладываются все сообщения об ошибках. Поэтому в стандарте C++ указано, что поиск независимого имени, присутствующего в шаблоне, происходит немедленно после того, как компилятор встречает его. Рассмотрим с учетом сказанного приведенный ниже пример.

```

template<typename T>
class DD : public Base<T>      // Зависимый базовый класс

```

```

{
public:
    void f()
    {
        basefield = 0;           // #1   Проблема...
    }
};

template<>                     // явная специализация
class Base<bool>
{
public:
    enum { basefield = 42 }; // #2   Трюк!
};

void g(DD<bool>& d)
{
    d.f();                   // #3   Хмм?
}

```

В точке #1 имеется ссылка на независимое имя `basefield`, поиск которого следует провести немедленно. Предположим, что оно найдено в шаблоне `Base` и связано с членом класса с типом `int` в этом классе. Однако вскоре после этого компилятор встречает явную специализацию данного класса. Когда это происходит, смысл члена класса `basefield` изменяется — при том, что его старый смысл уже использован! Так, при инстанцировании определения `DD::f` в точке #3 выясняется, что независимое имя в точке #1 связано с членом класса типа `int` преждевременно — в `DD<bool>`, специализированном в точке #2, не существует переменной `basefield`, которой можно было бы присвоить новое значение, так что компилятором будет выдано сообщение об ошибке.

Чтобы обойти эту проблему, стандарт C++ гласит, что поиск независимых имен *не* проводится в зависимых базовых классах¹⁵ (однако сам поиск выполняется, как только эти имена встречаются компилятором). Таким образом, соответствующий стандарту C++ компилятор выдаст диагностику в точке #1. Для исправления этого кода достаточно сделать имя `basefield` зависимым, поскольку поиск зависимых имен может проводиться только во время инстанцирования шаблона, а к этому моменту точная специализация базового класса, где будет вестись поиск, уже будет известна. Например, в точке #3 компилятор уже будет знать, что базовым по отношению к `DD<bool>` является класс `Base<bool>`, явно специализированный программистом. Сделать имя зависимым можно, например, как показано ниже.

```
// Вариант 1:
template<typename T>
class DD1 : public Base<T>
{
public:
```

¹⁵ Это часть так называемого *правила двухфазного поиска*, в котором различаются первая фаза, когда определения шаблона встречаются впервые, и вторая фаза, когда происходит инстанцирование шаблона (см. раздел 14.3.1).

```

void f()
{
    this->basefield = 0;      // Поиск отложен
}
};

```

Еще один вариант — введение зависимости с помощью квалифицированного имени.

```

// Вариант 2:
template<typename T>
class DD2 : public Base<T>
{
public:
    void f()
    {
        Base<T>::basefield = 0;
    }
};

```

Применение этого варианта требует особой тщательности, поскольку если неквалифицированное независимое имя используется для формирования вызова виртуальной функции, то квалификация подавляет механизм виртуального вызова и смысл программы изменяется. Несмотря на это, существуют ситуации, когда первый вариант нельзя использовать и приходится применять альтернативный.

```

template<typename T>
class B
{
public:
    enum E { el = 6, e2 = 28, e3 = 496 };
    virtual void zero(E e = el);
    virtual void one(E&);
};

template<typename T>
class D : public B<T>
{
public:
    void f()
    {
        typename D<T>::E e; // this->E синтаксически неверно
        this->zero();       // D<T>::zero() подавляет виртуальность
        one(e);             // one является зависимым именем
    }                     // из-за зависимости аргумента
};

```

Обратите внимание на то, что в этом примере мы использовали `D<T>::E` вместо `B<T>::E`. В этом случае работает любой вариант. Но в случаях множественного наследования мы не можем знать, какой именно базовый класс предоставляет требуемый член (в таком случае работает квалификация с помощью производного класса), или несколько базовых классов могут объявлять одно и то же имя (в этом случае может потребоваться использовать конкретное имя базового класса для устранения неоднозначности).

Заметим, что имя `one` в вызове `one (e)` зависито от параметра шаблона просто потому, что тип одного из явно заданных аргументов вызова является зависимым. Неявно используемые аргументы по умолчанию с типом, который зависит от параметра шаблона, во внимание не принимаются, поскольку компилятор не может их проверить до тех пор, пока не будет проведен поиск, — все та же проблема курицы и яйца. Чтобы избежать таких нюансов, предпочтительно использовать префикс `this->` во всех ситуациях, где только можно, — даже для нешаблонного кода.

Если вы обнаружите, что повторяющиеся квалификаторы загромождают ваш код, можно внести имя из зависимого базового класса в порожденный класс раз и навсегда.

```
// Версия 3:
template<typename T>
class DD3 : public Base<T>
{
public:
    using Base<T>::basefield; // #1      Теперь зависимое имя
                                // в области видимости
    void f()
    {
        basefield = 0;          // #2      Все в порядке
    }
};
```

Поиск в точке #2 успешен и находит *объявление using* в точке #1. Однако объявление *using* не проверяется до инстанцирования, так что поставленная цель достигнута. Эта схема имеет несколько тонких ограничений. Например, если осуществляется множественное наследование из нескольких базовых классов, программист должен точно указать, какой из них содержит необходимый член.

При поиске квалифицированного имени в текущем инстанцировании стандарт C++ указывает, что сначала имя ищется в текущем инстанцировании и всех независимых базовых классах, подобно выполнению неквалифицированного поиска для этого имени. Если найдено любое имя, то квалифицированное имя ссылается на член текущего инстанцирования и не будет зависимым именем¹⁶. Если такое имя не найдено, и класс имеет любые зависимые базовые классы, то квалифицированное имя ссылается на член неизвестной специализации. Например:

```
class NonDep
{
public:
    using Type = int;
};

template<typename T>
class Dep
{
public:
```

¹⁶ Тем не менее поиск повторяется при инстанцировании шаблона, и, если в этом контексте получается иной результат, программа считается некорректной.

```

        using OtherType = T;
};

template<typename T>
class DepBase : public NonDep, public Dep<T>
{
public:
    void f()
    {
        typename DepBase<T>::Type t;
        // Находится NonDep::Type; ключевое
        // слово typename не обязательное

        typename DepBase<T>::OtherType* ot;
        // Не находится ничего; DepBase<T>::OtherType
        // является членом неизвестной специализации
    }
};

```

13.5. Заключительные замечания

Первый компилятор, который действительно выполнял синтаксический анализ определений шаблонов, был разработан компанией Telligent в середине 1990-х годов. До этого — и даже несколько лет после этого — большинство компиляторов обрабатывали шаблоны как последовательность токенов, синтаксический анализ которой выполнялся во время инстанцирования. Поэтому никакой синтаксический анализ, за исключением минимального, достаточного для выявления конца определения шаблона, не выполнялся. На момент написания данной книги компилятор Microsoft Visual C++ все еще работает именно таким образом. Компилятор Edison Design Group (EDG) использует гибридный метод, когда шаблоны внутренне рассматриваются как последовательность аннотированных токенов, но “обобщенный синтаксический анализ” осуществляется для проверки синтаксиса в режимах, где он желателен (продукт EDG эмулирует несколько других компиляторов; в частности, он может точно эмулировать поведение компилятора от Microsoft).

Билл Гиббонс (Bill Gibbons) был представителем Telligent в Комитете стандартизации C++ и главным адвокатом однозначного синтаксического анализа шаблонов. Усилия Telligent не увенчались выпуском продукта до тех пор, пока компилятор не был приобретен и завершен Hewlett-Packard (HP) под именем компилятора aC++. Среди конкурентных преимуществ компилятора aC++ было высокое качество его диагностики. Тот факт, что диагностика шаблонов не всегда задерживалась до времени инстанцирования, несомненно, способствовал его успеху.

В относительно ранние времена разработки шаблонов Том Пеннелло (Tom Pennello) — широко признанный эксперт в области синтаксического анализа, работавший на Metaware, — отметил некоторые из проблем, связанных с угловыми скобками. Страуструп также высказал свои комментарии на эту тему в [66] и утверждал, что программисты предпочитают угловые скобки круглым. Однако

существуют и другие возможности; в частности, Пеннелло предложил использовать фигурные скобки (например, `List{::X}`) на симпозиуме по стандартизации C++ в Далласе в 1991 году¹⁷. В то время масштабы этой проблемы были более ограничены, потому что еще не было шаблонов, вложенных в другие шаблоны, — *шаблонов членов*, так что обсуждение из раздела 13.3.3 было бы неуместным. В конечном итоге Комитет отклонил предложение о замене угловых скобок.

Правило поиска имен для независимых имен и зависимых базовых классов, описанное в разделе 13.4.2, было введено в стандарт C++ в 1993 году. Для “широкой общественности” оно стало доступно в работе Бъярне Страуструпа [66] в начале 1994 года. Тем не менее первой общедоступной реализацией этого правила не было до начала 1997 года, когда НР включила ее в свой компилятор aC++, и к тому времени уже имелось большое количество кода производных шаблонов классов из зависимых базовых классов. Когда инженеры НР начали тестирование своей реализации, они обнаружили, что большинство программ, использующих шаблоны нетривиальными способами, больше не компилируются¹⁸. В частности, все реализации стандартной библиотеки шаблонов (STL) нарушали правило во многих сотнях, а иногда и тысячах мест¹⁹. Чтобы облегчить своим клиентам процесс перехода, в НР смягчили диагностику, связанную с кодом, который полагал, что независимые имена могут быть найдены в зависимых базовых классах следующим образом: когда независимое имя, используемое в области видимости шаблона класса, не найдено с помощью стандартных правил, а C++ заглядывал внутрь зависимых базовых классов. Если имя все равно не было обнаружено, выводилась ошибка, и компиляция завершалась неудачно. Однако, если имя было найдено в зависимом базовом классе, выдавалось предупреждение, а имя рассматривалось, как если бы оно было зависимым, так что во время инстанцирования попытка поиска повторялась.

Правило поиска, которое приводило к тому, что имена в независимых базовых классах скрывали идентично именованные параметры шаблонов (раздел 13.4.1), является недосмотром, но предложение изменить это правило не получило поддержки со стороны Комитета стандартизации C++. Лучше всего избегать кода с именами параметров шаблонов, которые используются в независимых базовых классах. Хорошие соглашения по именованию помогают решать эти проблемы.

Внесение имен друзей считалось вредным, потому что делало корректность программ более чувствительной к порядку инстанцирований. Билл Гиббонс (который в то время работал над компилятором Telligent) был одним из самых ярых сторонников решения этой проблемы, поскольку устранение зависимости от порядка инстанцирований обеспечивало поддержку новых и интересных сред разработки на C++ (над которыми, по слухам, работала Telligent). Однако трюк Бартона–Накмана (раздел 21.2.1) требует разновидности внесения

¹⁷ Фигурные скобки также не лишены проблем. В частности, они требуют существенной адаптации синтаксиса специализации шаблонов классов.

¹⁸ К счастью, они обнаружили это до выпуска новой функциональности.

¹⁹ Забавно, но первая реализация STL была также разработана НР.

дружественных имен, и именно эта конкретная методика, вызвавшая его, осталась в языке в ее нынешнем (ослабленном) виде, основанном на ADL.

Эндрю Кёниг (Andrew Koenig) вначале предложил ADL только для функций операторов (почему ADL иногда называют *поиском Кёнига*). Мотивация была главным образом эстетической: явно квалифицированные имена операторов с охватывающими их пространствами имен выглядят в лучшем случае неудобно (например, вместо `a+b` нам может потребоваться написать `N::operator+(a, b)`), а необходимость писать объявления `using` для каждого оператора может привести к громоздкому коду. Поэтому было решено, что операторы ищутся в пространствах имен, связанных с их аргументами. Позднее ADL был распространен на имена обычных функций для разрешения ограниченных видов инъекций имен друзей и для поддержки модели двухфазного поиска для шаблонов и их инстанцирований (глава 14, “Инстанцирование”). Обобщенные правила ADL также называют *расширенным поиском Кёнига*.

Спецификация для хака с угловыми скобками была добавлена в C++11 Дэвидом Вандевурдом согласно его статье N1757. Благодаря ему в стандарт был также добавлен хак с диграфом.

Глава 14

Инстанцирование

Инстанцирование (*instantiation*) шаблонов — это процесс, при котором на основе обобщенного определения шаблонов генерируются типы и функции¹. В C++ концепция инстанцирования шаблонов играет фундаментальную роль, однако она несколько запутана. Одна из основных причин этой запутанности состоит в том, что определения генерируемых шаблоном элементов не сосредоточены в одном месте исходного кода. Местонахождение определения шаблона, его использования и определения аргументов — все это играет роль.

В данной главе объясняется, как организовать исходный код для надлежащего использования шаблонов. Кроме того, здесь представлены различные методы, которые используются в наиболее популярных компиляторах C++ для инстанцирования шаблонов. Хотя все эти методы должны быть семантически эквивалентны, полезно понимать основные принципы, лежащие в основе стратегии инстанцирования, которой придерживается ваш компилятор. С одной стороны, каждый механизм инстанцирования обладает своим набором мелких особенностей, с другой — он подвергается влиянию конечных спецификаций языка C++.

14.1. Инстанцирование по требованию

Когда компилятор C++ встречает использование специализации шаблона, он создает ее, подставляя вместо параметров шаблона необходимые аргументы². Эти действия выполняются автоматически и не требуют каких бы то ни было указаний от пользовательского кода (или от определения шаблона). Это инстанцирование по требованию ставит шаблоны C++ особняком по отношению к ряду подобных функциональных возможностей в других ранних компилируемых языках (наподобие Ada или Eiffel; некоторые из этих языков требуют явных директив инстанцирования, в то время как другие используют механизмы диспетчеризации времени выполнения для полного устранения процесса инстанцирования). Это инстанцирование шаблона по требованию называют *неявным* (*implicit*) или *автоматическим* (*automatic*) инстанцированием. Шаблоны C++ стоят особняком по отношению к подобным возможностям других компилируемых языков программирования.

При инстанцировании по требованию компилятор обычно нуждается в доступе к полному определению (другими словами, не только к объявлению) шаблона

¹ Иногда термин *инстанцирование* применяется также для обозначения процесса создания объектов типов. Однако в данной книге этот термин всегда будет относиться к *шаблонам*.

² Термин *специализация* (*specialization*) применяется в обобщенном смысле. Под ним подразумевается конкретный экземпляр шаблона (см. главу 10, “Основные термины в области шаблонов”). Этот термин не относится к механизму *явной специализации* (*explicit specialization*), описываемой в главе 16, “Специализация и перегрузка”.

и некоторых его членов в точке, где этот шаблон используется. Рассмотрим небольшой исходный текст.

```
template<typename T> class C; // #1 Только объявление
C<int>* p = 0; // #2 Определение C<int> не требуется

template<typename T>
class C
{
public:
    void f(); // #3 Объявление члена
}; // #4 Определение шаблона класса завершено

void g(C<int>& c) // #5 Использование только объявления шаблона класса
{
    c.f(); // #6 Использование определения шаблона класса
    // требует определения C::f() в данной
    // единице трансляции

template<typename T>
void C<T>::f() // Требуемое объявление для #6
{
}
```

В точке #1 исходного текста доступно только объявление шаблона, но не его определение (такое объявление иногда называют *предварительным объявлением* (forward declaration)). Как и для обычных классов, определение шаблона класса не обязано находиться в области видимости для объявления указателей или ссылок на данный тип (как это сделано в точке #2). Например, для указания типа, которому принадлежит параметр функции `g()`, не требуется полное определение шаблона `C`. Однако, как только компоненту понадобится информация о размере специализации шаблона, или при доступе к члену такой специализации, нужно, чтобы определение шаблона класса находилось полностью в области видимости. Этим объясняется, что в точке #6 исходного кода должно быть доступно определение шаблона класса; в противном случае компилятор не в состоянии проверить наличие и доступность членов (ни закрытых, ни защищенных). Кроме того, необходимо также определение функции-члена, поскольку вызов в точке #6 требует существования `C<int>::f()`.

Приведем еще одно выражение, требующее инстанцирования предыдущего шаблона класса, чтобы узнать размер конструкции `C<void>`:

```
C<void>* p = new C<void>;
```

В данном случае инстанцирование необходимо для того, чтобы компилятор мог определить размер объекта `C<void>`, который необходим выражению `new` для выяснения, какое количество памяти следует выделить. Возможно, вы заметили, что для данного конкретного шаблона тип аргумента `X`, который подставляется вместо параметра `T`, не влияет на размер шаблона, поскольку в любом случае класс `C<X>` будет пустым. Однако от компилятора не требуется избежать инстанцирования путем анализа определения шаблона (и все компиляторы на практике

выполняют инстанцирование). Кроме того, в данном примере инстанцирование необходимо для того, чтобы определить, доступен ли конструктор по умолчанию для класса `C<void>`, а также убедиться, что в этом классе не объявлены операторы-члены `new` или `delete`.

Необходимость доступа к члену шаблона класса не всегда явно видна в исходном коде. Например, для разрешения перегрузки в C++ требуется, чтобы типы классов, которым принадлежат параметры функции-кандидата, находились в области видимости.

```
template<typename T>
class C
{
public:
    C(int); // Конструктор, который может быть вызван с одним
};           // параметром, может использоваться для неявных
            // преобразований

void candidate(C<double>); // #1
void candidate(int) { }    // #2

int main()
{
    candidate(42);          // Могут быть вызваны обе функции,
}                           // объявления которых приведены выше
```

Вызов функции `candidate(42)` будет разрешен с помощью объявления #2. Однако объявление #1 также может быть инстанцировано, чтобы проверить, подходит ли оно для разрешения вызова (благодаря тому, что конструктор с одним аргументом способен неявно преобразовать аргумент 42 в г-значение типа `C<double>`). Заметим, что компилятор может (но не обязан) выполнить инстанцирование, даже если способен обойтись при разрешении вызова и без него (в приведенном примере именно так и происходит, поскольку предпочтеть неявное преобразование типов их точному совпадению невозможно). Заметим также, что инстанцирование `C<double>` может привести к ошибке (что, возможно, покажется удивительным).

14.2. Отложенное инстанцирование

Приведенные к настоящему моменту примеры иллюстрируют требования, которые существенно не отличаются от требований при использовании обычных, не шаблонных классов. Во многих случаях нужно, чтобы класс был *полным*, или *завершенным* (*complete*; см. раздел 10.3.1). В случае шаблона компилятор генерирует полное определение класса из определения шаблона класса.

В связи с этим возникает вопрос: какая часть шаблона инстанцируется? Можно было бы ответить так: ровно столько, сколько необходимо. Другими словами, при инстанцировании шаблонов компилятору следует быть максимально “ленивым”. Рассмотрим, что это означает.

14.2.1. Частичное и полное инстанцирование

Как мы видели, иногда компилятору не требуется подстановка полного определения шаблона класса или функции. Например:

```
template<typename T> T f(T p)
{
    return 2 * p;
}
decltype(f(2)) x = 2;
```

В этом примере тип, указанный как `decltype(f(2))`, не требует полного инстанцирования шаблона функции `f()`. Поэтому компилятору разрешена только подстановка объявления `f()`, но не ее “тела”. Иногда это называют частичным инстанцированием.

Аналогично, если экземпляр шаблона класса передается без необходимости иметь полный тип для этого экземпляра, компилятор не должен выполнять полное инстанцирование этого экземпляра шаблона класса. Рассмотрим следующий пример:

```
template<typename T> class Q
{
    using Type = typename T::Type;
};
Q<int>* p = 0; // OK: тело Q<int> не подставляется
```

Здесь полное инстанцирование `Q<int>` приводило бы к ошибке, поскольку `T::Type` не имеет смысла, когда `T` представляет собой `int`. Но, поскольку `Q<int>` в данном примере не обязан быть полным типом, полное инстанцирование не выполняется, и код оказывается корректным (хотя и подозрительным).

У шаблонов переменных также различаются “полное” и “частичное” инстанцирование. Это иллюстрирует следующий пример:

```
template<typename T> T v = T::default_value();
decltype(v<int>) s; // OK: инициализатор v<int> не инстанцирован
```

Полное инстанцирование `v<int>` могло бы вызвать ошибку, но оно не является необходимым, если нам нужен только тип экземпляра шаблона переменной.

Интересно, что шаблоны псевдонимов этим различием не обладают: не существует двух способов их подстановки.

В C++, когда речь идет об “инстанцировании шаблона” без уточнения, идет речь о полном или частичном инстанцировании, подразумевается первое. То есть инстанцирование по умолчанию является полным инстанцированием.

14.2.2. Инстанцированные компоненты

В процессе неявного (полного) инстанцирования шаблона класса инстанцируются все объявления его членов, но не соответствующие определения (т.е. члены инстанцируются частично). Из этого правила есть несколько исключений. Во-первых, если в шаблоне класса содержится безымянное объединение, члены

определения этого объединения также инстанцируются³. Другое исключение связано с виртуальными функциями-членами. При инстанцировании шаблона класса определения этих функций могут как инстанцироваться, так и нет. Во многих реализациях эти определения будут инстанцироваться в силу того, что внутренняя структура, обеспечивающая механизм виртуальных вызовов, требует, чтобы виртуальные функции существовали в виде объектов, доступных для связывания.

При инстанцировании шаблонов аргументы функции по умолчанию рассматриваются отдельно. В частности, они не инстанцируются, если только не вызывается именно та функция (или функция-член), в которой используются эти аргументы по умолчанию. Они не инстанцируются и в том случае, когда при вызове функции аргументы указываются явным образом, т.е. аргументы по умолчанию не используются.

Аналогично не инстанцируются спецификации исключений и инициализаторы членов по умолчанию, если они не требуются.

Приведем пример, иллюстрирующий все упомянутые случаи:

details/lazy1.hpp

```
template<typename T>
class Safe
{
};

template<int N>
class Danger
{
    int arr[N]; // OK, хотя некорректно при N<=0
};

template<typename T, int N>
class Tricky
{
public:
    void noBodyHere(Safe<T>=3); // OK до тех пор, пока использование
                                // значения по умолчанию не приведет
                                // к ошибке
    void inclass()
    {
        Danger<N> noBoomYet; // OK до тех пор, пока inclass()
                                // не используется с N<=0
        struct Nested
        {
            Danger<N> pfew; // OK до тех пор, пока Nested
                                // не используется с N<=0
        };
    }
};
```

³ Безымянные объединения всегда представляют собой особый случай в том плане, что их члены всегда можно рассматривать как члены класса, в котором эти объединения содержатся. Безымянное объединение — это, по сути, конструкция, с помощью которой сообщается, что некоторые члены класса совместно используют одно и то же место в памяти.

```

union                      // Благодаря безымянному объединению:
{
    Danger<N> anonymous; // OK до тех пор, пока Tricky не
    int align;           // инстанцируется с N<=0
};
void unsafe(T(*p)[N]);    // OK до тех пор, пока Tricky не
                          // инстанцируется с N<=0
void error()
{
    Danger <-1> boom; // Всегда ошибка (которую обнаруживают
}                         // не все компиляторы)
};

```

Соответствующий стандарту компилятор C++ будет рассматривать эти определения шаблонов, чтобы проверить их синтаксис и общие семантические ограничения. При проверке ограничений, включающих параметры шаблонов, компилятор предполагает, что “все обстоит наилучшим образом”. Например, параметр N в члене Danger::arr может быть равным нулю или отрицательным (что привело бы к ошибке), однако предполагается, что это не так⁴. Определения inclass(), struct Nested и безымянного объединения, таким образом, проблем не вызывают.

По той же причине объявление члена unsafe(T(*p)[N]) тоже не представляет проблемы до тех пор, пока вместо параметра шаблона N не подставляется конкретное значение.

Спецификация аргумента по умолчанию (=3) в объявлении члена noBodyHere() подозрительна, поскольку шаблон Safe<> не является инициализируемым целочисленным значением, однако предполагается, что аргумент по умолчанию для обобщенного определения Safe<T> не потребуется, либо что Safe<T> будет специализирован (см. главу 16, “Специализация и перегрузка”) таким образом, что будет допускать инициализацию целочисленным значением. Однако определение функции-члена error() является ошибкой, даже когда шаблон не инстанцирован, поскольку использование Danger<-1> требует завершенного определения класса Danger<-1>, а генерация этого класса приводит к попытке определить массив с отрицательным размером. Интересно, что в то время как стандарт ясно указывает, что этот код некорректен, он также позволяет компилятору не диагностировать ошибку, если экземпляр шаблона на самом деле не используется. Следовательно, поскольку Tricky<T,N>::error() не используется ни для каких конкретных T и N, компилятор не обязан выводить сообщение об ошибке в данном случае. Например, GCC и Visual C++ на момент написания данной книги эту ошибку не диагностировали.

Проанализируем теперь, что произойдет, если добавить следующее определение:

```
Tricky<int, -1> inst;
```

Это заставит компилятор выполнить (полное) инстанцирование Tricky<int, -1>, подставляя int вместо T и -1 вместо N в определении шаблона

⁴Некоторые компиляторы, такие как GCC, допускают массивы с нулевой длиной в качестве расширений, таким образом принимая данный код как корректный даже когда N равно 0.

`Tricky<>`. Необходимы не все определения членов, но конструктор по умолчанию и деструктор (оба в данном случае объявляемые неявно) определено вызываются, а следовательно, их определения должны быть так или иначе доступны (что так и есть в нашем примере, поскольку они генерируются неявно). Как пояснено выше, члены `Tricky<int, -1>` частично инстанцированы (то есть их *объявления заменены*): этот процесс потенциально может привести к ошибкам. Например, объявление `unsafe(T (*p)[N])` создает тип массива с отрицательным количеством элементов, что является ошибкой. Аналогично член `anonymous` теперь вызывает ошибку, потому что тип `Danger<-1>` не может быть завершен. Напротив, определения членов `inclass()` и `struct Nested` еще не инстанцируются, и, таким образом, никаких ошибок от их необходимости для полного типа `Danger<-1>` (который содержит определение недопустимого массива, как мы уже упоминали ранее), не возникает.

Как уже указано, при инстанцировании шаблона на практике должны также предоставляться определения виртуальных членов. В противном случае могут возникнуть ошибки компоновки. Например:

details/lazy2.cpp

```
template<typename T>
class VirtualClass
{
public:
    virtual ~VirtualClass() {}
    virtual T vmem(); // Вероятна ошибка при инстанцировании
                      // без определения
};
int main()
{
    VirtualClass<int> inst;
```

Наконец, замечание об `operator->`. Рассмотрим код:

```
template<typename T>
class C
{
public:
    T operator->();
```

Обычно `operator->` должен возвращать тип указателя или тип другого класса, к которому применим `operator->`. Это свидетельствует о том, что завершение `C<int>` вызовет ошибку, поскольку этот тип объявляет возвращаемый тип `operator->` как `int`. Поскольку определения такого рода основываются на некоторых определениях “естественных” шаблонов классов⁵, правила языка сделаны более гибкими. От пользовательского оператора `operator->`, требуется возврат типа, к которому применим другой (например, встроенный) `operator->`,

⁵ Примерами могут служить шаблоны *интеллектуальных указателей* (например, стандартного `std::unique_ptr<T>`).

только если он действительно выбран процедурой разрешения перегрузки. Это утверждение остается истинным и тогда, когда оно не относится к шаблонам (хотя в таком контексте от него меньше пользы). Таким образом, здесь объявление не вызывает ошибки, даже несмотря на то, что в качестве возвращаемого типа подставляется тип `int`.

14.3. Модель инстанцирования C++

Инстанцирование шаблона — это процесс, в результате которого из определенного шаблона путем подстановки его параметров генерируется обычный тип, функция или переменная. На первый взгляд может показаться, что здесь все довольно просто, однако на практике этот процесс обрастает множеством деталей.

14.3.1. Двухфазный поиск

В главе 13, “Имена в шаблонах”, мы видели, что зависимые имена нельзя разрешить при синтаксическом анализе шаблонов. Поэтому в месте инстанцирования шаблона его определение еще раз просматривается компилятором. Однако независимые имена можно обработать при первом просмотре шаблона, выявив при этом многие ошибки. В результате мы приходим к концепции *двухфазного поиска* (two-phase lookup⁶): первая фаза — синтаксический анализ шаблона, вторая — его инстанцирование.

- На первом этапе, во время *синтаксического анализа* шаблона, обрабатываются независимые имена; на этой стадии анализ шаблона проводится с помощью *правил обычного поиска* (ordinary lookup rules), а также правил поиска, зависящего от аргументов (ADL), если они применимы в данном конкретном случае. Неквалифицированные зависимые имена (которые являются зависимыми, как зависимости имен функций при вызове с зависимыми аргументами) тоже просматриваются с помощью правил обычного поиска. Однако результат этого поиска не рассматривается как завершенный до тех пор, пока в процессе инстанцирования шаблона не будет проведен дополнительный поиск.
- На втором этапе, выполняющемся при *инстанцировании* шаблона в *точке инстанцирования* (point of instantiation — POI), анализируются зависимые квалифицированные имена (в которых параметры шаблонов заменяются аргументами шаблонов для данного конкретного инстанцирования). Кроме того, выполняется дополнительный ADL для зависимых неквалифицированных имен, поиск которых на первом этапе выполнялся с помощью правил обычного поиска.

Для неквалифицированных зависимых имен первоначальный обычный поиск — пока еще неполный — используется для определения, является ли имя шаблоном. Рассмотрим следующий пример:

⁶Кроме того, применяются термины *двухэтапный* (two-stage lookup) или *двухфазный поиск имен* (two-phase name lookup).

```

namespace N
{
    template<typename> void g() {}
    enum E { e };
}

template<typename> void f() {}

template<typename T> void h(T P)
{
    f<int>(p); // #1
    g<int>(p); // #2 Ошибка
}

int main()
{
    h(N::e); // Вызов шаблона h с T = N::E
}

```

В строке #1, увидев имя *f*, за которым следует <, компилятор должен решить, является ли “<” угловой скобкой или знаком “меньше”. Это зависит от того, известно ли, что имя *f* является шаблоном или нет; в нашем случае обычный поиск находит объявление имени *f*, которое действительно является шаблоном, так что синтаксический анализ успешно выполняется, считая < угловой скобкой.

Строка #2, однако, приводит к ошибке, потому что с помощью обычного поиска шаблон *g* не обнаруживается. Таким образом, < рассматривается как знак “меньше”, который в этом коде является синтаксической ошибкой. Если бы мы могли обойти эту проблему, то в конечном итоге нашли бы шаблон *N::g* с использованием ADL при инстанцировании *h* для *T=N::E* (так как *N* — это пространство имен, связанное с *E*), но все это невозможно до тех пор, пока мы не проанализируем успешно обобщенное определение *h*.

14.3.2. Точки инстанцирования

Как уже было показано, в исходном коде, использующем шаблон, есть места, в которых компилятор C++ должен иметь доступ к объявлению или определению этого шаблона. *Точка инстанцирования* (point of instantiation – POI) создается в том случае, когда некоторая конструкция исходного кода ссылается на специализацию шаблона таким образом, что для создания этой специализации требуется инстанцирование определения соответствующего шаблона. Точка инстанцирования — это место кода, в которое можно вставить шаблон с подставленными аргументами. Например:

```

class MyInt
{
public:
    MyInt(int i);
};

MyInt operator - (MyInt const&);

bool operator > (MyInt const&, MyInt const&);

```

```

using Int = MyInt;

template<typename T>
void f(T i)
{
    if (i > 0)
    {
        g(-i);
    }
}
// #1
void g(Int)
{
    // #2
    f<Int>(42); // Точка вызова
    // #3
}
// #4

```

Когда компилятор C++ встречает вызов шаблона функции `f<Int>(42)`, он знает, что нужно инстанцировать шаблон `f`, подставив вместо параметра `T` тип `MyInt`. В результате создается точка инстанцирования. Точки #2 и #3 находятся совсем рядом с местом вызова, однако они не могут быть точками инстанцирования, потому что в языке C++ в этих точках нельзя вставить определение `::f<Int>(Int)`. Главное различие между точками #1 и #4 заключается в том, что в точке #4 функция `g(Int)` находится в области видимости, поэтому становится разрешимым вызов `g(-i)`. Если бы точка #1 была точкой инстанцирования, то этот вызов нельзя было бы разрешить, поскольку в этой точке функция `g(Int)` еще не видна. К счастью, в C++ определяется, что точка инстанцирования для ссылки на специализацию шаблона функции должна находиться сразу после ближайшего объявления или определения области видимости пространства имен, в котором содержится эта ссылка. В нашем примере это точка #4.

Возможно, вас удивит, что в этом примере вместо обычного типа `int` принимает участие тип `MyInt`. Дело в том, что на втором этапе поиска имен, который проводится в точке инстанцирования, используется только ADL. Поскольку с типом `int` не связано никакое пространство имен, то при его применении поиск в точке инстанцирования не проводился бы и функция `g` не была бы обнаружена. Следовательно, если мы заменим объявление псевдонима типа `Int` следующим:

```
using Int = int;
```

то код из предыдущего примера перестанет компилироваться. Приведенный далее пример страдает от подобной проблемы:

```

template<typename T>
void f1(T x)
{
    g1(x);      // #1
}
void g1(int)
{
}

```

```

int main()
{
    f1(7);      // Ошибка: g1 не найдена!
}
// #2 Точка инстанцирования f1<int>(int)

```

Вызов `f1(7)` создает точку инстанцирования для `f1<int>(int)` сразу за `main()` в точке #2. В этом инстанцировании ключевым вопросом является поиск функции `g1`. Когда определение шаблона `f1` встречается впервые, отмечается, что неквалифицированное имя `g1` является зависимым, поскольку это имя функции в вызове функции с зависимыми аргументами (тип аргумента `x` зависит от параметра шаблона `T`). Таким образом, `g1` ищется в точке #1 с помощью правил обычного поиска; однако `g1` в этот момент не является видимой. В точке #2, точке инстанцирования, функция ищется снова в связанных пространствах имен и классах, но тип аргумента является `int`, и он не имеет связанных пространств имен или классов. Таким образом, `g1` никогда не будет найдена, несмотря даже на то, что обычный поиск в точке инстанцирования нашел бы `g1`.

Точка инстанцирования для шаблонов переменных обрабатывается аналогично обработке шаблонов функций⁷.

Для специализаций шаблонов классов ситуация иная, как иллюстрирует следующий пример:

```

template<typename T>
class S
{
public:
    T m;
};

unsigned long h()          #1
{
    //                      #2
    return (unsigned long) sizeof(S<int>); #3
}
//                      #4

```

И вновь точки области видимости функции #2 и #3 не могут быть точками инстанцирования, потому что определение класса `S<int>` с областью видимости пространства имен не может в них находиться (шаблоны в общем случае не могут находиться в области видимости функций⁸). Если следовать правилу для экземпляров шаблона функции, точка инстанцирования будет в #4, но тогда недопустимо выражение `sizeof(S<int>)`, потому что размер `S<int>` не может быть определен, пока не будет достигнута точка #4. Таким образом, точка инстанцирования для ссылки на сгенерированный экземпляр класса определяется как

⁷ Удивительно, но это не было явно оговорено в стандарте на момент написания книги. Однако не ожидается, что это может стать спорным вопросом.

⁸ Тонким исключением из этого наблюдения является оператор вызова обобщенного лямбда-выражения.

точка непосредственно перед ближайшим объявлением или определением области видимости пространства имен, содержащей ссылку на этот экземпляр. В нашем примере это точка #1.

Когда шаблон в действительности инстанцируется, может появиться необходимость в дополнительных инстанцированиях. Рассмотрим небольшой пример:

```
template<typename T>
class S
{
public:
    using I = int;
};

// #1
template<typename T>
void f()
{
    S<char>::I var1 = 41;
    typename S<T>::I var2 = 42;
}

int main()
{
    f<double>(); // #2: #2a, #2б
}
```

Нашим предыдущим рассмотрением уже установлено, что точка инстанцирования для `f<double>()` находится в точке #2. Шаблон функции `f()` также обращается к специализации класса `S<char>` с точкой инстанцирования, которая, таким образом, находится в точке #1. Он обращается также к `S<T>`, но, поскольку он до сих пор остается зависимым, мы в действительности не можем инстанцировать его в этой точке. Однако если мы инстанцируем `f<double>()` в точке #2, то заметим, что нам также необходимо инстанцировать определение `S<double>`. Такие вторичные, или переходные, точки инстанцирования определены несколько иначе. Для шаблонов функций вторичные точки инстанцирования в точности те же, что и первичные точки инстанцирования. Для классов вторичная точка инстанцирования непосредственно предшествует (в ближайшей охватывающей области видимости пространства имен) первичной точке инстанцирования. В нашем примере это означает, что точка инстанцирования `f<double>()` может быть помещена в точке #2б, а прямо перед ней — в точке #2а — находится вторичная точка инстанцирования для `S<double>`. Обратите внимание на отличие описанной ситуации от точки инстанцирования для `S<char>`.

Единица трансляции часто содержит несколько точек инстанцирования для одного и того же экземпляра. Для экземпляров шаблона класса в каждой единице трансляции сохраняется только первая точка инстанцирования, а последующие игнорируются (они не рассматриваются в действительности как точки инстанцирования). Для экземпляров шаблонов функций и переменных все точки инстанцирования сохраняются. В любом случае правило одного определения требует, чтобы инстанцирования, происходящие в любой из сохраненных точек

инстанцирования, были эквивалентны, но компилятор C++ не обязан проверять и диагностировать нарушения этого правила. Это позволяет компилятору C++ выбрать только одну точку инстанцирования шаблона, не являющегося шаблоном класса, для выполнения фактического инстанцирования без беспокойства о том, что другая точка инстанцирования может привести к другому инстанцированию, отличному от данного.

На практике большинство компиляторов откладывают фактическое инстанцирование большинства шаблонов функций до конца единицы трансляции. Некоторые инстанцирования не могут быть отложены, включая случаи, где инстанцирование необходимо для определения выводимого возвращаемого типа (см. разделы 15.10.1 и 15.10.4), и случаи, когда функция является `constexpr`-функцией и должна вычисляться для получения константного результата. Некоторые компиляторы инстанцируют встраиваемые функции при первом использовании для обеспечения правильного встраивания⁹. Это фактически перемещает точки инстанцирования соответствующих специализаций шаблонов в конец единицы трансляции, который в стандарте C++ разрешается использовать как альтернативную точку инстанцирования.

14.3.3. Модель включения

Где бы ни находилась точка инстанцирования, в этом месте каким-то образом должен быть обеспечен доступ к соответствующему шаблону. Для специализации класса это означает, что определение шаблона класса должно быть видимым в точке, которая находится раньше в данной единице трансляции. Для точек инстанцирования шаблонов функций и переменных (а также функций-членов и статических членов-данных шаблонов классов) это тоже необходимо. Обычно определения шаблонов просто добавляются в заголовочные файлы, которые с помощью директивы `#include` включаются в единицу трансляции, даже когда они представляют собой шаблоны, не являющиеся типами. Такая модель, применяемая к определениям шаблонов, называется *моделью включения* (*inclusion model*) и представляет собой единственную автоматическую модель исходных текстов для шаблонов, поддерживаемую текущим стандартом C++¹⁰.

Хотя модель включения поощряет программистов размещать все определения шаблонов в заголовочных файлах, чтобы они были доступны и удовлетворяли любым точкам инстанцирования, которые могут возникнуть, можноенным образом управлять инстанцированиями с использованием *объявлений явного инстанцирования* (*explicit instantiation declarations*) и *определений явного*

⁹ В современных компиляторах встраивание вызовов обычно обрабатывается главным образом независимым от языка программирования компонентом компилятора, относящимся к оптимизации. Однако в более ранних компиляторах C++ таким встраиванием занимались компоненты, связанные с языком C++, потому что в те времена оптимизирующие компоненты были для этого слишком консервативными.

¹⁰ Стандарт C++98 поддерживал также *модель разделения* (*separation model*). Она никогда не пользовалась популярностью и была удалена из стандарта непосредственно перед публикацией стандарта C++11.

инстанцирования (*explicit instantiation definitions*) (см. раздел 14.5). Это технически не тривиально, так что большую часть времени программисты предпочитают полагаться на механизм автоматического инстанцирования. Одной из задач реализации с автоматической схемой является возможность иметь точки инстанцирования для одной и той же специализации шаблона функции или переменной (или одной и той же функции-члена или статических членов-данных экземпляра шаблона класса) в нескольких единицах трансляции. Далее мы обсудим подходы к этой проблеме.

14.4. Схемы реализации

В этом разделе рассматриваются некоторые способы, с помощью которых различные реализации C++ поддерживают модель включения. Все эти реализации основываются на двух классических компонентах: на *компиляторе* и *компоновщике*. Компилятор преобразует исходный код в объектные файлы, которые содержат машинный код и символические аннотации (перекрестные ссылки на другие объектные файлы и библиотеки). Компоновщик создает исполняемые программы или библиотеки, соединяя объектные файлы в одно целое и разрешая содержащиеся в них перекрестные ссылки. Все, о чем пойдет речь далее, относится именно к такой модели, хотя вполне возможны другие способы реализации языка C++ (которые не приобрели широкой популярности). Например, вполне можно представить себе интерпретатор C++.

Если специализация шаблона класса используется в нескольких единицах трансляции, компилятору придется повторить процесс инстанцирования в каждой из этих единиц. Количество возникающих в связи с этим проблем весьма незначительно, поскольку определения классов не генерируют непосредственно код низкого уровня. Эти определения используются только внутри реализации C++ для проверки и интерпретации различных других выражений и объявлений. В этом отношении множественное инстанцирование определения класса, по сути, не отличается от многократного включения определения класса (обычно с помощью включения заголовочного файла) в разные единицы трансляции.

Однако если происходит инстанцирование шаблона (невстраиваемой) функции, ситуация может измениться. Если использовать несколько определений обычных невстроенных функций, то это нарушит правило одного определения. Например, предположим, что компилируется и компонуется программа, состоящая из двух приведенных ниже файлов:

```
// === a.cpp:
int main()
{
}

// === b.cpp:
int main()
{
}
```

Компиляторы C++ будут компилировать каждый модуль отдельно, причем без каких-либо проблем, потому что эти единицы трансляции, безусловно, являются корректными с точки зрения C++. Однако попытка связать эти два файла, скорее всего, вызовет протест компоновщика. Дело в том, что дублирование определений не допускается.

Рассмотрим теперь другой пример, в котором участвуют шаблоны.

```
// === t.hpp:
// Общий заголовочный файл (модель включения)
template<typename T>
class S
{
public:
    void f();
};

template<typename T>
void S::f() // Определение члена
{
}

void helper(S<int>*);

// === a.cpp:
#include "t.hpp"
void helper(S<int>* s)
{
    s->f(); // #1 Первая точка инстанцирования S::f
}

// === b.cpp:
#include "t.hpp"
int main()
{
    S<int> s;
    helper(&s);
    s.f(); // #2 Вторая точка инстанцирования S::f
}
```

Если компоновщик рассматривает инстанцированные функции-члены шаблонов классов точно так же, как обычные функции или функции-члены, то компилятор должен гарантировать, что он генерирует код только в одной из двух точек инстанцирования: в точке #1 или #2, но не в обеих. Чтобы достичь этого, компилятор должен перенести информацию из одной единицы трансляции в другую, а это никогда не требовалось от компиляторов C++ до введения шаблонов в этот язык программирования. Далее рассматриваются три популярных класса решений, получивших широкое распространение среди разработчиков реализаций языка C++.

Заметим, что такая же проблема возникает во всех связываемых объектах, возникающих в результате инстанцирования шаблонов: как в инстанцированных шаблонах обычных функций и функций-членов, так и в инстанцированных статических данных-членах и инстанцированных шаблонах переменных.

14.4.1. Жадное инстанцирование

Первые компиляторы C++, которые сделали популярным так называемое жадное инстанцирование, были произведены компанией Borland. С тех пор этот подход стал одним из самых распространенных методов среди различных систем C++.

В процессе жадного инстанцирования предполагается, что компоновщик осведомлен о том, что некоторые объекты (в частности, подлежащие компоновке инстанцированные шаблоны) могут дублироваться в разных объектных файлах и библиотеках. Обычно компилятор помечает такие элементы особым образом. Когда компоновщик обнаруживает множественные инстанцирования, одно из них он оставляет, а остальные отбрасывает. В этом и заключается суть рассматриваемого подхода.

Теоретически жадное инстанцирование обладает некоторыми серьезными недостатками, перечисленными ниже.

- Компилятор может потратить время на генерацию и оптимизацию множества инстанцирований, из которых будет использоваться только одно.
- Обычно компоновщики не проверяют идентичность двух инстанцирований, так как код, сгенерированный для разных экземпляров одной и той же специализации шаблона, может незначительно варьироваться, что вполне допустимо. Нельзя допустить, чтобы из-за этих небольших различий в работе компоновщика произошел сбой. (Причиной этих различий могут быть несущественные расхождения в состоянии компилятора в моменты инстанцирования.) Однако часто это приводит к тому, что компоновщик не замечает более существенных различий, например когда одно из инстанцирований скомпилировано со строгими математическими правилами работы с числами с плавающей точкой, а другое — с ослабленными высокопроизводительными правилами¹¹.
- Потенциально объем всех объектных файлов может существенно превысить их объем при использовании иного подхода, поскольку один и тот же фрагмент кода дублируется несколько раз.

На практике оказывается, что эти недостатки не создают особых проблем. Возможно, так получается благодаря тому, что жадное инстанцирование очень выгодно отличается от альтернативных подходов в одном важном аспекте: оно сохраняет традиционную зависимость от исходного кода. В частности, из одной единицы трансляции генерируется только один объектный файл, и каждый объектный файл содержит скомпилированный код всех подлежащих компоновке определений из соответствующего исходного файла (включая инстанцированные определения). Другим преимуществом является то, что все экземпляры шаблона функции являются кандидатами для встраивания без применения дорогих механизмов оптимизации времени компоновки (на практике экземпляры шаблона функции часто представляют собой малые функции, встраивание которых приносит выгоду).

¹¹ Однако современные системы в состоянии обнаруживать некоторые другие отличия. Например, они могут сообщить, что одно инстанцирование имеет связанную с ним отладочную информацию, а другое — нет.

Другие механизмы инстанцирования обрабатывают *встраиваемые* экземпляры шаблонов функций особым образом, чтобы гарантировать, что они могут быть преобразованы во встроенный код. Однако жадное инстанцирование позволяет преобразовывать во встраиваемые даже функции, не объявленные таковыми.

Наконец, стоит заметить, что механизм компоновки, позволяющий дублировать определения компонуемых элементов, обычно используется для обработки *дублируемых встраиваемых функций*¹² и *таблиц диспетчеризации виртуальных функций*¹³. Если этот механизм недоступен, в качестве альтернативы эти элементы обычно генерируются с внутренним связыванием, но это приводит к увеличению объема генерируемого кода. Требование, чтобы встраиваемая функция имела единственный адрес, затрудняет реализацию альтернативных подходов соответствующим стандарту способом.

14.4.2. Инстанцирование по запросу

В середине 1990-х годов компания *Sun Microsystems*¹⁴ выпустила реализацию своего компилятора C++ версии 4.0 с новым и интересным решением проблемы инстанцирования, которое мы называем *инстанцированием по запросу* (queried instantiation). Концептуально инстанцирование по запросу отличается удивительной простотой и элегантностью, являясь при этом наиболее современной схемой инстанцирования классов из всех рассматриваемых здесь. В этой схеме создается и поддерживается специальная база данных, совместно используемая при компиляции всех единиц трансляции, имеющих отношение к программе. В нее заносятся сведения об инстанцированных специализациях шаблонов, а также о том, от какого элемента исходного кода они зависят. Сами генерированные специализации также обычно сохраняются в этой базе данных. При достижении точки инстанцирования подлежащего компоновке элемента происходит одно из трех перечисленных ниже событий.

1. Соответствующая специализация отсутствует. В этом случае происходит инстанцирование, а полученная в результате специализация заносится в базу данных.
2. Специализация имеется в наличии, однако она устарела, поскольку с момента ее создания произошли изменения в исходном коде. В этой ситуации также происходит инстанцирование, а полученная в результате специализация заносится в базу данных вместо предыдущей.
3. В базе данных содержится подходящая не устаревшая специализация. Делать ничего не нужно.

¹² Если компилятор не в состоянии “встраивать” все вызовы какой-то функции, обозначенной ключевым словом `inline`, в состав объектного файла вводится отдельная копия этой функции. Это может быть сделано в нескольких объектных файлах.

¹³ Обычно вызовы виртуальных функций реализуются как косвенные, причем это осуществляется с помощью таблиц указателей на функции. Фундаментальное исследование подобных аспектов реализации C++ можно найти в [50].

¹⁴ Позже приобретенная компанией Oracle.

Несмотря на концептуальную простоту описанной схемы, ее реализация связана с необходимостью решения некоторых практических задач.

- Далеко не просто поддерживать правильную взаимосвязь между структурными элементами базы данных, поскольку состояние исходного кода может меняться. Несмотря на то что не будет ошибкой принять третий случай за второй, это увеличит количество работы, которую необходимо выполнить компилятору (а значит, и время компиляции).
- Распространена ситуация, когда компиляторы выполняют параллельную компиляцию нескольких единиц трансляции, что также усложняет поддержку базы данных.

Несмотря на указанные трудности, схема может быть довольно эффективно реализована. Кроме того, в отличие, например, от жадного инстанцирования, которое может привести к большому количеству напрасно затраченной работы, при описанном решении практически отсутствуют патологические случаи, которые могли бы привести к излишней работе компилятора.

К сожалению, использование базы данных может также создать некоторые проблемы программисту. Причина большинства этих проблем заключается в том, что традиционная модель трансляции, унаследованная от большинства компиляторов C, больше не применима: в результате компиляции одной единицы трансляции теперь не создается отдельный объектный файл. Предположим, например, что нужно скомпоновать конечную программу. Для этого понадобится не только содержимое каждого объектного файла, связанного с различными единицами трансляции, но и тех объектных файлов, которые хранятся в базе данных. Аналогично в процессе создания бинарной библиотеки нужно убедиться в том, что инструмент, с помощью которого эта библиотека создается (обычно это компоновщик или архиватор), располагает сведениями из базы данных. По сути, любой инструмент, оперирующий с объектными файлами, должен иметь информацию о содержимом базы данных. Многие из этих проблем можно смягчить, не занося инстанцирование в базу данных, а размещая вместо этого их объектный код в объектный файл, вызвавший данное инстанцирование.

С библиотеками связана другая проблема. В одну и ту же библиотеку может быть упаковано несколько сгенерированных специализаций. При добавлении этой библиотеки в другой проект может понадобиться занести в базу данных нового проекта сведения об уже доступных инстанцированиях. Если этого не сделать и если в проекте создаются свои точки инстанцирования для специализаций, которые содержатся в библиотеке, инстанцирования могут дублироваться. Стратегия, которой следует придерживаться в такой ситуации, может состоять в применении той же технологии компоновки, что и при жадном инстанцировании: передать компоновщику сведения об имеющихся инстанцированиях, а затем избавиться от дубликатов (которые, однако, встречаются намного реже, чем в случае жадного инстанцирования). Другие варианты размещений исходных файлов, объектных файлов и библиотек могут вызвать новые проблемы, в частности отсутствие

инстанцирований из-за того, что объектный код, содержащий нужное инстанцирование, не скомпонован с конечной исполняемой программой.

В конечном счете инстанцирование по запросу на рынке не выжило, и даже компилятор Sun теперь использует жадное инстанцирование.

14.4.3. Итеративное инстанцирование

Первым компилятором, поддерживающим шаблоны C++, был Cfront 3.0 – прямой потомок компилятора, разработанного Бъярне Страуструпом в процессе создания языка программирования C++¹⁵. Одна из особенностей компилятора Cfront, ограничивающая его гибкость, заключалась в том, что он должен был обладать переносимостью на другие платформы. Это означает, что, во-первых, в качестве представления на всех целевых платформах используется язык C и, во-вторых, применяется локальный целевой компоновщик. В частности, при этом подразумевается, что компоновщик не способен обрабатывать шаблоны. Фактически компилятор Cfront генерировал инстанцирования шаблонов как обычные функции C, поэтому он должен был избегать повторных инстанцирований. Хотя исходная модель, на которой основан компилятор Cfront, отличалась от стандартной модели включения, можно добиться того, что используемая в этом компиляторе стратегия инстанцирования будет соответствовать модели включения. Таким образом, это первый компилятор, в котором было воплощено итеративное инстанцирование. Итерации компилятора Cfront описаны ниже.

1. Исходный код компилируется без инстанцирования каких бы то ни было связываемых специализаций.
2. Объектные файлы связываются с помощью *предварительного компоновщика* (prelinker).
3. Предварительный компоновщик вызывает компоновщик и анализирует сгенерированные сообщения об ошибках, чтобы определить, не вызваны ли они отсутствием инстанцирований; если причина ошибки именно в этом, предварительный компоновщик вызывает компилятор для обработки исходных файлов, содержащих необходимые определения шаблонов; при этом параметры компилятора настроены для генерирования отсутствующих инстанцирований.
4. Если сгенерированы какие-либо определения, повторяется пункт 3.

Повторение пункта 3 обусловлено тем, что на практике инстанцирование одного из подлежащих компоновке элементов может привести к необходимости инстанцировать шаблон в другом элементе, который еще не был обработан. Такой итеративный процесс в конечном счете сходится, и компоновщику удается создать завершенную программу.

¹⁵ Не поймите эту фразу превратно, прия к заключению, что компилятор Cfront был академическим прототипом. Напротив, он использовался в промышленных целях и послужил основой для многих коммерческих компиляторов C++. Версия 3.0 появилась в 1991 году, но страдала наличием ошибок. Вскоре за этим последовала версия 3.0.1, благодаря которой стало возможным применение шаблонов.

Схема, положенная в основу исходного компилятора Cfront, обладает весьма серьезными недостатками.

- Время, затрачиваемое на создание исполняемого файла, увеличивается не только из-за работы предварительного компоновщика, но и за счет повторных компиляций и компоновок. Согласно отчетам некоторых пользователей систем, в основе которых находится компилятор Cfront, время создания исполняемых файлов возросло до “нескольких дней” по сравнению с тем, что раньше в альтернативных схемах это занимало “около часа”.
- Выдача сообщений об ошибках и предупреждений откладывается до этапа компоновки. Это особенно неприятно, когда компоновка занимает много времени и разработчику часами приходится ждать, пока будет обнаружена всего лишь опечатка в определении шаблона.
- Необходимо особо позаботиться о том, чтобы запомнить, где находится исходный код, содержащий то или иное определение (п. 1). В частности, компилятор Cfront использовал специальное хранилище, с помощью которого решались некоторые проблемы, что, по сути, напоминает использование базы данных при инстанцировании по запросу. Исходный компилятор Cfront не был приспособлен для поддержки параллельной компиляции.

Несмотря на перечисленные недостатки, принцип итерации в улучшенном виде был использован в двух системах компиляции, устранивших недостатки исходной реализации Cfront. Одна из них создана группой Edison Design Group (EDG), а вторая известна под названием HP aC++¹⁶. На практике эти реализации работают довольно хорошо, и, хотя построение “с нуля” обычно занимает больше времени, чем альтернативные схемы, времена последующих построений вполне конкурентоспособны. Тем не менее итеративное инстанцирование используется только в относительно немногих компиляторах C++.

14.5. Явное инстанцирование

Точку инстанцирования для специализации шаблона можно создать явным образом. Конструкция, с помощью которой это достигается, называется *директивой явного инстанцирования* (explicit instantiation directive). Синтаксически она состоит из ключевого слова `template`, за которым следует объявление инстанцируемой специализации. Например:

```
template<typename T>
void f(T)
{
}
```

¹⁶ Компилятор HP aC++ возник на основе технологии, разработанной в компании Taligent (позже она была поглощена компанией IBM). Компания HP добавила в компилятор aC++ принцип жадного инстанцирования и сделала его механизмом, применяющимся по умолчанию.

```
// Четыре явных инстанцирования:
template void f<int>(int);
template void f<>(float);
template void f(long);
template void f(char);
```

Обратите внимание на то, что корректны все приведенные выше директивы инстанцирования. Аргументы шаблона могут быть выведены (см. главу 15, “Вывод аргументов шаблона”).

Члены шаблонов классов также можно явно инстанцировать:

```
template<typename T>
class S
{
public:
    void f()
{
}
};

template void S<int>::f();
template class S<void>;
```

Кроме того, можно явно инстанцировать все члены, входящие в состав специализации шаблона класса, путем явного инстанцирования специализации шаблона этого класса. Поскольку эти директивы явного инстанцирования гарантируют, что создается определение специализации именованного шаблона (или его членов), директивы явного инстанцирования более точно называть *определениями явного инстанцирования* (explicit instantiation definitions). Специализация явно инстанцируемого шаблона не должна быть явно специализированной и наоборот, потому что это будет означать, что два определения могут быть различными (нарушая правило одного определения).

14.5.1. Ручное инстанцирование

Многие программисты на C++ замечали, что автоматическое инстанцирование шаблона отрицательно сказывается на времени построения. Это особенно верно в случае компиляторов, реализующих жадное инстанцирование (раздел 14.4.1), потому что одни и те же специализации шаблонов могут инстанцироваться и оптимизироваться во многих разных единицах трансляции.

Методика, применяемая для улучшения времени построения, состоит в ручном инстанцировании тех специализаций шаблонов, которые требуются программе, в единственном месте, и препятствовании инстанцированиям во всех других единицах трансляции. Один из переносимых способов обеспечить это препятствование – не предоставять определение шаблона, за исключением единицы трансляции, где он инстанцируется явно¹⁷. Например:

¹⁷ В стандартах C++ 1998 и 2003 годов это был *единственный* переносимый способ предотвратить инстанцирование в других единицах трансляции.

```

// === Единица трансляции 1:
template<typename T> void f(); // Нет определения: предотвращает
                                // инстанцирование в этой
                                // единице трансляции
void g()
{
    f<int>();
}

// === Единица трансляции 2:
template<typename T> void f()
{
    // Реализация
}

template void f<int>();           // Ручное инстанцирование

void g();

int main()
{
    g();
}

```

В первой единице трансляции компилятор не может видеть определение шаблона функции `f`, поэтому он не может и не будет выполнять инстанцирование `f<int>`. Вторая единица трансляции предоставляет определение `f<int>` через явное определение инстанцирования; без него программа не может быть скомпонована.

Ручное инстанцирование имеет явный недостаток: мы должны тщательно отслеживать инстанцируемые сущности. Для крупных проектов это быстро становится чрезмерным бременем, поэтому мы не рекомендуем применять этот способ. Мы работали над несколькими проектами, которые первоначально недооценили всю тяжесть такого подхода, и позже пришли к сожалению о принятом решении.

Однако ручное инстанцирование имеет также и несколько преимуществ, поскольку инстанцирование может быть настроено для потребностей конкретной программы. Очевидно, что при этом удается избежать накладных расходов, связанных с большими заголовками, так как они связаны с многократным инстанцированием одних и тех же шаблонов с одними и теми же аргументами в нескольких единицах трансляции. Кроме того, исходный код определения шаблона может поддерживаться скрытым, но тогда клиентской программой не могут быть созданы никакие дополнительные инстанцирования.

Часть бремени ручного инстанцирования может быть облегчена путем размещения определений шаблонов в третьем исходном файле, обычно с расширением `.tpp`. Для нашей функции `f` шаблон распадается на следующие части:

```

// === f.hpp:
template<typename T> void f(); // Определения нет:
                                // препятствует инстанцированию

// === f.tpp:
#include "f.hpp"

```

```
template<typename T> void f() // Определение
{
    // Реализация
}

// === f.cpp:
#include "f.tpp"

template void f<int>();           // Ручное инстанцирование
```

Такая структура обеспечивает определенную гибкость. Можно включить только файл `f.hpp`, чтобы получить объявление `f` без автоматического инстанцирования. Явное инстанцирование может быть при необходимости добавлено в `f.cpp` вручную. Или, если ручное инстанцирование становится слишком обременительными, можно включать `f.tpp` для того, чтобы разрешить автоматическое инстанцирование.

14.5.2. Объявления явного инстанцирования

Более целенаправленным подходом к ликвидации избыточных автоматических инстанцирований является использование *объявления явного инстанцирования* (*explicit instantiation declaration*), представляющего собой директиву явного инстанцирования, которой предшествует ключевое слово `extern`. Объявление явного инстанцирования *в общем случае* подавляет автоматическое инстанцирование специализации именованного шаблона, поскольку объявляет, что специализация именованного шаблона будет определена где-то в программе (с помощью определения явного инстанцирования). Мы говорим *в общем случае*, потому что имеется множество исключений.

- Встраиваемые функции все равно могут быть инстанцированы для того, чтобы их можно было встроить в код (но отдельный объектный код не генерируется).
- Переменные с выведенными типами `auto` или `decltype(auto)` и функции с выведенными возвращаемыми типами могут быть инстанцированы для того, чтобы определить их типы.
- Переменные, значения которых используются как константные выражения, могут быть инстанцированы для вычисления их значений.
- Переменные ссылочных типов могут быть инстанцированы для того, чтобы сущности, на которые они ссылаются, могли быть разрешены.
- Шаблоны классов и шаблоны псевдонимов могут быть инстанцированы для того, чтобы проверить получающиеся в результате типы.

Используя объявления явного инстанцирования, мы можем предоставить определение шаблона для `f` в заголовке (`t.hpp`), а затем подавить автоматическое инстанцирование для часто используемых специализаций следующим образом:

```
// === t.hpp:
template<typename T> void f()
{
}

extern template void f<int>(); // Объявлен, но не определен
extern template void f<float>(); // Объявлен, но не определен

// === t.cpp:
template void f<int>(); // Определение
template void f<float>(); // Определение
```

Каждое объявление явного инстанцирования должно быть в паре с определением соответствующего явного инстанцирования, которое должно следовать за объявлением явного инстанцирования. Пропуск определения приведет к ошибке компоновщика.

Объявление явного инстанцирования может использоваться для ускорения компиляции или компоновки, когда во многих разных единицах трансляции часто используются определенные специализации. В отличие от ручного инстанцирования, при котором требуется вручную обновлять список определений явного инстанцирования всякий раз, когда требуется новая специализация, объявления явных инстанцирований могут быть введены в качестве оптимизации в любой точке. Однако преимущества во время компиляции могут оказаться не столь существенными, как при ручном инстанцировании, как потому что некоторые избыточные автоматические инстанцирования, скорее всего, будут встречаться¹⁸, так и потому, что определения шаблонов по-прежнему анализируются как часть заголовочного файла.

14.6. Инструкции `if` времени компиляции

Как было рассказано в разделе 8.5, в C++ 17 добавлен новый вид инструкций, который оказался чрезвычайно полезным при написании шаблонов: `if` времени компиляции. В нем также вводятся новые тонкости в процесс инстанцирования.

Приведенный далее пример иллюстрирует работу `if` времени компиляции:

```
template<typename T> bool f(T p)
{
    if constexpr(sizeof(T) <= sizeof(long long))
    {
        return p > 0;
    }
    else
    {
        return p.compare(0) > 0;
    }
}
```

¹⁸ Интересная часть этой проблемы оптимизации — точное определение, какие специализации являются хорошими кандидатами для объявлений явного инстанцирования. Низкоуровневые утилиты, такие как `pm` из `Upli`, могут пригодиться при определении, какие автоматические инстанцирования были в действительности выполнены в объектных файлах, составляющих программу.

```
bool g(int n)
{
    return f(n); // OK
}
```

`if` времени компиляции представляет собой инструкцию `if`, в которой за ключевым словом `if` непосредственно следует (как в приведенном примере) ключевое слово `constexpr`¹⁹. Следующее далее условие в скобках должно иметь константное логическое значение (разрешены неявные преобразования в `bool`). Таким образом, компилятору известно, какая из ветвей будет выбрана, а какая — отброшена. Особенно интересно, что во время инстанцирования шаблонов (включая обобщенные лямбда-выражения) отбрасываемые ветви *не* инстанцируются. Это необходимо, чтобы наш пример был корректным: мы выполняем инстанцирование `f(T)` с `T=int`, что означает, что ветвь `else` отбрасывается. Если бы она не отбрасывалась, то она была бы инстанцирована, и мы бы получили ошибку в выражении `p.compare(0)` (которое является ошибочным, когда `p` представляет собой простое целое число).

До C++17 и его инструкции *константного if*, чтобы избежать данной ошибки при получении того же результата, требовалась явная специализация шаблона или перегрузка (см. главу 16, “Специализация и перегрузка”).

Представленный выше пример может быть реализован на C++14 следующим образом:

```
template<bool b> struct Dispatch // Инстанцируется при b, равном
{
    static bool f(T p)           // false, из-за наличия
    {
        return p.compare(0) > 0;
    }
};

template<> struct Dispatch<true>
{
    static bool f(T p)
    {
        return p > 0;
    }
};

template<typename T> bool f(T p)
{
    return Dispatch<sizeof(T) <= sizeof(long long)>::f(p);
}

bool g(int n)
{
    return f(n);                 // OK
}
```

¹⁹Хотя эта инструкция записывается как `if constexpr`, ее часто называют “*константным if*” (`constexpr if`).

Очевидно, что *константный if* является альтернативным выражением нашего намерения, причем выражением гораздо более четким и лаконичным. Однако он требует от реализаций языка уточнения единицы инстанцирования: в то время как ранее определения функций всегда инстанцировались как единое целое, теперь должна быть обеспечена возможность подавления инстанцирования их частей.

Еще одним удобным применением *константного if* является выражение рекурсии, необходимой для обработки пакетов параметров функции. Вспомним пример из раздела 8.5.

```
template<typename Head, typename... Remainder>
void f(Head&& h, Remainder&&... r)
{
    doSomething(std::forward<Head>(h));
    if constexpr (sizeof... (r) != 0)
    {
        // Рекурсивная обработка остатка
        // (прямая передача аргументов):
        f(std::forward<Remainder>(r)...);
    }
}
```

Без инструкции `if constexpr` нам бы потребовалась дополнительная перегрузка шаблона `f()`, чтобы обеспечить завершение рекурсии.

Даже в контексте, не относящемся к шаблонам, `if constexpr` имеет несколько уникальное действие:

```
void h();
void g()
{
    if constexpr(sizeof(int) == 1)
    {
        h();
    }
}
```

На большинстве платформ условие в `g()` дает значение `false`, и поэтому вызов `h()` игнорируется. Как следствие, функции `h()` не обязательно быть определенной вообще (конечно, если она не используется в другом месте). Если в этом примере опустить ключевое слово `constexpr`, отсутствие определения `h()` будет часто приводить к ошибке времени компоновки²⁰.

14.7. В стандартной библиотеке

Стандартная библиотека C++ включает ряд шаблонов, которые широко используются только с несколькими основными типами. Например, шаблон класса `std::basic_string` наиболее часто используется с `char` (потому `std::string` и является псевдонимом типа `std::basic_string<char>`) или `wchar_t`, хотя можно инстанцировать его и с другими символьными типами. Поэтому

²⁰ Оптимизация может замаскировать ошибку; но применение `if constexpr` гарантирует отсутствие проблемы.

в реализациях стандартной библиотеки часто вводятся объявления явного инстанцирования для этих распространенных случаев. Например:

```
namespace std
{
    template<typename charT, typename traits = char_traits<charT>,
              typename Allocator = allocator<charT>>
    class basic_string
    {
        ...
    };
    extern template class basic_string<char>;
    extern template class basic_string<wchar_t>;
}
```

Исходные файлы реализации стандартной библиотеки будут содержать соответствующие определения явного инстанцирования, так что эти общие реализации могут использоваться всеми пользователями стандартной библиотеки. Часто подобные явные реализации имеются для различных классов потоков, например `basic_iostream`, `basic_istream` и так далее.

14.8. Заключительные замечания

Эта глава посвящена двум взаимосвязанным, но различным вопросам: модели компиляции шаблонов C++ и различным механизмам инстанцирования шаблонов C++.

Модель компиляции определяет смысл шаблона на различных этапах трансляции программы. В частности, она определяет, что означают различные конструкции шаблона при его инстанцировании. Важным компонентом модели компиляции является поиск имён.

Стандарт C++ поддерживает только одну модель компиляции — модель включения. Однако в стандартах 1998 и 2003 годов поддерживалась также модель разделения, которая позволяла писать и хранить определение шаблона в единице трансляции, отличной от единицы трансляции, в которой выполняется его инстанцирование. Эти экспортруемые шаблоны были реализованы только один раз, в компиляторе Edison Design Group (EDG)²¹. Их усилия по реализации показали, что 1) реализация модели разделения шаблонов C++ гораздо более сложна и трудоемка, чем ожидалось, и что 2) предполагаемые преимущества модели разделения, такие как ускорение компиляции, не оправдались из-за сложности данной модели. Ко времени принятия стандарта 2011 стало ясно, что никакие другие разработчики не собираются поддерживать эту возможность, так что Комитет по стандартизации C++ проголосовал за удаление экспортруемых шаблонов из языка. Читателям, которых интересуют детали модели разделения, мы рекомендуем обратиться к первому изданию данной книги [71], в которой описано поведение экспортруемых шаблонов.

²¹ Забавно, что именно EDG были самым активным противником этой модели при внесении ее в стандарт.

Механизмы инстанцирования являются внешними механизмами, которые позволяют реализации C++ корректно создавать экземпляры шаблонов. Эти механизмы могут быть ограничены требованиями компоновщика и другого программного обеспечения для построения программ. Механизмы инстанцирования, отличаясь от одной реализации к другой (и каждый имеет собственные компромиссы), обычно не оказывают значительного влияния на повседневное программирование в C++.

Вскоре после того, как стандарт C++11 был завершен, Уолтер Брайт (Walter Bright), Герб Саттер (Herb Sutter) и Андрей Александреску (Andrei Alexandrescu) предложили “статический if”, не слишком сильно отличающийся от “`constexpr if`” (см. статью N3329). Однако их конструкция была более общей возможностью, которая может появляться даже вне определений функций. (Уолтер Брайт – главный проектант и реализатор языка программирования D, имеющего аналогичную функциональную возможность.) Например:

```
template<unsigned long N>
struct Fact
{
    static if (N <= 1)
    {
        constexpr unsigned long value = 1;
    }
    else
    {
        constexpr unsigned long value = N * Fact<N-1>::value;
    }
};
```

Обратите внимание на то, что объявления с областью видимости класса сделаны в этом классе условными. Эта мощная возможность была не принята некоторыми членами Комитета, одни из которых опасались, что она может вызвать злоупотребления, а другим не нравились некоторые технические аспекты предложения (например, тот факт, что фигурные скобки не вводят область видимости и что отбрасываемые ветви не подлежат синтаксическому анализу).

Несколько лет спустя Вилль Вуттиайнен (Ville Voutilainen) вернулся с предложением P0128, которое и стало основой для конструкции `if constexpr`. Оно прошло через несколько итераций незначительных изменений в дизайне (включая предложения использовать ключевые слова `static_if` и `constexpr_if`), и с помощью Йенса Маурера (Jens Maurer) предложение Вилля в конечном итоге оказалось в стандарте языка (см. статью P0292r2).

Глава 15

Вывод аргументов шаблона

Если при каждом вызове шаблона функции явным образом задавать аргументы шаблона (например, `concat<std::string, int>(s, 3)`), то код может быстро стать громоздким. К счастью, компилятор C++ часто в состоянии автоматически определить, какими должны быть аргументы шаблона. Это достигается с помощью мощного механизма под названием *вывод аргументов шаблона* (template argument deduction).

В этой главе подробно объясняется, что происходит в процессе вывода аргументов шаблонов. Как это часто бывает в C++, с этим процессом связано множество правил, соблюдение которых обычно приводит к интуитивно понятному результату. Глубокое понимание материала, изложенного в этой главе, позволит избежать многих досадных неожиданностей.

Хотя вывод аргументов шаблона был впервые разработан для упрощения вызова шаблонов функций, он с тех пор был расширен для ряда других использований, включая определение типов переменных из их инициализаторов.

15.1. Процесс вывода

В процессе вывода типы аргументов вызова функции сравниваются с соответствующими типами параметров шаблона функции, и компилятор пытается сделать вывод о том, что именно нужно подставить вместо одного или нескольких выведенных параметров. Анализ каждой пары “аргумент-параметр” проводится независимо, и, если выводы в конечном итоге отличаются, процесс вывода завершается неудачей. Рассмотрим следующий пример:

```
template<typename T>
T max (T a, T b)
{
    return b < a ? a : b;
}

auto g = max(1, 1.0);
```

Здесь первый аргумент вызова имеет тип `int`, из чего можно заключить, что в роли параметра `T` в исходном шаблоне `max()` должен выступать тип `int`. Однако второй аргумент вызова имеет тип `double`, а это означает, что вместо параметра типа `T` нужно подставить тип `double`. Этот вывод противоречит предыдущему. Заметим, что утверждение “вывод выполнить не удается” не означает, что программа некорректна. В конце концов может случиться так, что этот процесс удастся провести для другого шаблона с именем `max` (шаблоны функций, как и обычные функции, можно перегружать; см. раздел 1.5 и главу 16, “Специализация и перегрузка”).

Даже если удалось вывести все параметры шаблона, это еще не означает, что вывод успешен. Бывает и так, что при подстановке выведенных аргументов в оставшуюся часть определения функции получается некорректная конструкция. Например:

```
template<typename T>
typename T::ElementT at(T a, int i)
{
    return a[i];
}

void f(int* p)
{
    int x = at(p, 7);
}
```

Здесь `T` выводится как `int*` (тут имеется только один тип параметра, в котором появляется `T`, так что, очевидно, никакой конфликт анализа невозможен). Однако подстановка `int*` вместо `T` в возвращаемом типе `T::ElementT`, очевидно, представляет собой некорректный код C++, и процесс вывода завершается неудачей¹.

Рассмотрим, как происходит процедура проверки соответствия параметра и аргумента. Опишем его в терминах соответствия типа *A* (выведенного из типа аргумента) параметризованному типу *P* (выведенному из объявления параметра вызова). Если параметр объявлен как ссылка, считаем, что *P* — это тип, на который делается ссылка, а *A* — тип аргумента. В противном случае *P* представляет собой объявленный тип параметра, а тип *A* получается из типа аргумента путем низведения (decaying)² типов массива или функции к указателю на соответствующий тип. При этом квалификаторы верхнего уровня `const` и `volatile` игнорируются. Например:

```
template<typename T> void f(T); // Параметризованный тип P
                                // представляет собой тип T
template<typename T> void g(T&); // Параметризованный тип P также
                                // представляет собой T
double arr[20];
int const seven = 7;

f(arr); // Передача по значению: Т является double*
g(arr); // Передача по ссылке : Т является double[20]
f(seven); // Передача по значению: Т является int
g(seven); // Передача по ссылке : Т является int const
f(7); // Передача по значению: Т является int
g(7); // Передача по ссылке : Т является int => Ошибка: нельзя
                                // передать 7 в int&
```

¹ В данном случае неуспешный вывод приводит к ошибке. Однако в силу принципа SFINAE (см. раздел 8.4) при наличии функции, для которой вывод завершается успешно, код оказывается корректным.

² Термин, которым обозначается неявное преобразование типов массивов и функций в соответствующие типы-указатели.

При вызове `f(arr)` тип массива `arr` низводится к типу `double*`, который представляет собой тип, выведенный для `T`. В `f(seven)` квалификатор `const` убирается и, следовательно, `T` выводится как `int`. Напротив, вызов `g(x)` приводит к выводу `T` как типа `double[20]` (низведение не выполняется). Аналогично `g(seven)` имеет аргумент, являющийся l-значением типа `int const`, а, поскольку квалификаторы `const` и `volatile` не отбрасываются при сопоставлении ссылочных параметров, `T` выводится как `int const`. Обратите, однако, внимание на то, что `g(7)` приводит к выводу `T` как `int` (поскольку выражение г-значения, не являющееся классом, никогда не имеет квалификаторы типа `const` или `volatile`), и вызов будет неудачным, так как аргумент `7` не может быть передан параметру типа `int&`.

Тот факт, что для аргументов, которые передаются по ссылке, низведение не выполняется, может привести к неожиданным результатам в тех случаях, когда эти аргументы являются строковыми литералами. Еще раз рассмотрим шаблон `max()`, объявленный со ссылками:

```
template<typename T>
T const& max(T const& a, T const& b);
```

Разумно было бы ожидать, что в выражении `max("Apple", "Pie")` параметр `T` будет выведен как тип `char const*`. Однако строка `"Apple"` имеет тип `char const[6]`, а строка `"Pie"` — тип `char const[4]`. Никакого низведения массива к указателю на массив не происходит (поскольку вывод типа выполняется на основе параметров, передаваемых по ссылке). Таким образом, вместо параметра `T` нужно одновременно подставить и тип `char[6]`, и тип `char[4]`, а это, конечно же, невозможно. Более подробное обсуждение этой темы можно найти в разделе 7.4.

15.2. Выводимые контексты

Типу аргумента могут соответствовать значительно более сложные параметризованные типы, чем просто `T`. Ниже приведено несколько (все еще не слишком сложных) примеров:

```
template<typename T>
void f1(T*);
```



```
template<typename E, int N>
void f2(E(&) [N]);
```



```
template<typename T1, typename T2, typename T3>
void f3(T1(T2::*) (T3*));
```



```
class S
{
public:
    void f(double*);
```

```
void g(int*** ppp)
{
    bool b[42];
    f1(ppp); // Т выводится как int**
    f2(b); // Е выводится как bool, а N - как 42
    f3(&S::f); // Выводится: T1 = void, T2 = S и T3 = double
}
```

Сложные объявления типов составляются из более простых конструкций (деклараторов указателей, ссылок, массивов и функций, деклараторов указателей на члены, идентификаторов шаблонов и т.д.). Процесс определения нужного типа происходит в нисходящем порядке, начиная с конструкций высокого уровня и рекурсивно продвигаясь к составляющим их элементам. Уместно заметить, что этим путем можно подобрать тип для большинства конструкций объявлений типов; в этом случае они называются *выводимым контекстом* (deduced context). Однако некоторые конструкции выводимым контекстом не являются. К их числу относятся следующие.

- Квалифицированное имя типа. Например, имя типа наподобие `Q<T>::X` никогда не используется для вывода параметра шаблона `T`.
- Выражения, не являющиеся типами, но которые при этом не являются не-типовыми параметрами (не являющимися типами). Например, ни имя типа наподобие `S<I+1>` никогда не используется для вывода `I`, ни параметр `T` не выводится путем сравнения с параметром типа `int (&) [sizeof(S<T>)]`.

Эти ограничения не должны вызывать удивления, поскольку в общем случае вывод может оказаться неоднозначным (может даже оказаться, что подходящих типов бесконечно много), хотя ограничение на квалифицированные имена типов иногда легко просмотреть. Если в программе встречается невыводимый контекст, это еще не означает, что программа содержит ошибку или что анализируемый параметр не может принимать участия в выводе типа. Чтобы это проиллюстрировать, рассмотрим более сложный пример.

details/fppm.cpp

```
template<int N>
class X
{
public:
    using I = int;
    void f(int)
    {
    }
};

template<int N>
void fppm(void (X<N>::*p)(typename X<N>::I));
int main()
{
    fppm(&X<33>::f); // Все в порядке: N выводится как 33
}
```

Конструкция `X<N>::I`, которая находится в шаблоне функции `format()`, является невыводимым контекстом; однако использующийся в ней компонент `X<N>` член-на класса с типом указателя на член класса является выводимым контекстом. Когда выведенный из этого компонента параметр `N` подставляется в невыводимый контекст, получается тип, совместимый с типом фактического аргумента (`&X<33>::f`). Таким образом, для этой пары “аргумент-параметр” вывод удаётся успешно выполнить до конца.

И наоборот, если параметр типа состоит только из выводимого контекста, то это еще не означает, что вывод не приведет к противоречиям. Например, предположим, что у нас имеются надлежащим образом объявленные шаблоны классов `X` и `Y`. Рассмотрим приведенный ниже код.

```
template<typename T>
void f(X<Y<T>, Y<T>);

void g()
{
    f(X<Y<int>, Y<int>()); // OK
    f(X<Y<int>, Y<char>()); // Ошибка: неудачный вывод
}
```

Проблема, связанная со вторым вызовом шаблона функции `f()`, заключается в том, что для параметра `T` на основе двух аргументов функции выводятся разные типы, что приводит к противоречию. (В обоих вызовах аргументы функции являются временными объектами, полученными путем вызова конструктора по умолчанию для шаблона класса `X`.)

15.3. Особые ситуации вывода

Возможны несколько ситуаций, в которых использующаяся для вывода пара (P, A) получается не из аргументов вызова функции и параметров шаблона функции. Первая ситуация осуществляется, когда используется адрес шаблона функции. В этом случае P – это параметризованный тип объявления шаблона функции, а A – тип функции, на которую ссылается инициализируемый указатель или указатель, которому присваивается значение. Например:

```
template<typename T>
void f(T, T);
void (*pf)(char, char) = &f;
```

В этом примере P – `void(T, T)`, а A – `void(char, char)`. Вывод успешен, при этом `T` заменяется на `char`, а `pf` инициализируется адресом специализации `f<char>`.

Аналогично типы функций для P и A используются и в некоторых других особых ситуациях.

- Определение частичного упорядочения между перегруженными шаблонами функций.
- Соответствие явной специализации шаблону функции.

- Соответствие явного инстанцирования шаблону.
- Соответствие шаблону специализации шаблона дружественной функции.
- Соответствие шаблонов размещающих³ операторов `operator delete` или `operator delete[]` соответствующим шаблонам размещающих операторов `operator new` или `operator new[]`.

Некоторые из этих тем, наряду с использованием вывода аргумента шаблона для частичной специализации шаблона класса, получат дальнейшее развитие в главе 16, “Специализация и перегрузка”.

Еще одна особая ситуация связана с шаблоном оператора преобразования типа. Например:

```
class S
{
public:
    template<typename T> operator T& () ;
};
```

В этом случае пара (P, A) получается таким образом, как если бы в нее входил аргумент того типа, к которому мы пытаемся выполнить преобразование, и тип параметра, возвращаемый оператором преобразования. Приведенный ниже код иллюстрирует один из возможных вариантов этой ситуации.

```
void f(int (&)[20]);
void g(S s)
{
    f(s);
}
```

В этом фрагменте делается попытка преобразовать `S` к типу `int (&) [20]`. Поэтому тип A – это `int [20]`, а тип P – это T . Процесс вывода выполняется успешно, и T заменяется типом `int [20]`.

Наконец, отдельное рассмотрение требуется для вывода заместителя типа `auto` (см. раздел 15.10.4).

15.4. Списки инициализаторов

Когда в качестве аргумента функции выступает список инициализации, такой аргумент не имеет определенного типа, так что в общем случае вывод из данной пары (P, A) не выполняется за отсутствием A . Например:

```
#include <initializer_list>
template<typename T> void f(T p);

int main()
{
    f({1,2,3}); // Ошибка: невозможно вывести Т
                  // из списка в фигурных скобках
```

³ Так у авторов книги. В стандарте C++ размещающих операторов `delete[]` нет. — *Примеч. пер.*

Однако, если тип P параметра после удаления ссылочности и квалификаторов верхнего уровня `const` и `volatile` эквивалентен `std::initializer_list<P>` для некоторого типа P , который имеет выводимую схему, вывод продолжается путем сравнения P с типом каждого элемента в списке инициализаторов, и успешно завершается, только если все элементы имеют один и тот же тип:

deduce/initlist.cpp

```
#include <initializer_list>
template<typename T> void f(std::initializer_list<T>);
int main()
{
    f({2, 3, 5, 7, 9});           // OK: Т выводится как int
    f({'a','e','i','o','u',42}); // Ошибка: Т выводится
                                // и как char, и как int
}
```

Аналогично, если тип параметра P является ссылкой на тип массива с типом элементов P для некоторого типа P , имеющего выводимую схему, то вывод выполняется путем сравнения P с типом каждого элемента в списке инициализаторов, и является успешным, только если все элементы имеют один и тот же тип. Кроме того, если граница имеет выводимую схему (т.е. просто именует параметр шаблона, не являющийся типом), то эта граница выводится как количество элементов в списке.

15.5. Пакеты параметров

Процесс вывода проверяет соответствие каждого аргумента каждому параметру для определения значений аргументов шаблона. Однако при выполнении вывода аргумента шаблона для вариативных шаблонов отношение “один к одному” между параметрами и аргументами больше не выполняется, потому что пакету параметров могут соответствовать несколько аргументов. В этом случае один и тот же пакет параметров (P) сопоставляется нескольким аргументам (A), и каждое соответствие производит дополнительные значения для любого пакета параметров шаблонов в P :

```
template<typename First, typename... Rest>
void f(First first, Rest... rest);
void g(int i, double j, int* k)
{
    f(i, j, k); // Вывод First как int, Rest как {double, int*}
}
```

Здесь вывод первого параметра функции прост, поскольку он не включает каких-либо пакетов параметров. Второй параметр функции, `rest`, представляет собой пакет параметров функции. Его тип — раскрытие пакета (`Rest...`), схема которого имеет тип `Rest`: этот шаблон служит в качестве P , для сравнения с типами A второго и третьего аргументов. При сравнении с первым таким A (тип `double`), первое значение в пакете параметров шаблона `Rest` выводится как `double`. Аналогично, когда сравнивается второй такой A (тип `int*`), второе значение в пакете параметров шаблона `Rest` выводится как `int*`. Таким образом,

вывод определяет значение пакета параметров Rest как последовательность `{double, int*}`. Подстановка результатов этого вывода и вывода для первого параметра функции дает тип функции `void(int, double, int*)`, который соответствует типам аргументов в точке вызова.

Поскольку вывод для пакетов параметров функций использует для сравнения схему раскрытия, схема может быть произвольно сложной, а значения для нескольких параметров шаблона и пакетов параметров могут быть определены из каждого типа аргументов. Рассмотрим поведение вывода для представленных ниже функций `h1()` и `h2()`:

```
template<typename T, typename U> class pair { };

template<typename T, typename... Rest>
void h1(pair<T, Rest> const& ...);

template<typename... Ts, typename... Rest>
void h2(pair<Ts, Rest> const& ...);

void foo(pair<int, float> pif, pair<int, double> pid,
         pair<double, double> pdd)
{
    h1(pif, pid); // OK: T - int, Rest - {float, double}
    h2(pif, pid); // OK: Ts - {int, int}, Rest - {float, double}
    h1(pif, pdd); // Ошибка: T - int bp первого аргумента,
                  // но double из второго
    h2(pif, pdd); // OK: Ts - {int, double}, Rest - {float, double}
}
```

И для `h1()`, и для `h2()` *P* — это ссылочный тип, приведенный к неквалифицированной версии ссылки (`pair<T, Rest>` или `pair<Ts, Rest>` соответственно) для вывода типа каждого аргумента. Так как все параметры и аргументы являются специализациями шаблона класса `pair`, сравниваются аргументы шаблона. Для `h1()` первый аргумент шаблона (`T`) не является пакетом параметров, поэтому его значение выводится независимо для каждого аргумента. Если выведенные типы отличаются, как во втором вызове `h1()`, вывод является неуспешным. Для второго аргумента шаблона `pair` в `h1()` и `h2()` (`Rest`) и для первого аргумента `pair` в `h2()` (`Ts`) вывод из каждого из типов аргументов в *A* определяет последовательные значения пакетов параметров шаблонов.

Вывод для пакетов параметров не ограничивается пакетами параметров функции, где из аргументов вызова получаются пары “аргумент-параметр”. Фактически этот вывод используется везде, где раскрытие пакета находится в конце списка параметров функции или списка аргументов шаблонов⁴. Рассмотрим две аналогичные операции над простым типом `Tuple`:

```
template<typename... Types> class Tuple { };

template<typename... Types>
bool f1(Tuple<Types...>, Tuple<Types...>);
```

⁴ Если раскрытие пакета происходит в некоторой иной точке в списке параметров функции или списке аргументов шаблона, то раскрытие пакета рассматривается как невыводимый контекст.

```

template<typename... Types1, typename... Types2>
bool f2(Tuple<Types1...>, Tuple<Types2...>);

void bar(Tuple<short, int, long> sv,
         Tuple<unsigned short, unsigned, unsigned long> uv)
{
    f1(sv, sv); // OK: Types выводится как {short, int, long}
    f2(sv, sv); // OK: Types1 выводится как {short, int, long},
                 //      Types2 выводится как {short, int, long}
    f1(sv, uv); // Ошибка: Types выводится как {short, int, long}
                 //      из первого аргумента, но как {unsigned
                 //      short, unsigned, unsigned long} из второго
    f2(sv, uv); // OK: Types1 выводится как {short, int, long},
                 //      Types2 выводится как {unsigned short,
                 //      unsigned, unsigned long}
}

```

И в f1(), и в f2() пакеты параметров шаблонов выводятся путем сравнения схемы раскрытия пакета, встроенного в тип Tuple (например, Types для h1()) с каждым из аргументов шаблона типа Tuple, предоставляемым аргументом вызова, и выведением последовательных значений для соответствующего пакета параметров шаблона. Функция f1() использует один и тот же пакет параметров шаблона Types в обоих параметрах функции, гарантируя, что вывод завершается успешно только тогда, когда аргументы вызова двух функций имеют ту же специализацию Tuple, что и их тип. С другой стороны, функция f2() использует различные пакеты параметров для типов Tuple в каждом из своих параметров функции, так что типы аргументов вызова функции могут быть различными – при условии, что оба являются специализациями Tuple.

15.5.1. Шаблоны оператора литерала

Шаблоны оператора литерала (literal operator template) должны определять свои аргументы уникальным способом, иллюстрируемым следующим примером:

```

template<char...> int operator "" _B7(); // #1
***                                                 // #2
int a = 121_B7;

```

Здесь инициализатор в #2 содержит пользовательский литерал, который превращается в вызов шаблона оператора литерала #1 со списком аргументов шаблона <'1','2','1'>. Таким образом, реализация оператора литерала, такая как

```

template<char... cs>
int operator"" _B7()
{
    std::array<char, sizeof...(cs)> // Инициализация массива
        chars{cs...}; // переданными символами
    for (char c : chars)           // и их использование (в
    {                            // данном случае – печать)
        std::cout << "!" << c << " ";
    }

    std::cout << '\n';
    return ...;
}

```

выведет для литерала `121.5_B7` следующее:

```
'1' '2' '1' '.' '5'
```

Обратите внимание на то, что эта методика поддерживается только для числовых литералов, корректных даже без суффикса. Например:

```
auto b = 01.3_B7;      // OK: выводит <'0', '1', '.', '3'>
auto c = 0xFF00_B7;    // OK: выводит <'0', 'x', 'F', 'F', '0', '0'>
auto d = 0815_B7;      // Ошибка: 8 не является корректным
                      // восьмеричным литералом
auto e = hello_B7;    // Ошибка: идентификатор hello_B7 не определен
auto f = "hello"_B7;  // Ошибка: нет соответствия оператора _B7
```

В разделе 25.6 описывается применение этой возможности для вычисления целочисленных литералов во время компиляции.

15.6. Ссылки на *r*-значения

В C++11 вводятся ссылки на *r*-значения (*rvalue references*), которые позволяют применять новые методы работы, включая семантику перемещений и прямую передачу. В этом разделе описывается взаимодействие между ссылками на *r*-значения и выводом типов.

15.6.1. Правила свертки ссылок

Программисты не могут объявить “ссылку на ссылку” непосредственно:

```
int const& r = 42;
int const&& ref2ref = i; // Ошибка: ссылка на ссылку
```

Однако при создании типов путем подстановки параметров шаблонов, применения псевдонимов типов или конструкций `decltype`, такие ситуации допускаются. Например:

```
using RI = int&;
int i = 42;
RI r = i;
RI const& rr = r; // OK: rr имеет тип int&
```

Правила, которые определяют тип, получающийся при такой композиции, известны как правила *свертки ссылок* (*reference collapsing*)⁵. Во-первых, любые квалификаторы `const` или `volatile`, применяемые поверх внутренней ссылки, просто отбрасываются (т.е. сохраняются только квалификаторы *под* знаком внутренней ссылки). Затем две ссылки сводятся к одной согласно табл. 15.1, которую можно резюмировать как “если любая из ссылок является ссылкой на *l*-значение, то таковым будет и результирующий тип; в противном случае получаем ссылку на *r*-значение”.

⁵ Свертка ссылок была введена в C++ стандартом 2003 года, когда было замечено, что стандартный шаблон класса `pair` не работает со ссылочными типами. Стандарт 2011 года расширил эти правила, включив в них ссылки на *r*-значения.

Таблица 15.1. Правила свертки ссылок

Внутренняя ссылка	Внешняя ссылка	Результирующая ссылка
&	+	& ⇒ &
&	+	&& ⇒ &
&&	+	& ⇒ &
&&	+	&& ⇒ &&

Еще один пример показывает эти правила в действии:

```
using RCI = int const&;
RCI volatile&& r = 42; // OK: r имеет тип int const&
using RRI = int&& ;
RRI const&& rr = 42; // OK: rr имеет тип int&&
```

Здесь `volatile` применяется поверх ссылочного типа `RCI` (псевдоним для `int const &`) и поэтому игнорируется. Затем поверх этого типа применяется ссылка на `r`-значение, но, поскольку базовый тип представляет собой ссылку на `l`-значение, а ссылки на `l`-значение имеют более высокий “приоритет” в правилах свертки ссылок, в конечном итоге получается тип `int const &` (или эквивалентный псевдоним `RCI`). Аналогично отбрасывается `const` поверх `RRI`, а применение ссылки на `r`-значение поверх получающейся ссылки на `r`-значение по-прежнему оставляет нас с типом ссылки на `r`-значение (позволяя связывать ее с таким `r`-значением, как 42).

15.6.2. Передаваемые ссылки

Как рассказывалось в разделе 6.1, вывод аргумента шаблона ведет себя особым образом, когда параметр функции представляет собой *передаваемую ссылку* (forwarding reference) (ссылку на `g`-значение на параметр шаблона данного шаблона функции). В этом случае вывод аргумента шаблона рассматривает не только тип аргумента вызова функции, но и выясняет, является этот аргумент `l`-значением или `g`-значением. В случаях, когда аргумент является `l`-значением, тип, определяемый выводом аргумента шаблона, представляет собой ссылку на `l`-значение на тип аргумента, а рассмотренные выше правила свертки ссылок приводят к тому, что подставляемый параметр будет ссылкой на `l`-значение. В противном случае выведенный для параметра шаблона тип является просто типом аргумента (не ссылочным типом), и подставляемый параметр является ссылкой на `g`-значение на этот тип. Например:

```
template<typename T> void f(T& p); // p - передаваемая ссылка
void g()
{
    int i;
    int const j = 0;
    f(i);    // Аргумент является l-значением; Т выводится как
              // int&, а параметр p имеет тип int&
    f(j);    // Аргумент является l-значением; Т выводится как
              // int const&, а параметр p имеет тип int const&
```

```
f(2); // Аргумент является l-значением; T выводится как
      // int, а параметр p имеет тип int&&
}
```

В вызове `f(i)` параметр шаблона `T` выводится как `int&`, поскольку выражение `i` является `l`-значением типа `int`. Подстановка `int&` вместо `T` в тип параметра `T&&` требует свертки ссылки, и мы применяем правило `&+&&-&`, чтобы заключить, что результирующий параметр типа — `int&`, который идеально подходит для принятия `l`-значения типа `int`. Напротив, в вызове `f(2)`, аргумент `2` представляет собой `g`-значение, и поэтому параметр шаблона выводится просто как тип `g`-значения (т.е. `int`). Для результирующего параметра функции, который представляет собой простой `int&&`, применение правил свертки ссылок не требуется (а параметр подходит для переданного аргумента).

Вывод `T` как ссылочного типа может иметь некоторое интересное влияние на инстанцирование шаблона. Например, локальная переменная, объявленная с типом `T`, после инстанцирования для `l`-значения имеет ссылочный тип и поэтому требует наличия инициализатора:

```
template<typename T> void f(T&&) // p - передаваемая ссылка
{
    T x; // Для переданного l-значения x является ссылкой
    ...
}
```

Это означает, что при определении функции `f()` выше необходимо быть осторожным при использовании типа `T`, иначе сам шаблон функции не будет корректно работать с аргументами, являющимися `l`-значениями. Чтобы справиться с этой ситуацией, часто используется свойство типа `std::remove_reference` для гарантии, что `x` не является ссылкой:

```
template<typename T> void f(T&&) // p - передаваемая ссылка
{
    std::remove_reference_t<T> x; // x никогда не является ссылкой
    ...
}
```

15.6.3. Прямая передача

Сочетание специальных правил вывода для ссылок на `g`-значения и правил свертки ссылок позволяет написать шаблон функции с параметром, который принимает практически любой аргумент⁶ и фиксирует его основные свойства (его тип и то, является он `l`-значением или `g`-значением). Шаблон функции может затем “передать” аргумент другой функции следующим образом:

```
class C
{
    ...
};

void g(C&);
```

⁶ Исключением являются битовые поля.

```

void g(C const&);
void g(C&&);

template<typename T>
void forwardToG(T&& x)
{
    g(static_cast<T&&>(x)); // Передача x в g()
}

void foo()
{
    C v;
    C const c;
    forwardToG(v);           // Вызов g(C&)
    forwardToG(c);           // Вызов g(C const&)
    forwardToG(C());          // Вызов g(C&&)
    forwardToG(std::move(v)); // Вызов g(C&&)
}

```

Проиллюстрированная здесь методика называется *прямой передачей* (perfect forwarding), потому что результат вызова `g()` косвенно через `forwardToG()` будет тем же, как если бы код вызывал `g()` непосредственно: не делается никаких дополнительных копий, и выбирается в точности та же перегрузка `g()`.

Применение `static_cast` в функции `forwardToG()` требует некоторых дополнительных объяснений. В каждом инстанцировании `forwardToG()` параметр `x` будет иметь тип либо ссылки на l-значение, либо ссылки на r-значение. Независимо от этого выражение `x` будет l-значением того типа, на который указывает ссылка⁷. `static_cast` приводит `x` к исходному типу и l- или r-“значимости”. Тип `T&&` будет либо свернут к ссылке на l-значение (если исходный аргумент был l-значением, заставляя `T` стать ссылкой на l-значение) или ссылке на r-значение (если исходный аргумент был r-значением), так что результат применения `static_cast` имеет тот же тип и l- или r-“значимость”, что и исходный аргумент, обеспечивая тем самым прямую передачу.

Как было сказано в разделе 6.1, стандартная библиотека C++ предоставляет шаблон функции `std::forward<>()` в заголовочном файле `<utility>`, которая должна использоваться для корректной передачи вместо `static_cast`. С помощью этого вспомогательного шаблона намерения программиста оказываются лучше документированы, чем при применении непрозрачного `static_cast`, он также препятствует таким ошибкам, как пропуск одного символа &. То есть приведенный выше пример более ясно записывается следующим образом:

⁷ Рассмотрение параметра типа ссылки на r-значение как l-значения выполняется преднамеренно для большей безопасности, потому что по всему, что имеет имя (как, например, параметр), в функции может иметься множество обращений. Если каждое из этих обращений будет неявно рассматриваться как обращение к r-значению, то последнее может незаметно для программиста оказаться уничтоженным. Таким образом, следует явно указывать, когда именованные сущности следует рассматривать как r-значения. Для этой цели функция `std::move` стандартной библиотеки C++ позволяет трактовать любое значение как r-значение (или, точнее, x-значение; подробности см. в приложении Б, “Категории значений”).

```
#include <utility>
template<typename T> void forwardToG(T&& x)
{
    g(std::forward<T>(x)); // Передача x в g()
}
```

Прямая передача для вариативных шаблонов

Прямая передача хорошо сочетается с вариативными шаблонами, позволяя шаблону функции принимать любое количество аргументов вызова функции и передавать каждый из них в другую функцию:

```
template<typename... Ts> void forwardToG(Ts&&... xs)
{
    g(std::forward<Ts>(xs)...); // Передача всех xs в g()
}
```

Аргументы в вызове `forwardToG()` (независимо) приводят к выводу последовательных значений пакета параметров `Ts` (см. раздел 15.5), так что сохраняются типы и l- или r-“значимость”. Раскрытие пакета (см. раздел 12.4.1) в вызове `g()` выполняет передачу каждого из этих аргументов с использованием технологии прямой передачи, объясненной выше.

Несмотря на свое название, прямая передача по сути является не совсем “прямой”, в том смысле, что она не охватывает все интересные свойства выражения. Например, она не в состоянии выяснить, ни является ли l-значение значением битового поля, ни имеет ли выражение определенное константное значение. Последнее может вызвать проблемы, в частности, когда мы имеем дело с константой нулевого указателя, которая представляет собой константное значение целочисленного типа, равное нулю. Поскольку константное значение выражения при прямой передаче не захватывается, разрешение перегрузки в следующем примере будет вести себя по-разному для прямого и переадресованного вызова `g()`:

```
void g(int*);
void g(...);

template<typename T> void forwardToG(T&& x)
{
    g(std::forward<T>(x)); // Передача x в g()
}

void foo()
{
    g(0);                  // Вызов g(int*)
    forwardToG(0);         // Вызов g(...)
}
```

Это еще одна причина использовать значение `nullptr` (введенное в C++11) вместо константы нулевого указателя:

```
g(nullptr);           // Вызов g(int*)
forwardToG(nullptr); // Вызов g(int*)
```

Все наши примеры прямой передачи были сосредоточены на передаче аргументов функций при сохранении их точного типа и l- или r-“значимости”. Та же проблема

возникает при пересылке возвращаемого значения вызова в другую функцию с точно тем же типом и *категорией значения*, обобщением l- и r-значений, рассматривающимся в приложении Б, “Категории значений”. Конструкция `decltype`, введенная в C++11 (и описанная в разделе 15.10.2), обеспечивает применение показанной далее несколько многословной идиомы:

```
template<typename... Ts>
auto forwardToG(Ts&&... xs) -> decltype(g(std::forward<Ts>(xs)...))
{
    return g(std::forward<Ts>(xs)...); // Передача всех xs в g()
}
```

Обратите внимание на то, что выражение в операторе `return` скопировано в тип `decltype`, так что вычисляется точный тип возвращаемого выражения. Кроме того, используется *завершающий возвращаемый тип* (trailing return type) функции (т.е. для указания возвращаемого типа перед именем функции ставится заместитель `auto`, а после него — `->`), так что пакет параметров параметров функции `xs` находится в области видимости данного типа `decltype`. Эта функция переадресации выполняет прямую передачу всех аргументов функции `g()`, а затем выполняет такую же прямую передачу результата вызова обратно в вызывающий код.

C++14 вводит дополнительные возможности упрощения данного кода:

```
template<typename... Ts>
decltype(auto) forwardToG(Ts&& ... xs)
{
    return g(std::forward<Ts>(xs)...); // Передача всех xs в g()
}
```

Использование `decltype(auto)` в качестве типа возвращаемого значения указывает, что компилятор должен вывести тип возвращаемого значения из определения функции (см. разделы 15.10.1 и 15.10.3).

15.6.4. Сюрпризы вывода

Результаты специальных правил вывода для ссылок на r-значения являются очень полезными для прямой передачи. Однако они могут оказаться неожиданными, потому что шаблоны функций обычно обобщают типы в сигнатуре функции, не учитывая, какие разновидности аргументов (l-значения или r-значения) она допускает. Рассмотрим следующий пример:

```
void int lvalues(int&); // Принимает l-значения типа int
template<typename T>
void lvalues(T&); // Принимает l-значения любого типа

void int_rvalues(int&&); // Принимает r-значения типа int

template<typename T>
void anything(T&&); // Сюрприз: принимает l- и r-значения
// любого типа
```

Программисты, которые просто абстрагируют конкретные функции наподобие `int_rvalues` в их шаблонные эквиваленты, вероятно, будут удивлены тем

фактом, что шаблон функции `anything` принимает `l`-значения. К счастью, это поведение вывода применяется только тогда, когда параметр функции, записанный в виде *параметра шаблона* `&&`, является частью шаблона функции, и именованный параметр шаблона объявляется этим шаблоном функции. Таким образом, данное правило вывода *не* применяется ни в одной из следующих ситуаций:

```
template<typename T>
class X
{
public:
    X(X&&);           // X не является параметром шаблона
    X(T&&);           // Этот конструктор не является шаблоном функции
    template<typename U>
        X(X<U>&&); // X<U> не является параметром шаблона
    template<typename U>
        X(U, T&&); // T является параметром внешнего шаблона
};
```

Несмотря на удивительное поведение, которое дает это правило вывода, случаи, когда такое поведение вызывает проблемы, на практике встречается нечасто. Когда это происходит, можно использовать сочетание SFNAE (см. разделы 8.4 и 15.7) и свойство типа, такое как `std::enable_if` (см. разделы 6.3 и 20.3), для того, чтобы ограничить шаблон только `g`-значениями:

```
template<typename T>
typename std::enable_if<!std::is_lvalue_reference<T>::value>::type
rvalues(T&&); // Принимает r-значения любого типа
```

15.7. SFNAE

Принцип SFNAE (substitution failure is not an error — ошибка подстановки ошибкой не является), представленный в разделе 8.4, является важным аспектом вывода аргументов шаблона, не позволяющим шаблонам не относящихся к делу функций приводить к ошибкам при разрешении перегрузки⁸.

Рассмотрим, например, пару шаблонов функций, которые получают итератор, указывающий на начало контейнера или массива:

```
template<typename T, unsigned N>
T* begin(T(&array)[N])
{
    return array;
}

template<typename Container>
typename Container::iterator begin(Container& c)
{
    return c.begin();
}

int main()
```

⁸ Принцип SFNAE также применим к подстановке частичных специализаций шаблонов классов (см. раздел 16.4).

```

{
    std::vector<int> v;
    int a[10];
    ::begin(v); // OK: соответствует только begin() для контейнера,
    // поскольку первый вывод неудачен
    ::begin(a); // OK: соответствует только begin() для массивов,
    // поскольку вторая подстановка неудачна
}

```

Первый вызов `begin()`, аргумент которого представляет собой `std::vector<int>`, пытается вывести аргумент шаблона для обоих шаблонов функций.

- Вывод аргумента шаблона `begin()` для массива неуспешен, поскольку `std::vector` не является массивом, а потому этот шаблон игнорируется.
- Вывод аргумента шаблона `begin()` для контейнера успешен при выводе `Container` как `std::vector<int>`, так что инстанцируется и вызывается именно этот шаблон функции.

Второй вызов `begin()`, аргумент которого представляет собой массив, также частично неудачен.

- Вывод аргумента шаблона `begin()` для массива успешен при выводе `T` как `int`, а `N` — как `10`.
- Вывод аргумента шаблона `begin()` для контейнера определяет, что `Container` должен быть заменен на `int [10]`. В то время как в общем случае эта подстановка корректна, получающийся возвращаемый тип `Container::iterator` является неверным, поскольку тип массива не имеет вложенного типа по имени `iterator`. В любом ином контексте попытка обращения к несуществующему вложенному типу привела бы к немедленной ошибке времени компиляции. При подстановке аргументов шаблона SFNAE превращает такие ошибки в сбои вывода, и соответствующие шаблоны функций удаляются из рассмотрения. Таким образом, второй кандидат `begin()` игнорируется, и вызывается специализация первого шаблона функции `begin()`.

15.7.1. Непосредственный контекст

SFINAE защищает от попыток формирования недопустимых типов или выражений, включая ошибки из-за неоднозначностей или нарушений контроля доступа, которые происходят в *непосредственном контексте* (*immediate context*) подстановки шаблона функции. Определение *непосредственного контекста* подстановки шаблона функции более легко дать путем определения, что *не является* этим контекстом⁹. В частности, во время подстановки в шаблон функции с целью вывода ничто из того, что происходит во время инстанцирования,

⁹ *Непосредственный контекст* включает многое, в том числе различные разновидности поиска, подстановки шаблонов псевдонимов, разрешение перегрузки и т.п. Возможно, этот термин является несколько неправильным, потому что некоторые из действий, которые он включает в себя, не связаны с шаблоном функции, для которого выполняется подстановка.

не является частью непосредственного контекста подстановки этого шаблона функции, в частности:

- определение шаблона класса (т.е. его тело и список базовых классов);
- определение шаблона функции (тело и, в случае конструктора, его инициализаторы);
- инициализатор шаблона переменной;
- аргумент по умолчанию;
- инициализатор члена по умолчанию;
- спецификации исключений.

Также не является частью непосредственного контекста подстановки любое неявное определение специальных функций-членов, вызванное процессом подстановки. Все остальное *является* частью этого контекста.

Так что если подстановка параметров шаблона объявления шаблона функции требует инстанцирования тела шаблона класса, потому что ссылается на член этого класса, ошибка при таком инстанцировании не находится в *непосредственном контексте* подстановки шаблона функции, а потому является реальной ошибкой (даже если для другого шаблона функции соответствие определяется без ошибок). Например:

```
template<typename T>
class Array
{
public:
    using iterator = T*;
};

template<typename T>
void f(Array<T>::iterator first, Array<T>::iterator last);

template<typename T>
void f(T*, T*);

int main()
{
    f<int&>(0, 0); // Ошибка: подстановка int& вместо T в первом
}                      // шаблоне функции инстанцирует Array<int&>,
                      // что приводит к ошибке
```

Основное различие между этим примером и предыдущим заключается в том, где происходит сбой. В предыдущем примере ошибка происходила при формировании типа `typename Container::iterator`, который находился в непосредственном контексте подстановки шаблона функции `begin()`. В этом же примере сбой происходит при инстанцировании `Array<int&>`, которое, хотя и запущено из контекста шаблона функции, на самом деле происходит в контексте шаблона класса `Array`. Таким образом, принцип SFINAE не применяется, и компилятор выдаст сообщение об ошибке.

Далее приведен пример на C++14, который основан на выводе возвращаемого типа (см. раздел 15.10.1) и включает ошибку во время инстанцирования определения шаблона функции:

```
template<typename T> auto f(T p)
{
    return p->m;
}

int f(...);

template<typename T> auto g(T p) -> decltype(f(p));

int main()
{
    g(42);
}
```

Вызов `g(42)` выводит тип `T` как `int`. Выполнение соответствующей подстановки в объявлении `g()` требует от нас определить тип `f(p)` (где `p`, как теперь известно, имеет тип `int`), а следовательно, и возвращаемый тип `f()`. Для `f()` есть два кандидата. Нешаблонный кандидат обеспечивает совпадение, но не очень хорошее — потому что это совпадение с параметром, являющимся многоточием. К сожалению, шаблонный кандидат имеет выведенный тип возвращаемого значения, так что мы должны инстанцировать его определение для вывода типа возвращаемого значения. Это инстанцирование неудачное, потому что `p->m` не является допустимым, когда `p` представляет собой `int`; а поскольку отказ происходит вне непосредственного контекста подстановки (так как она происходит в последующем инстанцировании определения функции), этот отказ приводит к ошибке. Из-за этого мы рекомендуем избегать вывода типов возвращаемых значений, если они легко могут быть заданы явно.

Принцип SFINAE первоначально предназначался для устранения странно выглядящих ошибок из-за непреднамеренных соответствий с перегрузками шаблонов функций, как в случае с примером `begin()` для контейнера. Однако способность обнаруживать недопустимое выражение или тип обеспечивает работоспособность замечательных методов времени компиляции, позволяя определить, корректен ли тот или иной конкретный синтаксис. Эти методы рассматриваются в разделе 19.4.

Особого внимания заслуживает раздел 19.4.4, где приводится пример создания свойства типа, *дружественного к SFINAE*, чтобы избежать проблем, связанных с *непосредственным контекстом*.

15.8. Ограничения вывода

Вывод аргумента шаблона — это мощная функциональная возможность, устраниющая необходимость явно указывать аргументы шаблона в большинстве вызовов шаблонов функций, и обеспечивающая как перегрузку шаблонов функций (см. раздел 1.5), так и частичную специализацию шаблонов классов

(см. раздел 16.4). Однако есть несколько ограничений, с которыми при использовании шаблонов могут столкнуться программисты. Эти ограничения и обсуждаются в данном разделе.

15.8.1. Допустимые преобразования аргументов

Обычно вывод шаблонов пытается найти подстановку для параметров шаблона функции, которая делает параметризованный тип P идентичным типу A . Однако когда это невозможно, допускаются следующие различия, когда P содержит параметр шаблона в выводимом контексте.

- Если исходный параметр был объявлен с декларатором ссылки, то заменяемый тип P может иметь дополнительные квалифиликаторы `const/volatile` по сравнению с типом A .
- Если тип A представляет собой тип указателя или член, он может быть преобразуем в заменяемый тип P путем преобразования квалификации (другими словами, преобразования, которое добавляет квалифиликаторы `const` и/или `volatile`).
- Если только вывод не выполняется для шаблона оператора преобразования типа, заменяемый тип P может быть типом базового класса для типа A или указателем на тип базового класса для типа класса, для которого A является типом указателя. Например:

```
template<typename T>
class B
{
};

template<typename T>
class D : public B<T>
{
};

template<typename T> void f(B<T>* );
void g(D<long> d1)
{
    f(&d1); // Вывод успешен; Т заменяется на long
}
```

Если P не содержит параметр шаблона в выводимом контексте, допустимы *все* неявные преобразования. Например:

```
template<typename T> int f(T, typename T::X);

struct V
{
    V();
    struct X
    {
        X(double);
    };
} v;
```

```
int r = f(v, 7.0); // OK: Т из первого параметра выводится как V,
                  // что приводит к тому, что второй параметр имеет
                  // тип V::X, который может быть построен из double
```

Ослабленные требования соответствия рассматриваются только тогда, когда точное соответствие невозможно. Даже в этом случае вывод успешен, только если обнаруживается ровно одна подстановка, которая позволяет “подогнать” тип *A* к заменяемому типу *P* с указанными дополнительными преобразованиями.

Обратите внимание на то, что эти правила очень узки и игнорируют, например, различные преобразования, которые могут быть применены к аргументам функции, чтобы сделать вызов успешным. Рассмотрим такой вызов шаблона функции `max()`, показанной в разделе 15.1:

```
std::string maxWithHello(std::string s)
{
    return ::max(s, "hello");
}
```

Здесь вывод аргумента шаблона из первого аргумента дает в качестве *T* тип `std::string`, в то время как из второго аргумента *T* выводится как `char[6]`, поэтому вывод аргументов шаблона неудачен — так как оба параметра используют один и тот же параметр шаблона. Эта неудача может показаться удивительной, поскольку строковый литерал "hello" неявно преобразуем в `std::string`, и вызов

```
::max<std::string>(s, "hello")
```

должен быть успешен.

Возможно, еще более удивительно то, что, когда два аргумента имеют различные типы классов, производных от общего базового класса, вывод не рассматривает этот общий базовый класс в качестве кандидата для выведенного типа. Этот вопрос и возможные его решения рассматриваются в разделе 1.2.

15.8.2. Аргументы шаблона класса

До C++17 вывод аргумента шаблона применяется исключительно к шаблонам функций и функций-членов. В частности, аргументы для шаблона класса не выводились из аргументов вызова одного из его конструкторов. Например:

```
template<typename T>
class S
{
public:
    S(T b) : a(b)
    {
    }
private:
    T a;
};
S x(12); // Ошибка до C++17: параметр T шаблона класса
          // не выводится из аргумента вызова конструктора 12
```

Это ограничение снято в C++17 — см. раздел 15.12.

15.8.3. Аргументы вызова по умолчанию

Аргументы вызова функции по умолчанию могут быть указаны в шаблонах функций так же, как и в обычных функциях:

```
template<typename T>
void init(T* loc, T const& val = T())
{
    *loc = val;
}
```

На самом деле, как показывает этот пример, аргумент вызова функции по умолчанию может зависеть от параметра шаблона. Такой зависимый аргумент по умолчанию инстанцируется только тогда, когда не указаны явные аргументы — принцип, который делает следующий пример корректным:

```
class S
{
public:
    S(int, int);
};

S s(0, 0);

int main()
{
    init(&s, S(7, 42)); // T() некорректно в случае T = S, но
                        // аргумент вызова по умолчанию T() не
                        // инстанцируется, так как имеется
    }                   // явно указанный аргумент.
```

Даже когда аргумент вызова по умолчанию не является зависимым, он не может использоваться для вывода аргументов шаблона. Это означает, что следующий код — неверный код C++:

```
template<typename T>
void f(T x = 42)
{
}

int main()
{
    f<int>(); // OK: T = int
    f();       // Ошибка: невозможно вывести T
}           // из аргумента вызова по умолчанию
```

15.8.4. Спецификации исключений

Подобно аргументам вызова по умолчанию спецификации исключений инстанцируются только тогда, когда они необходимы. Это означает, что они не участвуют в выводе аргумента шаблона. Например:

```
template<typename T>
void f(T, int) noexcept(nonexistent(T())); // #1
```

```
template<typename T>
void f(T, ...); // #2 (вариативная функция в стиле C)

void test(int i)
{
    f(i, i); // Ошибка: выбран #1, но выражение
} // nonexistent(T()) некорректно
```

Спецификация поexcept в функции, помеченной как #1, пытается вызвать несуществующую функцию. Обычно такие ошибки непосредственно в объявлении шаблона функции приводят к неудаче вывода аргумента шаблона (SFINAE), позволяя вызову `f(i, i)` достичь успеха, выбрав функцию, помеченную как #2, которая без этого имела бы худшее соответствие (соответствие многоточию — худший вид соответствия с точки зрения разрешения перегрузки; смите приложение B, “Разрешение перегрузки”). Однако поскольку спецификации исключений не участвуют в выводе аргумента шаблона, разрешение перегрузки выбирает #1, и программа становится некорректной при более позднем инстанцировании спецификации поexcept.

То же правило применимо к спецификациям исключений, которые перечисляют потенциально возможные типы исключений:

```
template<typename T>
void g(T, int) throw(typename T::Nonexistent); // #1

template<typename T>
void g(T, ...); // #2

void test(int i)
{
    g(i, i); // Ошибка: выбирает #1, но тип T::Nonexistent некорректен
}
```

Однако эти “динамические” спецификации исключений не рекомендованы к применению начиная с C++11, и были полностью удалены в C++17.

15.9. Явные аргументы шаблонов функций

Когда аргумент шаблона функции не может быть выведен, может оказаться возможным явно указать его после имени шаблона функции. Например:

```
template<typename T> T default_value()
{
    return T{};
}

int main()
{
    return default_value<int>();
```

Это также может быть сделано и для выводимых параметров шаблонов:

```
template<typename T> void compute(T p)
{
    ...
}
```

```
int main()
{
    compute<double>(2);
}
```

После того как аргумент шаблона указан явно, соответствующий ему параметр больше не подлежит выводу. Это, в свою очередь, позволяет выполнить преобразования над параметрами вызова функции, которые не были бы возможны в выведенном вызове. В примере выше аргумент 2 в вызове `compute<double>(2)` неявно преобразуется в `double`.

Это позволяет явно указать некоторые аргументы шаблона, в то время как остальные могут быть выведены. Однако явно указанные аргументы всегда должны соответствовать параметрам шаблона слева направо. Поэтому в объявлении шаблона сначала указываются параметры, которые нельзя вывести (или которые, скорее всего, будут указаны явно). Например:

```
template<typename Out, typename In>
Out convert(In p)
{
    ...
}
int main()
{
    auto x = convert<double>(42); // Тип параметра p выводится,
                                    // возвращаемый тип указан явно
}
```

Иногда полезно указать пустой список аргументов шаблона, чтобы гарантировать, что выбранная функция является экземпляром шаблона, но при этом использует вывод для определения аргументов шаблона:

```
int f(int);                                // #1
template<typename T> T f(T);                // #2

int main()
{
    auto x = f(42);                         // Вызов #1
    auto y = f<>(42);                      // Вызов #2
}
```

Здесь для вызова `f(42)` выбирается нешаблонная функция, потому что разрешение перегрузки предпочитает при прочих равных условиях обычные функции шаблонам функций. Однако для вызова `f<>(42)` наличие списка аргументов шаблона требует выбора шаблона функции (даже если фактические аргументы шаблона не указаны).

В контексте объявлений дружественных функций наличие явного списка аргументов шаблона имеет интересный эффект. Рассмотрим следующий пример:

```
void f();
template<typename> void f();
```

```
namespace N
{
    class C
    {
        friend int f(); // OK
        friend int f<>(); // Ошибка: конфликт возвращаемого типа
    };
}
```

Когда простой идентификатор используется для именования дружественной функции, эта функция ищется только в ближайшей охватывающей области видимости, и если она там не найдена, объявляется новая сущность в данной области видимости (но она остается “невидимой”, за исключением поиска, зависящего от аргументов (ADL); см. раздел 13.2.2). Это и происходит с нашим первым объявлением `friend` выше: `f` не объявлена в пространстве имен `N`, поэтому “невидимо” объявляется новая функция `N::f()`.

Однако когда за идентификатором, именующим друга, следует список аргументов шаблона, шаблон в этой точке должен быть виден с помощью обычного поиска, и этот обычный поиск будет проходить по любому необходимому количеству областей видимости. Таким образом, наше второе объявление выше найдет глобальный шаблон функции `f()`, но затем компилятор выдаст ошибку, поскольку типы возвращаемых значений не совпадают (поскольку здесь не выполняется ADL, объявление, созданное предыдущим объявлением дружественной функции, игнорируется).

Явно указанные аргументы шаблона заменяются с использованием принципов SFINAE: если подстановка приводит к ошибке в непосредственном контексте этой подстановки, шаблон функции отбрасывается, но другие шаблоны могут по-прежнему быть успешны. Например:

```
template<typename T> typename T::EType f(); // #1
template<typename T> T f(); // #2

int main()
{
    auto x = f<int*>();
}
```

Здесь подстановка `int*` вместо `T` в кандидате #1 является неудачной, но в кандидате #2 она вполне успешна, поэтому выбирается этот кандидат. Фактически если после подстановки остается ровно один кандидат, то имя шаблона функции с явными аргументами шаблона ведет себя очень похоже на имя обычной функции, включая низведение к типу указателя на функцию во многих контекстах. То есть, заменив приведенную выше функцию `main()` следующей

```
int main()
{
    auto x = f<int*>; // OK: x - указатель на функцию
}
```

мы получим корректную единицу трансляции. Однако следующий код:

```
template<typename T> void f(T);
template<typename T> void f(T, T);

int main()
{
    auto x = f<int*>; // Ошибка: две возможных f<int*>
}
```

будет некорректным, поскольку `f<int*>` в данном случае не идентифицирует единственную функцию.

Вариативные шаблоны функций также могут использоваться с явными аргументами шаблона:

```
template<typename ... Ts> void f(Ts ... ps);
int main()
{
    f<double,double,int>(1,2,3); // OK: 1 и 2 преобразуются в double
}
```

Интересно, что пакет может быть частично указан явно, а частично — выведен:

```
template<typename ... Ts> void f(Ts ... ps);
int main()
{
    f<double,int>(1,2,3); // OK: аргументы шаблона - <double,int,int>
}
```

15.10. Вывод из инициализаторов и выражений

Стандарт C++11 включает возможность объявления переменной, тип которой выводится из ее инициализатора. Он также обеспечивает механизм для выражения типа именованной сущности (переменной или функции) или выражения. Эти возможности оказались очень удобными, и в стандартах C++14 и C++17 были внесены новые дополнения на эту тему.

15.10.1. Спецификатор типа `auto`

Спецификатор типа `auto` может использоваться в ряде мест (главным образом, в областях видимости пространства имен и локальных) для вывода типа переменной из ее инициализатора. В таких случаях `auto` называется *типов-заместителем* (placeholder type). Еще один тип-заместитель, `decltype(auto)`, будет описан чуть позже в разделе 15.10.2. Например:

```
template<typename Container>
void useContainer(Container const& container)
{
    auto pos = container.begin();

    while (pos != container.end())
    {
        auto& element = *pos++;
        ... // Операции над элементом
    }
}
```

Эти два применения `auto` в приведенном примере устраниют необходимость записывать два длинных и потенциально сложных типа, — тип итератора контейнера и тип значения итератора:

```
typename Container::iterator pos = container.begin();
...
typename std::iterator_traits<typename Container::iterator>::reference
element = *pos++;
```

Вывод для `auto` использует тот же механизм, что и вывод аргумента шаблона. Спецификатор типа `auto` заменяется параметром типа `T` шаблона, после чего вывод выполняется так, как если бы переменная была параметром функции, а ее инициализатор — соответствующим аргументом функции. В первом примере `auto` это соответствует следующей ситуации:

```
template<typename T> void deducePos(T pos);
deducePos(container.begin());
```

где `T` — тип, который выводится для `auto`. Одним из непосредственных последствий этого является то, что переменная типа `auto` никогда не будет ссылочным типом. Использование `auto&` во втором примере иллюстрирует, как получить ссылку на выведенный тип. Его вывод эквивалентен следующему шаблону функции и вызову:

```
template<typename T> deduceElement(T& element);
deduceElement(*pos++);
```

Здесь `element` всегда имеет ссылочный тип, и его инициализатор не может производить временное значение.

Можно также скомбинировать `auto` со ссылкой на `r`-значение, но это приведет к поведению, похожему на поведение *передаваемой ссылки*, поскольку модель вывода для

```
auto&& fr = ...;
```

основана на шаблоне функции:

```
template<typename t> // auto заменяется
void f(T&& fr); // параметром шаблона T
```

Это поясняет следующий пример:

```
int x;
auto&& rr = 42; // OK: ссылка на r-значение связана
                 // с r-значением (auto = int)
auto&& lr = x; // Também OK: auto = int&, и свертка ссылок
                 // делает lr ссылкой на l-значение
```

Этот метод часто используется в обобщенном коде для привязки результата вызова функции или оператора, категория значения которого (`l`- или `r`-значение) не известна, без копирования этого результата. Например, она часто является предпочтительным способом объявить итерируемое значение в цикле `for` для диапазона:

```
template<typename Container> void g(Container c)
{
    for (auto && x : c)
    {
        ***
    }
}
```

Здесь мы не знаем сигнатуры итерирующих интерфейсов контейнера, но с помощью `auto &&` мы можем быть уверены, что никаких дополнительных копий итерируемых значений создано не будет. Как обычно, можно использовать `std::forward<T>()`, если требуется прямая передача связанного значения. Это обеспечивает “отложенную” прямую передачу (см. пример в разделе 11.3).

В дополнение к ссылкам можно комбинировать спецификатор `auto` с другими для получения константной переменной, указателя, указателя на член и так далее, но `auto` должен быть “главным” спецификатором объявления. Он не может быть вложенным в аргумент шаблона или частью декларатора, за которым следует спецификатор типа. Следующий пример иллюстрирует различные возможности:

```
template<typename T> struct X
{
    T const m;
};

auto const N = 400u;      // OK: константа типа unsigned int
auto* gp =
    (void*)nullptr;      // OK: gp имеет тип void*
auto const S::* pm =
    &X<int>::m;          // OK: pm имеет тип int const X<int>::*
X<auto> xa = X<int>(); // Ошибка: auto в аргументе шаблона
int const auto::* pm2 =
    &X<int>::m;          // Ошибка: auto является частью "декларатора"
```

Нет никаких технических причин, по которым C++ не мог бы поддерживать все случаи в последнем примере, но Комитет по стандартизации C++ считает, что получаемые выгоды перевешивались бы стоимостью дополнительных реализаций и потенциальных злоупотреблений.

Во избежание путаницы как для программиста, так и для компилятора, старое применение `auto` как “спецификатора класса хранения” в C++11 (и более поздних стандартах) запрещено:

```
int g()
{
    auto int r = 24; // OK в C++03, ошибка в C++11
    return r;
}
```

Это старое применение `auto` (унаследованное от C) всегда было излишним. Большинство компиляторов могут (но не должны) разрешить неоднозначность и определить, используется ли `auto` по-новому, как тип-заместитель, и предложить вариант перехода от старого кода C++ к новому. Однако на практике старое применение `auto` встречается крайне редко.

Выводимые возвращаемые типы

C++14 добавил еще одну ситуацию, где может встретиться выводимый тип-заместитель `auto`: типы возвращаемых значений функций. Например:

```
auto f() { return 42; }
```

Этот код определяет функцию с возвращаемым типом `int` (тип значения 42). Это можно также выразить с помощью синтаксиса *завершающего возвращаемого типа* (*trailing return type*):

```
auto f() -> auto { return 42; }
```

В этом последнем случае первый `auto` завершающий возвращаемый тип, а второй `auto` является выводимым типом-заместителем. Однако имеется мало доводов в пользу более многословного синтаксиса.

Тот же механизм по умолчанию используется в лямбда-выражениях: если тип возвращаемого значения не указан явным образом, то возвращаемый тип лямбда-выражения выводится так, как будто это `auto`¹⁰:

```
auto lm = [](int x)
{
    return f(x);
};  
// То же, что и: [] (int x) -> auto { return f(x); };
```

Функции могут быть объявлены отдельно от их определений. Это верно и для функций, возвращаемый тип которых выводится:

```
auto f(); // Предварительное объявление
auto f()
{
    return 42;
}
```

Однако предварительное объявление в подобном случае имеет весьма ограниченное применение, так как в любой точке, где используется функция, должно быть видимым ее определение. Возможно, это покажется удивительным, но предоставление предварительного объявления с “разрешенным” возвращаемым типом некорректно. Например:

```
int known();
auto known()
{
    return 42;      // Ошибка: несовместимые возвращаемые типы
}
```

Главным образом возможность предварительного объявления функции с выводимым возвращаемым типом полезна только для того, чтобы иметь возможность

¹⁰ Хотя C++14 вводит выводимые возвращаемые типы в целом, они уже были доступны для лямбда-выражений C++11 с использованием спецификации, которая не была сформулирована в терминах вывода. В C++14 эта спецификация была обновлена с использованием общего механизма вывода `auto` (с точки зрения программиста, никакой разницы между ними нет).

переместить определение функции-члена вне определения класса из-за стилистических предпочтений:

```
struct S
{
    auto f(); // Определение находится после определения класса
};

auto S::f()
{
    return 42;
}
```

Выводимые параметры, не являющиеся типами

До C++17 аргументы шаблонов, не являющиеся типами, должны были объявляться с указанием конкретного типа. Однако этот тип может быть типом параметра шаблона, например:

```
template<typename T, T V> struct S;
S<int, 42>* ps;
```

В этом примере необходимость указывать тип аргумента шаблона, не являющегося типом (т.е. указывать `int` в дополнение к `42`), может быть утомительной. Поэтому стандарт C++17 добавил возможность объявления параметров шаблонов, не являющихся типами, фактические типы которых выводятся из соответствующих аргументов шаблонов. Они объявляются следующим образом:

```
template<auto V> struct S;
```

что позволяет записать

```
S<42>* ps;
```

Здесь тип `V` для `S<42>` выводится как `int`, потому что `42` имеет тип `int`. Если мы напишем вместо этого `S<42u>`, тип `V` будет выведен как `unsigned int` (см. подробности выведения спецификатора типа `auto` в разделе 15.10.1).

Обратите внимание на то, что общие ограничения на тип параметров шаблонов, не являющихся типами, остаются в силе. Например:

```
S<3.14>* pd; // Ошибка: аргумент, не являющийся типом,
               // представляет собой число с плавающей точкой
```

Определение шаблона с такого рода выводимыми параметрами, не являющимися типами, часто требует выражения фактического типа соответствующего аргумента. Это легко сделать с помощью конструкции `decltype` (см. раздел 15.10.2). Например:

```
template<auto V> struct Value
{
    using ArgType = decltype(V);
};
```

`auto`-параметры шаблона, не являющиеся типами, полезны также для параметризации шаблонов членов или классов. Например:

```

template<typename> struct PMClassT;
template<typename C, typename M> struct PMClassT<M C::*>
{
    using Type = C;
};
template<typename PM> using PMClass = typename PMClassT<PM>::Type;

template<auto PMD> struct CounterHandle
{
    PMClass<decltype(PMD)>& c;
    CounterHandle(PMClass<decltype(PMD)>& c) : c(c)
    {
    }
    void incr()
    {
        ++(c.*PMD);
    }
};

struct S
{
    int i;
};

int main()
{
    S s{41};
    CounterHandle<&S::i> h(s);
    h.incr(); // Увеличивает s.i
}

```

Здесь мы использовали вспомогательный шаблон класса `PMClassT` для извлечения из типа указателя на член тип “родительского” класса, с помощью частичной специализации шаблона класса¹¹ (описан в разделе 16.4). С параметром шаблона `auto` нам нужно только задать константный указатель на член `&S::i` в качестве аргумента шаблона. До C++17 мы должны были также указывать тип указателя на член; т.е. писать что-то вроде

`OldCounterHandle<int S::*, &S::i>`

что, конечно, и громоздко и излишне.

Как вы и ожидали, эта возможность может также использоваться для пакетов параметров, не являющихся типами:

```

template<auto... VS> struct Values
{
};
Values<1, 2, 3> beginning;
Values<1, 'x', nullptr> triplet;

```

Пример `triplet` показывает, что каждый элемент пакета параметров, не являющихся типами, может быть выведен как отдельный тип. В отличие от случая нескольких деклараторов переменных (см. раздел 15.10.4) здесь нет требования, чтобы все выводы были эквивалентными.

¹¹ Этот же способ может использоваться для извлечения связанного типа члена: вместо `using Type = C;` используется `using Type=M;`.

Если мы хотим обеспечить однородный пакет параметров, не являющихся типами, это также возможно:

```
template<auto V1, decltype(V1)... VRest> struct HomogeneousValues
{
};
```

Однако в данном конкретном случае список аргументов шаблона не может быть пустым.

Завершенный пример использования `auto` в качестве параметра типа шаблона имеется в разделе 3.4.

15.10.2. Запись типа выражения с помощью `decltype`

Ключевое слово `auto` позволяет избежать необходимости писать тип переменной, но не позволяет легко использовать тип этой переменной. Эту проблему решает ключевое слово `decltype`: оно позволяет программисту выразить точный тип выражения или объявления. Однако программисты должны быть осторожными и понимать тонкие различия между тем, что дает `decltype`, в зависимости от того, является переданный аргумент объявлением сущностью или выражением.

- Если `e` является *именем сущности* (такой как переменная, функция, перечислитель или член-данные) или обращением к члену класса, `decltype(e)` дает *объявленный тип* этой сущности или упомянутого члена класса. Таким образом, `decltype` может использоваться для изучения типа переменной. Это полезно, когда нам нужно точное соответствие типу существующего объявления. Рассмотрим, например, следующие переменные `y1` и `y2`:

```
auto x = ...;
auto y1 = x + 1;
decltype(x) y2 = x + 1;
```

В зависимости от инициализатора `x`, `y1` может иметь тот же тип, что и `x` (но может и не иметь): это зависит от поведения `+`. Если переменная `x` была выведена как `int`, переменная `y1` также должна быть типа `int`. Если `x` была выведена как `char`, то `y1` должна быть `int`, поскольку сумма `char` и `1` (значение `1` имеет тип `int` по определению) имеет тип `int`. Использование `decltype(x)` в качестве типа `y2` гарантирует, что эта переменная всегда будет иметь тот же тип, что и переменная `x`.

- В противном случае, если `e` является некоторым другим выражением, `decltype(e)` дает тип, отражающий *тип и категорию значения* этого выражения следующим образом.
 - Если `e` представляет собой `l`-значение типа `T`, `decltype(e)` дает тип `T&`.
 - Если `e` представляет собой `x`-значение типа `T`, `decltype(e)` дает тип `T&&`.
 - Если `e` представляет собой `rg`-значение типа `T`, `decltype(e)` дает тип `T`.

О категориях значений рассказывается в приложении Б, “Категории значений”.

Указанные различия могут быть продемонстрированы следующим примером:

```
void g(std::string&& s)
{
    // Проверка типа s:
    std::is_lvalue_reference<decltype(s)>::value;           // false
    std::is_rvalue_reference<decltype(s)>::value;           // true
                                                        // (s как объявлено)
    std::is_same<decltype(s), std::string&>::value;         // false
    std::is_same<decltype(s), std::string&&>::value;        // true

    // Проверка категории значения s, использованного как выражение:
    std::is_lvalue_reference<decltype((s))>::value;          // true
                                                                // (s - l-значение)
    std::is_rvalue_reference<decltype((s))>::value;          // false
    std::is_same<decltype((s)), std::string&>::value;        // true
                                                                // (T& говорит об l-значении)
    std::is_same<decltype((s)), std::string&&>::value;       // false
}
```

В первых четырех выражениях `decltype` применяется к переменной `s`:

```
decltype(s) // Объявленный тип сущности e определяется s
```

Это означает, что `decltype` производит объявленный тип `s`, т.е. `std::string &&`. В последних четырех выражениях операнд конструкции `decltype` является не просто *именем*, потому что в каждом случае выражение представляет собой `(s)`, которое является именем в скобках. В этом случае тип будет отражать категорию значения `(s)`:

```
decltype((s)) // Проверка категории значения (s)
```

Наше выражение ссылается на переменную по имени `i`, таким образом, является l-значением¹²: согласно приведенным выше правилам, это означает, что `decltype(s)` является обычной ссылкой (т.е. ссылкой на l-значение) на `std::string` (поскольку тип `(s)` – `std::string`). Это одно из немногих мест в C++, где скобки меняют смысл программы (помимо воздействия на порядок вычислений операторов).

Тот факт, что `decltype` вычисляет тип произвольного выражения `e`, может оказаться полезным в различных местах. В частности, `decltype(e)` сохраняет достаточно информации о выражении, чтобы сделать возможным точно описать тип возвращаемого значения функции, которая возвращает это выражение `e`: `decltype` вычисляет тип этого выражения, а также передает категорию значения выражения вызывающему данную функцию коду. Рассмотрим простую передающую функцию `g()`, которая возвращает результат вызова `f()`:

```
??? f();
decltype(f()) g()

{ return f(); }
```

¹² Как уже отмечалось, трактовка параметра типа ссылки на l-значение как l-значения, а не x-значения, предусмотрена с целью безопасности, потому что на нечто с именем (наподобие параметра) можно легко сослаться в функции несколько раз. Если бы это было x-значение, его первого использования могло хватить для того, чтобы оно было “потеряно”, что привело бы к сюрпризам при последующих использованиях этого значения. См. разделы 6.1 и 15.6.3.

Тип возвращаемого значения `g()` зависит от возвращаемого типа `f()`. Если `f()` будет возвращать `int &`, вычисление типа возвращаемого значения `g()` сначала определит, что выражение `f()` имеет тип `int`. Это выражение является `l`-значением, поскольку `f()` возвращает ссылку на `l`-значение, поэтому объявленным возвращаемым типом `g()` становится `int &`. Аналогично, если тип возвращаемого значения `f()` представляет собой ссылку на `g`-значение, то вызов `f()` будет `x`-значением, и `decltype` будет давать тип ссылки на `g`-значение, который точно соответствует типу, возвращаемому `f()`. По существу, эта форма `decltype` принимает основные характеристики произвольного выражения — его тип и категорию значения — и кодирует их в системе типов таким образом, чтобы обеспечить корректную прямую передачу возвращаемого значения.

`decltype` также может быть полезным, когда вывода `auto` оказывается недостаточно. Например, предположим, что у нас есть переменная `pos` некоторого неизвестного типа итератора, и мы хотим создать переменную `element`, которая ссылается на элемент, указываемый `pos`. Мы могли бы написать

```
auto element = *pos;
```

Однако таким образом всегда будет создаваться копия элемента. Если же мы попытаемся написать

```
auto& element = *pos;
```

то мы всегда будем получать ссылку на элемент, но программа завершится ошибкой, если `operator*` итератора возвращает значение, а не ссылку¹³. Для решения этой проблемы можно использовать `decltype`, так что “значимость” или “ссылочность” `operator*` итератора сохраняется:

```
decltype(*pos) element = *pos;
```

Этот код будет использовать ссылку, когда итератор поддерживает ее, и копировать значение в противном случае. Его основной недостаток в том, что он требует записи выражения инициализатора дважды: один раз в `decltype` (где выражение не вычисляется) и один раз в качестве фактического инициализатора. C++14 для решения этого вопроса вводит конструкцию `decltype(auto)` (которую мы рассмотрим ниже).

15.10.3. `decltype(auto)`

Стандарт C++14 добавляет функциональную возможность, которая представляет собой сочетание `auto` и `decltype`: `decltype(auto)`. Подобно спецификатору типа `auto`, это *тип-заместитель*, и тип переменной, возвращаемый тип или аргумент шаблона определяется из типа связанного выражения (инициализатора, возвращаемого значения или аргумента шаблона). Однако в отличие от простого

¹³ Когда мы использовали такую запись в нашем вступительном примере с `auto`, мы неявно предполагали, что итераторы дают ссылки на некоторое хранилище. Хотя это в целом верно для итераторов контейнеров (и является требованием для всех стандартных контейнеров, кроме `vector<bool>`), это справедливо не для всех итераторов.

`auto`, который использует правила вывода аргумента шаблона для определения интересующего типа, фактический тип определяется путем применения конструкции `decltype` непосредственно к выражению. Приведенный далее пример иллюстрирует это:

```
int i = 42;           // i имеет тип int
int const& ref = i;   // ref имеет тип int const& и ссылается на i
auto x = ref;         // x имеет тип int и представляет собой
                      // новый независимый объект
decltype(auto) y = ref; // y имеет тип int const&
                      // и также ссылается на i
```

Тип `y` получается путем применения `decltype` к выражению инициализатора, здесь — `ref`, которая имеет тип `int const&`. Правила же вывода типа для `auto` дают тип `int`.

Еще один пример демонстрирует разницу при индексации `std::vector` (дающем l-значение):

```
std::vector<int> v = { 42 };
auto x = v[0];           // x – новый объект типа int
decltype(auto) y = v[0]; // y – ссылка (типа int&)
```

Эта конструкция аккуратно устраниет избыточность в нашем предыдущем примере:

```
decltype(*pos) element = *pos;
```

Теперь его можно переписать как

```
decltype(auto) element = *pos;
```

Ее часто удобно использовать и для возвращаемых типов. Рассмотрим следующий пример:

```
template<typename C> class Adapt
{
    C container;
    ***
    decltype(auto) operator[](std::size_t idx)
    {
        return container[idx];
    }
};
```

Если `container[idx]` дает l-значение, мы хотим передать его вызывающему коду как l-значение (кто-то может захотеть получить его адрес или изменить его). Это требует тип ссылки на l-значение, в который разрешается конструкция `decltype(auto)`. Если же вместо этого получается rg-значение, ссылочный тип приведет к висячим ссылкам, — но, к счастью, для этого случая `decltype(auto)` будет давать тип объекта (а не ссылочный тип).

В отличие от `auto`, `decltype(auto)` не допускает спецификаторы или операторы, которые модифицируют выводимый тип. Например:

```
decltype(auto)* p = (void*)nullptr; // Неверно
int const N = 100;
decltype(auto) const NN = N * N; // Неверно
```

Обратите также внимание на то, что скобки в инициализаторе могут быть значимыми (поскольку они являются существенными для конструкции `decltype`, как описано в разделе 6.1):

```
int x;
decltype(auto) z = x; // Объект типа int
decltype(auto) r = (x); // Ссылка типа int&
```

Это, в частности, означает, что скобки могут иметь серьезное влияние на корректность операторов `return`:

```
int g();
...
decltype(auto) f()
{
    int r = g();
    return (r); // Ошибка времени выполнения: возврат
} // ссылки на временную переменную
```

Начиная с C++17, `decltype(auto)` может также использоваться для выводимых параметров, не являющихся типами (см. раздел 15.10.1). Это проиллюстрировано в приведенном ниже примере:

```
template<decltype(auto) Val> class S
{
    ...
};

constexpr int c = 42;
extern int v = 42;
S<c> sc; // #1 создает S<42>
S<(v)> sv; // #2 создает S<(int&)v>
```

В строке #1 отсутствие скобок вокруг `c` приводит к тому, что выводимый параметр имеет тип самого `c` (т.е. `int`). Поскольку `c` является константным выражением со значением 42, это эквивалентно `S<42>`. В строке #2 скобки приводят к тому, что `decltype(auto)` становится ссылкой типа `int&`, которую можно связать с глобальной переменной `v` типа `int`. Следовательно, при таком объявлении шаблон класса зависит от ссылки на `v`, и любые изменения значения `v` могут повлиять на поведение класса `S` (см. подробности в разделе 11.4). (`S<v>` без скобок будет ошибкой, поскольку `decltype(v)` имеет тип `int`, и поэтому ожидается константный аргумент типа `int`. Однако `v` не имеет константного значения типа `int`.)

Обратите внимание на то, что природа этих двух случаев несколько отличается. Мы считаем, что такие параметры шаблона, не являющиеся типами, могут привести к сюрпризам в поведении, и не думаем, что они будут широко использоваться.

Наконец, небольшой комментарий об использовании выводимых параметров, не являющихся типами, в шаблонах функций:

```
template<auto N> struct S {};
template<auto N> int f(S<N> p);
S<42> x;
int r = f(x);
```

В этом примере тип параметра N шаблона функции `f<>()` выводится из типа параметра S, не являющегося типом. Это возможно, поскольку имя вида `X<...>`, где X является шаблоном класса, представляет собой выводимый контекст. Однако есть также много схем, которые не могут быть выведены таким образом:

```
template<auto V> int deduce decltype(V) p;
int rl = deduce<42>(42); // OK
int r2 = deduce(42);      // Ошибка: decltype(V) является
                           // невыводимым контекстом
```

В этом случае `decltype(V)` является невыводимым контекстом: отсутствует уникальное значение V, которое соответствует аргументу 42 (например, `decltype(7)` производит тот же тип, что и `decltype(42)`). Таким образом, чтобы иметь возможность вызывать эту функцию, следует явно указать параметр шаблона, не являющийся типом.

15.10.4. Особые случаи вывода `auto`

Существует несколько особых ситуаций для (простых в прочих случаях) правил вывода `auto`. Во-первых, когда инициализатор переменной представляет собой список инициализаторов. Соответствующий вывод для вызова функции может потерпеть неудачу, потому что мы не можем вывести параметр типа шаблона из аргумента, который представляет собой список инициализаторов:

```
template<typename T>
void deduceT(T);
...
deduceT({ 2, 3, 4}); // Ошибка
deduceT({ 1 });      // Ошибка
```

Однако если наша функция имеет более точно определенный параметр, как здесь:

```
template<typename T>
void deduceInitList(std::initializer_list<T>);
...
deduceInitList({ 2, 3, 5, 7 }); // OK: T выводится как int
```

тот вывод оказывается успешным. Следовательно, переменная `auto`, инициализированная копированием (т.е. с помощью инициализации с использованием `=`) со списком инициализации, определяется в терминах этого более точного параметра:

```
auto primes = { 2, 3, 5, 7 }; // primes имеет тип
                             // std::initializer_list<int>
deduceT(primes);           // T выводится как
                           // std::initializer_list<int>
```

До C++17 соответствующая непосредственная инициализация переменных `auto` (т.е. без `=`) обрабатывалась таким же способом, но в C++17 это было изменено для лучшего соответствия поведению, ожидаемому большинством программистов:

```
auto oops { 0, 8, 15 }; // Ошибка в C++17
auto val { 2 };        // OK: val имеет тип int в C++17
```

До C++17 обе инициализации были корректны, и обе инициализировали и `oops`, и `val` как переменные типа `initializer_list<int>`.

Интересно, что возврат списка инициализаторов в фигурных скобках для функции с выводимым типом-заместителем недопустим:

```
auto subtleError()
{
    return { 1, 2, 3 }; // Ошибка
}
```

Дело в том, что список инициализаторов в области видимости функции является объектом, который указывает на лежащий в его основе объект массива (со значениями элементов, указанными в списке), и который исчезает, когда функция возвращает значение. Допуск такой конструкции будет способствовать образованию висячих ссылок.

Во-вторых, особая ситуация возникает, когда несколько объявлений переменных разделяют одно и то же ключевое слово `auto`, как показано ниже:

```
auto first = container.begin(), last = container.end();
```

В таких случаях вывод осуществляется независимо для каждого объявления. Другими словами, имеется найденный для `first` параметр типа шаблона `T1`, и другой параметр типа шаблона `T2` для `last`. Только если оба вывода успешны, и выводы `T1` и `T2` дают одинаковый тип, эти объявления являются корректными. Это может приводить к некоторым интересным ситуациям¹⁴:

```
char c;
auto *cp = &c, d = c; // OK
auto e = c, f = c + 1; // Ошибка: несоответствие выводов char и int
```

Здесь две пары переменных объявляются с помощью общего спецификатора `auto`. Объявления `cp` и `d` выводят для `auto` один и тот же тип `char`, так что это корректный код. Однако объявления `e` и `f` приводят к выводу `char` и `int` из-за повышения к `int` при вычислении `c+1`, и это несоответствие приводит к ошибке.

В чем-то схожая особая ситуация может также возникнуть с заместителями для выводимых возвращаемых типов. Рассмотрим следующий пример:

```
auto f(bool b)
{
    if (b)
    {
        return 42.0; // Выводит возвращаемый тип double
    }
    else
    {
        return 0;     // Ошибка: конфликт вывода
    }
}
```

¹⁴ В этом примере не используется наш обычный стиль размещения `*` непосредственно прилегающим к `auto`, потому что он может ввести читателя в заблуждение, будто мы объявляем два указателя. С другой стороны, непрозрачность этих объявлений является хорошим аргументом в пользу консервативности при объявлении нескольких сущностей в одном объявлении.

В этом случае вывод для каждого оператора `return` выполняется отдельно, но если выводятся различные типы, то программа является некорректной. Если возвращаемое выражение рекурсивно вызывает ту же функцию, выполнение вывода невозможно, и программа является некорректной — если только возвращаемый тип не был ранее определен с помощью другого вывода. Это означает, что следующий код некорректен:

```
auto f(int n)
{
    if (n > 1)
    {
        return n * f(n - 1); // Ошибка: тип f(n-1) неизвестен
    }
    else
    {
        return 1;
    }
}
```

Однако следующий эквивалентный код вполне работоспособен:

```
auto f(int n)
{
    if (n <= 1)
    {
        return 1;           // Возвращаемый тип выводится как int
    }
    else
    {
        return n*f(n-1); // OK: тип f(n-1) является int, и
                           // таковым же является тип n*f(n-1)
    }
}
```

Выводимые возвращаемые типы имеют еще один особый случай, когда отсутствуют выводимые типы переменных или выводимые типы параметров, не являющихся типами:

```
auto f1() {} // OK: возвращаемый тип - void
auto f2()
{
    return;     // OK: возвращаемый тип - void
}
```

И `f1()`, и `f2()` корректны и имеют возвращаемый тип `void`. Однако если схема возвращаемого типа не может соответствовать `void`, такие случаи являются некорректными:

```
auto* f3() {} // Ошибка: auto* не может быть выведено как void
```

Как ожидается, любое использование шаблона функции с выводимым типом возвращаемого значения требует немедленного инстанцирования этого шаблона для уверенного определения возвращаемого типа. Что, однако, имеет удивительное следствие, когда дело доходит до SFINAE (принцип SFINAE описан в разделах 8.4 и 15.7). Рассмотрим следующий пример:

deduce/resulttypetmpl.cpp

```
template<typename T, typename U>
auto addA(T t, U u) -> decltype(t + u)
{
    return t + u;
}

void addA(...);

template<typename T, typename U>
auto addB(T t, U u) -> decltype(auto)
{
    return t + u;
}

void addB(...);
struct X
{
};

// OK: AddResultA представляет собой void
using AddResultA = decltype(addA(X(),X()));

// Ошибка: инстанцирование addB<X> некорректно
using AddResultB = decltype(addB(X(),X()));
```

Здесь использование `decltype(auto)` вместо `decltype(t+u)` для `addB()` вызывает ошибку в процессе разрешения перегрузки: тело функции шаблона `addB()` для определения ее возвращаемого типа должно быть полностью инстанцировано. Это инстанцирование не находится в непосредственном контексте (см. раздел 15.7.1) вызова `addB()` и поэтому не подпадает под фильтр SFINAE, а приводит к ошибке. Поэтому важно помнить, что *выводимые возвращаемые типы* не являются просто сокращением для сложного явно определенного возвращаемого типа, и должны использоваться с осторожностью (т.е. с пониманием, что они не должны вызываться в сигнатурах других шаблонов функций, чтобы можно было рассчитывать на свойства SFINAE).

15.10.5. Структурированное связывание

C++17 добавляет новую возможность, известную как *структурированное связывание* (structured bindings)¹⁵. Наиболее просто понять, о чем идет речь, с помощью маленького примера:

```
struct MaybeInt
{
    bool valid;
    int value;
};
```

¹⁵ Термин *структурированное связывание* был использован в первоначальном предложении этой возможности и в конечном итоге оказался в формальной спецификации языка. Однако некоторое время спецификация использовала вместо него термин *объявление разложения* (decomposition declaration).

```
MaybeInt g();
auto const& [b,N] = g(); // Связывает b и N с членами результата g()
```

Вызов `g()` производит значение (в данном случае — простой класс-агрегат типа `MaybeInt`), которое можно разложить на “элементы” (в данном случае — члены данных `MaybeInt`). Значение этого вызова производится как если бы список идентификаторов в квадратных скобках `[b, N]` был заменен уникальным именем переменной. Если это имя `e`, то указанная инициализация эквивалентна следующей:

```
auto const& e = g();
```

Идентификаторы в квадратных скобках затем связываются с элементами `e`. Таким образом, вы можете думать о `[b, N]` как введении имен для частей `e` (некоторые детали этого связывания мы будем обсуждать позже).

Синтаксически структурированное связывание должно всегда иметь тип `auto` с необязательными квалификаторами `const` и/или `volatile` и/или операторами деклараторов `&` и/или `&&` (но не декларатором указателя `*` или некоторыми другими). Далее следует список в квадратных скобках, содержащий по крайней мере один идентификатор (напоминает список захвата лямбда-выражения). Все это, в свою очередь, должно сопровождаться инициализатором.

Структурированное связывание можно инициализировать сущностями трех разных видов.

- Первый случай — простой тип класса, в котором все нестатические члены-данные открыты (как в примере выше). Чтобы этот случай был применим, все нестатические члены-данные должны быть открытыми (либо непосредственно в самом классе, либо в однозначном открытом базовом классе; анонимные объединения использовать не могут). В этом случае количество идентификаторов в квадратных скобках должно равняться числу членов, и использование одного из этих идентификаторов в области видимости структурированного связывания равносильно использованию соответствующего члена объекта, обозначенного `e` (со всеми связанными свойствами; например, если соответствующий член является битовым полем, невозможно получить его адрес).

- Второй случай соответствует *массивам*. Вот соответствующий пример:

```
int main()
{
    double pt[3];
    auto& [x, y, z] = pt;
    x = 3.0;
    y = 4.0;
    z = 0.0;
    plot(pt);
}
```

Неудивительно, что переменные в квадратных скобках просто представляют собой сокращения для соответствующих элементов неименованной

переменной массива. Количество элементов массива должно совпадать с количеством инициализаторов в квадратных скобках.

Вот другой пример:

```
auto f() -> int(&)[2]; // f() возвращает ссылку на массив int
auto [ x, y ] = f(); // #1
auto& [ r, s ] = f(); // #2
```

Строка #1 особенная: обычно сущность *e*, описанная ранее, выводится для этого случая из следующего кода:

```
auto e = f();
```

Однако так будет выведен низведенный указатель на массив, и это совсем не то, что происходит при выполнении структурированного связывания массива. Вместо этого *e* выводится как переменная типа массива, соответствующего типу инициализатора. Затем этот массив поэлементно *копируется* из инициализатора: для встроенных массивов это несколько необычная концепция¹⁶. Наконец, *x* и *y* становятся псевдонимами для выражений *e[0]* и *e[1]* соответственно.

Строка #2 не выполняет копирования массивов и следует обычным правилам для *auto*. Так что гипотетически *e* объявляется следующим образом:

```
auto& e = f();
```

Это дает нам ссылку на массив, и *x* и *y* вновь становятся псевдонимами для выражений *e[0]* and *e[1]* соответственно (которые представляют собой l-значения, ссылающиеся непосредственно на элементы массива, создаваемого вызовом *f()*).

3. И, наконец, третий случай позволяет *классам наподобие std::tuple* разложение на элементы с помощью протокола на основе шаблона с использованием *get<i>()*. Пусть *E* – тип выражения (*e*), в котором *e* объявлено, как указано выше. Поскольку *E* является типом выражения, оно никогда не является ссылочным типом. Если выражение *std::tuple_size<E>::value* является корректным целочисленным константным выражением, оно должно быть равно количеству идентификаторов в скобках (протокол отдает предпочтение этому варианту по сравнению с первым, но не со вторым вариантом для массивов). Давайте обозначим идентификаторы в квадратных скобках как *n₀*, *n₁*, *n₂* и т. д. Если *e* имеет любой член с именем *get*, то поведение получается таким, как если бы эти идентификаторы были объявлены как

```
std::tuple_element<i, E>::type& ni = e.get<i>();
```

если *e* выводится как имеющее ссылочный тип, или

```
std::tuple_element<i, E>::type&& ni = e.get<i>();
```

¹⁶Два других места, где происходит копирование встроенных массивов – это захват лямбда-выражением и генерируемый копирующий конструктор.

в противном случае. Если у *e* нет члена *get*, то соответствующие объявления приобретают вид

```
std::tuple_element<i, E>::type& ni = get<i>(e);
```

или

```
std::tuple_element<i, E>::type&& ni = get<i>(e);
```

где *get* ищется только в связанных классах и пространствах имен. (Во всех случаях предполагается, что *get* является шаблоном и, следовательно, за ним следует угловая скобка <.) Все шаблоны — *std::tuple*, *std::pair* и *std::array* — реализуют этот протокол, делая корректным, например, следующий код:

```
#include <tuple>
std::tuple<bool, int> bi {true, 42};
auto [b, i] = bi;
int r = i;      // Инициализация r значением 42
```

Однако несложно добавить специализации *std::tuple_size*, *std::tuple_element* и шаблон функции или функции-члена *get<>()*, которые сделают этот механизм работающим для произвольного типа класса или перечисления. Например:

```
#include <utility>

enum M {};

template<> class std::tuple_size<M>
{
public:
    static unsigned const value = 2; // Отображение M на пару значений
};

template<> class std::tuple_element<0, M>
{
public:
    using type = int;      // Первое значение будет иметь тип int
};

template<> class std::tuple_element<1, M>
{
public:
    using type = double; // Второе значение будет иметь тип double
};

template<int> auto get(M);
template<> auto get<0>(M)
{
    return 42;
}
template<> auto get<1>(M)
{
    return 7.0;
}

auto [i, d] = M(); // Как если бы int& i = 42; double& d = 7.0;
```

Учтите, что вам нужно включить заголовочный файл `<utility>`, чтобы использовать две вспомогательные функции `std::tuple_size<>` и `std::tuple_element<>`.

Кроме того, обратите внимание, что третий из показанных выше случаев (использующий протокол `tuple`) выполняет фактическую инициализацию инициализаторов в скобках, а связывания представляют собой фактические ссылочные переменные, а не просто псевдонимы для других выражений (в отличие от первых двух случаев, использующих простые типы классов и массивов). Это интересно, потому что инициализация ссылки может выполниться неправильно; например, она может сгенерировать исключение, и такое исключение теперь является неизбежным. Однако комитет по стандартизации C++ обсуждал также возможность не связывать идентификаторы с инициализированными ссылками, а вместо этого при каждом последующем использовании идентификаторов вычислять выражение `get<>()`. Это позволило бы использовать структурированное связывание с типами, в которых “первое” значение должно быть протестировано до обращения ко “второму” (например, основанными на `std::optional`).

15.10.6. Обобщенные лямбда-выражения

Лямбда-выражения быстро стали одной из наиболее популярных возможностей C++11, отчасти потому, что они значительно облегчили использование функциональных конструкций в стандартной библиотеке C++ (и многих других современных библиотеках C++) благодаря значительно сокращенному синтаксису. Однако в рамках самих шаблонов лямбда-выражения могут стать довольно громоздкими, что обусловлено необходимостью уточнения типов параметров и результатов. Рассмотрим, например, шаблон функции для поиска первого отрицательного значения в последовательности:

```
template<typename Iter>
Iter findNegative(Iter first, Iter last)
{
    return std::find_if(
        first, last,
        [](typename std::iterator_traits<Iter>::value_type value)
        { return value < 0; });
}
```

В этом шаблоне функции наиболее сложной частью лямбда-выражения является тип параметра. C++14 вводит понятие “обобщенных” лямбда-выражений, в которых один или несколько типов параметров используют ключевое слово `auto` для автоматического вывода типа вместо явного его указания:

```
template<typename Iter>
Iter findNegative(Iter first, Iter last)
{
    return std::find_if(first, last,
        [](auto value){ return value < 0; })
    );
}
```

`auto` в параметре лямбда-выражения обрабатывается аналогично `auto` в позиции типа переменной на основе типа инициализатора: это ключевое слово заменяется открытым параметром типа шаблона `T`. Однако, в отличие от случая переменной, здесь вывод не выполняется немедленно, потому что аргумент в момент создания лямбда-выражения неизвестен. Вместо этого само лямбда-выражение становится обобщенным (если оно еще не было таковым), и типовой параметр шаблона добавляется в список параметров шаблона. Таким образом, приведенное выше лямбда-выражение может быть вызвано с любым типом аргумента, лишь бы этот тип аргумента поддерживал операцию `< 0`, результат которой был преобразован в значение типа `bool`. Например, это лямбда-выражение может быть вызвано со значением типа `int` или `double`.

Чтобы понять, что для лямбда-выражения означает быть обобщенным, сначала рассмотрим модель реализации лямбда-выражений, не являющихся обобщенными. Данное лямбда-выражение

```
[](int i)
{
    return i < 0;
```

компилятор C++ транслирует в экземпляр вновь создаваемого специально для данного лямбда-выражения класса. Этот экземпляр называется *замыканием* (*closure*) или *объектом замыкания*, а класс называется *типов замыкания*. Тип замыкания имеет оператор вызова функции, и, следовательно, замыкание является функциональным объектом¹⁷. Для данного лямбда-выражения тип замыкания будет выглядеть примерно следующим образом (для краткости и простоты мы опустили преобразование функции в значение указателя на функцию):

```
class SomeCompilerSpecificNameX
{
public:
    SomeCompilerSpecificNameX() // Вызывается только компилятором
    bool operator()(int i) const
    {
        return i < 0;
    }
};
```

При проверке категории типа для лямбда-выражения `std::is_class<>` вернет `true` (см. раздел Г.2.1).

Лямбда-выражение, таким образом, оказывается объектом данного класса (типа замыкания). Например, код

¹⁷ Эта модель трансляции лямбда-выражений используется в спецификации языка C++, что делает ее удобным и точным описанием семантики. Захваченные переменные становятся членами-данными; преобразование незахватывающих лямбда-выражений в указатели на функции моделируется преобразованием функции в класс, и так далее. А поскольку лямбда-выражения являются функциональными объектами, то, если определены правила для функциональных объектов, они применимы и к лямбда-выражениям.

```
foo(...,
    [](int i)
{
    return i < 0;
});
```

создает объект (**замыкание**) внутреннего, специфичного для конкретного компилятора класса `SomeCompilerSpecificNameX`:

```
foo(...,
    SomeCompilerSpecificNameX{}); // Передача объекта типа замыкания
```

Если лямбда-выражение должно захватывать локальные переменные:

```
int x, y;
...
[x, y](int i)
{
    return i > x && i < y;
}
```

то эти захваты моделируются в виде инициализируемых членов соответствующего типа класса:

```
class SomeCompilerSpecificNameY
{
    private
        int _x, _y;
    public:
        SomeCompilerSpecificNameY(int x, int y) // Вызывается только
            : _x(x), _y(y) // компилятором
        {
        }
        bool operator()(int i) const
        {
            return i > _x && i < _y;
        }
};
```

В случае обобщенного лямбда-выражения оператор вызова функции становится шаблоном функции-члена, поэтому наше простое обобщенное лямбда-выражение

```
[](auto i)
{
    return i < 0;
}
```

преобразуется в следующий созданный компилятором класс (мы опять игнорируем функцию преобразования, которая в обобщенном лямбда-выражении становится **шаблоном функции**):

```
class SomeCompilerSpecificNameZ
{
    public:
        SomeCompilerSpecificNameZ(); // Вызывается только компилятором

        template<typename T>
        auto operator()(T i) const
```

```

    {
        return i < 0;
    }
};

```

Шаблон функции-члена инстанцируется при вызове замыкания, что обычно происходит не там, где появляется лямбда-выражение. Например:

```

#include <iostream>
template<typename F, typename... Ts> void invoke(F f, Ts... ps)
{
    f(ps...);
}

int main()
{
    invoke([](auto x, auto y)
    {
        std::cout << x + y << '\n'
    },
    21, 21);
}

```

Здесь лямбда-выражение появляется в функции `main()`, где и создается связанное замыкание. Однако оператор вызова замыкания при этом не инстанцируется. Вместо этого инстанцируется шаблон функции `invoke()` с типом замыкания в качестве первого параметра типа и типом `int` (тип значения 21) в качестве второго и третьего параметров типа. Это инстанцирование `invoke()` вызывается с копией замыкания (которое до сих пор остается замыканием, связанным с исходным лямбда-выражением), и инстанцирует шаблон `operator()` замыкания для удовлетворения инстанцированного вызова `f(ps...)`.

15.11. Шаблоны псевдонимов

Шаблоны псевдонимов (см. раздел 2.8) “прозрачны” по отношению к выводам. Это означает, что везде, где появляется шаблон псевдонима с некоторыми аргументами шаблона, выполняется подстановка аргументов в определение псевдонима (т.е. типа справа от `=`), и получающийся в результате шаблон используется для вывода. Например, вывод аргумента шаблона успешен в трех следующих вызовах:

`deduce/aliastemplate.cpp`

```

template<typename T, typename Cont>
class Stack;

template<typename T>
using DequeStack = Stack<T, std::deque<T>>;

template<typename T, typename Cont>
void f1(Stack<T, Cont>);

template<typename T>
void f2(DequeStack<T>);

```

```
template<typename T>
void f3(Stack<T, std::deque<T>); // Эквивалентно f2

void test(DequeStack<int> intStack)
{
    f1(intStack); // OK: Т выводится как int,
                  // Cont выводится как std::deque<int>
    f2(intStack); // OK: Т выводится как int
    f3(intStack); // OK: Т выводится как int
}
```

В первом вызове (`f1()`) использование шаблона псевдонима `DequeStack` в типе `intStack` на вывод влияния не оказывает: указанный тип `DequeStack<int>` рассматривается как подставляемый тип `Stack<int, std::deque<int>>`. Второй и третий вызовы имеют то же поведение вывода, поскольку `DequeStack<T>` в `f2()` и подставляемая форма `Stack<T, std::deque<T>>` в `f3()` эквивалентны. Для вывода аргументов шаблона шаблонов псевдонимов прозрачны: они могут использоваться для прояснения и упрощения кода, но при этом не влияют на то, как работает вывод.

Обратите внимание, что это возможно потому, что шаблоны псевдонимов не могут быть специализированы (см. в главе 16, “Специализация и перегрузка”, подробную информацию о специализации шаблонов). Предположим, что было бы возможно следующее:

```
template<typename T> using A = T;
template<> using A<int> = void; // Ошибка, но предположим,
                                // что это возможно...
```

тогда мы бы не могли проверить соответствие `A<T>` типу `void` и заключить, что тип `T` должен быть `void`, потому что и `A<int>`, и `A<void>` эквивалентны `void`. Тот факт, что это невозможно, гарантирует, что каждое использование псевдонима может быть обобщенно расширено согласно его определению, что и делает шаблоны псевдонимов прозрачными для вывода.

15.12. Вывод аргументов шаблонов классов

C++17 вводит новую разновидность вывода: вывод параметров шаблонов типа класса из аргументов, указанных в инициализаторе объявления переменной или преобразовании типа в стиле функции. Например:

```
template<typename T1, typename T2, typename T3 = T2>
class C
{
public:
    // Конструктор для 0, 1, 2 или 3 аргументов:
    C(T1 x = T1{}, T2 y = T2{}, T3 z = T3{});
    ...
};

C c1(22, 44.3, "hi"); // OK в C++17: T1 = int, T2 = double,
                      // T3 = char const*
```

```
C c2(22,44.3);           // OK в C++17: T1 = int, T2 = T3 = double
C c3("hi","guy");        // OK в C++17: T1 = T2 = T3 = char const*
C c4;                   // Ошибка: T1 и T2 не определены
C c5("hi");             // Ошибка: T2 не определен
```

Обратите внимание, что *все* параметры должны определяться либо путем процесса вывода, либо из аргументов по умолчанию. Невозможно явно указать только несколько аргументов и вывести другие. Например:

```
C<string> c10("hi","my",42);      // Ошибка: явно указан только T1,
                                  // T2 не выведен
C<> c11(22,44.3,42);            // Ошибка: ни T1, ни T2
                                  // явно не указаны
C<string,string> c12("hi","my"); // OK: T1 и T2 выводятся,
                                  // T3 имеет значение по умолчанию
```

15.12.1. Правила вывода

Рассмотрим сначала небольшое изменение в нашем приведенном ранее примере из раздела 15.8.2:

```
template<typename T>
class S
{
private:
    T a;
public:
    S(T b) : a(b)
    {
    }
};

template<typename T> S(T) -> S<T>; // Правило вывода

S x{12};           // OK с C++17, то же что и S<int> x{12};
S y{12};           // OK с C++17, то же что и S<int> y{12};
auto z = S{12};    // OK с C++17, то же что и auto z = S<int>{12};
```

Обратите внимание на новую шаблонообразную конструкцию, именуемую *правилом вывода* (deduction guide). Она выглядит подобно шаблону функции, но синтаксически отличается от шаблона функции несколькими моментами.

- Часть, которая выглядит подобно завершающему возвращаемому типу, не может быть записана как традиционный возвращаемый тип. Тип, который она обозначает (`S<T>` в нашем примере), мы называем *типовом, задаваемым правилом* (guided type).
- Нет ведущего ключевого слова `auto` для указания наличия завершающего возвращаемого типа.
- “Имя” правила вывода должно быть неквалифицированным именем шаблона класса, объявленного ранее в той же самой области видимости.
- Тип, задаваемый правилом, должен быть *идентификатором шаблона*, имя шаблона которого соответствует имени правила.
- Оно может быть объявлено со спецификатором `explicit`.

В объявлении `S x(12);` спецификатор `S` называется типом — заместителем класса (placeholder class type)¹⁸. При использовании такого заместителя имя объявляемой переменной должно следовать непосредственно за ним, а за именем, в свою очередь, должен следовать инициализатор. Так что следующий код некорректен:

```
S* p = &x; // Ошибка: неразрешенный синтаксис
```

При наличии правила, приведенного в нашем примере, объявление `S x(12);` выводит тип переменной, рассматривая правила вывода, связанные с классом `S`, как множество перегрузок, и пытаясь выполнить разрешение перегрузки с инициализатором с помощью этого множества. В нашем случае это множество состоит только из одного правила, и оно успешно выводит `T` как `int`, а *тип, задаваемый правилом*, как `S<int>`¹⁹. Поэтому этот тип выбирается как тип объявления.

Обратите внимание на то, что в случае нескольких деклараторов, следующих за именем шаблона класса, требующего вывода, инициализатор для каждого из этих деклараторов должен давать один и тот же тип. Например, для приведенных выше объявлений:

```
S s1(1), s2(2.0); // Ошибка: вывод S и как S<int>, и как S<double>
```

Это аналогично ограничению при выводе типа-заместителя C++11 `auto`.

В предыдущем примере имеется неявная связь между правилом вывода, которое мы объявили, и конструктором `S(T b)`, объявленным в классе `S`. Однако такая связь не является обязательной, что означает, что правила вывода могут использоваться с шаблонами классов-агрегатов:

```
template<typename T>
struct A
{
    T val;
};

template<typename T> A(T) -> A<T>; // Правило вывода
```

Без этого правила вывода мы должны всегда (даже в C++17) указывать явные аргументы шаблона:

```
A<int> a1{42};      // OK
A<int> a2(42);     // Ошибка: не агрегатная инициализация
A<int> a3 = {42};   // OK
A a4 = 42;          // Ошибка: невозможно вывести тип
```

¹⁸ Обратите внимание на разницу между *типом-заместителем*, который представляет собой `auto` или `decltype(auto)` и может разрешаться в тип любого вида, и *типом — заместителем класса*, который представляет собой имя шаблона и может разрешаться только в тип класса, который является экземпляром указанного шаблона.

¹⁹ Как и в случае обычного вывода шаблона функции, SFINAE может быть применен, если, например, подстановка выведенных аргументов в тип, задаваемый правилом, окажется неудачной. В нашем простом случае этого не происходит.

Но при наличии такого правила, как приведенное выше, можно записать:

```
A a4 = { 42 }; // OK
```

Однако в случаях, подобных этому, тонкостью является то, что инициализатор должен оставаться корректным инициализатором агрегата; т.е. он должен использовать список инициализации в фигурных скобках. Поэтому не допускаются следующие альтернативы:

```
A a5(42); // Ошибка: не агрегатная инициализация
A a6 = 42; // Ошибка: не агрегатная инициализация
```

15.12.2. Неявные правила вывода

Довольно часто правила вывода желательны для каждого конструктора в шаблоне класса. Это привело разработчиков вывода аргументов шаблонов классов к включению неявного механизма вывода, что эквивалентно представлению для каждого конструктора и шаблона конструктора²⁰ *неявного правила вывода* следующим образом.

- Список параметров шаблона для неявного правила состоит из параметров шаблона для шаблона класса, за которыми (в случае шаблона конструктора) следуют параметры шаблонов шаблона конструктора. Эти последние сохраняют любые аргументы по умолчанию.
- “Функциообразные” параметры правила копируются из конструктора или шаблона конструктора.
- Тип, задаваемый правилом, представляет собой имя шаблона с аргументами, которые представляют собой параметры шаблонов, взятые из шаблона класса.

Применим сказанное к нашему простому шаблону класса:

```
template<typename T>
class S
{
private:
    T a;
public:
    S(T b) : a(b)
    {
    }
};
```

Список параметров шаблона представляет собой `typename T`, “функциообразный” список параметров становится просто `(T b)`, а тип, задаваемый правилом, — `S<T>`. Таким образом, мы получаем правило, эквивалентное объявлению пользователем, которое мы писали ранее: следовательно, получается, что это правило

²⁰ В главе 16, “Специализация и перегрузка”, рассматривается возможность “специализировать” шаблоны классов различными способами. Такие специализации не участвуют в выводе аргументов шаблонов классов.

не требовалось для получения желаемого результата! Иными словами, при работе с такими простыми шаблонами классов, как рассмотренный выше (и без правил вывода) можно вполне корректно просто записать `S x(12)`; и получить ожидаемый результат — что `x` имеет тип `S<int>`.

Правила выводов содержат неоднозначность. Снова рассмотрим наш простой шаблон класса `S` и следующие инициализации:

```
S x{12}; // x имеет тип S<int>
S y{x};
S z(x);
```

Мы уже видели, что `x` имеет тип `S<int>`, но какими должны быть типы `y` и `z`? Интуитивно эти два типа должны быть `S<S<int>>` и `S<int>`. Комитет принял несколько спорное решение, что это должен быть тип `S<int>` в обоих случаях. Почему это спорно? Рассмотрим похожий пример с типом `vector`:

```
std::vector v{1, 2, 3}; // vector<int> - ничего удивительного
std::vector w2{v, v}; // vector<vector<int>>
std::vector w1{v}; // vector<int>!
```

Другими словами, инициализатор с одним элементом в фигурных скобках приводит к иному выводу, чем инициализатор с фигурными скобками с несколькими элементами. Часто результат в виде вектора с одним элементом является желательным, но непоследовательность оказывается тонким вопросом. В обобщенном коде эту тонкость легко упустить:

```
template<typename T, typename... Ts>
auto f(T p, Ts... ps)
{
    std::vector v{p, ps...}; // Тип зависит от длины пакета
    ...
}
```

Здесь легко забыть, что если `T` выводится как векторный тип, то тип `v` будет принципиально различен в зависимости от того, является ли `ps` пустым или не-пустым пакетом.

Добавление неявных правил само по себе несколько противоречиво. Основной аргумент против их включения заключается в том, что эта возможность автоматически добавляет интерфейсы к существующим библиотекам. Чтобы понять, о чём идет речь, рассмотрим еще раз наш простой шаблон класса `S`, приведенный выше. Его определение было корректным с момента ввода шаблонов в C++. Предположим, однако, что автор `S` расширяет библиотеку, что приводит к определению `S` более сложным способом:

```
template<typename T>
struct ValueArg
{
    using Type = T;
};
```

```
template<typename T>
class S
{
private:
    T a;
public:
    using ArgType = typename ValueArg<T>::Type;
    S(ArgType b) : a(b)
    {
    }
};
```

До C++17 подобные преобразования (мало распространенные) не влияли на существующий код. Однако в C++17 они отключают неявные правила вывода. Чтобы увидеть это, напишем правило вывода, соответствующее правилу, генерируемому процессом создания неявных правил вывода, описанным выше: список параметров шаблона и правило вывода остаются неизменными, но “функциообразный” параметр теперь записывается в терминах ArgType, который представляет собой не что иное, как typename ValueArg<T>::Type:

```
template<typename> S(typename ValueArg<T>::Type) -> S<T>;
```

Вспомним из раздела 15.2, что квалификатор имени наподобие ValueArg<T>:: не является выводимым контекстом. Так что правило вывода такого вида является бесполезным и не разрешает объявление наподобие S x (12);. Другими словами, разработчик библиотеки, выполнивший такое преобразование, нарушит работу клиентского кода в C++17.

Что же должен делать разработчик библиотеки в такой ситуации? Наш совет заключается в том, чтобы тщательно рассмотреть каждый конструктор — хотите ли вы предложить его в качестве источника для неявного правила вывода на все оставшееся время жизни библиотеки. Если нет, то замените каждый экземпляр выводимого параметра конструктора типа X на что-то вроде typename ValueArg<X>::Type. К сожалению, более простого способа избавиться от неявных правил вывода не имеется.

15.12.3. Прочие тонкости

Внедренные имена классов

Рассмотрим следующий пример:

```
template<typename T> struct X
{
    template<typename Iter> X(Iter b, Iter e);

    template<typename Iter> auto f(Iter b, Iter e)
    {
        return X(b, e); // Что это?
    }
};
```

Этот код корректен в C++14: `X` в `X(b, e)` представляет собой *внедренное имя класса* (injected class name), в данном контексте эквивалентное `X<T>` (см. раздел 13.2.3). Правила вывода аргументов шаблонов класса, однако, естественным образом делают `X` эквивалентным `X<Iter>`.

Тем не менее для обеспечения обратной совместимости вывод аргументов шаблона класса отключен, если имя шаблона представляет собой внедренное имя класса.

Передаваемые ссылки

Рассмотрим другой пример:

```
template<typename T> struct Y
{
    Y(T const&);
    Y(T&&);
};

void g(std::string s)
{
    Y y = s;
}
```

Очевидно, что здесь преследуется цель вывести `T` как `std::string` с помощью неявных правил вывода, связанных с копирующим конструктором. Написание неявных правил вывода как явно объявленных приводит к сюрпризу:

```
template<typename T> Y(T const&) -> Y<T>; // #1
template<typename T> Y(T&&) -> Y<T>; // #2
```

Вспомним из раздела 15.6, что `T&&` во время вывода аргументов шаблона ведет себя особым образом: будучи *передаваемой ссылкой* (forwarding reference), она приводит к тому, что `T` выводится как ссылочный тип, если соответствующий аргумент вызова представляет собой l-значение. В нашем примере выше в процессе вывода аргументом является выражение `s`, которое представляет собой l-значение. Неявное правило #1 выводит `T` как `std::string`, но требует, чтобы аргумент был исправлен с `std::string` на `std::string const`. Однако правило #2 обычно выводит `T` как ссылочный тип `std::string&` и генерирует параметр того же типа (из-за правила свертки ссылок), который является подходящим в большей степени, поскольку не требует добавления `const`.

Этот результат кажется несколько удивительным и скорее всего приведет к ошибке инстанцирования (при использовании параметра шаблона класса в контекстах, которые не допускают ссылочные типы) или, хуже того, к молчаливому неверному инстанцированию (приводящему, например, к висящим ссылкам).

Поэтому Комитет по стандартизации C++ пришел к решению удалить особое правило вывода для `T&&` при выполнении вывода для неявных правил вывода, если `T` изначально был параметром шаблона класса (в отличие от параметра шаблона конструктора; для них особое правило вывода остается в силе). Таким образом, приведенный выше пример выводит `T` как `std::string`, что и следовало ожидать.

Ключевое слово `explicit`

Правило вывода может быть объявлено с ключевым словом `explicit`. В таком случае оно рассматривается только при непосредственной, но не при копирующей инициализации. Например:

```
template<typename T, typename U> struct Z
{
    Z(T const&);
    Z(T&&);

};

template<typename T> Z(T const&) -> Z<T, T&>;           // #1
template<typename T> explicit Z(T&&) -> Z<T, T>;      // #2

Z z1 = 1; // Рассматривается только #1; аналог Z<int,int&>z1=1;
Z z2{2}; // Предпочтительно #2;           аналог Z<int,int> z2{2};
```

Обратите внимание на то, что инициализация `z1` является копирующей, а потому правило вывода #2 при этом не рассматривается, будучи объявлено как `explicit`.

Копирующее конструирование и списки инициализации

Рассмотрим следующий шаблон класса:

```
template<typename ... Ts> struct Tuple
{
    Tuple(Ts...);
    Tuple(Tuple<Ts...> const&);

};
```

Чтобы понять результат применения неявных правил, давайте запишем их как явные объявления:

```
template<typename... Ts> Tuple(Ts...) -> Tuple<Ts...>;
template<typename... Ts> Tuple(Tuple<Ts...> const&) -> Tuple<Ts...>;
```

Теперь рассмотрим несколько примеров:

```
auto x = Tuple{1,2};
```

Ясно, что здесь выбирается первое правило, а значит, и первый конструктор: `x`, таким образом, представляет собой `Tuple<int,int>`. Давайте продолжим и рассмотрим некоторые примеры, использующие синтаксис, который наводит на мысль о копировании `x`:

```
Tuple a = x;
Tuple b(x);
```

Оба правила походят и для `a`, и для `b`. Первое правило выбирает тип `Tuple <Tuple<int,int>>`, в то время как правило, связанное с копирующим

конструктором, дает `Tuple<int, int>`. К счастью, второе правило обеспечивает лучшее соответствие, а потому и `a`, и `b` конструируются копированием из `x`.

Рассмотрим теперь несколько правил с использованием списков инициализации в фигурных скобках:

```
Tuple c{x, x};
Tuple d{x};
```

В первом из этих примеров (`x`) может соответствовать только первому правилу, так что получается `Tuple< Tuple<int, int>, Tuple<int, int>>`. Это соответствует интуитивным представлениям и не является чем-то удивительным. Точно так же кажется, что во втором примере тип `d` должен быть выведен как `Tuple< Tuple<int>>`. Однако вместо этого второй пример рассматривается как копирующее конструирование (т.е. предпочтительнее оказывается второе неявное правило). То же самое происходит и при выполнении функционального приведения типов:

```
auto e = Tuple{x};
```

Здесь `e` выводится как `Tuple<int, int>`, а не как `Tuple< Tuple<int>>`.

Правила используются только для вывода

Правила вывода не предназначены для шаблонов функций — они используются только для вывода параметров шаблонов и не “вызываются”. Это означает, что разница между передачей аргументов по ссылке или по значению не важна для объявлений, управляемых правилами. Например:

```
template<typename T> struct X
{
    ***
};

template<typename T> struct Y
{
    Y(X<T> const&);
    Y(X<T>&&);
};

template<typename T> Y(X<T>) -> Y<T>;
```

Обратите внимание, что правило вывода не совсем соответствует двум конструкторам `Y`. Однако это не имеет значения, потому что правило используется только для вывода. Для данного значения `xtt` типа `X<TT>` — `l`- или `r`-значения — будет выбран выведенный тип `Y<TT>`. Затем инициализация выполнит разрешение перегрузки конструкторов `Y<TT>`, чтобы решить, какой из них следует вызвать (что будет зависеть от того, является `xtt` `l`- или `r`-значением).

15.13. Заключительные замечания

Вывод аргументов шаблонов для шаблонов функций был частью исходного дизайна C++. Фактически альтернатива, предоставляемая явными аргументами шаблона, так и не стала частью C++ за много лет.

SFINAE — термин, который появился в первом издании этой книги. Он быстро стал очень популярным у всего сообщества программистов на C++. Однако в C++98 принцип SFINAE был не столь мощным средством, как сейчас: он применялся только к ограниченному числу операций над типами и не охватывал произвольные выражения или управление доступом. По мере того как шаблонные технологии начали полагаться на SFINAE (см. раздел 19.4), стала очевидной необходимость обобщения условий SFINAE. Стив Адамчик (Steve Adamczyk) и Джон Спайсер (John Spicer) разработали формулировки, которые были приняты в C++11 (документ N2634). Хотя изменения в формулировках стандарта были относительно малы, усилия по их реализации в некоторых компиляторах оказались непропорционально велики.

Спецификатор типа `auto` и конструкция `decltype` были среди самых ранних дополнений в C++03, которые в конечном итоге появились в C++11. Их развитие возглавили Бъярне Страуструп (Bjarne Stroustrup) и Яакко Ярви (Jaakko Järví) (см. документы N1607 (`auto`) и N2343 (`decltype`)).

Страуструп рассматривал синтаксис `auto` еще в своей исходной реализации C++ (известной как Cfront). Когда эта функциональная возможность была добавлена в C++11, первоначальный смысл `auto` как спецификатора памяти (унаследованный от языка C) остался в языке, и было добавлено правило, которое решало, как именно следует интерпретировать это ключевое слово. При реализации этой функциональной возможности в компиляторе Edison Design Group Дэвид Вандевурд обнаружил, что это правило приводит к ряду неприятных сюрпризов для программистов C++11 (документ N2337). После изучения вопроса Комитет по стандартизации постановил отказаться от традиционного использования `auto` вообще (везде, где в программе C++03 используется ключевое слово `auto`, оно может быть безболезненно опущено) (см. документ N2546 Дэвида Вандевурда и Йенса Маурера (Jens Maurer)). Это был необычный прецедент удаления некоторой функциональной возможности из языка без первоначального перевода в состояние не рекомендованного к применению, но практика показала правильность этого решения.

В компиляторе GNU GCC было принято расширение `typeof`, по сути, не отличавшееся от `decltype`, и программисты нашли, что оно очень полезно в программировании шаблонов. К сожалению, это средство было разработано в контексте языка C и не совсем подходило для C++. Поэтому Комитет по стандартизации C++ не мог как включить его “как есть”, так и не мог его изменить, потому что это нарушило бы работу существующего кода, который использовал эту возможность GCC. Вот почему мы пишем `decltype`, а не `typeof`. Джейсон Меррилл (Jason Merrill) и другие привели серьезные аргументы в пользу того, что было бы предпочтительнее иметь собственные операторы вместо текущего

тонкого различия между `decltype(x)` и `decltype((x))`, но их сочли недостаточно убедительными, чтобы изменять окончательную спецификацию.

Возможность объявления параметров шаблона, не являющихся типами, с помощью `auto` в C++17 была в основном разработана Майком Спертусом (Mike Speratus) при участии Джеймса Тутона (James Touton), Дэвида Вандевурда и многих других. Изменения спецификации для этого средства описаны в документе P0127R2. Интересно, что то, что возможность использования `decltype(auto)` вместо `auto` была намеренно сделана частью языка, не очевидно (по-видимому, этот вопрос не обсуждался Комитетом, но он выпадает из спецификации).

Майк Спертус также способствовал развитию *вывода аргументов шаблонов классов* в C++17; значительные технические идеи были привнесены Ричардом Смитом (Richard Smith) и Файсалом Вали (Faisal Vali) (включая идею правил вывода). Документ P0091R3 содержит спецификации, которые были приняты в рабочий документ для следующего стандарта языка.

Структурированное связывание в основном продвигал Герб Саттер (Herb Sutter), который в документе P0144R1 в соавторстве с Габриэлем Дос Рейсом (Gabriel Dos Reis) и Бъярне Страуструпом предложил это средство. Во время обсуждений в Комитете было внесено множество поправок, включая использование квадратных скобок для разделения идентификаторов. Йенс Маурер перевел предложение в окончательную спецификацию стандарта (P0217R3).

Глава 16

Специализация и перегрузка

Сейчас вы уже знаете, как шаблоны C++ обеспечивают расширение обобщенного определения в семейство связанных классов, функций или переменных. Хотя это и мощный механизм, существует много ситуаций, в которых при замене параметров шаблона обобщенная форма работы далека от оптимальной.

Язык C++ в определенной мере уникален среди прочих популярных языков программирования с поддержкой обобщенного программирования, поскольку обладает богатым набором возможностей, позволяющих осуществлять прозрачную подмену обобщенного определения более специализированными. В этой главе представлены два механизма языка C++, которые позволяют реализовать полезные отступления от обобщенного подхода: специализация шаблона и перегрузка шаблонов функций.

16.1. Когда обобщенный код недостаточно хорош

Рассмотрим следующий пример:

```
template<typename T>
class Array
{
private:
    T* data;

    ...
public:
    Array(Array<T> const&);
    Array<T>& operator= (Array<T> const&);
    void exchangeWith(Array<T>* b)
    {
        T* tmp = data;
        data = b->data;
        b->data = tmp;
    }
    T& operator[](std::size_t k)
    {
        return data[k];
    }
    ...
};

template<typename T> inline
void exchange(T* a, T* b)
{
    T tmp(*a);
    *a = *b;
    *b = tmp;
}
```

Для простых типов обобщенная реализация функции `exchange()` работает хорошо. Однако для типов с дорогостоящими операциями копирования обобщенная реализация может быть значительно более ресурсоемкой (как в плане использования машинного времени, так и памяти), чем реализация, настроенная под конкретную структуру данных. В нашем примере обобщенная реализация требует одного вызова конструктора копирования шаблона `Array<T>` и двух вызовов оператора копирующего присваивания. Для больших структур данных создание таких копий часто сопровождается копированием относительно больших объемов памяти. Однако функциональность `exchange()` часто может заменяться простым обменом указателями, подобно тому, как это делается в функции-члене `exchangeWith()`.

16.1.1. Прозрачная настройка

В предыдущем примере функция-член `exchangeWith()` обеспечивала эффективную альтернативу обобщенной функции `exchange()`. Тем не менее по ряду причин использование другой функции неудобно.

- Пользователи класса `Array` должны помнить о дополнительном интерфейсе и по возможности аккуратно им пользоваться.
- Обобщенные алгоритмы в общем случае не могут различать различные возможные варианты. Например:

```
template<typename T>
void genericAlgorithm(T* x, T* y)
{
    ...
    exchange(x, y); // Как выбрать правильный алгоритм?
    ...
}
```

По этим соображениям шаблоны C++ обеспечивают прозрачные способы настройки шаблонов функций и классов. Для шаблонов функций это достигается через механизм перегрузки. Например, можно написать перегруженный набор шаблонов функций `quickExchange()`, как показано ниже.

```
template<typename T>
void quickExchange(T* a, T* b) // #1
{
    T tmp(*a);
    *a = *b;
    *b = tmp;
}

template<typename T>
void quickExchange(Array<T>* a, Array<T>* b) // #2
{
    a->exchangeWith(b);
}
```

```

void demo(Array<int>* p1, Array<int>* p2)
{
    int x = 42, y = -7;
    quickExchange(&x, &y);                                // Используется #1
    quickExchange(p1, p2);                                // Используется #2
}

```

Первый вызов `quickExchange()` имеет два аргумента типа `int*`, поэтому вывод аргументов выполняется успешно только для первого шаблона, объявленного в строке #1, когда тип `T` заменяется типом `int`. Поэтому не возникает сомнений относительно того, какую функцию нужно вызвать. Второй же вызов соответствует обоим шаблонам: подходящие функции для вызова `quickExchange(p1, p2)` получаются как подстановкой `Array<int>` вместо `T` в первом шаблоне, так и подстановкой `int` во втором шаблоне. Кроме того, обе подстановки дают функции с типами параметров, которые точно соответствуют типам аргументов во втором вызове. Обычно это позволяет заключить, что вызов неоднозначен, однако (как выяснится позже) язык C++ считает второй шаблон “более специализированным”, чем первый. При прочих равных условиях разрешение перегрузки отдает предпочтение более специализированному шаблону, и поэтому выбирается шаблон из строки #2.

16.1.2. Семантическая прозрачность

Использование перегрузки, как было показано в предыдущем разделе, очень полезно при достижении прозрачной настройки процесса инстанцирования. При этом важно понимать, что такая “прозрачность” существенно зависит от деталей реализации. Чтобы проиллюстрировать это, рассмотрим наше решение `quickExchange()`. Хотя и обобщенный алгоритм, и алгоритм, настроенный для типов `Array<T>`, заканчиваются обменом значений, на которые указывают указатели, побочные эффекты этих операций существенно отличаются. Яркой иллюстрацией тому может служить код, который сравнивает обмен структурных объектов с обменом шаблонов `Array<T>`.

```

struct S
{
    int x;
} s1, s2;
void distinguish(Array<int> a1, Array<int> a2)
{
    int* p = &a1[0];
    int* q = &s1.x;
    a1[0] = s1.x = 1;
    a2[0] = s2.x = 2;
    quickExchange(&a1, &a2); // После этого *p == 1 (все еще)
    quickExchange(&s1, &s2); // После этого *q == 2
}

```

Этот пример показывает, что после вызова `quickExchange()` указатель `p` на первый массив `Array` становится указателем на второй массив. Однако указатель на поле объекта `s1`, не являющегося массивом, продолжает указывать

на поле структуры `s1` даже после выполнения операции обмена. Единственное изменение — поменялись местами значения, на которые указывают указатели. Это весьма существенное отличие, которое может сбивать с толку пользователей шаблона. Применение префикса `quick` позволяет привлечь внимание к тому, что данная реализация представляет собой сокращенный вариант нужной операции. Однако первоначальный обобщенный шаблон `exchange()` может при этом содержать оптимизацию для шаблонов `Array<T>`:

```
template<typename T>
void exchange(Array<T>* a, Array<T>* b)
{
    T* p = &(*a)[0];
    T* q = &(*b)[0];

    for (std::size_t k = a->size(); k-- != 0;)
    {
        exchange(p++, q++);
    }
}
```

Преимущество этой версии обобщенного кода заключается в том, что при этом не требуется создавать (потенциально) большой временный массив `Array<T>`. Шаблон `exchange()` вызывается рекурсивно, чем достигается хорошая производительность даже для таких типов, как `Array<Array<char>>`. Отметим также, что более специализированная версия шаблона не объявлена как `inline`, поскольку выполняет значительный объем работы. В то же время первоначальная обобщенная реализация является встроенной, поскольку выполняет только несколько операций (каждая из которых потенциально является дорогостоящей).

16.2. Перегрузка шаблонов функций

В предыдущем разделе было показано, что возможно сосуществование двух шаблонов функций с одним и тем же именем, даже если они могут быть инстанцированы с параметрами идентичных типов. Приведем еще один простой пример этого.

`details/funcoverload1.hpp`

```
template<typename T>
int f(T)
{
    return 1;
}

template<typename T>
int f(T*)
{
    return 2;
}
```

Когда тип `T` заменяется типом `int*` в первом шаблоне, получается функция, у которой есть точно такие же типы параметров (и возвращаемых значений),

что и у функции, получаемой при замене типа `int` типом `T` во втором шаблоне. Существовать могут не только эти шаблоны, но и их инстанцирования, даже если у них идентичны типы параметров и возвращаемых значений.

Приведенный ниже пример демонстрирует, как можно вызвать две такие сгенерированные функции с помощью синтаксиса явного аргумента шаблона (в предположении предыдущих объявлений шаблона):

`details/funcoverload1.cpp`

```
#include <iostream>
#include "funcoverload1.hpp"
int main()
{
    std::cout << f<int*>((int*)nullptr); // Вызов f<T>(T)
    std::cout << f<int>((int*)nullptr); // Вызов f<T>(T*)
}
```

Результатом выполнения этой программы будет следующий вывод:

12

Чтобы объяснить работу программы, подробно проанализируем вызов `f<int*>((int*)nullptr)`. Синтаксис `f<int*>` обозначает, что первый параметр шаблона `f` нужно заменить значением типа `int*` без использования вывода аргумента шаблона. В этом случае существует более одного шаблона `f` и поэтому создается набор перегрузки, содержащий две функции, сгенерированные из шаблонов: `f<int*>(int*)` (сгенерированная из первого шаблона) и `f<int*>(int**)` (сгенерированная из второго шаблона). Аргумент вызова `(int*)nullptr` имеет тип `int*`, который соответствует только функции, сгенерированной из первого шаблона. Следовательно, это и есть функция, которая будет вызвана в конечном итоге.

С другой стороны, для второго вызова множество перегрузки содержит функции `f<int>(int)` (генерируется из первого шаблона) и `f<int>(int*)` (генерируется из второго шаблона), так что соответствует аргументу только второй шаблон.

16.2.1. Сигнатуры

Две функции могут существовать в программе, если у них разные сигнатуры. Определим сигнатуру как приведенную ниже информацию¹.

1. Неквалифицированное имя функции (или имя шаблона функции, из которого она генерируется).
2. Область видимости класса или пространства имен и, если это имя имеет внутреннее связывание, единица трансляции, в которой объявлено имя.
3. Классификация функции как `const`, `volatile` или `const volatile` (если это функция-член с данным квалификатором).

¹ Это определение отличается от того, которое дано в стандарте C++, однако следствия у них эквиваленты.

4. Квалификаторы & или && функции (если она является функцией-членом с таким квалификатором).
5. Типы параметров функции (перед подстановкой параметров шаблона, если функция генерируется из шаблона функции).
6. Если функция генерируется из шаблона функции, то тип ее возвращаемого значения.
7. Параметры и аргументы шаблона, если функция генерируется из шаблона функции.

Это означает, что в одной и той же программе могут сосуществовать следующие шаблоны и их инстанцирования:

```
template<typename T1, typename T2>
void f1(T1, T2);

template<typename T1, typename T2>
void f1(T2, T1);

template<typename T>
long f2(T);

template<typename T>
char f2(T);
```

Однако их не всегда можно использовать, если они объявлены в одной и той же области видимости, поскольку при их инстанцировании возникает неоднозначность перегрузки. Например, очевидно, что вызов `f2(42)` приводит к неоднозначности, если объявлены оба соответствующих шаблона, приведенных выше. Еще один пример показан ниже:

```
#include <iostream>

template<typename T1, typename T2>
void f1(T1, T2)
{
    std::cout << "f1(T1, T2)\n";
}

template<typename T1, typename T2>
void f1(T2, T1)
{
    std::cout << "f1(T2, T1)\n";
}

// Пока все в порядке
int main()
{
    f1<char, char>('a', 'b'); // Ошибка: неоднозначность
}
```

Здесь функция

```
f1<T1 = char, T2 = char>(T1, T2)
```

может сосуществовать с функцией

```
f1<T1 = char, T2 = char>(T2, T1)
```

однако разрешение перегрузки никогда не отдаст предпочтения одной из них. Если эти шаблоны появляются в различных единицах трансляции, эти два экземпляра действительно могут существовать в одной и той же программе (при этом компоновщик не должен жаловаться на двойное определение, поскольку сигнатуры инстанций различны):

```
// === Единица трансляции 1:
#include <iostream>

template<typename T1, typename T2>
void f1(T1, T2)
{
    std::cout << "f1(T1, T2)\n";
}

void g()
{
    f1<char, char>('a', 'b');
}

// === Единица трансляции 2:
#include <iostream>

template<typename T1, typename T2>
void f1(T2, T1)
{
    std::cout << "f1(T2, T1)\n";
}

extern void g(); // Определена в единице трансляции 1

int main()
{
    f1<char, char>('a', 'b');
    g();
}
```

Эта программа корректна и выдает следующее:

```
f1(T2, T1)
f1(T1, T2)
```

16.2.2. Частичное упорядочение перегруженных шаблонов функций

Вернемся к рассмотренному ранее примеру. Мы обнаружили, что после установки данных списков аргументов (`<int*>` и `<int>`), разрешение перегрузки в конечном итоге выбирает для вызова нужную функцию:

```
std::cout << f<int*>((int*)nullptr); // Вызов f<T>(T)
std::cout << f<int>">((int*)nullptr); // Вызов f<T>(T*)
```

Однако выбор функции происходит даже тогда, когда явные аргументы шаблона не указаны. В этом случае вступает в игру вывод аргумента шаблона. Чтобы обсудить этот механизм, давайте немного модифицируем функцию `main()` из предыдущего примера.

`details/funcoverload2.cpp`

```
#include <iostream>

template<typename T>
int f(T)
{
    return 1;
}

template<typename T>
int f(T*)
{
    return 2;
}

int main()
{
    std::cout << f(0);           // Вызов f<T>(T)
    std::cout << f(nullptr);     // Вызов f<T>(T)
    std::cout << f((int*)nullptr); // Вызов f<T>(T*)
}
```

Рассмотрим первый вызов, `f(0)`: здесь тип аргумента — `int`, который соответствует типу параметра первого шаблона, если заменить `T` на `int`. Однако тип параметра второго шаблона — это всегда указатель, и, следовательно, после вывода кандидатом для вызова будет только экземпляр, сгенерированный из первого шаблона. В этом случае разрешение перегрузки тривиально.

То же самое можно сказать и о втором вызове, `f(nullptr)`. Тип аргумента в этом случае `std::nullptr_t`, который опять же соответствует только первому шаблону.

Третий вызов (`f((int*)nullptr)`) более интересен: осуществить вывод аргумента удается для обоих шаблонов, что дает функции `f<int*>(int*)` и `f<int>(int*)`. С точки зрения традиционного разрешения перегрузки обе функции одинаково хороши для вызова с аргументом `int*`, что соответствует неоднозначности вызова (см. приложение B, “Разрешение перегрузки”). Однако в таких случаях вступает в игру дополнительный критерий перегрузки: выбирается функция, сгенерированная из более специализированного шаблона. Здесь, как вы скоро увидите, второй шаблон считается более специализированным, а потому результатом работы этой программы будет

16.2.3. Правила формального упорядочения

В нашем последнем примере интуитивно вполне понятно, что второй шаблон “более специальный”, чем первый, поскольку первый может быть подстроен

почти под любой тип аргумента, тогда как второй разрешает только типы-указатели. Однако другие примеры могут оказаться не столь очевидными. Далее описана точная процедура определения того, является ли один шаблон, участвующий в наборе перегрузки, более специализированным, чем другой. Отметим, однако, что это правила лишь *частичного* упорядочения: возможна ситуация, когда ни один из шаблонов не будет считаться более специализированным, чем другой. Если разрешение перегрузки должно выбирать между такими шаблонами, решение принято не будет и в программе возникнет ошибка неоднозначности.

Предположим, сравниваются два идентично именованных шаблона, которые кажутся подходящими для данного вызова функции. Разрешение перегрузки принимает решение следующим образом.

- Параметры вызова функции, которые используют аргументы по умолчанию или многоточия, игнорируются.
- Создаются два искусственных списка типов аргументов (а для шаблона функции преобразования типов — возвращаемого типа) путем подстановки каждого параметра шаблона следующим образом.
 1. Заменим каждый типовой параметр шаблона уникальным искусственным типом.
 2. Заменим каждый шаблонный параметр шаблона уникальным искусственным шаблоном класса.
 3. Заменим каждый шаблонный параметр, не являющийся типом, уникальным искусственным значением соответствующего типа.

(Искусственные типы, шаблоны и значения в этом контексте отличаются от любых прочих типов, шаблонов или значений, как используемых программистом, так и создаваемых компилятором в других контекстах.)

- Если вывод аргументов второго шаблона из первого синтезированного списка типов аргументов происходит успешно при точном соответствии, но не наоборот, то говорят, что первый шаблон является *более специализированным*, чем второй. Если вывод аргументов первого шаблона для второго синтезированного списка типов аргументов происходит успешно при точном соответствии, но не наоборот, то говорят, что второй шаблон является более специализированным, чем первый. В ином случае (если нет ни одного успешного вывода или же оба вывода успешны) упорядочения шаблонов не происходит.

Попробуем применить этот подход к двум шаблонам в предыдущем примере. Для этих шаблонов синтезируются два списка типов аргументов путем замены шаблонных параметров описанным выше способом: (A_1) и (A_2^*), где A_1 и A_2 — уникальные искусственные типы. Очевидно, что вывод первого шаблона для второго списка типов аргументов происходит успешно при замене A_2^* на T . Однако тип T^* из второго шаблона невозможно сделать соответствующим типу A_1 из первого списка, который не является типом указателя. Следовательно, формально можно заключить, что второй шаблон более специализирован, чем первый.

Рассмотрим более сложный пример с использованием нескольких параметров функций.

```
template<typename T>
void t(T*, T const* = nullptr, ...);

template<typename T>
void t(T const*, T*, T* = nullptr);

void example(int* p)
{
    t(p, p);
}
```

Прежде всего, поскольку реальный вызов не использует параметр многоточия для первого шаблона, а последний параметр второго шаблона покрывается аргументом по умолчанию, эти аргументы при частичном упорядочении игнорируются. Отметим, что аргумент первого шаблона по умолчанию не используется. Поэтому соответствующий параметр участвует в упорядочении.

Созданные списки типов аргументов – это $(A1^*, A1\ const^*)$ и $(A2\ const^*, A2^*)$. Вывод аргументов шаблона $(A1^*, A1\ const^*)$ для второго шаблона успешен при замене T на $A1\ const$, однако результирующее соответствие не точное, поскольку для вызова $t<A1\ const>(A1\ const^*, A1\ const^*, A1\ const^*=0)$ с аргументами типов $(A1^*, A1\ const^*)$ требуется дополнительное уточнение типов. Точно так же нельзя найти точное соответствие при выводе аргументов шаблона для первого шаблона из списка типов аргументов $(A2\ const^*, A2^*)$. Следовательно, между двумя шаблонами нет отношения упорядочения и вызовы неоднозначны.

Формальные правила упорядочения обычно обеспечивают возможность интуитивно очевидного выбора шаблонов функций. Тем не менее можно привести множество примеров, когда интуитивно очевидный выбор оказывается невозможным. Вероятно, данные правила упорядочения в будущем могут быть пересмотрены, чтобы такие ситуации стали разрешимыми.

16.2.4. Шаблоны и нешаблоны

Шаблоны функций можно перегружать нешаблонными функциями. При прочих равных условиях при выборе реальной функции вызова нешаблонная функция предпочтительнее. Приведенный ниже пример иллюстрирует это.

details/nontmp11.cpp

```
#include <string>
#include <iostream>

template<typename T>
std::string f(T)
{
    return "Template";
}
```

```
std::string f(int&)
{
    return "Nontemplate";
}

int main()
{
    int x = 7;
    std::cout << f(x) << '\n'; // Выводит: Nontemplate
}
```

Результат выполнения программы:

```
Nontemplate
```

Однако при различных ссылочных квалификаторах и `const` приоритеты разрешения перегрузки могут измениться. Например:

`details/nontmpl2.cpp`

```
#include <string>
#include <iostream>

template<typename T>
std::string f(T&)
{
    return "Template";
}

std::string f(int const&)
{
    return "Nontemplate";
}

int main()
{
    int x = 7;
    std::cout << f(x) << '\n'; // Выводит: Template
    int const c = 7;
    std::cout << f(c) << '\n'; // Выводит: Nontemplate
}
```

Результат выполнения программы:

```
Template
Nontemplate
```

Теперь шаблон функции `f<>(T&)` демонстрирует лучшее соответствие при передаче неконстантного `int`. Дело в том, что для `int` инстанцированная функция `f<>(int &)` демонстрирует лучшее соответствие, чем `f(int const &)`. Таким образом, разница не только в том, что одна функция является шаблоном, а другая — нет. В этом случае применяются общие правила разрешения перегрузки (см. раздел B.2). И только при вызове `f()` для `const int` обе сигнатуры имеют один и тот же тип `int const &`, так что предпочтительным оказывается нешаблонный вариант.

По этой причине хорошая идея объявлять шаблон функции-члена как

```
template<typename T>
std::string f(T const&)
{
    return "Template";
}
```

Тем не менее этот эффект легко может оказаться случайным и привести к удивительному поведению, когда функции-члены определены так, что принимают те же аргументы, что и копирующие или перемещающие конструкторы. Например:

details/tmpconstr.cpp

```
#include <string>
#include <iostream>
class C
{
public:
    C() = default;
    C(C const&)
    {
        std::cout << "copy constructor\n";
    }
    C(C&&)
    {
        std::cout << "move constructor\n";
    }
    template<typename T>
    C(T&&)
    {
        std::cout << "template constructor\n";
    }
};

int main()
{
    C x;
    C x2{x};           // Выводит: template constructor
    C x3{std::move(x)}; // Выводит: move constructor
    C const c;
    C x4{c};           // Выводит: copy constructor
    C x5{std::move(c)}; // Выводит: template constructor
}
```

Результат выполнения программы:

```
template constructor
move constructor
copy constructor
template constructor
```

Таким образом, шаблон функции-члена лучше подходит для копирования С, чем копирующий конструктор. И для `std::move(c)`, который дает тип С `const &` (возможный, но обычно не имеющий значимой семантики тип), шаблон

функции-члена также демонстрирует лучшее соответствие, чем перемещающий конструктор.

По этой причине обычно следует частично отключать такие шаблоны функций-членов, когда они могут скрывать копирующие или перемещающие конструкторы. Этот вопрос объясняется в разделе 6.4.

16.2.5. Вариативные шаблоны функций

Вариативные шаблоны функций (см. раздел 12.4) требуют при частичном упорядочении особого подхода, потому что при выводе для пакета параметров (описан в разделе 15.5) один параметр соответствует нескольким аргументам. Это поведение приводит к некоторым интересным ситуациям при упорядочении шаблонов функций, иллюстрируемым в следующем примере:

details/variadicoverload.cpp

```
#include <iostream>

template<typename T>
int f(T*)
{
    return 1;
}

template<typename... Ts>
int f(Ts...)
{
    return 2;
}

template<typename... Ts>
int f(Ts* ...)
{
    return 3;
}

int main()
{
    std::cout << f(0, 0.0);           // Вызов f<>(Ts...)
    std::cout << f((int*)nullptr,
                    (double*)nullptr); // Вызов f<>(Ts*...)
    std::cout << f((int*)nullptr);   // Вызов f<>(T*)
}
```

Вывод этой программы, которую мы сейчас детально разберем, имеет вид

231

В первом вызове, `f(0, 0.0)`, рассматриваются все шаблоны функций с именем `f`. Для первого шаблона функции, `f(T*)`, вывод оказывается неудачен, как потому, что параметр шаблона `T` не может быть выведен, так и потому, что аргументов функции больше, чем параметров в этом невариативном шаблоне функции. Второй шаблон функции, `f(Ts...)`, является вариативным: вывод в этом

случае сравнивает схему пакета параметров функции (Ts) с типами двух аргументов (`int` и `double`, соответственно), выводя Ts как последовательность (`int, double`). Для третьего шаблона функции, $f(Ts^* \dots)$, вывод сравнивает схему пакета параметров функции Ts^* с каждым из типов аргументов. Этот вывод оказывается неудачным (Ts не может быть выведен), оставляя в результате в качестве кандидата только один второй шаблон функции. Упорядочение шаблонов функций не требуется.

Второй вызов, $f((int*)nullptr, (double*)nullptr)$, более интересен: вывод неудачен для первого шаблона функции, потому что аргументов функции больше, чем параметров, но успешен для второго и третьего шаблонов. Будучи записанными явно, результирующие вызовы имеют вид

```
f<int*,double*>((int*)nullptr, (double*)nullptr) // Второй шаблон
```

и

```
f<int,double>((int*)nullptr, (double*)nullptr) // Третий шаблон
```

Затем частичное упорядочение рассматривает второй и третий шаблоны (оба они являются вариативными) следующим образом: при применении формальных правил упорядочения, описанных в разделе 16.2.3, к вариативному шаблону каждый пакет параметров шаблона заменяется одним искусственным типом, шаблоном класса или значением. Например, это означает, что сгенерированные типы аргументов для второго и третьего шаблонов функций представляют собой $A1$ и $A2^*$, соответственно, где $A1$ и $A2$ – уникальные, искусственно введенные типы. Вывод для второго шаблона для третьего списка типов аргументов успешен при подстановке одноэлементной последовательности ($A2^*$) вместо пакета параметров Ts . Однако не существует способа сделать схему Ts^* пакета параметров третьего шаблона соответствующей типу $A1$, не являющемуся типом указателя; поэтому третий шаблон функции (принимающий аргументы-указатели), считается более специализированным, чем второй шаблон функции (который принимает любые аргументы).

Третий вызов, $f((int*)nullptr)$, вводит новые сложности: вывод успешен для всех трех шаблонов функций, что требует частичного упорядочения для сравнения вариативных и невариативных шаблонов. Для иллюстрации сравним первый и третий шаблоны функций. Синтезированные типы аргументов представляют собой $A1^*$ и $A2^*$, где $A1$ и $A2$ являются уникальными искусственными типами. Вывод первого шаблона для третьего списка синтезированных аргументов успешен при подстановке $A2$ вместо T . В другом направлении вывод третьего шаблона для первого списка синтезированных аргументов также успешен при подстановке одноэлементной последовательности ($A1$) вместо пакета параметров Ts . Частичное упорядочение между первым и третьим шаблонами должно бы вести к неоднозначности. Однако имеется специальное правило, которое запрещает со-поставление аргумента, изначально пришедшего из пакета параметров функции (например, пакет параметров третьего шаблона $Ts^* \dots$), параметру, который не является пакетом параметров (параметр первого шаблона T^*). Следовательно, вывод первого шаблона для третьего списка синтезированных аргументов неудачен,

и первый шаблон рассматривается как более специализированный, чем третий. Это специальное правило рассматривает невариативные шаблоны (т.е. шаблоны с фиксированным количеством параметров) как более специализированные, чем вариативные шаблоны (с переменным количеством параметров).

Описанные выше правила точно так же применяются к раскрытиям пакетов, происходящим в типах в сигнатуре функции. Например, мы можем “обернуть” параметры и аргументы каждого из шаблонов функций в нашем предыдущем примере в вариативный шаблон класса `Tuple` для получения аналогичного примера, не связанного с пакетами параметров функции:

`details/tupleoverload.cpp`

```
#include <iostream>

template<typename... Ts> class Tuple
{
};

template<typename T>
int f(Tuple<T*>)
{
    return 1;
}

template<typename... Ts>
int f(Tuple<Ts...>)
{
    return 2;
}

template<typename... Ts>
int f(Tuple<Ts* ...>)
{
    return 3;
}

int main()
{
    std::cout << f(Tuple<int,double>()); // Вызов f<>(Tuple<Ts...>)
    std::cout << f(Tuple<int*,double*>()); // Вызов f<>(Tuple<Ts*...>)
    std::cout << f(Tuple<int*>()); // Вызов f<>(Tuple<T*>)
}
```

Упорядочение шаблонов функций рассматривает раскрытие пакета в аргументах шаблона в `Tuple` аналогично пакетам параметров функций в нашем предыдущем примере, что приводит к такому же, как и выше, результату:

231

16.3. Явная специализация

Возможность перегружать шаблоны функций в сочетании с правилами частичного упорядочения при выборе обеспечивающего наилучшее соответствие

шаблона функции позволяет добавлять к обобщенной реализации специализированные шаблоны для повышения эффективности кода. Однако перегружать шаблоны классов и шаблоны переменных нельзя. Вместо этого для обеспечения прозрачной настройки шаблонов классов выбран другой механизм — *явная специализация* (explicit specialization). Стандартный термин *явная специализация* ссылается на возможность языка, которую мы называем *полной специализацией* (full specialization). Она обеспечивает реализацию шаблона с полностью замененными параметрами шаблонов, когда никаких неизвестных шаблонных параметров не остается. Шаблоны классов, шаблоны функций и шаблоны переменных могут быть полностью специализированными².

Члены шаблонов классов (т.е. функции-члены, вложенные классы, статические члены-данные и члены с типами перечислений) могут быть определены за пределами тела определения класса.

В одном из следующих разделов рассматривается *частичная специализация* (partial specialization). Она напоминает полную специализацию, но вместо полной замены шаблонных параметров в ней остается некоторая параметризация. Полная и частичная специализации одинаково “явно” присутствуют в нашем исходном коде, поэтому при обсуждении мы избегаем термина *явная специализация*. Ни полная, ни частичная специализация не вводят полностью новый шаблон или его экземпляр. Вместо этого данные конструкции предоставляют возможность альтернативного определения для экземпляров, которые уже неявно определены в обобщенном (*неспециализированном*) шаблоне. Это довольно важное концептуальное отличие от перегруженных шаблонов.

16.3.1. Полная специализация шаблона класса

Полная специализация вводится последовательностью трех лексем: `template`, `<` и `>`³. Кроме того, после имени класса идут аргументы шаблона, для которых объявляется специализация. Это проиллюстрировано ниже.

```
template<typename T>
class S
{
public:
    void info()
    {
        std::cout << "generic (S<T>::info())\n";
    }
};
```

² Шаблоны псевдонимов являются единственной формой шаблонов, которая не может быть специализирована, ни полностью, ни частично. Это ограничение необходимо для того, чтобы сделать использование шаблонов псевдонимов прозрачным для процесса вывода аргумента шаблона (раздел 15.11).

³ Тот же префикс требуется и при объявлении полной специализации шаблона функции. Ранние версии языка C++ не включали этот префикс, однако добавление шаблонов-членов потребовало дополнительного синтаксиса для разрешения неоднозначности в сложных случаях специализации.

```

template<>
class S<void>
{
public:
    void msg()
    {
        std::cout << "fully specialized (S<void>::msg())\n";
    }
};

```

Обратите внимание на то, что реализация полной специализации не требует какой-либо связи с обобщенным определением. Это позволяет создавать функции-члены с различными именами (`info` и `msg`). Связь между ними определяется исключительно именем шаблона класса.

Список указываемых аргументов шаблона должен соответствовать списку параметров шаблона. Например, некорректно использовать значение, не являющееся типом, вместо шаблонного параметра типа. Однако указывать аргументы для параметров со значениями по умолчанию необязательно:

```

template<typename T>
class Types
{
public:
    using I = int;
};

template<typename T, typename U = typename Types<T>::I> // #1
class S;

template<>
class S<void> // #2
{
public:
    void f();
};

template<> class S<char, char>; // #3

template<> class S<char,0>; // Ошибка: 0 нельзя подставить вместо U

int main()
{
    S<int>* pi; // OK: использует #1, определение не нужно
    S<int> el; // Ошибка: использует #1, но нет определения
    S<void>* pv; // OK: использует #2
    S<void, int> sv; // OK: использует #2, определение имеется
    S<void, char> e2; // Ошибка: использует #1, но нет определения
    S<char, char> e3; // Ошибка: использует #3, но нет определения
}

template<>
class S<char, char> // Определение для #3
{
};

```

Данный пример также показывает, что объявления полной специализации (и шаблонов) не обязательно должны быть определениями. Однако, если объявлена полная специализация, для данного набора аргументов шаблона обобщенное определение никогда не используется. Следовательно, если определение необходимо, но его нет, в программе содержится ошибка. Для специализации шаблонов класса иногда полезно предварительное объявление типов, что позволяет создавать взаимно зависимые типы. Объявление полной специализации идентично объявлению обычного класса (это *не* шаблонное объявление). Все, что их отличает, — это синтаксис и тот факт, что объявление должно соответствовать предыдущему объявлению шаблона. Поскольку это не объявление шаблона, члены полной специализации шаблона класса могут быть определены с помощью обычного синтаксиса определения члена вне класса (иными словами, нельзя указывать префикс `template<>`):

```
template<typename T>
class S;

template<> class S<char**>
{
public:
    void print() const;
};

// Следующее определение не может предваряться template<>
void S<char**>::print() const
{
    std::cout << "pointer to pointer to char\n";
}
```

Ниже приведен более сложный пример этой концепции:

```
template<typename T>
class Outside
{
public:
    template<typename U>
    class Inside
    {
    };
};

template<>
class Outside<void>
{
    // Нет никакой особой связи между следующим вложенным
    // классом и определенным в обобщенном шаблоне
    template<typename U>
    class Inside
    {
        private:
            static int count;
    };
};
```

```
// Следующее определение не может предваряться template<>
template<typename U>
int Outside<void>::Inside<U>::count = 1;
```

Полная специализация — это замена инстанцирования некоторого обобщенного шаблона. При этом некорректно одновременно иметь как явную, так и сгенерированную версии шаблона в одной и той же программе. Попытка использовать их обе в одном и том же файле обычно отслеживается компилятором:

```
template<typename T>
class Invalid
{
};

Invalid<double> xl;      // Вызывает инстанцирование Invalid<double>
template<>
class Invalid<double>; // Ошибка: Invalid<double> уже инстанцирован
```

К сожалению, при использовании в различных единицах трансляции проблема не отслеживается так легко. Следующий некорректный пример кода на C++ состоит из двух файлов. Этот код компилируется и компонуется многими реализациями, однако он некорректен и опасен.

```
// ===== Единица трансляции 1:
template<typename T>
class Danger
{
public:
    enum { max = 10 };
};

char buffer[Danger<void>::max]; // Использует обобщенное значение

extern void clear(char*);

int main()
{
    clear(buffer);
}

// ===== Единица трансляции 2:
template<typename T>
class Danger;

template<>
class Danger<void>
{
public:
    enum { max = 100 };
};

void clear(char* buf)
{
    // Несоответствие в границе массива:
    for (int k = 0; k < Danger<void>::max; ++k)
    {
```

```

    buf[k] = '\0';
}
}

```

Этот пример был придуман специально, чтобы как можно более коротко показать, насколько необходимо следить за тем, чтобы объявление специализации было видно всем пользователям обобщенного шаблона. С практической точки зрения это означает, что объявление специализации должно идти после объявления шаблона в его заголовочном файле. Если обобщенная реализация берет начало из внешнего источника (такого, что соответствующие заголовочные файлы не должны изменяться), то стоит, хотя это и не обязательно, создать заголовочный файл, включающий обобщенный шаблон с последующим объявлением специализаций, чтобы избежать таких труднообнаруживаемых ошибок. В целом лучше избегать специализации шаблона, происходящего из внешнего источника, если не указано, что он для этого предназначен.

16.3.2. Полная специализация шаблона функции

Синтаксис и принципы, лежащие в основе (явной) полной специализации шаблона функции, во многом такие же, как и в случае полной специализации шаблона класса. Однако здесь вступают в игру перегрузка и вывод аргумента.

При объявлении полной специализации можно пропускать явные аргументы шаблона, если этот шаблон можно определить с помощью вывода аргумента (используя в качестве типов аргументов типы параметров, указанные в объявлении) и частичного упорядочения. Например:

```

template<typename T>
int f(T)                                // #1
{
    return 1;
}

template<typename T>
int f(T*)                                 // #2
{
    return 2;
}

template<> int f(int)                      // OK: специализация #1
{
    return 3;
}

template<> int f(int*)
// OK: специализация #2
{
    return 4;
}

```

Полная специализация шаблона функции не может включать значения аргумента по умолчанию. Однако любые аргументы по умолчанию, указанные для шаблона, подвергаемого специализации, остаются применимыми и для явной специализации.

```

template<typename T>
int f(T, T x = 42)
{
    return x;
}

template<> int f(int, int = 35) // Ошибка
{
    return 0;
}

```

(Дело в том, что полная специализация обеспечивает альтернативное определение, но не альтернативное объявление. В точке вызова шаблона функции вызов полностью разрешен на основе шаблона функции.)

Полная специализация во многом подобна обычному объявлению (точнее, обычному *повторному* объявлению). В частности, она не объявляет шаблон и, следовательно, в программе должно быть только *одно определение* невстраиваемой полной специализации шаблона функции. Однако необходимо следить за тем, чтобы *объявление* полной специализации следовало после шаблона, что предотвратит попытки использования функции, сгенерированной из шаблона. Поэтому объявления шаблона `g()` и его полной специализации в предыдущем примере лучше размещать в двух файлах следующим образом.

- Файл интерфейса содержит определения первичных шаблонов и частичных специализаций, но объявляет только полные специализации:

```

#ifndef TEMPLATE_G_HPP
#define TEMPLATE_G_HPP

// Определение шаблона должно находиться в заголовочном файле:
template<typename T>
int g(T, T x = 42)
{
    return x;
}

// Объявление специализации запрещает инстанцирования шаблона;
// определение не должно появляться здесь во избежание ошибок
// многократных определений
template<> int g(int, int y);
#endif // TEMPLATE_G_HPP

```

- Соответствующий файл реализации определяет полную специализацию:

```

#include "template_g.hpp"

template<> int g(int, int y)
{
    return y / 2;
}

```

В качестве альтернативы специализация может быть встраиваемой; в этом случае ее объявление может (и должно) быть помещено в заголовочном файле.

16.3.3. Полная специализация шаблона переменной

Шаблоны переменных также могут быть полностью специализированы. Синтаксис уже должен быть интуитивно понятен для вас:

```
template<typename T> constexpr std::size_t SZ = sizeof(T);
template<> constexpr std::size_t SZ<void> = 0;
```

Очевидно, что специализация может предоставить инициализатор, который отличается от получающегося из шаблона. Интересно, что специализация шаблона переменной не обязана иметь тип, соответствующий типу в специализируемом шаблоне.

```
template<typename T> typename T::iterator null_iterator;
template<> BitIterator null_iterator<std::bitset<100>>;
// BitIterator не соответствует T::iterator, и это нормально
```

16.3.4. Полная специализация члена

Полностью специализироваться могут не только шаблоны членов, но и обычные статические члены-данные и функции-члены шаблонов класса. Синтаксис требует наличия префикса `template<>` для каждого охватывающего шаблона класса. Если специализируется шаблон члена, то для указания этой специализации необходимо добавить префикс `template<>`. Чтобы проиллюстрировать это, будем считать, что у нас имеются следующие объявления:

```
template<typename T>
class Outer                                // #1
{
public:
    template<typename U>
    class Inner                               * // #2
    {
private:
        static int count;                  // #3
    };
    static int code;                      // #4
    void print() const                   // #5
    {
        std::cout << "generic";
    }
}

template<typename T>
int Outer<T>::code = 6;                    // #6

template<typename T> template<typename U>
int Outer<T>::Inner<U>::count = 7;          // #7

template<>
class Outer<bool>                         // #8
{
public:
    template<typename U>
    class Inner                           // #9
```

```

{
    private:
        static int count;           // #10
    };
    void print() const           // #11
    {
    }
};

};

```

Обычные члены code в точке #4 и print() в точке #5 обобщенного шаблона Outer #1 имеют единственный охватывающий шаблон класса и, следовательно, требуют одного префикса template<> для полной специализации для конкретного набора аргументов шаблона:

```

template<>
int Outer<void>::code = 12;

template<>
void Outer<void>::print() const
{
    std::cout << "Outer<void>";
}

```

Эти определения используются поверх обобщенных в точках #4 и #5 для класса Outer<void>, однако другие члены класса Outer<void> все еще генерируются из шаблона в точке #1. Заметим, что после этих объявлений он утрачивает силу в плане обеспечения явной специализации для Outer<void>.

Как и в случае полной специализации шаблона функции, нам нужен способ объявления специализации обычного члена шаблона класса без указания определения (чтобы избежать многократных определений). Хотя для функций-членов и статических членов-данных обычных классов в C++ не разрешены неопределяющие объявления вне класса, они будут корректными при специализации членов шаблонов классов. Предыдущие определения могут быть объявлены следующим образом:

```

template<>
int Outer<void>::code;

template<>
void Outer<void>::print() const;

```

Внимательный читатель может заметить, что неопределяющее объявление полной специализации Outer<void>::code имеет точно тот же синтаксис, что и при определении с помощью конструктора по умолчанию. Это действительно так, однако такие объявления всегда интерпретируются как неопределяющие. Для полной специализации статического члена-данных с типом, который может быть инициализирован только с помощью конструктора по умолчанию, мы должны прибегнуть к синтаксису списка инициализаторов. При наличии

```

class DefaultInitOnly
{
public:
    DefaultInitOnly() = default;

```

```

DefaultInitOnly(DefaultInitOnly const&) = delete;
};

template<typename T>
class Statics
{
private:
    static T sm;
};

```

следующий код является объявлением:

```
template<>
DefaultInitOnly Statics<DefaultInitOnly>::sm;
```

в то время как код, приведенный ниже, является определением, которое вызывает конструктор по умолчанию:

```
template<>
DefaultInitOnly Statics<DefaultInitOnly>::sm{};
```

До C++11 это было невозможно. Таким образом, инициализация по умолчанию была для таких специализаций недоступной. Обычно использовался инициализатор, копирующий значение по умолчанию:

```
template<>
DefaultInitOnly Statics<DefaultInitOnly>::sm = DefaultInitOnly();
```

К сожалению, в нашем примере это было невозможно, потому что копирующий конструктор удален. Однако в C++17 введено обязательное к исполнению правило *пропуска копирования* (copy-elision), которое делает эту альтернативу корректной — вызова копирующего конструктора здесь больше нет.

Шаблон члена `Outer<T>::Inner` также можно специализировать для данного аргумента шаблона без влияния на другие члены `Outer<T>`, для которого специализируется шаблон члена. И вновь наличие охватывающего шаблона приводит к необходимости префикса `template<>`. В результате получается такой код:

```

template<>
template<typename X>
class Outer<wchar_t>::Inner
{
public:
    static long count; // Тип члена изменен
};

template<>
template<typename X>
long Outer<wchar_t>::Inner<X>::count;

```

Шаблон `Outer<T>::Inner` также может быть полностью специализированным, однако только для данного экземпляра `Outer<T>`. Теперь нам нужно два префикса `template<>`: один из-за охватывающего класса, и один из-за того, что мы полностью специализируем (внутренний) шаблон.

```

template<>
template<>
class Outer<char>::Inner<wchar_t>
{
public:
    enum { count = 1 };
};

// Приведенный далее код не является корректным кодом C++:
// template<> не может следовать за списком параметров шаблона
template<typename X>
template<> class Outer<X>::Inner<void>; // Ошибка

```

Сравните это со специализацией шаблона-члена `Outer<bool>`. Поскольку он уже полностью специализирован, охватывающего шаблона нет и нам нужен только один префикс `template<>`.

```

template<>
class Outer<bool>::Inner<wchar_t>
{
public:
    enum { count = 2 };
};

```

16.4. Частичная специализация шаблона класса

Полная специализация шаблона часто полезна, но иногда естественным оказывается желание специализировать шаблон класса для семейства аргументов шаблона, а не только для конкретного набора аргументов. Например, предположим, что у нас есть шаблон класса, реализующий связанный список:

```

template<typename T>
class List           // #1
{
public:
    ...
    void append(T const&);
    inline std::size_t length() const;
    ...
};

```

Большой проект с использованием этого шаблона может инстанцировать свои члены для многих типов. Для невстраиваемых функций-членов (скажем, `List<T>::append()`) это может вызывать заметный рост объектного кода. Однако с низкоуровневой точки зрения код `List<int*>::append()` и код `List<void*>::append()` идентичны. Иными словами, хотелось бы указать, что все списки указателей используют одну и ту же реализацию. Хотя это нельзя выразить кодом C++, можно приблизиться к цели, отметив, что все списки указателей `List` должны инстанцироваться из другого определения шаблона:

```

template<typename T>
class List<T*>           // #2
{

```

```

private:
    List<void*> impl;
    ...
public:
    ...
    inline void append(T* p)
    {
        impl.append(p);
    }
    inline std::size_t length() const
    {
        return impl.length();
    }
    ...
};

};

```

В этом контексте исходный шаблон в точке #1 называется *первичным шаблоном*, а следующее за ним определение — *частичной специализацией* (поскольку аргументы шаблона, для которых это определение шаблона должно быть использовано, указаны только частично). Синтаксис, который характеризует частичную специализацию, является комбинацией объявления списка параметров шаблона (`template<...>`) и множества явно указанных аргументов шаблона с именем шаблона класса (в нашем примере это `<T*>`).

Наш код таит в себе проблему, поскольку `List<void*>` рекурсивно содержит член того же типа `List<void*>`. Для прерывания рекурсии перед предыдущей частичной специализацией можно ввести полную специализацию.

```

template<>
class List<void*>           // #3
{
    ...
    void append(void* p);
    inline std::size_t length() const;
    ...
};

```

Этот код вполне корректно работает, поскольку полная специализация предпочтительнее частичной. В результате все функции-члены списков указателей `List` переадресовываются (с помощью легко встраиваемых функций) реализации списка `List<void*>`. Это эффективный способ борьбы с так называемым *разбуханием кода* (code bloat, в чем часто обвиняют шаблоны C++).

Существует ряд ограничений на объявления списков параметров и аргументов частичной специализации. Приведем некоторые из них.

1. Аргументы частичной специализации должны отвечать виду (параметры, представляющие собой тип, параметры, не являющиеся типом, шаблонные параметры) соответствующих параметров первичного шаблона.
2. Список параметров частичной специализации не может иметь аргументов по умолчанию; вместо них используются аргументы по умолчанию первичного шаблона класса.

3. Аргументы частичной специализации, не являющиеся типами, должны быть либо независимыми значениями, либо простыми параметрами, не являющимися типами. Они не могут быть сложными зависимыми выражениями наподобие 2^N (где N — параметр шаблона).
4. Список аргументов шаблона частичной специализации не должен быть идентичен (без учета переименования) списку параметров первичного шаблона.
5. Если один из аргументов шаблона представляет собой раскрытие пакета, он должен находиться в конце списка аргументов шаблона.

Вот пример, иллюстрирующий описанные ограничения:

```
template<typename T, int I = 3>
class S;                                // Первичный шаблон

template<typename T>
class S<int, T>;                      // Ошибка: несоответствие вида параметра

template<typename T = int>
class S<T, 10>;                      // Ошибка: аргументы по умолчанию
                                         //      не разрешены

template<int I>
class S<int, I * 2>;                  // Ошибка: не разрешены выражения,
                                         //      не являющиеся типами

template<typename U, int K>
class S<U, K>;                      // Ошибка: нет существенных отличий
                                         //      от первичного шаблона

template<typename... Ts>
class Tuple;

template<typename Tail, typename... Ts>
class Tuple<Ts..., Tail>; // Ошибка: раскрытие пакета не в конце

template<typename Tail, typename... Ts>
class Tuple<Tuple<Ts...>, Tail>; // OK: раскрытие пакета — в конце вложенного списка аргументов шаблона
```

Любая частичная специализация, как и любая полная специализация, связана с первичным шаблоном. При использовании шаблона сначала всегда ищется первичный шаблон, но затем проверяется соответствие аргументов аргументам связанных специализаций (с использованием вывода аргументов шаблона, описанного в главе 15, “Вывод аргументов шаблона”) для определения того, какая реализация шаблона должна быть выбрана. Здесь так же, как и при выводе аргументов шаблона функции, применим принцип SFINAE: если при попытке сопоставления частичной специализации образуется недопустимая конструкция, такая специализация молча отбрасывается и проверяется другой кандидат (если таковой имеется). Если соответствующие специализации не найдены, выбирается первичный шаблон. Если найдено несколько соответствующих специализаций, из

них выбирается наиболее специализированная (в смысле, определенном для перегруженных шаблонов функций). Если ни одну из них нельзя назвать “наиболее специализированной”, в программе имеется ошибка неоднозначности.

Наконец, следует заметить, что частичная специализация шаблона класса вполне может иметь большее или меньшее количество параметров, чем первичный шаблон. Рассмотрим снова наш обобщенный шаблон `List` (объявленный в точке #1). Мы уже рассматривали оптимизацию списка указателей, но у нас может возникнуть желание сделать то же самое и с определенными типами “указатель на член”. Ниже приведен код, который оптимизирует список для указателей на указатель на член:

```
// Частичная специализация для любого члена-указателя на void*
template<typename C>
class List<void* C::*> // #4
{
public:
    using ElementType = void* C::*;
    ...
    void append(ElementType pm);
    inline std::size_t length() const;
    ...
};

// Частичная специализация для любого типа указателя на
// указатель на член, за исключением члена типа указателя
// на void*, который обработан ранее (заметим, что у этой
// частичной специализации два шаблонных параметра, тогда
// как у первичного шаблона - только один).
//
// Эта специализация использует предыдущую для достижения
// желаемой оптимизации
template<typename T, typename C>
class List<T* C::*> // #5
{
private:
    List<void* C::*> impl;
    ...
public:
    using ElementType = T * C::*;
    ...
    inline void append(ElementType pm)
    {
        impl.append(static_cast<void* C::*>(pm));
    }
    inline std::size_t length() const
    {
        return impl.length();
    }
    ...
};

};

В дополнение к замечаниям относительно числа шаблонных параметров отметим, что общая реализация, определенная в точке #4, которую используют все
```

остальные из объявления в точке #5, сама является частичной специализацией (для случая простого указателя — это полная специализация). Однако очевидно, что специализация в точке #4 более специализирована, чем в точке #5, так что неоднозначности не возникает.

Кроме того, возможно даже, что количество явно указанных *аргументов* шаблона может отличаться от количества параметров первичного шаблона. Это может произойти как с аргументами шаблона по умолчанию, так и гораздо более полезным способом с вариативными шаблонами:

```
template<typename... Elements>
class Tuple;                                // Первичный шаблон

template<typename T1>
class Tuple<T>;                            // Одноэлементный кортеж

template<typename T1, typename T2, typename... Rest>
class Tuple<T1, T2, Rest...>; // Кортеж с двумя и более элементов
```

16.5. Частичная специализация шаблонов переменных

Когда в стандарт C++11 были добавлены шаблоны переменных, оказались упущенными из виду некоторые аспекты их спецификации, и ряд этих вопросов до сих пор официально не разрешен. Однако фактические реализации языка обычно согласованы в плане обработки таких вопросов.

Пожалуй, самое удивительное в том, что стандарт ссылается на возможность частичной специализации шаблонов переменных, но не описывает, как ее объявить или что она означает. Таким образом, текст этого раздела основан на практических реализациях C++ (которые разрешают такие частичные специализации), а не на стандарте C++.

Как можно было бы ожидать, синтаксис частичной специализации шаблона переменной похож на полную специализацию переменной шаблона, за исключением того, что `template<>` заменяется на заголовок фактического объявления шаблона, а список аргументов шаблона, следующий за именем шаблона переменной, должен зависеть от параметров шаблона. Например:

```
template<typename T> constexpr std::size_t SZ = sizeof(T);
template<typename T> constexpr std::size_t SZ<T*> = sizeof(void*);
```

Как и в случае полной специализации шаблонов переменных, тип частичной специализации не обязан соответствовать типу первичного шаблона:

```
template<typename T> typename T::iterator null_iterator;
template<typename T, std::size_t N> T* null_iterator<T[N]> = nullptr;
// T* не соответствует T::iterator, но все работает
```

Правила, касающиеся видов аргументов шаблона, которые могут указываться в частичной специализации шаблона переменной, идентичны правилам для специализации шаблона класса. Аналогично правила для выбора специализации для данного списка конкретных аргументов шаблона также идентичны.

16.6. Заключительные замечания

Полная специализация шаблона была частью механизма шаблонов C++ с самого начала. Перегрузка шаблона функции и частичная специализация шаблона класса появились значительно позже. Компилятор HP aC++ стал первым компилятором, реализовавшим перегрузку шаблона функции, а EDG C++ первым реализовал частичную специализацию шаблона класса. Принципы частичного упорядочения, описанные в этой главе, первоначально были разработаны Стивом Адамчиком (Steve Adamczyk) и Джоном Спайсером (John Spicer) (оба из EDG).

Возможность применения специализаций шаблонов для завершения рекурсивного шаблонного определения (как в примере с `List<T*>`, приведенном в разделе 16.4) известна уже давно. Однако Эрвин Анрюх (Erwin Unruh), вероятно, первый заметил, что это приводит к интересной концепции *шаблонного метапрограммирования*: использованию механизма инстанцирования шаблонов для выполнения нетривиальных вычислений во время компиляции. Данной теме посвящена глава 23, “Метапрограммирование”.

Возникает вполне резонный вопрос: почему частично специализировать можно только шаблоны классов и шаблоны переменных? Причины этого в основном исторические. Вероятно, можно определить такой же механизм и для шаблонов функций (см. главу 17, “Дальнейшее развитие”). В ряде аспектов тот же эффект достигается путем перегрузки шаблонов функций, но здесь есть и некоторые тонкие отличия, в основном связанные с тем, что при использовании этого механизма осуществляется поиск только первичного шаблона. Специализации рассматриваются только впоследствии для определения того, какая именно реализация должна использоваться. В отличие от этого все перегруженные шаблоны функций должны вноситься в набор перегрузки для выполнения поиска; при этом они могут находиться в различных пространствах имен или классах. Это несколько увеличивает вероятность непреднамеренной перегрузки имени шаблона.

С другой стороны, можно представить возможность перегрузки шаблонов классов и шаблонов переменных, например:

```
// Неверная перегрузка шаблонов класса
template<typename T1, typename T2> class Pair;
template<int N1, int N2> class Pair;
```

Однако наущной необходимости использования такого механизма не видно.

Глава 17

Дальнейшее развитие

Шаблоны C++ существенно эволюционировали со времени их первоначального появления в 1988 году, пройдя через различные этапы стандартизации в 1998, 2011, 2014 и 2017 годах. Можно сказать, что большинство добавлений в стандарт после разработки стандарта 1998 было связано с шаблонами.

В первом издании этой книги перечислен ряд расширений, которые мы могли бы увидеть в последующих стандартах; некоторые из них стали реальностью.

- Проблема с угловыми скобками: в C++11 убрана необходимость вставки пробела между двумя закрывающими угловыми скобками.
- Аргументы шаблонов функций по умолчанию: C++11 позволяет шаблонам функций иметь аргументы шаблонов по умолчанию.
- Шаблоны `typedef`: в C++11 в стандарт введены подобные по свойствам шаблоны псевдонимов.
- Оператор `typeof`: в C++11 введен оператор `decltype`, который играет ту же роль (но использует иное имя во избежание конфликтов с существующим расширением, которое не вполне отвечает нуждам сообщества программистов C++).
- Статические свойства: в первом издании предполагается выбор свойств типов, непосредственно поддерживаемых компиляторами. Это сделано, хотя интерфейс выражается с использованием стандартной библиотеки (которая для многих свойств реализуется с помощью расширений компилятора).
- Пользовательская диагностика инстанцирования: новое ключевое слово `static_assert` реализует идею, изложенную в описании `std::instantiation_error` в первом издании данной книги.
- Параметры-списки: реализованы как *пакеты параметров* в C++11.
- Управление размещением в памяти: операторы C++11 `alignof` и `alignas` охватывают нужды программистов, описанные в первом издании книги. Кроме того, в стандартную библиотеку C++17 добавлен шаблон `std::variant`, поддерживающий размеченные объединения.
- Вывод на основе инициализаторов: в C++17 добавлен вывод аргументов шаблонов классов, который решает те же вопросы.
- Функциональные выражения: лямбда-выражения C++11 в точности обеспечивают данную функциональность (с синтаксисом, несколько отличным от обсуждавшегося в первом издании).

Прочие предположительные направления развития, о которых писалось в первом издании, в современном языке отсутствуют, но большинство из них времена от времени по-прежнему обсуждаются, так что мы оставим их и в этой книге. Тем временем появляются новые идеи, некоторые из которых также будут представлены далее.

17.1. Ослабленные правила применения `typename`

В первом издании книги в этом разделе было высказано предположение, что будущее может принести два вида ослабления правил использования `typename` (см. раздел 13.3.2): разрешить применение `typename` там, где оно не было разрешено ранее, и сделать необязательным использование `typename` там, где компилятор может относительно легко определить, что квалифицированное имя с зависимым квалификатором должно быть именем типа. Первое предположение сбылось (`typename` в C++11 может избыточно использоваться во многих местах), а последнее — нет.

Недавно, однако, наблюдалась новая попытка сделать `typename` необязательным в различных распространенных контекстах, где однозначно предполагается спецификатор типа.

- Тип возвращаемого значения и типы параметров функций в объявлениях функций и функций-членов в областях видимости пространств имен и классов. То же относится и к шаблонам функций и функций-членов, а также лямбда-выражений в любой области видимости.
- Типы в объявлениях переменных, шаблонов переменных и статических членов-данных. То же самое относится и к шаблонам переменных.
- Тип после токена `=` в объявлении псевдонима или шаблона псевдонима.
- Аргумент по умолчанию параметра типа шаблона.
- Тип в угловых скобках после конструкций `static_cast`, `const_cast`, `dynamic_cast` или `reinterpret_cast`.
- Тип, именованный в выражении `new`.

Хотя это не слишком продуманный список, оказывается, что такие изменения в языке позволили бы устраниТЬ большинство применений `typename`, что сделало бы код более компактным и удобочитаемым.

17.2. Обобщенные параметры шаблонов, не являющиеся типами

Пожалуй, и для начинающих, и для опытных программистов, пишущих шаблоны, самое удивительное из всех ограничений на аргументы шаблонов, не являющиеся типами, состоит в том, что строковый литерал невозможно использовать в качестве аргумента шаблона.

Приведенный ниже пример интуитивно понятен:

```
template<char const* msg>
class Diagnoser
{
public:
    void print();
};
```

```
int main()
{
    Diagnoser<"Surprise!">().print();
}
```

Однако здесь есть ряд потенциальных проблем. В стандарте C++ два экземпляра класса `Diagnoser` будут принадлежать одному и тому же типу тогда и только тогда, когда у них одни и те же аргументы. В данном случае аргумент является указателем, другими словами — адресом. Однако у двух идентичных строковых литералов, расположенных в разных местах исходного кода, не обязательно должен быть один и тот же адрес. Таким образом, можно оказаться в неловкой ситуации, когда классы `Diagnoser<"X">` и `Diagnoser<"X">` будут представлять два разных несовместимых типа! (Обратите внимание на то, что "X" имеет тип `char const[2]`, но когда он передается в качестве аргумента шаблона, его тип низводится до `char const*`.)

Исходя из этих (и других, связанных с ними) соображений, в стандарте C++ запрещено использовать строковые литералы в качестве аргументов шаблонов. Однако в некоторых реализациях такая возможность существует в виде расширения. Это обеспечивается использованием содержимого строковых литералов во внутреннем представлении экземпляра шаблона. Хотя это вполне осуществимо, некоторые толкователи языка C++ полагают, что не являющийся типом параметр шаблона, который может замещаться строковым литералом, должен объявляться иначе, чем параметр, который может замещаться адресом. Одной из возможностей является захват строковых литералов в пакет параметров символов, например:

```
template<char... msg>
class Diagnoser
{
public:
    void print();
};

int main()
{
    // Инстанцирует Diagnoser<'S', 'u', 'r', 'p', 'r', 'i', 's', 'e', '!'>
    Diagnoser<"Surprise!">().print();
}
```

Кроме того, следует отметить, что в данном вопросе кроется один дополнительный технический недостаток. Рассмотрим следующие объявления шаблонов и предположим, что язык расширен так, чтобы строковые литералы могли использоваться в качестве аргументов шаблонов:

```
template<char const* str>
class Bracket
{
public:
    static char const* address();
    static char const* bytes();
};
```

```

template<char const* str>
char const* Bracket<str>::address()
{
    return str;
}

template<char const* str>
char const* Bracket<str>::bytes()
{
    return str;
}

```

В этом коде две функции-члена идентичны во всем, кроме своих имен, — ситуация не такая уж и редкая. Предположим, что некоторая реализация инстанцирует `Bracket<"X">` с помощью процесса, подобного макрорасширению: тогда, если эти две функции-члена инстанцируются в разных единицах трансляции, они могут возвращать разные значения. Интересно, что тестирование некоторых компиляторов языка C++, имеющих данное расширение, показало, что они обладают таким удивительным поведением.

С этим вопросом связан и вопрос о возможности использования литералов с плавающей точкой (и простых константных выражений с плавающей точкой) в качестве аргументов шаблонов, например:

```

template<double Ratio>
class Converter
{
public:
    static double convert(double val)
    {
        return val * Ratio;
    }
};

using InchToMeter = Converter<0.0254>;

```

Эта возможность также имеется в некоторых реализациях языка C++, и она не представляет собой серьезной технической задачи (в отличие от использования строковых литералов в качестве аргументов).

В C++11 вводится понятие *литерального типа класса*: тип класса, который может принимать константное значение, вычисляемое во время компиляции (включая нетривиальные вычисления с использованием `constexpr`-функций). После того как такие типы классов стали доступны, тут же стало желательно позволить им быть параметрами шаблонов, не являющимися типами. Однако при этом возникают проблемы, аналогичные описанным выше проблемам, связанным с использованием в качестве параметров строковых литералов. В частности, “равенство” двух значений типа класса является отнюдь не тривиальным вопросом, потому что в общем случае оно определяется с помощью оператора `operator==`. Это равенство определяет, эквивалентны ли два экземпляра, но на практике такая эквивалентность должна проверяться компоновщиком путем сравнения декорированных имен. Одним из выходов может быть вариант, при котором определенные литеральные классы помечаются как имеющие тривиальный критерий

равенства, обеспечивающий попарное сравнение скалярных членов класса, после чего в качестве параметров шаблонов, не являющихся типами, могут использоваться только типы классов с таким тривиальным критерием равенства.

17.3. Частичная специализация шаблонов функций

В главе 16, “Специализация и перегрузка”, отмечалось, что шаблоны классов можно частично специализировать, тогда как шаблоны функций просто перегружают. Эти два механизма различаются между собой.

При частичной специализации не создается полностью новый шаблон: это просто расширение существующего (*первичного*) шаблона. Когда выбирается шаблон класса, сначала рассматриваются только первичные шаблоны. Если после выбора первичного шаблона оказывается, что существует его частичная специализация с аргументами шаблона, соответствующими данному инстанцированию, *его определение* (или, другими словами, *его тело*) инстанцируется вместо определения первичного шаблона (при полной специализации шаблона все выполняется точно так же).

Перегруженные шаблоны функций, напротив, являются отдельными шаблонами, полностью независимыми друг от друга. Когда компилятор решает, какой именно шаблон инстанцировать, он рассматривает все перегруженные шаблоны и выбирает наиболее подходящий. На первый взгляд такой подход кажется вполне адекватным, однако на практике имеется ряд ограничений.

- Можно специализировать шаблоны — члены класса без изменения определения этого класса. Однако добавление перегруженного члена требует изменения в определении класса. Во многих случаях этот вариант невозможен, так как у программиста может не быть на это прав. Более того, существующий стандарт языка C++ не позволяет программистам добавлять новые шаблоны в пространство имен `std`, но позволяет специализировать шаблоны из этого пространства имен.
- Чтобы перегрузка шаблонов функций была возможна, параметры этих функций должны различаться каким-то существенным образом. Рассмотрим шаблон функции `R convert(T const&)`, где `R` и `T` — параметры шаблона. Этот шаблон вполне можно специализировать для `R = void`, но с помощью перегрузки этого сделать нельзя.
- Код, который корректен для неперегруженной функции, может перестать быть корректным, когда эта функция перегружается. В частности, если есть два шаблона функций `f(T)` и `g(T)` (где `T` — параметр шаблона), выражение `g(&f<int>)` корректно только в случае, если функция `f` не перегружена (в противном случае будет невозможно решить, какая именно функция `f` имеется в виду).
- Дружественные объявления относятся к определенному шаблону функции или инстанцированию определенного шаблона функции. Перегруженная версия шаблона функции не будет автоматически иметь привилегии, которыми обладает исходный шаблон.

В целом этот список представляет собой убедительный аргумент в пользу частичной специализации шаблонов функций.

Естественным синтаксисом частичной специализации шаблонов функций является обобщение записи для шаблона класса.

```
template<typename T>
T const& max(T const&, T const&); // Первичный шаблон

template<typename T>
T* const& max <T*>(T* const&, T* const&); // Частичная специализация
```

Некоторых разработчиков языка беспокоит взаимодействие такого подхода к частичной специализации и перегрузки шаблонов функций. Например:

```
template<typename T>
void add(T& x, int i); // Первичный шаблон

template<typename T1, typename T2>
void add(T1 a, T2 b); // Другой (перегруженный) первичный шаблон

template<typename T>
void add<T*>(T*&, int); // Какой из первичных шаблонов специализирован?
```

Однако мы ожидаем, что такие ошибки не окажут значительного влияния на полезность данной функциональной возможности.

Это расширение вкратце обсуждалось во время работы над стандартом C++11, но в итоге собрало относительно мало голосов. Тем не менее эта тема время от времени возрождается, потому что такое нововведение могло бы аккуратно решать некоторые общие проблемы программирования. Возможно, к ее рассмотрению вновь вернутся в некоторых будущих версиях стандарта C++.

17.4. Именованные аргументы шаблонов

В разделе 21.4 описывается методика, позволяющая предоставлять для определенного параметра аргумент шаблона, не используемый по умолчанию, без необходимости задавать другие аргументы шаблона, для которых используется значение по умолчанию. Это интересная методика, но ясно, что она требует значительного объема работы для достижения относительно простого результата. Поэтому вполне естественным представляется создание механизма для именования аргументов шаблонов.

Здесь следует отметить, что в процессе стандартизации языка C++ подобное расширение (иногда называемое *ключевыми аргументами*) предлагалось ранее Роландом Хартингером (Roland Hartinger) (см. раздел 6.5.1 в [66]). Это предложение, хотя и вполне обоснованное технически, в конечном итоге не было принято по ряду причин. В настоящее время нет никаких оснований полагать, что именованные аргументы шаблонов когда-нибудь попадут в язык, хотя эта тема регулярно поднимается на заседаниях Комитета.

Однако для полноты картины упомянем об одной синтаксической идеи, которая бродила в умах некоторых разработчиков.

```

template<typename T,
         typename Move = defaultMove<T>,
         typename Copy = defaultCopy<T>,
         typename Swap = defaultSwap<T>,
         typename Init = defaultInit<T>,
         typename Kill = defaultKill<T>>
class Mutator
{
    ...
};

void test(MatrixList ml)
{
    mySort(ml, Mutator <Matrix, .Swap = matrixSwap>);
}

```

Здесь точка, предшествующая имени аргумента, используется для указания, что мы ссылаемся на аргумент шаблона по имени. Этот синтаксис похож на синтаксис инициализации членов структур, введенный в стандарт языка С в 1999 году:

```

struct Rectangle
{
    int top, left, width, height;
};

struct Rectangle r = { .width = 10, .height = 10, .top = 0, .left = 0 };

```

Конечно, введение именованных аргументов шаблона означает, что имена параметров шаблона теперь являются частью общего интерфейса данного шаблона и не могут быть свободно изменены. Этую проблему можно решить путем более явного синтаксиса, например, такого:

```

template<typename T,
         Move: typename M = defaultMove<T>,
         Copy: typename C = defaultCopy<T>,
         Swap: typename S = defaultSwap<T>,
         Init: typename I = defaultInit<T>,
         Kill: typename K = defaultKill<T>>
class Mutator
{
    ...
};

void test(MatrixList ml)
{
    mySort(ml, Mutator <Matrix, .Swap = matrixSwap>);
}

```

17.5. Перегруженные шаблоны классов

Вполне представима перегрузка шаблонов классов на основе их параметров шаблонов. Например, можно представить создание семейства шаблонов `Array`, которые содержат массивы как с динамическими, так и со статическими размерами:

```

template<typename T>
class Array
{
    // массив с динамическим размером
    ***
};

template<typename T, unsigned Size>
class Array
{
    // Массив с фиксированным размером
    ***
};

```

Перегрузка не обязательно ограничена количеством параметров шаблона; может также варьироваться и вид параметров:

```

template<typename T1, typename T2>
class Pair
{
    // Пара полей
    ***
};

template<int I1, int I2>
class Pair
{
    // Пара константных целочисленных значений
    ***
};

```

Хотя неофициально эта идея обсуждалась некоторыми разработчиками языка, однако официально она все еще не была представлена на рассмотрение Комитета по стандартизации языка C++.

17.6. Вывод неконечных раскрытий пакетов

Вывод аргумента шаблона для раскрытия пакета работает только в том случае, когда раскрытие пакета происходит в конце списка параметров или аргументов. Это означает, что, хотя довольно просто получить первый элемент из списка:

```

template<typename... Types>
struct Front;

template<typename FrontT, typename... Types>
struct Front<FrontT, Types...>
{
    using Type = FrontT;
};

```

извлечь последний элемент из списка невозможно из-за ограничений, накладываемых на частичные специализации (см. раздел 16.4):

```

template<typename... Types>
struct Back;

```

```
template<typename BackT, typename... Types>
struct Back<Types..., BackT> // Ошибка: раскрытие пакета не в
{                                // конце списка аргументов шаблона
    using Type = BackT;
};
```

Вывод аргумента шаблона для вариативных шаблонов функций ограничен аналогично. Вполне вероятно, что правила, касающиеся вывода аргумента шаблона раскрытия пакета и частичных специализаций, будут смягчены и станут допускать раскрытие пакета в любом месте списка аргументов шаблона, что сделает операции этого вида гораздо проще. Кроме того, возможно (хотя и менее вероятно), что вывод будет позволять несколько раскрытий пакетов в пределах одного и того же списка параметров:

```
template<typename... Types> class Tuple
{
};

template<typename T, typename... Types>
struct Split;

template<typename T, typename... Before, typename... After>
struct Split<T, Before..., T, After...>
{
    using before = Tuple<Before...>;
    using after = Tuple<After...>;
};
```

Разрешение нескольких раскрытий пакетов привносит дополнительные сложности. Например, должен ли `Split` отделять первое вхождение `T`, последнее его вхождение или некоторое между ними? Насколько сложным станет процесс вывода, прежде чем компилятору будет позволено от него отказаться?

17.7. Регуляризация `void`

При программировании шаблонов регулярность является добродетелью: если единственная конструкция может охватывать все случаи, это делает наш шаблон проще. Одним из нескольких иррегулярных аспектов наших программ являются *типы*. Например, рассмотрим следующий код:

```
auto&& r = f();           // Ошибка, если f() возвращает void
```

Это выражение работает почти для любого типа, который возвращает `f()`, за исключением `void`. Так же проблема возникает и при использовании `decltype(auto)`:

```
decltype(auto) r = f(); // Ошибка, если f() возвращает void
```

`void` является не единственным иррегулярным типом: типы функций и ссылочные типы также часто демонстрируют поведение, делающее их в том или ином отношении исключительными. Однако оказывается, что `void` часто осложняет наши шаблоны, и что нет никаких глубоких причин для того, чтобы `void` был таким необычным. Например, в разделе 11.1.3 имеется пример того, как это осложняет реализацию идеальной оболочки `std::invoke()`.

Можно было бы просто сказать, что `void` — это нормальный тип значения с уникальным значением (как тип `std::nullptr_t` для значения `nullptr`). Для обеспечения обратной совместимости нам бы по-прежнему пришлось поддерживать частный случай для объявлений функций наподобие следующих:

```
void g(void); // То же, что и void g();
```

Однако в большинстве других применений `void` стал бы полным типом значений. Тогда мы могли бы, например, объявлять переменные и ссылки `void`:

```
void v = void{};  
void&& rrv = f();
```

И, что более существенно, многие шаблоны больше не требовали бы специализаций для случая `void`.

17.8. Проверка типов для шаблонов

Большая часть сложности программирования шаблонов связана с неспособностью компилятора локально проверить правильность определения шаблона. Вместо этого большинство проверок шаблонов выполняется уже во время инстанцирования, когда контекст определения шаблона и контекст инстанцирования шаблона тесно переплетаются. Это смешение различных контекстов затрудняет выяснение того, кто виноват: виновно ли определение шаблона, потому что неверно использует свои аргументы шаблона, или виновен пользователь шаблона, предоставивший аргументы, не отвечающие требованиям шаблона? Эту проблему можно проиллюстрировать простым примером, который мы аннотировали диагностикой, генерируемой типичным компилятором:

```
template<typename T>  
T max(T a, T b)  
{  
    return b < a ? a : b; // Ошибка: "нет соответствующего оператора  
                         // operator < (типы оператора 'X' и 'X')"  
}  
  
struct X  
{  
};  
  
bool operator> (X, X);  
  
int main()  
{  
    X a, b;  
    X m = max(a, b); // Примечание: "в запрошенном здесь  
                      // инстанцировании специализации шаблона функции 'max<X>'"  
}
```

Обратите внимание на то, что фактическая ошибка (отсутствие корректного оператора `operator>`) обнаруживается внутри определения шаблона функции `max()`. Возможно, это истинная ошибка — может быть, функция `max()` должна

использовать вместо него оператор `operator>`? Однако компилятор выводит также примечание в месте, которое приводит к инстанцированию `max<X>`, которое и может быть реальной ошибкой — возможно, шаблон функции `max()` документирован как требующий наличия оператора `operator<`? Именно неспособность ответить на этот вопрос часто приводит к “роману ошибок”, описанному в разделе 9.4, где компилятор предоставил всю историю инстанцирования шаблона — от исходного шаблона до определения шаблона, в котором была обнаружена ошибка. Как оказывается, программист после этого сможет определить, какое из определений шаблонов (или, возможно, исходное использование шаблона в программе) на самом деле содержит ошибку.

Идея, лежащая в основе проверки типов шаблонов, состоит в описании требований шаблона в самом шаблоне, таким образом, чтобы компилятор мог определить, является причиной сбоя компиляции определение или использование шаблона. Одним из решений этой проблемы является описание требований шаблона как часть его сигнатуры с помощью *концепта*:

```
template<typename T> requires LessThanComparable<T>
T max(T a, T b)
{
    return b < a ? a : b;
}

struct X { };

bool operator> (X, X);

int main()
{
    X a, b;
    X m = max(a,b); // Ошибка: X не отвечает
} // требованию LessThanComparable
```

С помощью описания требования к параметру шаблона `T` компилятор способен гарантировать, что шаблон функции `max()` использует только те операции над `T`, наличие которых пользователь, как ожидается, должен обеспечить (в данном случае `LessThanComparable` описывает потребность в операторе `operator<`). Кроме того, при использовании шаблона компилятор может проверить, что переданный аргумент шаблона предоставляет все необходимое для правильной работы шаблона функции `max()` поведение. Путем разделения проблемы проверки типов для компилятора становится гораздо проще обеспечить точный диагноз проблемы.

В приведенном выше примере `LessThanComparable` называется *концептом*: он представляет собой ограничения, накладываемые на тип (в более общем случае, ограничения на множество типов), которые компилятор может проверить. Системы концептов могут быть спроектированы по-разному.

В процессе цикла стандартизации C++11 была полностью спроектирована и реализована сложная система концептов, которые оказались достаточно мощными для выполнения проверок как точек инстанцирования, так и определений

шаблона. Первое для нашего примера выше означает, что ошибка в функции `main()` может быть обнаружена очень рано, с диагностикой, которая объясняет, что тип `X` не удовлетворяет ограничениям `LessThanComparable`. Последнее же означает, что при обработке шаблона функции `max()` компилятором он проверяет, что не используется ни одна операция, которая не разрешена концептом `LessThanComparable` (и при нарушении этого ограничения выводится соответствующая диагностика). В конечном итоге это предложение в стандарт C++11 было удалено из спецификации языка исходя из различных практических соображений (например, оставалось много незначительных вопросов спецификации, решение которых угрожало затянуться, при том что принятие стандарта и так запаздывало).

После выхода C++11 членами Комитета было разработано и представлено новое предложение (сначала называвшееся *concepts lite*). Эта система не ставила целью проверку правильности шаблонов на основе присоединенных к нему ограничений. Вместо этого они сосредоточивались только на точке инстанцирования. Так что если в нашем примере `max()` был реализован с использованием оператора `>`, сообщение об ошибке в этот момент выводиться не будет. Однако оно по-прежнему будет выводиться в функции `main()`, потому что тип `X` не удовлетворяет ограничениям `LessThanComparable`. Новое предложение было реализовано и специфицировано под названием *Concepts TS* (где *TS* означало *техническую спецификацию*), и окончательно получило название *Расширения C++ для концептов*¹. В настоящее время основные элементы этой технической спецификации интегрированы в проект следующего стандарта (ожидается, что им станет C++20). Приложение Д, “Концепты”, описывает эту функциональную возможность языка в том виде, в котором она подана в проект языка на момент печати данной книги.

17.9. Рефлексивное метапрограммирование

В контексте программирования *рефлексия* (*reflection*) относится к способности программно инспектировать функциональные возможности самой программы (например, получить ответы на вопросы: “Это целочисленный тип?” “Какие нестатические члены-данные содержит этот класс?”). Метапрограммирование – это возможность “программирования программы”, которая обычно важна для программной генерации нового кода. *Рефлексивное метапрограммирование* предоставляет возможность автоматически генерировать код, который приспособливается к существующим свойствам (часто – типам) программы.

В части III, “Шаблоны и проектирование”, этой книги мы будем изучать, как шаблоны позволяют достичь некоторых простых форм рефлексии и метапрограммирования (в определенном смысле инстанцирование шаблона является одной из форм метапрограммирования, поскольку приводит к генерации нового кода). Однако возможности шаблонов C++17 являются весьма ограниченными,

¹ C++ extensions for Concepts. См., например, версию *Concepts TS* в документе N4641.

когда дело доходит до рефлексии (например, не представляется возможным ответить на вопрос, какие нестатические члены-данные содержит некоторый тип класса), и способы метaprogramмирования оказываются крайне неудобными (в частности, синтаксис становится громоздким, а производительность — разочаровывающей).

Признавая потенциал новых объектов в этой области, Комитет по стандартизации C++ создал исследовательскую группу (SG7) для изучения возможностей для более мощных рефлексий. Позже деятельность группы была распространена также и на метaprogramмирование. Вот пример одного из рассматриваемых группой вариантов:

```
template<typename T> void report(T p)
{
    constexpr
    {
        std::meta::info infoT = reflexpr(T);

        for (std::meta::info : std::meta::data_members(infoT))
        {
            -> {
                std::cout << (: std::meta::name(info) :)
                << ":" << p.(.info.) << '\n';
            }
        }
    }
    // Код будет внедрен сюда
}
```

В этом коде присутствует довольно много новых вещей. Во-первых, конструкция `constexpr{...}` обеспечивает вычисление инструкций во время компиляции, но если она находится в шаблоне, то это вычисление выполняется только при инстанцировании шаблона. Во-вторых, оператор `reflexpr()` генерирует выражение непрозрачного типа `std::meta::info`, которое является дескриптором рефлексивной информации о его аргументе (тип, в этом примере представленный вместо `T`). Библиотека стандартных мetaфункций разрешает запрос этой метаинформации. Одной из таких стандартных мetaфункций является `std::meta::data_members`, которая создает последовательность объектов `std::meta::info`, описывающих непосредственные нестатические члены-данные своего операнда. Так что показанный выше цикл `for` действительно является циклом по всем нестатическим членам-данным `p`.

В основе метaprogramмирования в этой системе лежит возможность “внедрения” кода в различные области видимости. Конструкция `->{...}` внедряет инструкции и/или объявления сразу после инструкции или объявления, которые вызывают вычисление `constexpr`. В приведенном примере это означает — после конструкции `constexpr{...}`. Фрагменты внедряемого кода могут содержать определенные схемы, заменяемые вычисляемыми значениями. В нашем примере `(:...:)` создает значение строкового литерала (выражение `std::meta::name(info)` производит строковый объект, представляющий

неквалифицированное имя сущности (в данном случае — член-данные), представленный именем `info`). Аналогично выражение `(.info.)` создает идентификатор, именующий сущность, представленную именем `info`. Предлагаются также и другие шаблоны для генерации типов, списков аргументов шаблона, и т.д.

С учетом всего сказанного инстанцирование шаблона функции `report()` для типа

```
struct X
{
    int x;
    std::string s;
};
```

будет давать инстанцирование наподобие

```
template<> void report(X const& p)
{
    std::cout << "x" << ":" << "p.x" << '\n';
    std::cout << "s" << ":" << "p.s" << '\n';
}
```

Т.е. эта функция автоматически генерирует функцию для вывода значений нестатических членов-данных типа класса.

Для этих возможностей существует множество приложений. Хотя вполне вероятно, что-то вроде этого в конечном итоге войдет в стандарт языка, неясно, когда именно можно этого ожидать. Тем не менее на момент написания книги было продемонстрировано несколько экспериментальных реализаций таких систем. (Перед самой передачей этой книги в печать группа SG7 согласовала общее направление использования вычислений `constexpr` и типов значений наподобие `std::meta::info` для работы с рефлексивным метапрограммированием. Однако представленный здесь механизм внедрения согласован не был и, скорее всего, будет реализована другая система.)

17.10. Работа с пакетами

Пакеты параметров были введены в стандарт C++11, но работа с ними часто требует применения рекурсивных технологий инстанцирования шаблонов. Вернемся к наброску кода, который рассматривался в разделе 14.6:

```
template<typename Head, typename... Remainder>
void f(Head&& h, Remainder&& ... r)
{
    doSomething(h);

    if constexpr(sizeof...(r) != 0)
    {
        // Рекурсивная обработка остальных параметров
        // (прямая передача аргументов):
        f(r...);
    }
}
```

Этот пример можно сделать более простым, используя такую функциональную возможность C++17, как `if` времени компиляции (см. раздел 8.5), но метод рекурсивного инстанцирования остается в силе, что может оказаться чрезмерно дорогим для компиляции.

Несколько предложений Комитета направлены на упрощение текущего положения дел. Одним из примеров таких предложений является введение обозначений, позволяющих выбирать определенный элемент из пакета. В частности, для пакета `P` в качестве способа обозначения $N+1$ -го элемента этого пакета предлагалась запись `P. [N]`. Были сделаны и другие предложения, в частности, для обозначения “срезов” пакетов (например, с использованием обозначений `P. [b, e]`).

При изучении таких предложений становится понятно, что они в определенной мере взаимодействуют с рассматривавшимся выше рефлексивным метапрограммированием. На данный момент неясно, будут ли добавлены в стандарт некоторые механизмы выбора пакетов, или вместо этого данная потребность будет удовлетворяться с помощью возможностей метапрограммирования.

17.11. Модули

Еще одно грядущее крупное расширение — *модули*, которые слабо связаны с шаблонами, но которые стоит здесь отметить, потому что наибольшую выгоду от них получат библиотеки шаблонов.

В настоящее время интерфейсы библиотек указываются в *заголовочных файлах*, которые текстуально включаются (с помощью директив `#include`) в единицы трансляции. У этого подхода имеется ряд недостатков, но два наиболее нежелательных таковы: 1) значение текста интерфейса может быть случайно изменено ранее включенным кодом (например, макросами) и 2) время повторной обработки начинает быстро доминировать во времени построения.

Модули представляют собой функциональную возможность, которая позволяет скомпилировать библиотеку интерфейсов в формате, специфичном для данного компилятора, а затем эти интерфейсы можно “импортировать” в единицы трансляции, не опасаясь влияния макросов или модификации значения кода из-за случайного добавления объявления. Более того, компилятор может обеспечить чтение только тех частей файла скомпилированного модуля, которые имеют отношение к клиентскому коду, и тем самым резко ускорить процесс компиляции.

Вот как может выглядеть определение модуля:

```
module MyLib;

void helper()
{
    ...
}

export inline void libFunc()
{
    ...
}
```

```
    helper();  
    ...  
}
```

Этот модуль экспортирует функцию `libFunc()`, которая может быть использована в клиентском коде следующим образом:

```
import MyLib;  
  
int main()  
{  
    libFunc();  
}
```

Заметим, что `libFunc()` становится видимой для клиентского кода, но функция `helper()` таковой не является, несмотря на то, что файл скомпилированного модуля, скорее всего, будет содержать информацию о функции `helper()` для обеспечения возможности встраивания.

Предложение добавить модули в C++ разумно, и Комитет по стандартизации планирует его интеграцию после C++17. Одной из проблем в разработке такого предложения является переход от мира заголовочных файлов в мир модулей. Уже имеются средства, как в определенной степени это позволяющие (например, возможность включать заголовочные файлы, не делая их содержимое частью модуля), так и все еще находящиеся на стадии обсуждения (например, возможность экспортить макросы из модулей).

Модули особенно полезны для библиотек шаблонов, потому что шаблоны почти всегда полностью определены в заголовочных файлах. Даже включение базового стандартного заголовочного файла наподобие `<vector>` приводит к обработке десятков тысяч строк кода на языке C++ (даже при очень малом количестве ссылок на объявление в этом заголовочном файле). У ряда популярных библиотек эта величина больше на порядок. Избежать расходов на компиляцию всего этого кода — мечта программистов на C++, работающих с большими и сложными проектами.

ЧАСТЬ III

Шаблоны и проектирование

Обычно программы создаются с использованием конструкций, которые относительно хорошо отражают возможности механизмов, предлагаемых выбранным языком программирования. Поскольку шаблоны являются совершенно новым механизмом языка, неудивительно, если они будут вызывать к жизни новые элементы проектирования. Данная часть книги посвящена изучению именно этих элементов.

Шаблоны отличаются от более традиционных конструкций языка тем, что позволяют параметризовать типы и константы нашего кода. Их комбинирование с 1) частичной специализацией и 2) рекурсивным инстанцированием приводит к необычайному увеличению выразительности языка.

Наша представление материала ставит целью не только продемонстрировать различные полезные элементы в листингах, но и рассказать о принципах проектирования, обеспечивающих создание новых методов. В следующих главах будет проиллюстрировано большое количество методов проектирования, в частности, такие как

- расширенная полиморфная диспетчеризация;
- обобщенное программирование с использованием свойств;
- работа с перегрузкой и наследованием;
- метапрограммирование;
- гетерогенные структуры и алгоритмы;
- шаблоны выражений.

Мы также представим некоторые материалы, которые помогут вам в отладке шаблонов.

Глава 18

Полиморфная мощь шаблонов

Полиморфизм представляет собой способность связывать различные специфические виды поведения с помощью единой общей записи¹. Кроме того, полиморфизм является краеугольным камнем парадигмы объектно-ориентированного программирования, которая в C++ поддерживается главным образом через наследование свойств классов и виртуальные функции. Поскольку этот механизм (по крайней мере частично) работает во время выполнения программы, можно употребить термин *динамический полиморфизм*. Обычно так говорят, когда речь идет об обычном полиморфизме в C++. Однако шаблоны также позволяют связывать различные специфические виды поведения единой общей записью, но это связывание обрабатывается, как правило, в процессе компиляции, так что в данном случае следует говорить о *статическом полиморфизме*. В этой главе приводится обзор обоих вариантов полиморфизма и обсуждается вопрос, какой из них соответствует той или иной конкретной ситуации.

Обратите внимание на то, что в главе 22, “Статический и динамический полиморфизм”, обсуждаются некоторые способы работы с полиморфизмом после введения и обсуждения ряда вопросов проектирования.

18.1. Динамический полиморфизм

Исторически сложилось так, что язык C++ начался с поддержки полиморфизма только посредством наследования, объединенного с виртуальными функциями². В этом контексте искусство полиморфного дизайна состоит в идентификации общего набора возможностей среди связанных типов объектов и объявлении их в качестве интерфейсов виртуальных функций в общем базовом классе.

Наглядным примером этого подхода к конструированию является приложение, которое управляет построением геометрических фигур с возможностью их воспроизведения определенным способом (например, на экране). В таком приложении можно указать так называемый *абстрактный базовый класс* (abstract base class — ABC) `GeoObj`, который объявляет общие операции и свойства, применимые к геометрическим объектам вообще. Каждый конкретный класс для конкретных геометрических объектов будет затем порождаться из абстрактного базового класса `GeoObj` (рис. 18.1).

¹ Полиморфизм буквально означает условие существования нескольких форм или видов (от греческого *polymorphos*).

² Строго говоря, макросы также могут рассматриваться как ранняя форма статического полиморфизма. Однако они остаются за пределами данного обсуждения, поскольку обычно никак не связаны с другими механизмами языка.

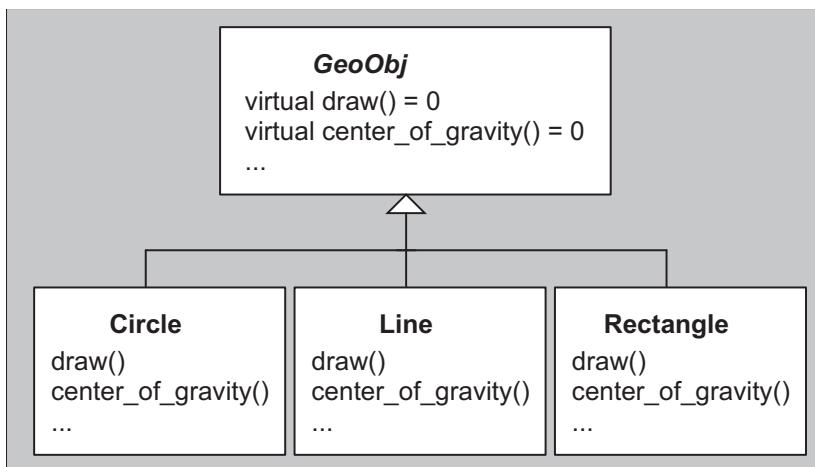


Рис. 18.1. Полиморфизм, реализованный с помощью наследования

poly/dynahier.hpp

```

#include "coord.hpp"

// Общий абстрактный базовый класс GeoObj
// для геометрических объектов
class GeoObj
{
public:
    // Черчение геометрического объекта:
    virtual void draw() const = 0;
    // Возврат центра масс геометрического объекта:
    virtual Coord center_of_gravity() const = 0;
    ...
    virtual ~GeoObj() = default;
};

// Конкретный класс геометрического объекта Circle
// - унаследован от GeoObj
class Circle : public GeoObj
{
public:
    virtual void draw() const override;
    virtual Coord center_of_gravity() const override;
    ...
};

// Конкретный класс геометрического объекта Line
// - унаследован от GeoObj
class Line : public GeoObj
{
public:
    virtual void draw() const override;
    virtual Coord center_of_gravity() const override;
    ...
};
...

```

После создания конкретных объектов клиентский код может управлять этими объектами через ссылки или указатели на базовый класс, который дает возможность задействовать механизм диспетчеризации виртуальных функций. В результате вызова виртуальной функции-члена посредством указателя или ссылки на подобъект базового класса происходит вызов соответствующего члена объекта, на который осуществлялась ссылка.

В нашем примере конкретный код может выглядеть, как показано ниже.

poly/dynapoly.cpp

```
#include "dynahier.hpp"
#include <vector>

// Чертение любого GeoObj
void myDraw(GeoObj const& obj)
{
    obj.draw();           // Вызов draw() в соответствии
}                           // с типом объекта

// Вычисление расстояния между центрами масс двух GeoObjs
Coord distance(GeoObj const& x1, GeoObj const& x2)
{
    Coord c = x1.center_of_gravity() - x2.center_of_gravity();
    return c.abs();        // Возврат абсолютного значения
}

// Чертение неоднородного набора объектов GeoObjs
void drawElems(std::vector<GeoObj*> const& elems)
{
    for (std::size_type i = 0; i < elems.size(); ++i)
    {
        elems[i]->draw(); // Вызов draw() в соответствии
    }                     // с типом элемента
}

int main()
{
    Line l;
    Circle c, c1, c2;
    myDraw(l);           // myDraw(GeoObj&) => Line::draw()
    myDraw(c);           // myDraw(GeoObj&) => Circle::draw()
    distance(c1, c2);   // distance(GeoObj&, GeoObj&)
    distance(l, c);     // distance(GeoObj&, GeoObj&)
    std::vector<GeoObj*> coll; // Неоднородная коллекция
    coll.push_back(&l);   // Вставка линии
    coll.push_back(&c);   // Вставка окружности
    drawElems(coll);     // Чертение разных GeoObjs
}
```

Ключевыми элементами полиморфного интерфейса являются функции `draw()` и `center_of_gravity()`. Обе функции являются виртуальными функциями-членами. В нашем примере продемонстрировано их использование в функциях `myDraw()`, `distance()` и `drawElems()`. Последние три функции записаны с использованием общего базового типа `GeoObj`. Вследствие этого в процессе

компиляции нельзя определить, какая именно версия функции `draw()` или `center_of_gravity()` должна использоваться. Однако в процессе выполнения программы при диспетчеризации вызовов функций определяется полный динамический тип объектов, для которых вызываются виртуальные функции³. Следовательно, соответствующая операция выполняется в зависимости от фактического типа геометрического объекта: если `myDraw()` вызывается для объекта `Line`, то выражение `obj.draw()` вызывает функцию `Line::draw()`, тогда как для объекта `Circle` вызывается функция `Circle::draw()`. Подобным же образом в вызове `distance()` функции-члены `center_of_gravity()` соответствуют переданным в качестве параметров объектам.

Пожалуй, наиболее впечатляющей возможностью динамического полиморфизма является способность обрабатывать разнородные коллекции объектов. Эта концепция иллюстрируется функцией `drawElems()`; простое выражение `elems[i] -> draw()`

выполняет вызов разных функций-членов, в зависимости от типа итерируемого элемента.

18.2. Статический полиморфизм

Шаблоны также могут использоваться для реализации полиморфизма. Однако они не зависят от фактора общего поведения, свойственного базовым классам. Вместо этого общность подразумевает поддержку операций с использованием общего синтаксиса (т.е. соответствующие функции имеют одни и те же имена). Конкретные классы при этом определяются независимо друг от друга (рис. 18.2), а сам полиморфизм проявляется при инстанцировании шаблонов с конкретными классами.

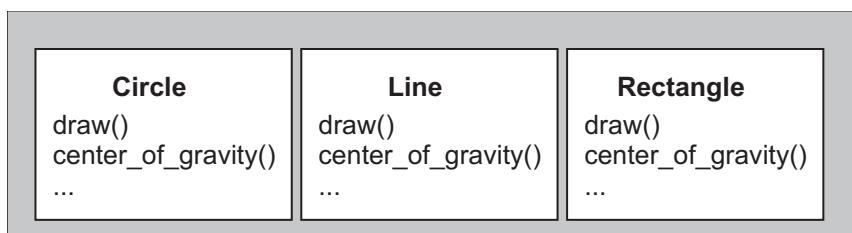


Рис. 18.2. Полиморфизм, реализованный с помощью шаблонов

Например, функция `myDraw()` из предыдущего раздела

```
void myDraw (GeoObj const& obj) // GeoObj – абстрактный базовый класс
{
    obj.draw();
}
```

³ Т.е. кодировка подобъектов полиморфного базового класса включает некоторые (в основном скрытые) данные, которые обеспечивают такую диспетчеризацию времени выполнения.

может быть переписана следующим образом:

```
template<typename GeoObj>
void myDraw (GeoObj const& obj) // GeoObj - параметр шаблона
{
    obj.draw();
}
```

Сравнивая эти две реализации функции `myDraw()`, можно видеть, что основное различие состоит в указании `GeoObj` в качестве параметра шаблона вместо указания в качестве общего базового класса. Имеются, однако, и более существенные различия. Например, при использовании динамического полиморфизма в процессе выполнения у нас была только одна функция `myDraw()`, тогда как, применяя шаблон, мы имеем различные функции, такие как `myDraw<Line>()` и `myDraw<Circle>()`.

Можно попытаться переписать весь пример из предыдущего раздела с использованием статического полиморфизма. При этом вместо иерархии геометрических классов у нас появится несколько индивидуальных геометрических классов.

poly/statichier.hpp

```
#include "coord.hpp"

// Конкретный класс геометрического объекта Circle
// - не являющийся производным от какого-либо иного класса
class Circle
{
public:
    void draw() const;
    Coord center_of_gravity() const;
    ...
};

// Конкретный класс геометрического объекта Line
// - не являющийся производным от какого-либо иного класса
class Line
{
public:
    void draw() const;
    Coord center_of_gravity() const;
    ...
};
...
```

Теперь применение этих классов имеет следующий вид:

poly/staticpoly.cpp

```
#include "statichier.hpp"
#include <vector>

// Чертение любого GeoObj
template<typename GeoObj>
void myDraw(GeoObj const& obj)
```

```

{
    obj.draw();           // Вызов draw() в соответствии с типом объекта
}

// Вычисление расстояния между центрами масс двух GeoObjs
template<typename GeoObj1, typename GeoObj2>
Coord distance(GeoObj1 const& x1, GeoObj2 const& x2)
{
    Coord c = x1.center_of_gravity() - x2.center_of_gravity();
    return c.abs(); // Возврат абсолютного значения
}
// Чертение однородного набора объектов GeoObjs
template<typename GeoObj>
void drawElems(std::vector<GeoObj> const& elems)
{
    for (unsigned i = 0; i < elems.size(); ++i)
    {
        elems[i].draw(); // Вызов draw() в соответствии
                          // с типом объекта
    }
}

int main()
{
    Line l;
    Circle c, c1, c2;
    myDraw(l); // myDraw<Line>(GeoObj&) => Line::draw()
    myDraw(c); // myDraw<Circle>(GeoObj&) => Circle::draw()
    distance(c1, c2); // distance<Circle, Circle>(GeoObj1&, GeoObj2&)
    distance(l, c); // distance<Line, Circle>(GeoObj1&, GeoObj2&)
    // std::vector<GeoObj*> coll; // Ошибка: гетерогенная
    //                                // коллекция невозможна
    std::vector<Line> coll; // OK: гомогенная коллекция
    coll.push_back(l); // Вставка линии
    drawElems(coll); // Чертение всех линий
}

```

Тип `GeoObj` больше не может использоваться в качестве конкретного параметра типа как для функции `distance()`, так и в функции `myDraw()`. Вместо этого в функции `distance()` предусмотрены два параметра шаблона — `GeoObj1` и `GeoObj2`. Два разных параметра шаблона позволяют вычислять расстояние между разными типами геометрических объектов:

```
distance(l, c); // distance<Line, Circle>(GeoObj1&, GeoObj2&)
```

Теперь, однако, разнородные коллекции больше не могут быть прозрачно обработаны. Это тот случай, когда *статическая часть статического полиморфизма* налагает свои ограничения, а именно: все типы должны быть определены во время компиляции. Взамен предоставляется возможность легко вводить разные коллекции для различных типов геометрических объектов; к тому же больше не требуется, чтобы коллекция была ограничена указателями, что дает существенные преимущества в аспекте производительности и безопасности типов.

18.3. Сравнение динамического и статического полиморфизма

А теперь классифицируем и сравним обе формы полиморфизма.

Терминология

Динамический и статический полиморфизм обеспечивает поддержку различных идиом языка программирования C++⁴.

- Полиморфизм, реализованный с использованием наследования, является *ограниченным*, или *связанным* (bounded) и *динамическим* (dynamic).
 - Термин *ограниченный* означает, что интерфейсы типов, участвующих в процессе полиморфизма, предопределены дизайном общего базового класса (другими терминами для обозначения данной концепции являются *инвазивный* (invasive) или *интрузивный* (intrusive)).
 - Термин *динамический* означает, что связывание интерфейсов происходит в процессе выполнения программы (т.е. динамически).
- Полиморфизм, реализованный с использованием шаблонов, является *неограниченным* (unbounded) и *статическим* (static).
 - Термин *неограниченный* означает, что интерфейсы типов, участвующих в процессе полиморфизма, не предопределены заранее (другими терминами для обозначения данной концепции являются *неинвазивный* (noninvasive) или *неинтрузивный* (nonintrusive)).
 - Термин *статический* означает, что связывание интерфейсов происходит в процессе компиляции (т.е. статически).

Строго говоря, в терминах языка C++ понятия *динамический полиморфизм* и *статический полиморфизм* – это сокращенные варианты понятий *ограниченный динамический полиморфизм* и *неограниченный статический полиморфизм*. В других языках используются иные комбинации (например, Smalltalk предоставляет неограниченный динамический полиморфизм). Однако более краткие термины *динамический полиморфизм* и *статический полиморфизм* в контексте языка C++ не приводят к возникновению путаницы.

Преимущества и недостатки

Динамический полиморфизм в C++ обладает рядом преимуществ.

- Элегантная обработка разнородных коллекций.
- Размер исполняемого кода потенциально меньше (поскольку в данном случае нужна только одна полиморфная функция, тогда как для шаблонов с разными параметрами типов должны быть сгенерированы отдельные экземпляры).

⁴ С терминологией, касающейся полиморфизма, более детально можно ознакомиться в [34], разделы 6.5–6.7.

- Код полностью компилируем; таким образом, исходные тексты не обязательно должны быть опубликованы (распространение библиотек шаблонов обычно требует распространения исходного кода реализации шаблонов).

Приведем преимущества статического полиморфизма в C++.

- Легко реализуются коллекции встроенных типов. Общность интерфейса не обязательно должна выражаться через общий базовый класс.
- Сгенерированный код потенциально выполняется быстрее (поскольку отсутствует необходимость в косвенном обращении через указатели, а невиртуальные функции могут быть встраиваемыми намного чаще).
- Могут использоваться конкретные типы, в которых имеются только частичные интерфейсы (только если приложение ограничивается использованием этого частичного интерфейса).

Часто статический полиморфизм расценивается как более надежный в плане *безопасности типов*, чем динамический, поскольку все связывания выполняются в процессе компиляции. Например, опасность того, что в контейнер, реализованный шаблоном, будет вставлен объект неправильного типа, крайне мала; в то же время в контейнере, который содержит указатели на общий базовый класс, существует возможность непреднамеренного использования указателей на объекты совершенно иного типа.

На практике инстанцирование шаблонов может вызвать определенные неприятности в том случае, когда за идентично выглядящими интерфейсами скрываются разные семантические допущения. Например, неприятные сюрпризы могут произойти тогда, когда шаблон предполагает наличие ассоциативного оператора + у типа, который таким оператором не обладает. Обычно этот вид семантического несоответствия встречается гораздо реже в иерархиях, основанных на наследовании; вероятно, это связано с более явным и точным определением интерфейса.

Объединение обеих форм

Конечно, можно совместить обе формы наследования. Например, различные виды геометрических объектов можно порождать из общего базового класса, для того чтобы иметь возможность обрабатывать неоднородные коллекции геометрических объектов. Однако одновременно можно использовать и шаблоны в целях написания кода для некоторого отдельного вида геометрического объекта.

Комбинация наследования и шаблонов описана в главе 21, “Шаблоны и наследование”. В ней рассматривается (помимо прочего), как может быть параметризована виртуальность функции-члена и как можно предоставить дополнительную гибкость статическому полиморфизму, используя основанную на наследовании модель *необычного рекуррентного шаблона* (*curiously recurring template pattern – CRTP*).

18.4. Применение концептов

Одним из аргументов против статического полиморфизма с шаблонами является то, что связывание интерфейсов выполняется путем инстанцирования соответствующих шаблонов. Это означает, что не существует некоторого общего интерфейса (класса), а работает любое использование шаблона, лишь бы весь инстанцированный код был корректен. Если же это не так, то это может привести к трудно понимаемым сообщениям об ошибках или даже вызвать корректное, но нежелательное поведение.

По этой причине разработчики языка C++ работали над возможностью явно предоставлять (и проверять) *интерфейсы* для параметров шаблонов. Обычно такой интерфейс называется в C++ *концептом*. Он обозначает набор ограничений, которым должны отвечать аргументы шаблона для того, чтобы инстанцирование шаблона было успешным.

Несмотря на многие годы работы в этой области концепты по-прежнему не являются частью стандарта C++ — по крайней мере до стандарта C++17 включительно. Некоторые компиляторы предоставляют экспериментальную поддержку такой возможности⁵, так что концепты, вероятно, станут частью следующего после C++17 стандарта.

Концепты можно рассматривать как своего рода “интерфейс” для статического полиморфизма. В нашем примере это может выглядеть следующим образом:

poly/conceptsreq.hpp

```
#include "coord.hpp"

template<typename T>
concept GeoObj = requires(T x)
{
    {
        x.draw()
    } -> void;
    {
        x.center_of_gravity()
    } -> Coord;
    ...
};
```

Здесь ключевое слово *concept* использовано для определения концепта *GeoObj*, которая ограничивает тип наличием вызываемых членов *draw()* и *center_of_gravity()* с соответствующими возвращаемыми типами.

Теперь можно переписать некоторые из наших примеров шаблонов, включив конструкцию *requires*, которая ограничивает параметры шаблона концептом *GeoObj*:

⁵ Например, у GCC 7 имеется параметр командной строки *-fconcepts*.

poly/conceptspoly.hpp

```
#include "conceptsreq.hpp"
#include <vector>

// Чертение любого GeoObj
template<typename T>
requires GeoObj<T>
void myDraw(T const& obj)
{
    obj.draw();      // Вызов draw() в соответствии с типом объекта
}

// Вычисление расстояния между центрами масс между двумя GeoObjs
template<typename T1, typename T2>
requires GeoObj<T1>&& GeoObj<T2>
Coord distance(T1 const& x1, T2 const& x2)
{
    Coord c = x1.center_of_gravity() - x2.center_of_gravity();
    return c.abs(); // Возврат абсолютного значения
}

// Чертение однородной коллекции GeoObjs
template<typename T>
requires GeoObj<T>
void drawElems(std::vector<T> const& elems)
{
    for (std::size_type i = 0; i < elems.size(); ++i)
    {
        elems[i].draw(); // Вызов draw() в соответствии
                         // с типом элемента
    }
}
```

Этот подход по-прежнему неинвазивный по отношению к типам, которые могут участвовать в (статическом) полиморфном поведении:

```
// Конкретный класс геометрического объекта Circle
// - не является производным от другого класса
// и не реализующий никакой интерфейс
class Circle
{
public:
    void draw() const;
    Coord center_of_gravity() const;
    ...
};
```

То есть такие типы по-прежнему определяются без каких-либо конкретных базовых классов или конструкций требований и могут быть фундаментальными типами данных или типов из независимых каркасов.

В приложении Д, “Концепты”, концепты в C++ обсуждаются более подробно, поскольку ожидается, что они войдут в следующий стандарт языка.

18.5. Новые виды проектных шаблонов

Следствием использования новой формы статического полиморфизма являются новые пути реализации проектных шаблонов. Возьмем, например, шаблон “Мост” (Bridge pattern), который играет большую роль в программах на C++. Одна из задач использования этого проектного шаблона состоит в переключении между различными реализациями интерфейса.

Согласно [35] обычно это переключение осуществляется с использованием класса интерфейса, который содержит указатель для обращения к фактической реализации и делегирования всех обращений к этому классу (рис. 18.3).

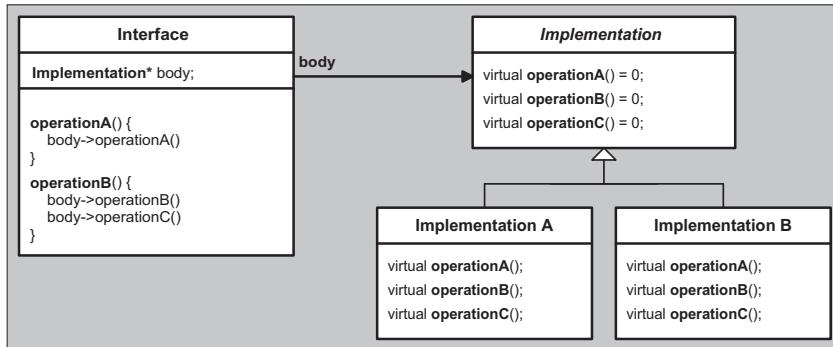


Рис. 18.3. Проектный шаблон “Мост”, реализованный с использованием наследования

Однако если тип реализации известен во время компиляции, то вместо этого можно использовать подход с применением шаблонов (рис. 18.4). Это приведет к большей безопасности типов и позволит избежать использования указателей, что должно способствовать более высокой производительности программы.

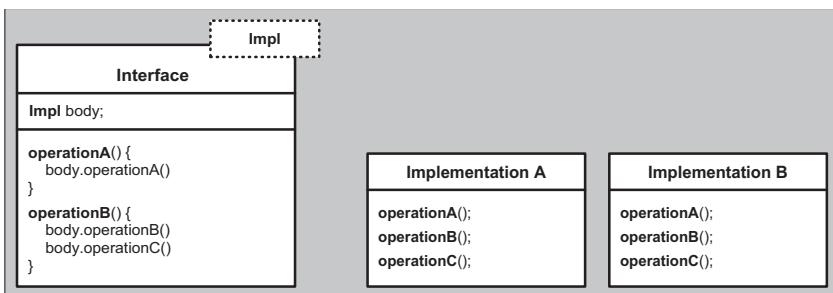


Рис. 18.4. Проектный шаблон “Мост”, реализованный с использованием шаблонов

18.6. Обобщенное программирование

Статический полиморфизм порождает концепцию *обобщенного программирования* (generic programming). Однако единого универсального установившегося определения этого понятия не существует (как не существует и единого установившегося определения понятия *объектно-ориентированного программирования*).

Согласно [34] имеются определения от *программирования с обобщенными параметрами* (programming with generic parameters) до *поиска наиболее абстрактного представления эффективных алгоритмов* (finding the most abstract representation of efficient algorithms). Книга резюмирует:

Обобщенное программирование – это поддисциплина информатики, которая имеет дело с поиском абстрактных представлений эффективных алгоритмов, структур данных и других понятий программного обеспечения, вместе с организацией их систематики... Обобщенное программирование сосредоточивает внимание на представлении семейств концепций доменов (С. 169–170).

В контексте C++ обобщенное программирование иногда определяется как *программирование с шаблонами* (programming with templates), а объектно-ориентированное программирование рассматривается как *программирование с виртуальными функциями* (programming with virtual functions). В этом смысле почти любое использование шаблонов в C++ можно рассматривать как пример обобщенного программирования. Однако практикующие программисты часто рассматривают обобщенное программирование как имеющее дополнительный существенный компонент, а именно: шаблоны должны конструироваться в целях предоставления большого числа полезных комбинаций.

Наиболее значительный вклад в этой области принадлежит *стандартной библиотеке шаблонов* (Standard Template Library – STL), которая позже была адаптирована и включена в стандартную библиотеку C++. STL является каркасом, который предоставляет большое количество полезных операций, называемых *алгоритмами*, для ряда линейных структур данных для хранения коллекции объектов (*контейнеров*). И алгоритмы и контейнеры являются шаблонами; однако ключевой момент состоит в том, что алгоритмы *не являются* функциями-членами контейнеров. Они написаны *обобщенным* способом, так что их можно использовать с любым контейнером (и линейной коллекцией элементов). Для обеспечения такого использования проектировщики STL определили абстрактное понятие *итераторов*, которые могут быть предоставлены для любого вида линейной коллекции. По существу, аспекты функционирования контейнера, специфические для данной коллекции, оказались переложенными на функциональность итераторов.

Вследствие этого операция наподобие вычисления максимального значения в последовательности может быть выполнена без знания того, каким образом в этой последовательности хранятся значения.

```
template<typename Iterator>
Iterator max_element(Iterator beg, // Начало коллекции
                     Iterator end) // Конец коллекции
{
    // Используются определенные операции итератора
    // для обхода всех элементов коллекции с целью
    // поиска элемента с максимальным значением и
    // возврата его позиции посредством итератора
    ...
}
```

Вместо того чтобы обеспечить полезными операциями, подобными `max_element()`, каждый из линейных контейнеров, контейнер должен предоставить итератор для обхода всех содержащихся в нем значений, а также функции-члены, необходимые для создания таких итераторов.

```
namespace std
{
    template<typename T, ...>
    class vector
    {
        public:
            using const_iterator = ...; // Зависящий от реализации
            ...                         // итератор для константных
                                         // векторов
            const_iterator begin() const; // Итератор начала коллекции
            const_iterator end() const;  // Итератор конца коллекции
            ...
    };

    template<typename T, ...>
    class list
    {
        public:
            using const_iterator = ...; // Зависящий от реализации
            ...                         // итератор для константных
                                         // списков
            const_iterator begin() const; // Итератор начала коллекции
            const_iterator end() const;  // Итератор конца коллекции
            ...
    };
}
```

Теперь можно находить максимум любой коллекции, вызывая *общенную* операцию `max_element()` с указанием начала и конца коллекции в качестве аргументов (здесь опущен частный случай пустой коллекции):

poly/printmax.cpp

```
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include "MyClass.hpp"

template<typename T>
void printMax(T const& coll)
{
    // Вычисление позиции максимального значения
    auto pos = std::max_element(coll.begin(), coll.end());

    // Вывод значения максимального элемента coll
    // (если таковой имеется):
    if (pos != coll.end())
    {
        std::cout << *pos << '\n';
    }
}
```

```

    else
    {
        std::cout << "empty" << '\n';
    }
}

int main()
{
    std::vector<MyClass> c1;
    std::list<MyClass> c2;
    ...
    printMax(c1);
    printMax(c2);
}

```

Параметризируя свои операции в терминах итераторов, STL избегает резкого увеличения количества определений операций. Вместо того чтобы реализовать каждую операцию для каждого контейнера, нужно реализовать алгоритм всего лишь один раз, после чего его можно будет использовать для каждого контейнера. *Обобщенная связка* (generic glue) — это итераторы, которые обеспечиваются контейнерами и используются алгоритмами. Этот метод работоспособен, поскольку итераторы имеют определенный интерфейс, который обеспечивается контейнерами и используется алгоритмами. Этот интерфейс обычно называется *концептом*, что обозначает набор ограничений, которым должен удовлетворять шаблон, чтобы вписаться в соответствующую схему.

Вспомните, что мы уже описывали *концепты* ранее в разделе 18.4 (более подробно они изложены в приложении Д, “Концепты”), и фактически данная возможность языка точно отображается на приведенное здесь понятие. В самом деле, термин *концепт* в данном контексте был впервые введен разработчиками STL для формализации их работы. Вскоре после этого начались работы по введению этого понятия в шаблоны явным образом.

Предлагаемая функциональная возможность языка поможет указывать и дважды проверять требования к итераторам (поскольку имеются различные категории итераторов, такие как одно- и двунаправленные итераторы, будет использоваться несколько соответствующих концептов; см. раздел D.3.1). Однако в современном C++ концепты в основном используются в спецификациях наших обобщенных библиотек (в том числе стандартной библиотеки C++) неявно. К счастью, некоторые функциональные возможности и методы (например, `static_assert` и SFINAE) позволяют выполнить часть автоматизированных проверок.

В принципе, функциональность наподобие STL-подобной может быть реализована и с использованием динамического полиморфизма. Однако на практике она имела бы ограниченное применение, поскольку концепция итераторов слишком “легковесная” по сравнению с механизмом вызова виртуальной функции. Добавление уровня интерфейса на основе виртуальных функций, скорее всего, замедлит наши операции на порядок (а то и больше).

Обобщенное программирование практически именно потому, что оно основано на статическом полиморфизме, который выполняет разрешение интерфейсов во время компиляции. С другой стороны, требование, чтобы интерфейсы разрешались во время компиляции, вызывает к жизни новые принципы проектирования, которые во многих отношениях отличаются от принципов объектно-ориентированного проектирования. Некоторые из наиболее важных из этих *принципов обобщенного проектирования* описываются в оставшейся части книги. Кроме того, обобщенное программирование подробно рассмотрено в приложении D, “Концепты”.

18.7. Заключительные замечания

Контейнерные типы были первым толчком для введения шаблонов в язык программирования C++. До шаблонов наиболее популярным подходом при разработке контейнеров были полиморфные иерархии. В качестве широко известного примера можно привести библиотеку классов Национального института здравоохранения (National Institutes of Health Class Library – NIHCL), в которой была расширена иерархия контейнеров Smalltalk (рис. 18.5).

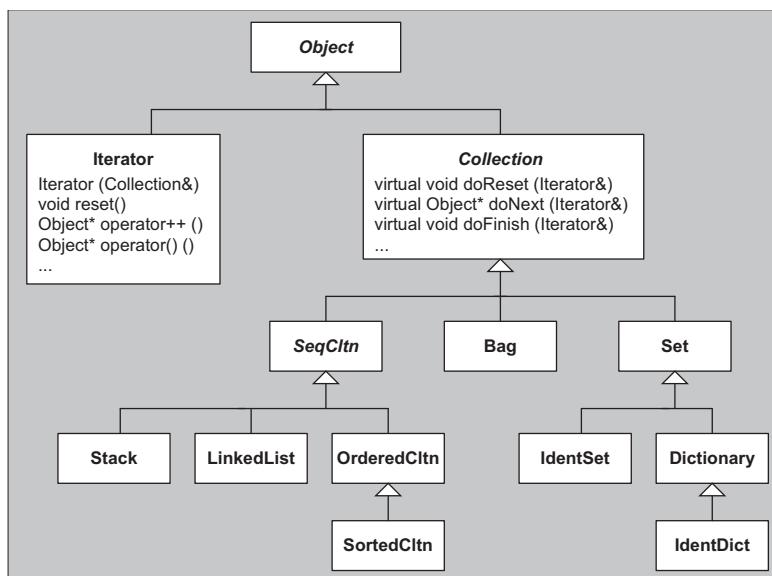


Рис. 18.5. Иерархия классов NIHCL

Во многом подобно стандартной библиотеке C++ библиотека NIHCL поддерживала широкое разнообразие контейнеров и итераторов. Однако реализация библиотеки следовала стилю динамического полиморфизма Smalltalk: для работы с коллекциями разных типов итераторы использовали абстрактный базовый класс `Collection`:

```
Bag c1;
Set c2;
...
Iterator i1(c1);
Iterator i2(c2);
...
```

К сожалению, цена такого подхода была весьма высока, это касалось как времени работы, так и используемой памяти. Обычно время работы оказывалось на порядок больше, чем у эквивалентного кода, использующего стандартную библиотеку C++, поскольку большинство операций приводили к виртуальным вызовам (в то время как в стандартной библиотеке C++ многие операции являются встраиваемыми, а в интерфейсах итераторов и контейнеров нет никаких виртуальных функций). Кроме того, поскольку (в отличие от Smalltalk) интерфейсы были ограниченными, встроенные типы должны были быть “обернутыми” в большие полиморфные классы (что и обеспечивала NIHCL), а это, в свою очередь, увеличивало потребность в памяти.

Даже в нынешнюю эпоху шаблонов во многих проектах все еще делается неоптимальный выбор в пользу использования полиморфизма. Очевидно, что существует множество ситуаций, когда следует отдать предпочтение динамическому полиморфизму (в качестве яркого примера можно привести гетерогенные коллекции). Однако ничуть не меньше задач программирования естественно и эффективно решаются с использованием шаблонов.

Использование статического полиморфизма хорошо подходит для кодирования наиболее фундаментальных вычислительных структур; необходимость же выбора общего базового типа приводит к тому, что динамическая полиморфная библиотека обычно хорошо удовлетворяет требованиям конкретной предметной области. Поэтому не должен вызывать никакого удивления тот факт, что STL-часть стандартной библиотеки C++ никогда не включала в свой состав полиморфные контейнеры, но зато содержит богатый набор контейнеров и итераторов, которые используют статический полиморфизм (как показано в разделе 18.6).

Средние и большие программы, написанные на C++, обычно работают с обобщими видами полиморфизма, рассмотренными в данной главе. Порой может даже возникнуть необходимость их весьма тесной комбинации. Во многих случаях выбор оптимального варианта проектирования в свете нашего обсуждения изначально представляется совершенно ясным, однако спустя некоторое время приходит понимание того, что здесь, как нигде, важна роль долгосрочного планирования с учетом всех возможных путей эволюции разрабатываемого проекта.

Глава 19

Реализация свойств типов

Шаблоны дают возможность параметризовать классы и функции для различных типов. Кажется весьма заманчивым вводить столько параметров шаблонов, сколько нужно для того, чтобы настроить каждый аспект поведения типа или алгоритма. Таким образом, наши “шаблонизированные” компоненты могли бы быть реализованы так, чтобы удовлетворять любым потребностям пользовательского кода. Однако с практической точки зрения нежелательно вводить большое количество параметров шаблонов для их максимально возможной параметризации. Необходимость указания всех соответствующих аргументов в пользовательском коде чрезмерно утомительна, а каждый дополнительный параметр шаблона усложняет контракт между компонентом и его клиентом.

К счастью, оказывается, что большинству дополнительных параметров можно назначить приемлемые значения по умолчанию. В ряде случаев дополнительные параметры полностью определяются несколькими *основными* параметрами и поэтому могут быть вообще опущены. Для других параметров могут быть заданы значения по умолчанию, зависящие от основных параметров, которые отвечают нуждам большинства ситуаций, но тем не менее в ряде случаев все же должны заменяться реальными значениями. Некоторые параметры оказываются не связанными с основными параметрами, и в этом смысле сами являются основными параметрами.

Свойства (*traits*), или *шаблоны свойств*, являются теми программными устройствами C++, которые значительно облегчают управление множеством дополнительных параметров, появляющихся при разработке мощных шаблонов. В этой главе приведен ряд ситуаций, в которых они доказывают свою несомненную эффективность, а также демонстрируются различные методы разработки мощных и надежных компонентов для ваших собственных программ.

Большинство представленных здесь свойств доступны в той или иной форме в стандартной библиотеке C++. Однако для ясности мы часто представляем упрощенные реализации, которые опускают некоторые детали, имеющиеся в реализациях промышленного уровня (как, например, в стандартной библиотеке). По этой причине мы также используем нашу собственную схему именования, которая, однако, легко отображается на стандартные свойства.

19.1. Пример: суммирование последовательности

Вычисление суммы последовательности значений – довольно тривиальная вычислительная задача. Однако эта простая на вид задача может служить прекрасным примером использования классов стратегий и свойств на разных уровнях.

19.1.1. Фиксированные свойства

Предположим для начала, что значения, сумму которых необходимо вычислить, хранятся в массиве, и нам заданы указатели на первый суммируемый элемент и на элемент, следующий за последним. Естественно, потребуется написать шаблон, который будет применим для различных типов. Приведенный ниже код может показаться вам очень простым¹.

traits/accum1.hpp

```
#ifndef ACCUM_HPP
#define ACCUM_HPP

template<typename T>
T accum(T const* beg, T const* end)
{
    T total{}; // Считаем, что создается нулевое значение

    while (beg != end)
    {
        total += *beg;
        ++beg;
    }

    return total;
}

#endif // ACCUM_HPP
```

Здесь есть только один тонкий момент: как создать *нулевое значение* корректного типа для начала процесса суммирования. Мы используем здесь *инициализацию значением* (с использованием записи с фигурными скобками), рассмотренную в разделе 5.2. Это означает, что объект `total` инициализируется либо с помощью своего конструктора по умолчанию, либо нулем (что означает `nullptr` для указателей и `false` для логических значений).

Рассмотрим теперь код, в котором используется наша функция `accum()`.

traits/accum1.cpp

```
#include "accum1.hpp"
#include <iostream>

int main()
{
    // Создание массива из 5 целочисленных значений
    int num[] = { 1, 2, 3, 4, 5 };
```

¹ В большинстве примеров, предлагаемых в этом разделе, ради простоты используются обычные указатели. Ясно, что в серьезной разработке может оказаться предпочтительным использование итераторов (следуя соглашениям стандартной библиотеки C++ [45]). Мы рассмотрим этот аспект несколько позже.

```

// Вывод среднего значения
std::cout << "Среднее значение для int равно "
    << accum(num, num + 5) / 5
    << '\n';

// Создание массива символов
char name[] = "templates";
int length = sizeof(name) - 1;

// (Пытаемся) вывести среднее значение
std::cout << "Среднее значение для char в \""
    << name << "\"" равно "
    << accum(name, name + length) / length
    << '\n';
}

```

В первой половине этой программы `accum()` используется для суммирования пяти целочисленных значений:

```

int num[] = { 1, 2, 3, 4, 5 };
...
accum(num, num + 5)

```

После этого полученная сумма просто делится на количество значений в массиве, что дает нам целочисленное среднее значение.

Вторая половина программы пытается сделать то же самое для всех символов в слове `templates` (рассматривая символы от `a` до `z` как непрерывную последовательность в наборе символов, что справедливо для ASCII, но не для EBCDIC²). По-видимому, результат вычисления должен находиться между значением `a` и значением `z`. В настоящее время на большинстве платформ эти значения определяются ASCII-кодами: символ `a` имеет код 97, а символ `z` — 122. Следовательно, можно предположить, что результат должен находиться где-то между 97 и 122. Однако на нашем компьютере программа выводит следующее:

```

Среднее значение для int равно 3
Среднее значение для char в "templates" равно -5

```

Проблема заключается в том, что наш шаблон был инстанцирован для типа `char`, у которого оказался слишком маленький диапазон для накопления даже относительно небольших значений. Ясно, что можно было решить эту проблему, введя дополнительный параметр шаблона `AccT`, описывающий тип, который используется для переменной `total` (и, соответственно, возвращаемый тип). Однако тем самым мы бы добавили дополнительную работу всем пользователям: они были бы вынуждены указывать этот тип при каждом обращении к шаблону; например, рассмотренный ранее код использовал бы следующий вызов функции:

```
accum<int>(name, name+5)
```

²EBCDIC (Extended Binary-Coded Decimal Interchange Code) — это расширенный двоично-десятичный код обмена информацией, который представляет собой набор символов IBM, широко используемый на больших машинах IBM.

Это не столь существенное ограничение, но и его можно избежать.

Альтернативным по отношению к применению дополнительного параметра подходом является создание связи между каждым типом T , для которого вызывается функция `accum()`, и типом, который будет использоваться для хранения накопленного значения. Эта связь может рассматриваться в качестве характеристики типа T , и поэтому тип вычисляемой суммы иногда называется *свойством* (trait) T . Эта связь может быть закодирована в виде специализации шаблона:

traits/accumtraits2.hpp

```
template<typename T>
struct AccumulationTraits;

template<>
struct AccumulationTraits<char>
{
    using AccT = int;
};

template<>
struct AccumulationTraits<short>
{
    using AccT = int;
};

template<>
struct AccumulationTraits<int>
{
    using AccT = long;
};

template<>
struct AccumulationTraits<unsigned int>
{
    using AccT = unsigned long;
};

template<>
struct AccumulationTraits<float>
{
    using AccT = double;
};
```

Шаблон `AccumulationTraits` называется *шаблоном свойств* (traits template), поскольку он хранит свойство типа своего параметра. (В общем случае в шаблоне свойств может быть как несколько свойств, так и несколько параметров.) В данном случае обобщенного определения шаблона нет, так как нет хорошего способа для выбора подходящего типа накопления в случае неизвестного исходного типа. Однако можно считать, что таким типом может быть сам тип T .

С учетом сказанного можно переписать наш шаблон `accum()`, как показано ниже³:

traits/accum2.hpp

```
#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits2.hpp"

template<typename T>
auto accum(T const* beg, T const* end)
{
    // Возвращаемый тип является свойством типа элемента
    using AccT = typename AccumulationTraits<T>::AccT;
    AccT total{}; // Считаем, что создается нулевое значение

    while (beg != end)
    {
        total += *beg;
        ++beg;
    }

    return total;
}
#endif // ACCUM_HPP
```

Вывод нашей программы становится таким, как мы и ожидали:

Среднее значение для `int` равно 3

Среднее значение для `char` в "templates" равно 108

В целом внесенные изменения не очень впечатляющи, хотя добавлен очень полезный механизм настройки нашего алгоритма. Кроме того, если появятся новые типы, предназначенные для использования с `accum()`, соответствующий тип `AccT` может быть связан с ними посредством простого объявления дополнительной явной специализации класса `AccumulationTraits`. Обратите внимание на то, что эта операция может быть выполнена для любого типа: фундаментальных типов, типов, которые объявлены в других библиотеках, и т.д.

19.1.2. Свойства-значения

До сих пор речь шла о том, что свойства предоставляют дополнительную информацию о типах, имеющую отношение к данному "основному" типу. В этом разделе показано, что такая дополнительная информация не ограничивается только типами. С типом могут быть связаны константы и другие классы значений.

Наш исходный шаблон `accum()` использует конструктор по умолчанию, возвращающий значение для инициализации переменной-результата значением, аналогичным нулевому:

```
AccT total{}; // Считаем, что создается нулевое значение
...
return total;
```

³ В C++11 необходимо объявить возвращаемый тип аналогично типу `AccT`.

Разумеется, нет никакой гарантии, что этот код обеспечивает подходящее значение, необходимое для запуска цикла накопления. Ведь тип `AccT` может даже не иметь конструктора по умолчанию.

Но и в этом случае классы свойств могут спасти ситуацию. В данном примере можно добавить к нашему классу `AccumulationTraits` новое *свойство-значение* (*value trait*):

traits/accumtraits3.hpp

```
template<typename T>
struct AccumulationTraits;

template<>
struct AccumulationTraits<char>
{
    using AccT = int;
    static AccT const zero = 0;
};

template<>
struct AccumulationTraits<short>
{
    using AccT = int;
    static AccT const zero = 0;
};

template<>
struct AccumulationTraits<int>
{
    using AccT = long;
    static AccT const zero = 0;
};

...
```

В представленном фрагменте кода нашим новым свойством является константа `zero`, которая может быть вычислена в процессе компиляции. Ниже показано, какой вид принимает при этом функция `accum()`:

traits/accum3.hpp

```
#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits3.hpp"

template<typename T>
auto accum(T const* beg, T const* end)
{
    // Возвращаемый тип является свойством типа элемента
    using AccT = typename AccumulationTraits<T>::AccT;
    AccT total =                                     // Инициализация total
              AccumulationTraits<T>::zero; // значением свойства
```

```

while (beg != end)
{
    total += *beg;
    ++beg;
}

return total;
}
#endif // ACCUM_HPP

```

В данном коде инициализация переменной для накопления результата остается очень простой:

```
AccT total = AccumulationTraits<T>::zero;
```

Недостаток этого способа состоит в том, что C++ позволяет инициализировать статический константный член-данное внутри класса, только если он имеет целочисленный или перечислимый тип.

Статические `constexpr` члены-данные являются несколько более обобщенными, допуская значения типов с плавающей точкой, а также иных литеральных типов:

```

template<>
struct AccumulationTraits<float>
{
    using AccT = float;
    static constexpr float zero = 0.0f;
};

```

Однако ни `const`, ни `constexpr` не разрешают инициализировать таким путем нелитеральные типы. Например, пользовательский тип для вычислений с произвольной точностью `BigInt` не может быть литеральным типом, поскольку он обычно должен выделять память для компонентов в динамической памяти, что препятствует литеральности его типа, или просто потому, что требуемый конструктор не является `constexpr`. Поэтому следующая специализация является ошибкой:

```

class BigInt
{
    BigInt(long long);
    ...
};

template<>
struct AccumulationTraits<BigInt>
{
    using AccT = BigInt;
    static constexpr BigInt zero =      // Ошибка:
                                    BigInt{0}; // не литеральный тип
};

```

Простейший альтернативный способ состоит в том, чтобы не определять свойство-значение в классе.

```
template<>
struct AccumulationTraits<BigInt>
{
    using AccT = BigInt;
    static BigInt const zero; // Только объявление
};
```

Затем инициализатор включается в исходный текст и выглядит примерно так:

```
BigInt const AccumulationTraits<BigInt>::zero = BigInt{0};
```

Хотя этот способ вполне работоспособен, он является более многословным (код должен быть добавлен в двух местах) и потенциально менее эффективным, поскольку компиляторам обычно ничего не известно об определениях в других файлах.

В C++17 эту проблему можно решить с использованием *встраиваемых переменных* (*inline variables*):

```
template<>
struct AccumulationTraits<BigInt>
{
    using AccT = BigInt;
    inline static BigInt const zero =
        BigInt{0}; // OK, начиная с C++17
};
```

Альтернативой, которая работает до C++17, является использование встроенных функций-членов для значений свойств, которые не всегда дают целочисленные значения. Такая функция опять же может быть объявлена как `constexpr`, если она возвращает тип литерала⁴.

Например, можно переписать `AccumulationTraits` следующим образом:

traits/accumtraits4.hpp

```
template<typename T>
struct AccumulationTraits;

template<>
struct AccumulationTraits<char>
{
    using AccT = int;
    static constexpr AccT zero()
    {
        return 0;
    }
};

template<>
struct AccumulationTraits<short>
{
    using AccT = int;
    static constexpr AccT zero()
```

⁴ Большинство современных компиляторов C++ может смотреть “сквозь” вызовы простых встраиваемых функций. Кроме того, использование `constexpr` делает возможным применение значений свойств в контекстах, где выражение должно быть константой (например, в аргументе шаблона).

```

    {
        return 0;
    }
};

template<>
struct AccumulationTraits<int>
{
    using AccT = long;
    static constexpr AccT zero()
    {
        return 0;
    }
};

template<>
struct AccumulationTraits<unsigned int>
{
    using AccT = unsigned long;
    static constexpr AccT zero()
    {
        return 0;
    }
};

template<>
struct AccumulationTraits<float>
{
    using AccT = double;
    static constexpr AccT zero()
    {
        return 0;
    }
};
...

```

После этого данные свойства можно расширить на наши собственные типы:

traits/accumtraits4bigint.hpp

```

template<>
struct AccumulationTraits<BigInt>
{
    using AccT = BigInt;
    static BigInt zero()
    {
        return BigInt{0};
    }
};

```

В коде приложения при этом появляется единственное отличие — использование синтаксиса вызова функции вместо несколько более краткого доступа к статической переменной-члену класса:

```

AccT total =                                // Инициализация total
    AccumulationTraits<T>::zero(); // функцией-свойством

```

Ясно, что свойства могут быть чем-то гораздо большим, нежели просто дополнительными *типами*. В нашем примере они могут быть механизмом, обеспечивающим функцию `accum()` всей необходимой информацией о типе элемента, для которого она вызвана. В этом состоит ключевой момент концепции свойств, а именно: свойства обеспечивают средства *настройки* конкретных элементов (обычно типов) для обобщенных вычислений.

19.1.3. Параметризованные свойства

Использование свойств в `accum()`, показанное в предыдущих разделах, называется *фиксированным*, поскольку, как только будет определен отдельный класс свойств, его будет нельзя заменить в алгоритме; хотя бывают ситуации, когда такое переопределение желательно. Например, может оказаться, что набор значений типа `float` вполне можно суммировать в переменной этого же типа, а не `double`; к тому же это приведет к некоторому повышению эффективности.

Решить эту проблему можно, добавив параметр шаблона `AT` со значением по умолчанию, определяемым нашим шаблоном свойств.

traits/accum5.hpp

```
#ifndef ACCUM_HPP
#define ACCUM_HPP
#include "accumtraits4.hpp"

template<typename T, typename AT = AccumulationTraits<T>>
auto accum(T const* beg, T const* end)
{
    typename AT::AccT total = AT::zero();
    while (beg != end)
    {
        total += *beg;
        ++beg;
    }
    return total;
}
#endif // ACCUM_HPP
```

Таким образом, многие пользователи смогут просто опустить дополнительный аргумент шаблона, но те, у кого потребности выходят за стандартные рамки, смогут указать альтернативу предопределенному типу аккумулятора. Вероятно, большинству пользователей этого шаблона никогда не придется явно указывать второй параметр шаблона, поскольку его можно настроить так, чтобы он имел подходящее значение по умолчанию для каждого типа, выводимого из первого аргумента.

19.2. Стратегии и классы стратегий

До сих пор речь шла о *накоплении* применительно к *суммированию*. Очевидно, что можно представить и другие виды накопления, а не только суммирование. Например, можно перемножать заданную последовательность значений. Или,

если значения представляют собой строки, можно просто конкатенировать эти строки. Даже поиск максимального значения последовательности можно представить как задачу накопления. Во всех этих вариантах единственная операция `accum()`, которая должна измениться, — это `total+=*beg`. Эту операцию можно назвать *стратегией* (policy) нашего процесса накопления.

Вот пример того, как мы могли бы ввести такую стратегию в наш шаблон функции `accum()`:

traits/accum6.hpp

```
#ifndef ACCUM_HPP
#define ACCUM_HPP
#include "accumtraits4.hpp"
#include "sumpolicy1.hpp"

template<typename T,
         typename Policy = SumPolicy,
         typename Traits = AccumulationTraits<T>>
auto accum(T const* beg, T const* end)
{
    using AccT = typename Traits::AccT;
    AccT total = Traits::zero();

    while (beg != end)
    {
        Policy::accumulate(total, *beg);
        ++beg;
    }

    return total;
}

#endif // ACCUM_HPP
```

В этой версии функции `accum()` имеется класс `SumPolicy` — класс стратегии, т.е. класс, реализующий одну или несколько стратегий для алгоритма через согласованный интерфейс⁵. `SumPolicy` может быть записан следующим образом:

traits/sumpolicy1.hpp

```
#ifndef SUMPOLICY_HPP
#define SUMPOLICY_HPP
class SumPolicy
{
public:
    template<typename T1, typename T2>
    static void accumulate(T1& total, T2 const& value)
    {
        total += value;
    }
};

#endif // SUMPOLICY_HPP
```

⁵ Его можно обобщить в качестве параметра стратегии, который может быть как классом (что и обсуждается в тексте), так и указателем на функцию.

Указывая разные стратегии накопления значений, можно вычислять разные вещи. Рассмотрим, например, программу, с помощью которой предполагается определять результат произведения ряда значений:

traits/accum6.cpp

```
#include "accum6.hpp"
#include <iostream>

class MultPolicy
{
public:
    template<typename T1, typename T2>
    static void accumulate(T1& total, T2 const& value)
    {
        total *= value;
    }
};

int main()
{
    // Создание массива из 5 целочисленных значений
    int num[] = { 1, 2, 3, 4, 5 };
    // Вывод произведения всех значений
    std::cout << "Произведение всех значений равно "
          << accum<int, MultPolicy>(num, num + 5)
          << '\n';
}
```

Однако вывод программы окажется вовсе не тем, который ожидается:

Произведение всех значений равно 0

Проблема вызвана нашим выбором начального значения: хотя значение 0 вполне пригодно при суммировании, оно не годится для умножения (нулевое начальное значение приводит к нулевому конечному результату). Этот пример иллюстрирует взаимодействие разных свойств и стратегий друг с другом, что еще раз подчеркивает, насколько важно быть аккуратным при проектировании шаблонов.

В данном случае легко понять, что инициализация цикла накопления — это часть стратегии накопления. Данная стратегия может использовать свойство `zero()` (но может и не воспользоваться им). Не следует забывать и о других вариантах решения задачи — далеко не все должно решаться только с помощью свойств и стратегий. Например, функция `accumulate()` стандартной библиотеки C++ получает начальное значение в качестве третьего аргумента функции.

19.2.1. Различие между свойствами и стратегиями

Вполне логично предположить, что стратегии представляют собой частный случай свойств. И наоборот, можно утверждать, что свойства — просто закодированные стратегии.

Оксфордский словарь [59] дает следующие определения:

- **свойство** — ...отличительная особенность, характеризующая сущность вещи;
- **стратегия** — ..любой образ действия, принятый как полезный или целесообразный.

На основании этих определений мы вправе ограничить использование термина *стратегия* классами, которые кодируют определенные действия, слабосвязанные с другими аргументами шаблона, с которым это действие связано. Это положение согласуется с утверждением Андрея Александреску (Andrei Alexandrescu) из книги *Современное проектирование на C++* [3]⁶:

Стратегии имеют много общего со свойствами, но отличаются от них тем, что в них меньше внимания уделяется типам и больше — поведению.

Натан Майерс (Nathan Myers), разработавший метод использования свойств, предложил следующее, более открытое определение [57]:

Класс свойств — это класс, используемый вместо параметров шаблона. В качестве класса он объединяет полезные типы и константы; как шаблон, он является средством для обеспечения того “дополнительного уровня конкретности”, который решает все проблемы программного обеспечения.

Таким образом, мы можем использовать следующие (несколько расплывчатые) определения.

- **Свойства** представляют собой естественные дополнительные свойства параметра шаблона.
- **Стратегии** представляют настраиваемое поведение обобщенных функций и типов (зачастую с некоторыми значениями по умолчанию).

Для дальнейшей конкретизации возможных различий между двумя этими концепциями перечислим ряд замечаний, касающихся свойств.

- Свойства могут быть использованы и как *фиксированные свойства*, т.е. без передачи их шаблону в качестве параметров.
- Параметры свойств обычно имеют естественные значения по умолчанию (которые крайне редко переопределяются или попросту не могут быть переопределены).
- Параметры свойств имеют тенденцию к сильной зависимости от одного или нескольких основных параметров.
- Свойства обычно содержат типы и константы, а не функции-члены.
- Свойства имеют тенденцию к агрегации в *шаблоны свойств*.

⁶ Александреску играл ключевую роль в мире классов стратегий; им разработан широкий набор основанных на них методов.

О классах стратегий также можно сделать несколько замечаний.

- Классы стратегий практически всегда передаются в качестве параметров шаблона.
- Параметры стратегий не обязательно должны иметь значения по умолчанию и часто явно специализируются (хотя многие обобщенные компоненты обычно настраиваются с использованием стратегий, заданных по умолчанию).
- Параметры стратегий обычно слабо связаны с другими параметрами шаблона.
- Классы стратегий обычно объединяют функции-члены в единое целое.
- Стратегии могут объединяться в обычных классах или в шаблонах классов.

Следует отметить, однако, что грань между обоими терминами весьма нечеткая. Например, свойства символов стандартной библиотеки C++ определяют также функциональное поведение, в частности сравнение символов, их перемещение и поиск. Заменяя эти свойства другими, можно определять строковые классы, которые ведут себя иначе, например нечувствительны к регистру символов при использовании того же символьного типа (см. раздел 13.2.15 в [45]). Таким образом, называясь *свойствами*, они имеют ряд характеристик, присущих стратегиям.

19.2.2. Шаблоны членов и шаблонные параметры шаблонов

Для реализации стратегии накопления был выбран вариант, в котором `SumPolicy` и `MultPolicy` представляли собой обычные классы с шаблонами членов. Другой вариант заключается в конструировании интерфейса класса стратегии с использованием шаблона класса, который затем применяется в качестве шаблонного аргумента шаблона (см. разделы 5.7 и 12.2.3). Например, можно переписать `SumPolicy` в виде шаблона.

traits/sumpolicy2.hpp

```
#ifndef SUMPOLICY_HPP
#define SUMPOLICY_HPP

template<typename T1, typename T2>
class SumPolicy
{
public:
    static void accumulate(T1& total, T2 const& value)
    {
        total += value;
    }
};

#endif // SUMPOLICY_HPP
```

Интерфейс класса `Accum` можно затем адаптировать для использования шаблонного параметра шаблона:

traits/accum7.hpp

```
#ifndef ACCUM_HPP
#define ACCUM_HPP
#include "accumtraits4.hpp"
#include "sumpolicy2.hpp"

template<typename T,
         template<typename, typename> class Policy = SumPolicy,
         typename Traits = AccumulationTraits<T>>
auto accum(T const* beg, T const* end)
{
    using AccT = typename Traits::AccT;
    AccT total = Traits::zero();

    while (beg != end)
    {
        Policy<AccT, T>::accumulate(total, *beg);
        ++beg;
    }

    return total;
}

#endif // ACCUM_HPP
```

Такое же преобразование можно применить и к параметру-свойству. (Возможны и другие варианты: например, вместо явной передачи в стратегию типа `AccT` может оказаться полезной передача свойств накопления, а стратегия при этом определяет тип результата из параметра свойства.)

Главное преимущество использования стратегий посредством шаблонных параметров шаблона — упрощение ситуации, когда класс стратегии содержит некоторую статическую информацию (статический член-данные) с типом, зависящим от параметров шаблона. (В нашем первом подходе статические члены-данные должны быть встроены в шаблон члена класса.)

Слабой стороной подхода с использованием шаблонных параметров шаблона является то, что классы стратегий должны быть написаны как шаблоны, с точным набором параметров шаблона, определяемых нашим интерфейсом. Это может сделать выражение самих свойств более многословным и менее естественным, чем простой нешаблонный класс.

19.2.3. Комбинирование нескольких стратегий и/или свойств

Как показали наши примеры, и свойства, и стратегии не позволяют полностью избежать применения нескольких параметров шаблона. При этом, однако, они снижают их количество до вполне управляемого. Возникает интересный вопрос: каким образом упорядочить такие множественные параметры?

Простая стратегия состоит в упорядочении параметров согласно возрастанию вероятности выбора значения по умолчанию. Обычно это приводит к тому, что параметры свойств следуют за параметрами стратегий, поскольку они чаще определяются пользователями (возможно, вы уже заметили использование этого правила в наших примерах).

Для тех, кто все же склонен прибегать к значительному количеству параметров, тем самым существенно усложняя код, существует альтернатива, состоящая в задании параметров в любом порядке, без использования значений по умолчанию. Более подробно этот вопрос изложен в разделе 21.4.

19.2.4. Накопление с обобщенными итераторами

Прежде чем закончить введение в свойства и стратегии, рассмотрим еще одну версию `accum()`, которая позволяет работать с обобщенными итераторами вместо указателей, что и ожидается от обобщенного промышленного компонента. Интересно, что возможность вызова `accum()` с указателями при этом остается, поскольку стандартная библиотека C++ обеспечивает использование так называемых *свойств итераторов* (iterator traits). Для этого можно переписать начальную версию `accum()` (опускаем при этом более поздние усовершенствования)⁷:

traits/accum0.hpp

```
#ifndef ACCUM_HPP
#define ACCUM_HPP
#include <iostream>
#include <iterator>

template<typename Iter>
auto accum(Iter start, Iter end)
{
    using VT = typename std::iterator_traits<Iter>::value_type;
    VT total{}; // Считаем, что создается нулевое значение

    while (start != end)
    {
        total += *start;
        ++start;
    }

    return total;
}

#endif // ACCUM_HPP
```

Структура `iterator_traits` инкапсулирует все существенные свойства итератора. Благодаря наличию частичной специализации для указателей эти свойства могут использоваться для любых обычных указателей. Ниже показано, как стандартная библиотека может реализовать эту поддержку:

⁷ В C++11 нужно объявить возвращаемый тип как VT.

```

namespace std
{
    template<typename T>
    struct iterator_traits<T*>
    {
        using difference_type = ptrdiff_t;
        using value_type = T;
        using pointer = T*;
        using reference = T&;
        using iterator_category = random_access_iterator_tag;
    };
}

```

Однако типа для накопления значений, к которым обращается итератор, здесь нет, так что нам придется разрабатывать свой собственный класс `AccumulationTraits`.

19.3. Функции типов

В первоначальном примере использования свойств показано, что можно задавать поведение, зависящее от типов. Это отличается от того, что мы обычно делаем в программах. В языках программирования С и C++ функции более точно можно назвать *функциями значений* (*value functions*): они принимают одни значения в качестве параметров и возвращают другое значение в качестве результата. При работе с шаблонами мы сталкиваемся с *функциями типов* (*type functions*), т.е. функциями, которые принимают некоторые аргументы типа и возвращают тип или константу в качестве результата.

Весьма полезной встроенной функцией типа является `sizeof`, которая возвращает константу, указывающую размер (в байтах) данного аргумента типа. Шаблоны классов также могут играть роль функций типа. Параметры функции типа — это параметры шаблона, а результат получается как тип-член или константа-член. Например, оператор `sizeof` может быть использован с приведенным ниже интерфейсом:

`traits/sizeof.cpp`

```

#include <cstddef>
#include <iostream>

template<typename T>
struct TypeSize
{
    static std::size_t const value = sizeof(T);
};

int main()
{
    std::cout << "TypeSize<int>::value = "
           << TypeSize<int>::value << '\n';
}

```

Это может показаться не очень полезным, поскольку у нас есть доступный встроенный оператор `sizeof`, но обратите внимание на то, что `TypeSize<T>` является

типов, и поэтому он может быть передан в качестве аргумента шаблона класса. Кроме того, `TypeSize<T>` — это шаблон, и он может быть передан в качестве шаблонного аргумента шаблона.

В дальнейшем мы разработаем несколько более универсальных функций типов, которые могут использоваться в качестве свойств.

19.3.1. Типы элементов

Предположим, что у нас есть ряд шаблонов контейнеров, таких как `std::vector<>`, `std::list<>`, а также встроенные массивы. Нам нужна функция типа, которая для данного типа контейнера возвращает тип его элементов. Этого можно достичь с помощью частичной специализации:

traits/elementtype.hpp

```
#include <vector>
#include <list>

template<typename T>
struct ElementT; // Первичный шаблон

template<typename T> // Частичная специализация
struct ElementT<std::vector<T>> // для std::vector
{
    using Type = T;
};

template<typename T> // Частичная специализация
struct ElementT<std::list<T>> // для std::list
{
    using Type = T;
};

***

template<typename T, std::size_t N>
struct ElementT<T[N]> // Частичная специализация для
{ // массива с известными границами
    using Type = T;
};

template<typename T> // Частичная специализация для
struct ElementT<T[]> // массива с неизвестными границами
{
    using Type = T;
};

***
```

Учтите, что мы должны обеспечить частичную специализацию для всех возможных типов массивов (см. подробнее в разделе 5.4).

Функцию типа можно использовать следующим образом:

traits/elementtype.cpp

```
#include "elementtype.hpp"
#include <vector>
#include <iostream>
#include <typeinfo>

template<typename T>
void printElementType(T const& c)
{
    std::cout << "Контейнер из "
        << typeid(typename ElementT<T>::Type).name()
        << " элементов.\n";
}

int main()
{
    std::vector<bool> s;
    printElementType(s);
    int arr[42];
    printElementType(arr);
}
```

Использование частичной специализации позволяет реализовать эту функцию, не требуя, чтобы в типы контейнера были заложены сведения о ней. Зачастую, однако, функция типа разрабатывается вместе с соответствующими типами, так что ее реализация может быть существенно упрощена. Например, если типы контейнера определяют тип элемента `value_type` (как это делают стандартные контейнеры), то можно написать следующее:

```
template<typename C>
struct ElementT
{
    using Type = typename C::value_type;
};
```

Этот код может быть реализацией по умолчанию, что не исключает наличия специализаций для тех типов контейнеров, для которых не задан соответствующий тип элемента `value_type`.

Тем не менее обычно желательно обеспечить возможность определения типов для параметров шаблонов, чтобы к ним было легче обращаться в обобщенном коде. В следующем фрагменте кода представлен набросок этой идеи:

```
template<typename T1, typename T2, ...>
class X
{
public:
    using ... = T1;
    using ... = T2;
    ...
};
```

В чем заключается польза функции типа? Она позволяет параметризовать шаблон в терминах типа контейнера, не требуя при этом дополнительных параметров для типа элемента и других характеристик. Например, вместо

```
template<typename T, typename C>
T sumOfElements(C const& c);
```

(где требуется указание типа элемента в явном виде при помощи синтаксиса на-подобие `sumOfElements<int>(list)`) можно объявить

```
template<typename C>
typename ElementT<C>::Type sumOfElements(C const& c);
```

(где тип элемента определяется функцией типа).

Обратите внимание на то, что свойства могут быть реализованы как расширения существующих типов. Таким образом, функции типа можно определять даже для фундаментальных типов и типов из закрытых библиотек.

В данном случае тип `ElementT` назван *классом свойств* (traits class), поскольку он используется для обращения к свойствам типа данного контейнера `C` (в общем случае в таком классе может быть собрано несколько свойств). Таким образом, классы свойств не ограничиваются описанием лишь характеристик параметров контейнера, но могут использоваться для описания любого вида “основных параметров”.

Для удобства можно создать шаблон псевдонима для типа функции. Например, можно ввести

```
template<typename T>
using ElementType = typename ElementT<T>::Type;
```

что позволяет нам выполнить дальнейшее упрощение объявления `sumOfElements` до

```
template<typename C>
ElementType<C> sumOfElements(C const& c);
```

19.3.2. Преобразующие свойства

Помимо предоставления доступа к конкретным аспектам типа основного параметра, свойства могут также выполнять преобразования типов, такие как добавление или удаление ссылок или квалификаторов `const` и `volatile`.

Удаление ссылок

Например, можно реализовать свойство `RemoveReferenceT`, которое превращает ссылочные типы в лежащие в их основе типы объектов или функций, оставляя только типы, не являющиеся ссылочными:

traits/removerefERENCE.hpp

```
template<typename T>
struct RemoveReferenceT
{
    using Type = T;
};

template<typename T>
struct RemoveReferenceT<T&>
```

```
{
    using Type = T;
};

template<typename T>
struct RemoveReferenceT<T&&>
{
    using Type = T;
};
```

И вновь шаблоны псевдонимов упрощают использование:

```
template<typename T>
using RemoveReference = typename RemoveReferenceT<T>::Type;
```

Удаление ссылки из типа обычно полезно, когда тип получен с помощью конструкции, которая иногда производит ссылочные типы, как, например, специальное правило вывода для параметров функции типа `T&&`, которое обсуждается в разделе 15.6.

Стандартная библиотека C++ предоставляет соответствующее свойство типа `std::remove_reference<>`, описанное в разделе Г.4.

Добавление ссылки

Аналогичным образом можно взять существующий тип и создать из него ссылку на l-значение или на r-значение (вместе с обычным использованием шаблона псевдонима):

`traits/addreference.hpp`

```
template<typename T>
struct AddLValueReferenceT
{
    using Type = T&;
};

template<typename T>
using AddLValueReference = typename AddLValueReferenceT<T>::Type;

template<typename T>
struct AddRValueReferenceT
{
    using Type = T &&;
};

template<typename T>
using AddRValueReference = typename AddRValueReferenceT<T>::Type;
```

Здесь применяются правила свертки ссылок (раздел 15.6). Например, вызов `AddLValueReference<int&&>` производит тип `int&` (поэтому нет необходимости их реализации вручную с помощью частичной специализации).

Если мы оставим `AddLValueReferenceT` и `AddRValueReferenceT` как есть и не будем вводить их специализации, то псевдонимы, используемые для удобства, на самом деле могут быть упрощены до

```
template<typename T>
using AddLValueReferenceT = T&;
template<typename T>
using AddRValueReferenceT = T&&;
```

которые могут быть инстанцированы без инстанцирования шаблона класса (тем самым облегчая дело). Однако это рискованно, если мы захотим специализировать этот шаблон для особых случаев. Например, при написанном выше коде мы не сможем использовать `void` в качестве аргумента для этих шаблонов. Для обработки такой ситуации требуется явная специализация:

```
template<>
struct AddLValueReferenceT<void>
{
    using Type = void;
};

template<>
struct AddLValueReferenceT<void const>
{
    using Type = void const;
};

template<>
struct AddLValueReferenceT<void volatile>
{
    using Type = void volatile;
};

template<>
struct AddLValueReferenceT<void const volatile>
{
    using Type = void const volatile;
};
```

Аналогичный код следует написать для `AddRValueReferenceT`.

При наличии такого кода удобные шаблоны псевдонимов должны быть сформулированы в терминах шаблонов классов для гарантии работоспособности специализаций (поскольку шаблоны псевдонимов не могут быть специализированы).

Стандартная библиотека C++ предоставляет соответствующие свойства типов `std::add_lvalue_reference<>` и `std::add_rvalue_reference<>`, описанные в разделе Г.4. Стандартные шаблоны включают специализации для типов `void`.

Удаление квалификаторов

Преобразующие свойства могут работать с любыми составными типами, а не только со ссылками. Например, можно удалить квалификатор `const`, если такой имеется в наличии:

traits/removeconst.hpp

```
template<typename T>
struct RemoveConstT
{
    using Type = T;
};

template<typename T>
struct RemoveConstT<T const>
{
    using Type = T;
};

template<typename T>
using RemoveConst = typename RemoveConstT<T>::Type;
```

Кроме того, преобразующее свойство может быть составным, как, например, свойство RemoveCVT, удаляющее и `const`, и `volatile`:

traits/removecv.hpp

```
#include "removeconst.hpp"
#include "removevolatile.hpp"

template<typename T>
struct RemoveCVT : RemoveConstT<typename RemoveVolatileT<T>::Type>
{
};

template<typename T>
using RemoveCV = typename RemoveCVT<T>::Type;
```

Следует отметить пару моментов в определении `RemoveCVT`. Оно использует как `RemoveConstT`, так и связанный с ним `RemoveVolatileT`, сначала удаляя квалификатор `volatile` (при наличии такого) и передавая результирующий тип в `RemoveConstT`⁸. Затем свойство использует *метафункциональную передачу* (metafunction forwarding) для наследования члена `Type` из `RemoveConstT` вместо объявления собственного члена `Type`, идентичного члену в специализации `RemoveConstT`. Здесь метафункциональная передача используется просто для того, чтобы уменьшить количество вводимого текста в определении `RemoveCVT`. Однако она также полезна, когда метафункция определена не для всех входных данных; эта техника будет рассматриваться в разделе 19.4.

Используемый для удобства шаблон псевдонима `RemoveCV` можно упростить до

```
template<typename T>
using RemoveCV = RemoveConst<RemoveVolatile<T>>;
```

⁸ Порядок удаления данных квалификаторов не имеет семантических последствий — их можно удалять в любом порядке.

И вновь, этот способ работает, только если шаблон RemoveCVT не специализируется. В отличие от случая AddLValueReference и AddRValueReference, мы не можем придумать ни одной причины для таких специализаций.

Стандартная библиотека C++ предоставляет соответствующие свойства типов `std::remove_volatile<>`, `std::remove_const<>` и `std::remove_cv<>`, описанные в разделе Г.4.

Низведение

Чтобы завершить нашу дискуссию о преобразующих свойствах, разработаем свойство, имитирующее преобразование типов при передаче аргументов в качестве параметров по значению. Это означает (наследуя C), что выполняется низведение аргументов (превращение массивов в указатели, а типов функций — в указатели на функции; см. разделы 7.4 и 11.1.1) и удаляются любые квалификаторы `const`, `volatile` или ссылки верхнего уровня (поскольку квалификаторы типов верхнего уровня у типов параметров при разрешении вызова функции игнорируются).

Эффект от такой передачи по значению можно увидеть в следующей программе, которая выводит тип фактического параметра, полученного после того, как компилятор низводит указанный тип:

traits/passbyvalue.cpp

```
#include <iostream>
#include <typeinfo>
#include <type_traits>

template<typename T>
void f(T)
{
}

template<typename A>
void printParameterType(void (*)(A))
{
    std::cout << "Тип параметра: "
        << typeid(A).name() << '\n';
    std::cout << "- является int: "
        << std::is_same<A, int>::value << '\n';
    std::cout << "- является const: "
        << std::is_const<A>::value << '\n';
    std::cout << "- является указателем: "
        << std::is_pointer<A>::value << '\n';
}

int main()
{
    printParameterType(&f<int>);
    printParameterType(&f<int const>);
    printParameterType(&f<int[7]>);
    printParameterType(&f<int(int)>);
}
```

В выводе программы параметр `int` остается неизменным, но параметры `int const`, `int[7]` и `int(int)` низводятся до `int`, `int*` и `int(*)(int)` соответственно.

Мы можем реализовать свойство, которое выполняет такое же преобразование типа, что и при передаче по значению. Для соответствия свойству `std::decay` стандартной библиотеки C++ назовем его `DecayT`⁹. Его реализация сочетает ряд описанных выше методов. Сначала мы определяем случай, в котором нет массива или функции, когда мы просто удаляем любой квалификатор `const` и `volatile`:

```
template<typename T>
struct DecayT : RemoveCVT<T>
{
};
```

Затем мы обрабатываем низведение массива к указателю, которое требует от нас распознавания массивов любого типа (с указанными границами или без таковых) с помощью частичной специализации:

```
template<typename T>
struct DecayT<T[]>
{
    using Type = T*;
};

template<typename T, std::size_t N>
struct DecayT<T[N]>
{
    using Type = T*;
};
```

Наконец, мы обрабатываем низведение функции к указателю на функцию, которое должно соответствовать любому типу функции, независимо от типа возвращаемого значения или количества и типов параметров. Для этого мы используем вариативные шаблоны:

```
template<typename R, typename... Args>
struct DecayT<R(Args...)>
{
    using Type = R(*)(Args...);
};

template<typename R, typename... Args>
struct DecayT<R(Args..., ...)>
{
    using Type = R(*)(Args..., ...);
};
```

Обратите внимание на то, что вторая частичная специализация соответствует любому типу функции, использующей переменное количество аргументов

⁹ Использование термина **низведение** (*decay*) может немного запутывать, потому что в C он предполагает только преобразование из типов массива/функции в тип указателя, в то время как в C++ он также включает удаление квалификаторов `const` или `volatile` верхнего уровня.

в стиле C¹⁰. Вместе первичный шаблон DecayT и четыре его частичных специализации реализуют низведение типа параметра, как показывает приведенный пример программы:

traits/decay.cpp

```
#include <iostream>
#include <typeinfo>
#include <type_traits>
#include "decay.hpp"

template<typename T>
void printDecayedType()
{
    using A = typename DecayT<T>::Type;
    std::cout << "Тип параметра: "
           << typeid(A).name() << '\n';
    std::cout << "- является int: "
           << std::is_same<A, int>::value << '\n';
    std::cout << "- является const: "
           << std::is_const<A>::value << '\n';
    std::cout << "- является указателем: "
           << std::is_pointer<A>::value << '\n';
}

int main()
{
    printDecayedType<int>();
    printDecayedType<int const>();
    printDecayedType<int[7]>();
    printDecayedType<int(int)>();
}
```

Как обычно, мы предоставляем удобный шаблон псевдонима:

```
template< typename T >
using Decay = typename DecayT<T>::Type;
```

Как уже говорилось, стандартная библиотека C++ обеспечивает соответствующее свойство типа `std::decay<>`, описанное в разделе Г.4.

19.3.3. Свойства-предикаты

До сих пор мы изучали и разрабатывали функции типов от одного типа: для данного одного типа функция предоставляет другой связанный с ним тип или константу. Однако в общем случае можно разработать функцию типа, которая зависит от нескольких аргументов. Это приводит также к особой форме свойств типа — предикатам (функциям типов, возвращающим логическое значение).

¹⁰ Стого говоря, запятая перед вторым многоточием (...) не является обязательной, но приведена здесь для ясности. Благодаря необязательности многоточия тип функции в первой частичной специализации на самом деле синтаксически неоднозначен: он может быть проанализирован как `R(Args, ...)` (переменное количество параметров в стиле C), так и как `R(Args... name)` (пакет параметров). Вторая интерпретация выбирается из-за того, что `Args` представляет собой не раскрытий пакет параметров. Можно явно добавить запятую в тех (редких) случаях, когда желательна иная интерпретация.

IsSameT

Свойство IsSameT проверяет, эквивалентны ли два типа:

traits/issame0.hpp

```
template<typename T1, typename T2>
struct IsSameT
{
    static constexpr bool value = false;
};

template<typename T>
struct IsSameT<T, T>
{
    static constexpr bool value = true;
};
```

Здесь первичный шаблон определяет, что в общем случае два различных типа, передаваемые в качестве аргументов шаблона, отличны. Таким образом, значение члена `value` равно `false`. Однако с помощью частичной специализации для частного случая совпадения двух переданных типов `value` равно `true`.

Например, следующее выражение проверяет, является ли переданный параметр шаблона типом `int`:

```
if (IsSameT<T, int>::value) ...
```

Для свойств, дающих константное значение, мы не можем предоставить шаблон псевдонима, но *можем* предоставить шаблон `constexpr`-переменной, играющей ту же роль:

```
template<typename T1, typename T2>
constexpr bool isSame = IsSameT<T1, T2>::value;
```

Стандартная библиотека C++ предоставляет соответствующее свойство типа `std::is_same<>`, которое описано в разделе Г.3.3.

true_type и false_type

Можно значительно улучшить определение IsSameT, предоставляя различные типы для двух возможных результатов, `true` и `false`. Фактически, если мы объявим шаблон класса `BoolConstant` с двумя возможными реализациями `TrueType` и `FalseType`:

traits/boolconstant.hpp

```
template<bool val>
struct BoolConstant
{
    using Type = BoolConstant<val>;
    static constexpr bool value = val;
};

using TrueType = BoolConstant<true>;
using FalseType = BoolConstant<false>;
```

то мы можем определить `IsSameT` так, чтобы, в зависимости от соответствия двух типов, он был производным от `TrueType` или `FalseType`:

`traits/issame.hpp`

```
#include "boolconstant.hpp"

template<typename T1, typename T2>
struct IsSameT : FalseType
{
};

template<typename T>
struct IsSameT<T, T> : TrueType
{
};
```

Теперь результирующий *тип* выражения

`IsSameT<T, int>`

неявно конвертируется в базовый класс `TrueType` или `FalseType`, который не только предоставляет соответствующий член `value`, но также позволяет нам выполнять диспетчеризацию времени компиляции различным реализациям функции или частичным специализациям класса. Например:

`traits/issame.cpp`

```
#include "issame.hpp"
#include <iostream>

template<typename T>
void fooImpl(T, TrueType)
{
    std::cout << "fooImpl(T,true) для int\n";
}

template<typename T>
void fooImpl(T, FalseType)
{
    std::cout << "fooImpl(T,false) для не int\n";
}

template<typename T>
void foo(T t)
{
    fooImpl(t, IsSameT<T, int>{}); // Выбор реализации в зависимости
                                    // от того, является ли T - int'ом
}

int main()
{
    foo(42);    // Вызов fooImpl(42,  TrueType)
    foo(7.7);   // Вызов fooImpl(7.7, FalseType)
}
```

Эта методика называется *диспетчеризацией дескрипторов* (tag dispatching) и рассматривается в разделе 20.2.

Обратите внимание на то, что наша реализация `BoolConstant` включает член `Type`, который вновь позволяет нам ввести шаблон псевдонима для `IsSameT`:

```
template<typename T>
using IsSame = typename IsSameT<T>::Type;
```

Этот шаблон псевдонима может сосуществовать с шаблоном переменной `isSame`.

В общем случае свойства, дающие логические значения, должны поддерживать диспетчеризацию дескрипторов путем наследования таких типов, как `TrueType` и `FalseType`. Однако для максимальной обобщенности должен существовать только один тип, представляющий значение `true`, и один тип, представляющий значение `false`, вместо того, чтобы каждая обобщенная библиотека определяла собственные типы для логических констант.

К счастью, стандартная библиотека C++ обеспечивает соответствующие типы в заголовочном файле `<type_traits>`, начиная со стандарта C++11: `std::true_type` и `std::false_type`. В C++11 и C++14 они определены следующим образом:

```
namespace std
{
    using true_type = integral_constant<bool, true>;
    using false_type = integral_constant<bool, false>;
}
```

Начиная с C++17, они определены как

```
namespace std
{
    using true_type = bool_constant<true>;
    using false_type = bool_constant<false>;
}
```

где имя `bool_constant` определено в пространстве имен `std` следующим образом:

```
template<bool B>
using bool_constant = integral_constant<bool, B>;
```

(Подробности представлены в разделе Г.1.1.)

По этой причине в оставшейся части книги мы используем `std::true_type` и `std::false_type`, в особенности для определения предикатов типов.

19.3.4. Свойства типов результатов

Еще одним примером функций типов, которые имеют дело с несколькими типами, являются *свойства типа результата* (result type traits). Они очень полезны при написании шаблонов операторов. Чтобы разобраться с идеей, давайте напишем шаблон функции, которая позволяет нам суммировать два контейнера `Array`:

```
template<typename T>
Array<T> operator+ (Array<T> const&, Array<T> const &);
```

Все выглядит неплохо, но, поскольку язык позволяет нам суммировать значение типа `char` со значением типа `int`, мы бы предпочли разрешить такие смешанные операции и с массивами тоже. Но тогда мы сталкиваемся с необходимостью определить тип возвращаемого значения такого шаблона

```
template<typename T1, typename T2>
Array <???> operator+ (Array<T1> const&, Array<T2> const&);
```

Помимо различных подходов, представленных в разделе 1.3, шаблон типа результата позволяет нам заполнить вопросительные знаки в предыдущем объявлении следующим образом:

```
template<typename T1, typename T2>
Array<typename PlusResultT<T1, T2>::Type>
operator+ (Array<T1> const&, Array<T2> const&);
```

или в предположении, что доступен удобный шаблон псевдонима,

```
template<typename T1, typename T2>
Array<PlusResult<T1, T2>> operator+ (Array<T1> const&, Array<T2> const&);
```

Свойство `PlusResultT` определяет тип, генерируемый при сложении двух (возможно, различных) типов с помощью оператора `+`:

traits/plus1.hpp

```
template<typename T1, typename T2>
struct PlusResultT
{
    using Type = decltype(T1() + T2());
};

template<typename T1, typename T2>
using PlusResult = typename PlusResultT<T1, T2>::Type;
```

Этот шаблон свойства использует `decltype` для вычисления типа выражения `T1() + T2()`, оставляя компилятору сложную работу по определению типа результата (включая обработку правил расширения и перегрузки операторов).

Однако для целей нашего примера `decltype` в действительности сохраняет *слишком много* информации (см. описание поведения `decltype` в разделе 15.10.2). Например, наша формулировка `PlusResultT` может дать ссылочный тип, но вряд ли наш шаблон класса `Array` предназначен для работы со ссылками. Более реалистично, что перегруженный оператор `operator+` может возвращать значение типа константного класса:

```
class Integer
{
    ...
};

Integer const operator+ (Integer const&, Integer const&);
```

Сложение двух значений `Array<Integer>` дает в результате массив значений `Integer const`, который, скорее всего, представляет собой не то, что нам нужно. Фактически мы хотим преобразовать тип результата путем удаления ссылок и квалификаторов, как описано в предыдущем разделе:

```
template<typename T1, typename T2>
Array<RemoveCV<RemoveReference<PlusResult<T1, T2>>>>
operator+(Array<T1> const&, Array<T2> const&);
```

Такая вложенность свойств распространена в библиотеках шаблонов и часто используется в контексте метапрограммирования. Метапрограммирование будут подробно рассматриваться в главе 23, “Метапрограммирование”. (Шаблоны псевдонимов особенно полезны при такой многоуровневой вложенности, как показанная. Без них нам бы пришлось добавлять `typename` и суффикс `::Type` на каждом уровне.)

В данный момент оператор сложения массивов правильно вычисляет тип результата при сложении двух массивов с, возможно, различными типами элементов. Однако наша формулировка `PlusResultT` добавляет нежелательные ограничения на типы элементов `T1` и `T2`: поскольку выражение `T1() + T2()` пытается выполнить инициализацию значением типов `T1` и `T2`, оба этих типа должны иметь доступный, не удаленный конструктор по умолчанию (или быть типами, не являющимися типами классов). Сам класс `Array` может не требовать наличия такой инициализации элементов, так что это ограничение оказывается излишним.

declval

К счастью, довольно легко получить значения для выражения с “+” без необходимости в конструкторе с помощью функции, которая производит значения данного типа `T`. Для этого стандарт C++ предоставляет шаблон `std::declval<>`, введенный в разделе 11.2.3. Он определен в заголовочном файле `<utility>` следующим образом:

```
namespace std
{
    template<typename T>
    add_rvalue_reference_t<T> declval() noexcept;
}
```

Выражение `declval<T>()` создает значение типа `T`, не требуя наличия конструктора по умолчанию (или любой иной операции).

Этот шаблон функции преднамеренно оставлен неопределенным, потому что он предназначен только для использования внутри `decltype`, `sizeof` или в некоторых других контекстах, где не требуется определение. Он имеет два других интересных атрибута.

- Для типов, для которых может быть создана ссылка, тип возвращаемого значения всегда представляет собой ссылку на *г*-значение типа, что позволяет `declval` работать даже с типами, которые обычно не могут быть возвращены из функций, как, например, тип абстрактного класса (класса с чисто виртуальными функциями) или тип массива. В противном случае преобразование из `T` в `T&` не имеет практического влияния на поведение `declval<T>()` при использовании в качестве выражения: оба они являются *г*-значениями (если `T`

является типом объекта), а ссылки на l-значение остаются неизменными благодаря свертке ссылок (описанной в разделе 15.6)¹¹.

- Спецификация исключений noexcept документирует, что decltype сам по себе не требует рассматривать выражение как нечто, что может генерировать исключение. Это становится полезным при использовании decltype в контексте noexcept-оператора (раздел 19.7.2).

Используя decltype, можно убрать требование инициализации значением для PlusResultT:

traits/plus2.hpp

```
#include <utility>

template<typename T1, typename T2>
struct PlusResultT
{
    using Type = decltype(std::::declval<T1>() + std::::declval<T2>());
};

template<typename T1, typename T2>
using PlusResult = typename PlusResultT<T1, T2>::Type;
```

Свойства типа результата предлагают способ для точного определения возвращаемого типа конкретной операции и часто полезны при описании типов результатов шаблонов функций.

19.4. Свойства на основе SFINAE

Принцип SFINAE (см. разделы 8.4 и 15.7) превращает потенциальные ошибки при формировании недопустимых типов и выражений во время вывода аргументов шаблонов (которые делают программу некорректной) в простой сбой вывода, позволяя разрешению перегрузки выбрать другого кандидата. Хотя первоначально этот принцип предназначался для избегания ложных сбоев при перегрузке шаблонов функций, SFINAE обеспечивает также замечательную методику времени компиляции, которая может определить, является ли допустимым определенный тип или выражение. Это позволяет нам писать свойства, которые определяют, например, имеет ли тип определенный член, поддерживает ли конкретную операцию или является классом.

Два основных подхода к свойствам на основе SFINAE представляют собой применение SFINAE для перегрузки функций и для частичных специализаций.

19.4.1. Принцип SFINAE и перегрузки функций

Наше первое знакомство со свойствами на основе SFINAE иллюстрирует базовую технологию применения SFINAE с перегрузкой функций для выяснения,

¹¹ Разница между возвращаемыми типами T и T&& обнаруживается при использовании decltype. Однако при данном ограниченном использовании decltype это не представляет практического интереса

является ли тип конструируемым по умолчанию, т.е. можно ли создавать объекты без каких-либо значений для инициализации (т.е. является ли корректным выражение наподобие `T()` для данного типа `T`).

Базовая реализация может иметь следующий вид:

`traits/isdefaultconstructible1.hpp`

```
#include "issame.hpp"

template<typename T>
struct IsDefaultConstructibleT
{
    private:
        // test() пытается подставить вызов конструктора
        // по умолчанию для T, переданного как U:
        template<typename U, typename = decltype(U())>
        static char test(void*);

        // Резервный вариант test():
        template<typename>
        static long test(...);

    public:
        static constexpr bool value
            = IsSameT<decltype(test<T>(nullptr)), char>::value;
};


```

Обычный подход к реализации свойств на основе SFINAE с перегрузкой функций состоит в объявлении двух перегруженных шаблонов функций с именем `test()` и с различными возвращаемыми типами:

```
template<...> static char test(void*);  
template<...> static long test(...);
```

Первая перегрузка разработана так, что демонстрирует соответствие, только если запрашиваемая проверка проходит успешно (ниже мы рассмотрим, как это достигается). Вторая перегрузка является резервной¹²: она всегда соответствует вызову, но наличие многоточия (т.е. переменного количества параметров) делает ее наименее предпочтительной по сравнению с любой другой функцией (см. раздел B.2).

Наше “возвращаемое значение” `value` зависит от того, какой перегруженный член `test` выбран:

```
static constexpr bool value  
    = IsSameT<decltype(test<T>(nullptr)), char>::value;
```

Если выбрана функция-член `test()`, возвращающая `char`, то `value` инициализируется значением `isSame<char, char>`, которое равно `true`. В противном случае оно инициализируется значением `isSame<long, char>`, равным `false`.

Мы работаем с определенными характеристиками типа, которые хотим проверить. Цель заключается в том, чтобы сделать первую перегрузку `test()` действительной только в том случае, когда проверяемое условие выполняется.

¹² Это резервное объявление иногда может быть объявлением обычной функции, а не шаблоном функции-члена.

В рассматриваемом случае мы хотим узнать, возможно ли конструирование объекта переданного типа `T` по умолчанию. Для этого мы передаем `T` в качестве `U` и передаем нашему первому объявлению `test()` второй безымянный (фиктивный) аргумент шаблона, инициализированный конструкцией, которая является корректной тогда и только тогда, когда условие является выполненным. В нашем случае мы используем выражение, которое может быть корректным только если существует явный или неявный конструктор по умолчанию: `U()`. Это выражение помещено в `decltype`, чтобы использовать это выражение для инициализации параметра типа.

Второй параметр шаблона не может быть выведен, так как соответствующий аргумент не передается, и мы не предоставляем для него явный аргумент шаблона. Таким образом, будет выполнена подстановка аргумента шаблона по умолчанию, и если она будет неудачна, то согласно принципу SFINAE это объявление `test()` будет отброшено, так что соответствие будет демонстрировать только резервное объявление.

Итак, мы можем использовать разработанное свойство следующим образом:

```
IsDefaultConstructibleT<int>::value // Дает true
struct S
{
    S() = delete;
};
IsDefaultConstructibleT<S>::value // Дает false
```

Обратите внимание на то, что мы не можем использовать параметр шаблона `T` в первой функции `test()` непосредственно:

```
template<typename T>
struct IsDefaultConstructibleT
{
    private:
        // Ошибка: test() использует T непосредственно:
        template<typename, typename = decltype(T())>
        static char test(void*);

        // Резервный вариант test():
        template<typename>
        static long test(...);

    public:
        static constexpr bool value
            = IsSameT<decltype(test<T>(nullptr)), char>::value;
};
```

Этот вариант не работает, потому что для любого `T` всегда выполняется подстановка во все функции-члены, так что для типа, который не является конструируемым по умолчанию, код приводит к ошибке компиляции, а не к игнорированию первого перегруженного метода `test()`. Путем передачи параметра шаблона класса `T` в качестве параметра шаблона функции `U` мы создаем определенный контекст SFINAE.

Стратегии альтернативных реализаций свойств на основе SFINAЕ

Реализовать свойства на основе SFINAЕ было невозможно до тех пор, пока не был опубликован первый стандарт C++ в 1998 году¹³. Ключевым моментом этого подхода является объявление двух перегруженных шаблонов функций, имеющих различные типы возвращаемых значений:

```
template<...> static char test(void*);  
template<...> static long test(...);
```

Однако изначально опубликованная методика¹⁴ использовала размер возвращаемого типа для определения того, какая из перегрузок была выбрана (а кроме того, использовались 0 и enum, поскольку nullptr и constexpr тогда еще не были доступны):

```
enum { value = sizeof(test<...>(0)) == 1 };
```

На некоторых платформах может оказаться, что `sizeof(char) == sizeof(long)`. Например, в процессорах для обработки цифровых сигналов (digital signal processors – DSP) или старых машинах Стэй все целочисленные фундаментальные типы могут иметь один и тот же размер. По определению, `sizeof(char)` равен 1, но на этих машинах `sizeof(long)` и даже `sizeof(long long)` также равны 1.

С учетом этого замечания мы хотим добиться, чтобы возвращаемые типы функций `test()` имели различные размеры на всех plataформах. Например, после определения

```
using Size1T = char;  
using Size2T = struct  
{  
    char a[2];  
};
```

или

```
using Size1T = char(&)[1];  
using Size2T = char(&)[2];
```

мы можем определить перегрузки `test()` следующим образом:

```
template<...> static Size1T test(void*); // Проверяющая test()  
template<...> static Size2T test(...); // Резервная функция
```

Здесь возвращается либо тип `Size1T`, который представляет собой один символ размером 1, либо (структура, включающая) массив из двух `char`, который имеет размер по крайней мере 2 на всех plataформах.

Код, использующий один из этих подходов, все еще широко распространен.

¹³Однако тогда правила SFINAЕ были более ограниченными: когда подстановка аргументов шаблона приводила к неверной конструкции типа (например, `T : X`, где `T` – тип `int`), принцип SFINAЕ работал, как и ожидалось, но если он приводил к недопустимому выражению (например, `sizeof(f())`, где `f()` возвращает `void`), SFINAЕ не работал и выводилось сообщение об ошибке.

¹⁴Пожалуй, первое издание данной книги было первым источником указанной методики.

Обратите внимание на то, что тип аргумента вызова, передаваемого в `func()`, значения не имеет. Важно только, чтобы переданный аргумент соответствовал ожидаемому типу. Например, можно написать определения для передачи целого числа 42:

```
template<...> static Size1T test(int); // Проверяющая test()
template<...> static Size2T test(...); // Резервная функция
...
enum { value = sizeof(test<...>(42)) == 1 };
```

Создание свойств-предикатов на основе SFINAE

Введенные в разделе 19.3.3 свойства-предикаты, возвращающие логическое значение, должны возвращать значение, производное от `std::true_type` или `std::false_type`. Таким способом можно также решить проблему, связанную с тем, что на некоторых платформах `sizeof(char) == sizeof(long)`.

Для этого нам нужно косвенное определение `IsDefaultConstructibleT`. Это свойство само должно быть унаследовано от `Type` вспомогательного класса, который является необходимым базовым классом. К счастью, можно просто предоставить соответствующие базовые классы как возвращаемые типы перегрузок `test()`:

```
template<...> static std::true_type test(void*); // Проверяющая test()
template<...> static std::false_type test(...); // Резервная функция
```

Таким образом, член `Type` базового класса может быть объявлен следующим образом:

```
using Type = decltype(test<FROM>(nullptr));
```

и нам больше не нужно свойство `IsSameT`.

Таким образом, полная усовершенствованная реализация `IsDefaultConstructibleT` становится следующей:

traits/isdefaultconstructible2.hpp

```
#include <type_traits>

template<typename T>
struct IsDefaultConstructibleHelper
{
    private:
        // test() пытается подставить вызов конструктора
        // по умолчанию для T, переданного как U:
        template<typename U, typename = decltype(U())>
        static std::true_type test(void*);

        // Резервная версия test():
        template<typename>
        static std::false_type test(...);

    public:
        using Type = decltype(test<T>(nullptr));
};
```

```
template<typename T>
struct IsDefaultConstructibleT :
    IsDefaultConstructibleHelper<T>::Type
{
};
```

Теперь, если первый шаблон функции `test()` является допустимым, он представляет собой предпочтительную перегрузку, так что член `IsDefaultConstructibleHelper::Type` инициализируется его возвращаемым типом `std::true_type`. Как следствие, `IsConvertibleT<...>` оказывается производным от `std::true_type`.

Если же первый шаблон функции `test()` не является допустимым, из-за SFINAE он становится недоступным, и `IsDefaultConstructibleHelper::Type` инициализируется возвращаемым типом резервной версии `test()`, то есть `std::false_type`. В результате `IsConvertibleT<...>` наследуется от `std::false_type`.

19.4.2. SFINAE и частичные специализации

Второй подход к реализации свойств на основе SFINAE применяет частичную специализацию. Мы вновь используем пример, выясняющий, имеет ли тип `T` конструктор по умолчанию:

```
traits/isdefaultconstructible3.hpp

#include "issame.hpp"

#include <type_traits> // Определения true_type и false_type

// Вспомогательный тип для игнорирования
// количества параметров шаблона:
template<typename...> using VoidT = void;

// Первичный шаблон:
template<typename, typename = VoidT<>>
struct IsDefaultConstructibleT : std::false_type
{
};

// Частичная специализация (может быть удалена с помощью SFINAE):
template<typename T>
struct IsDefaultConstructibleT<T, VoidT<decltype(T())>> :
    std::true_type
{
```

Как и в усовершенствованной версии `IsDefaultConstructibleT` для свойства-предиката выше, определяем общий случай, производный от `std::false_type`, потому что в общем случае тип не имеет конструктора по умолчанию.

Здесь интересной особенностью является второй аргумент шаблона, который по умолчанию использует вспомогательный тип `VoidT`. Это позволяет нам

предоставлять частичные специализации, которые используют произвольное количество типовых конструкций времени компиляции.

В данном случае нам нужна только одна конструкция `decltype(T())` для проверки, допустим ли конструктор по умолчанию для `T`. Если для конкретного `T` данная конструкция является некорректной, в этот раз SFINAE удалит всю частичную специализацию, и мы вернемся к первичному шаблону. В противном случае частичная специализация оказывается корректной и предпочтительной.

В C++17 в стандартную библиотеку введено свойство типа `std::void_t<>`, соответствующее типу, который здесь представлен как `VoidT`. До C++17 может быть полезным определить его самостоятельно, как указано выше, или даже в пространстве имен `std` следующим образом¹⁵:

```
#include <type_traits>

#ifndef __cpp_lib_void_t
namespace std
{
    template<typename...> using void_t = void;
}
#endif
```

Начиная с C++14, Комитет по стандартизации C++ рекомендовал указывать, какие части стандарта реализованы в компиляторах и стандартных библиотеках, для чего определил согласованные *макросы свойств* (feature macros). Это не является *требованием* стандарта, но разработчики, как правило, следуют полезным для их пользователей рекомендациям¹⁶. Макрос `__cpp_lib_void_t` представляет собой макрос, который рекомендован для указания, что стандартная библиотека реализует `std::void_t`; таким образом, приведенный выше код является условно компилируемым в зависимости от определения этого макроса.

Очевидно, что определение свойств типа таким образом выглядит более сжатым, чем рассмотренный ранее первый подход с перегрузкой шаблонов функций. Однако он требует возможности формулировать условие внутри объявления параметра шаблона. Применение шаблона класса с перегрузкой функций позволяет нам использовать дополнительные вспомогательные функции или вспомогательные типы.

19.4.3. Применение обобщенных лямбда-выражений со SFINAE

Какой бы метод мы ни использовали, всегда существует некоторый стереотипный код для определения свойств: перегрузка и вызов двух функций-членов `test()` или реализация множественной частичной специализации. Далее мы

¹⁵ Определение `void_t` в пространстве имен `std` формально некорректно: пользовательскому коду не разрешается добавлять объявления в пространство имен `std`. На практике существующие компиляторы не запрещают такое добавление, и в то же время не демонстрируют никакого “неожиданного” поведения (несмотря на то, что стандарт указывает, что это ведет к “неопределенному поведению”, при котором может произойти что угодно).

¹⁶ Увы, на момент написания книги Microsoft Visual C++ являлся досадным исключением.

покажем, как в C++17 можно минимизировать этот код, указав проверяемое условие в обобщенном лямбда-выражении¹⁷.

Начнем с инструмента, построенного из двух обобщенных лямбда-выражений:

traits/isvalid.hpp

```
#include <utility>

// Вспомогательный шаблон проверки корректности
// f(args...) для F f и Args... args:
template<typename F, typename... Args,
         typename = decltype(
             std::declval<F>() (std::declval<Args...>()...))>
std::true_type isValidImpl(void*);

// Резервный вариант для случая, если SFINAE
// отвергнет вспомогательный шаблон:
template<typename F, typename... Args>
std::false_type isValidImpl(...);

// Определение лямбда-выражения, которое принимает
// лямбда-выражение f и проверяет, корректен ли вызов f с args
inline constexpr
auto isValid = [](auto f)
{
    return [](auto && ... args)
    {
        return decltype(isValidImpl<decltype(f),
                        decltype(args)&&...
                        >(nullptr)) {};
    };
};

// Вспомогательный шаблон для представления типа в виде значения
template<typename T>
struct TypeT
{
    using Type = T;
};

// Вспомогательный шаблон для обертки типа в значение
template<typename T>
constexpr auto type = TypeT<T> {};

// Вспомогательный шаблон для развертывания обернутого
// типа в невычислимых контекстах
template<typename T>
T valueT(TypeT<T>); // Определение не требуется
```

Начнем с определения `isValid`: это переменная `constexpr`, типом которой является лямбда-замыкание. Объявление обязательно должно использовать тип-заместитель (в нашем коде `auto`), поскольку C++ не имеет возможности

¹⁷ Мы выражаем признательность Луи Дионну (Louis Dionne) за указание на метод, использованный в данном разделе.

выражать типы замыканий непосредственно. До C++17 лямбда-выражения не могут появляться в константных выражениях; поэтому данный код является допустимым только в C++17. Поскольку `isValid` имеет тип замыкания, он может быть вызван, но возвращаемый элемент сам является объектом с типом лямбда-замыкания, сгенерированным внутренним лямбда-выражением.

Прежде чем углубляться в детали этого внутреннего лямбда-выражения, рассмотрим типичное применение `isValid`:

```
constexpr auto isDefaultConstructible
= isValid([](auto x) -> decltype((void) decltype(valueT(x)) {}));
```

Мы уже знаем, что `isDefaultConstructible` имеет тип лямбда-замыкания, и, как предполагает данное имя, это функциональный объект, который проверяет свойство типа быть конструируемым по умолчанию (вскоре мы увидим, как). Другими словами, `isValid` представляет собой *фабрику свойств*: компонент, генерирующий свойства, проверяющие объекты из своих аргументов.

Шаблон вспомогательной переменной `type` позволяет нам представить тип в виде значения. Значение `x`, полученное таким способом, может быть вновь преобразовано в исходный тип с помощью `decltype(valueT(x))`¹⁸, и именно это делает лямбда-выражение, переданное выше в `isValid`. Если этот извлеченный тип не может быть сконструирован по умолчанию, `decltype(valueT(x))()` является некорректным выражением, и мы либо получим сообщение об ошибке компиляции, либо соответствующее объявление будет удалено в соответствии с принципом SFINAE (именно этот результат мы получаем благодаря деталям определения `isValid`).

`isDefaultConstructible` может быть использован следующим образом:

```
isDefaultConstructible(
    type<int>) // true (int конструируем по умолчанию)
isDefaultConstructible(
    type<int&>) // false (ссылки не конструируются по умолчанию)
```

Чтобы увидеть, как все части работают совместно, рассмотрим, во что превращается внутреннее лямбда-выражение в `isValid` с параметром `f`, связанным с аргументом, которое представляет собой обобщенное лямбда-выражение, указанное в определении `isDefaultConstructible`. Выполнение подстановки в определении `isValid` дает код, эквивалентный следующему¹⁹:

```
constexpr auto isDefaultConstructible
= [](auto && ... args)
{
    return decltype(
        isValidImpl <
        decltype([](auto x)
```

¹⁸Эта очень простая пара вспомогательных шаблонов является фундаментальной методикой, лежащей в основе современных библиотек, таких как Boost.Hana!

¹⁹Этот код не является корректным кодом C++, поскольку лямбда-выражение не может появляться непосредственно в операнде `decltype` по техническим причинам, но его смысл очевиден.

```

        -> decltype((void)decltype(valueT(x))()),
           decltype(args)&&...
      > (nullptr) {};
```

Если мы вернемся к первому объявлению `isValidImpl()` выше, то заметим, что оно включает в себя аргумент шаблона по умолчанию вида

```
decltype(std::declval<F>() (std::declval<Args&>()...))>
```

который пытается вызвать значение типа своего первого аргумента шаблона, который имеет тип лямбда-замыкания в определении `isDefaultConstructible`, со значениями типов аргументов (`decltype(args)&&...`), переданных `isDefaultConstructible`. Поскольку у лямбда-выражения имеется только один параметр `x`, `args` должно раскрыться только в один аргумент; в наших примерах `static_assert` выше этот аргумент имеет тип `TypeT<int>` или `TypeT<int&>`. В случае `TypeT<int&>`, `decltype(valueT(x))` представляет собой `int&`, что делает `decltype(valueT(x))()` некорректным, и, таким образом, подстановка аргумента шаблона по умолчанию в первое объявление `isValidImpl()` оказывается неудачной, и согласно принципу SFINAE оно игнорируется. Это оставляет нам только второе объявление (которое в противном случае имело бы худшее соответствие), которое производит значение `false_type`. В целом, когда передается `type<int&>`, `isDefaultConstructible` производит `false_type`. Если же вместо этого передается `type<int>`, то подстановка оказывается успешной, и выбирается первое объявление `isValidImpl()`, генерирующее значение `true_type`.

Вспомним, что для работы SFINAE подстановка должна выполняться в *непосредственном контексте* замещаемых шаблонов. В нашем случае замещаемыми шаблонами являются первое объявление `isValidImpl` и оператор вызова обобщенного лямбда-выражения, передаваемого в `isValid`. Таким образом, тестируемая конструкция должна находиться в возвращаемом типе этого лямбда-выражения, а не в его теле!

Наше свойство `isDefaultConstructible` немного отличается от предыдущих реализаций свойства тем, что он требует вызова в стиле функции вместо указания аргументов шаблона. Это, вероятно, более удобочитаемая запись, но можно добиться и стиля, применявшегося ранее:

```
template<typename T>
using IsDefaultConstructibleT
    = decltype(isDefaultConstructible(std::declval<T>()));
```

Однако, поскольку это традиционное объявление шаблона, оно может находиться только в области видимости пространства имен, в то время как определение `isDefaultConstructible`, по-видимому, может быть введено в области видимости блока.

Пока что этот метод может показаться не слишком интересным, поскольку оба выражения, участвующие в реализации, и стиль использования оказываются более сложными, чем в предыдущих методах. Однако при наличии понятного

`isValid` многие свойства можно реализовать с помощью только одного объявления. Например, проверка доступности члена с именем `first` оказывается очень простой (см. полный пример в разделе 19.6.4):

```
constexpr auto hasFirst
= isValid([](auto x) -> decltype((void)valueT(x).first)
{
});
```

19.4.4. SFINAЕ и свойства

В общем случае свойство типа должно отвечать на конкретный вопрос, не приводя к некорректному коду программы. Свойства на основе SFINAЕ решают эту задачу путем выявления потенциальных проблем в контексте SFINAЕ, пре-вращая эти потенциальные ошибки в отрицательные результаты.

Однако некоторые свойства, представленные в книге к этому моменту (на-пример, свойство `PlusResultT`, описанное в разделе 19.3.4), ведут себя при на-личии ошибок не так хорошо. Вспомним определение `PlusResultT` из указан-ного раздела:

`traits/plus2.hpp`

```
#include <utility>

template<typename T1, typename T2>
struct PlusResultT
{
    using Type = decltype(std::declval<T1>() + std::declval<T2>());
};

template<typename T1, typename T2>
using PlusResult = typename PlusResultT<T1, T2>::Type;
```

В этом определении `+` используется в контексте, который не защищен с по-мощью принципа SFINAЕ. Поэтому, если программа попытается вычислить `PlusResultT` для типов, которые не имеют подходящего оператора `+`, программа станет некорректной, как показывает следующая попытка объявления типа воз-вращаемого значения суммирования массивов несвязанных типов А и В²⁰:

```
template<typename T>
class Array
{
    * * *

};

// Объявление + для массивов с разными типами элементов:
template<typename T1, typename T2>
Array<typename PlusResultT<T1, T2>::Type>
operator+ (Array<T1> const&, Array<T2> const&);
```

²⁰ Для простоты возвращаемое значение просто использует `PlusResultT<T1, T2>::Type`. На практике возвращаемый тип должен вычисляться с использованием `RemoveReferenceT<>` и `RemoveCVT<>` во избежание возврата ссылок.

Очевидно, что использование здесь `PlusResultT<>` приведет к ошибке, если для элементов массивов не существует соответствующего оператора `+`.

```
class A
{
};

class B
{
};

void addAB(Array<A> arrayA, Array<B> arrayB)
{
    auto sum = arrayA +      // Ошибка: сбой при инстанцировании
            arrayB;        // PlusResultT<A, B>
    ...
}
```

На практике проблема не в том, что этот сбой происходит в коде, который очевидно некорректен (нельзя сложить массив `A` и массив `B`), а в том, что он происходит во время вывода аргумента шаблона для оператора `+`, глубоко в инстанцировании `PlusResultT<A, B>`.

У этого факта есть замечательное следствие: это означает, что программа *может* не компилироваться, даже если мы добавим перегрузку для сложения массивов `A` и `B`, поскольку C++ не указывает, должны ли типы в шаблоне функции быть инстанцированы фактически, если другая перегрузка оказывается лучшей:

```
// Объявление обобщенного оператора +
// для массивов с различными типами элементов:
template<typename T1, typename T2>
Array<typename PlusResultT<T1, T2>::Type>
operator+ (Array<T1> const&, Array<T2> const&);

// Перегрузка оператора + для конкретных типов:
Array<A> operator+(Array<A> const& arrayA, Array<B> const& arrayB);
void addAB(Array<A> const& arrayA, Array<B> const& arrayB)
{
    auto sum = arrayA + arrayB; // Ошибка? Зависит от того,
    ...                         // инстанцирует ли компилятор
}                                // PlusResultT<A, B>
```

Если компилятор может без выполнения вывода и подстановки в первое (шаблонное) объявление `operator+` определить, что второе объявление `operator+` имеет лучшее соответствие, то он примет этот код.

Однако во время вывода и подстановки кандидата шаблона функции все, что происходит при инстанцировании определения шаблона класса, *не* является частью *непосредственного контекста* подстановки этого шаблона функции, так что SFINAE *не* защищает нас от попыток сформировать некорректные типы или выражения. Вместо того, чтобы просто отбросить кандидата шаблона функции, сразу же выводятся сообщения об ошибках, поскольку мы пытаемся вызвать `operator+` для двух *элементов* типа `A` и типа `B` внутри `PlusResultT<>`:

```
template<typename T1, typename T2>
struct PlusResultT
{
    using Type = decltype(std::declval<T1>() + std::declval<T2>());
};
```

Для решения этой проблемы необходимо сделать `PlusResultT` *дружественным по отношению к SFINAE*, т.е. сделать его более устойчивым, давая ему подходящее определение, даже когда его выражение `decltype` некорректно.

Следуя примеру `HasLessT`, описанному в предыдущем разделе, мы определяем свойство `HasPlusT`, которое позволяет нам обнаружить, существует ли подходящая операция `+` для данных типов:

traits/hasplus.hpp

```
#include <utility>      // Для declval
#include <type_traits> // Для true_type, false_type и void_t

// Первичный шаблон:
template<typename, typename, typename = std::void_t<>>
struct HasPlusT : std::false_type
{
};

// Частичная специализация (может быть удалена с помощью SFINAE):
template<typename T1, typename T2>
struct HasPlusT<T1, T2, std::void_t<decltype(std::declval<T1>()
                                              + std::declval<T2>())>>
: std::true_type
{};

};
```

Если шаблон дает результат `true`, `PlusResultT` может использовать существующую реализацию. В противном случае `PlusResultT` требует безопасное значение по умолчанию. Лучшее значение по умолчанию для свойства, которое не имеет значащего результата для множества аргументов шаблона, заключается в том, чтобы не предоставлять член `Type` вовсе. Таким образом, если это свойство используется в контексте SFINAE (например, в качестве возвращаемого типа шаблона оператора `operator+` для массивов), отсутствующий член `Type` сделает вывод аргумента шаблона неудачным, и это именно то поведение, которое требуется для шаблона оператора `operator+` для массивов.

Это поведение предоставляет следующая реализация `PlusResultT`:

traits/plus3.hpp

```
#include "hasplus.hpp"

template<typename T1, typename T2, bool = HasPlusT<T1, T2>::value>
struct PlusResultT
{
    // Первичный шаблон – используется,
    // когда HasPlusT дает true
    using Type = decltype(std::declval<T1>() + std::declval<T2>());
};
```

```
template<typename T1, typename T2>
struct PlusResultT<T1, T2, false> // Частичная специализация,
{                                // используемая в противном случае
};
```

В этой версии `PlusResultT` мы добавляем параметр шаблона с аргументом по умолчанию, который с помощью нашего свойства `HasPlusT`, рассматривавшегося выше, определяет, поддерживается ли сложение первых двух параметров. Затем мы частично специализируем `PlusResultT` для значения дополнительного параметра, равного `false`, и это наше определение частичной специализации не имеет никаких членов вообще (тем самым позволяя избежать описанной проблемы). Для случаев, когда поддерживается сложение, аргумент по умолчанию принимает значение `true`, и выбирается первичный шаблон с соответствующим определением члена `Type`. Таким образом, мы выполняем контракт, по которому `PlusResultT` обеспечивает результирующий тип, только если результат операции суммирования является корректным. (Обратите внимание на то, что добавленный параметр шаблона никогда не должен иметь явно указанный аргумент шаблона.)

Вновь рассмотрим суммирование `Array<A>` и `Array`: с использованием нашей последней реализации шаблона `PlusResultT` инстанцирование `PlusResultT<A, B>` не будет иметь член `Type`, потому что значения `A` и `B` являются не суммируемыми. Следовательно, тип результата шаблона оператора `operator+` массива является недопустимым, и SFINAE устранит данный шаблон функции из рассмотрения. Таким образом, будет выбран перегруженный `operator+`, специфичный для `Array<A>` и `Array`.

В качестве общего принципа проектирования шаблонов свойств никогда не должен сбить во время инстанцирования, если в качестве входных данных получены разумные аргументы шаблона. Этот общий подход часто выполняет соответствующую проверку дважды.

1. Один раз для проверки, что операция корректна.
2. Еще один раз — для вычисления результата.

Мы уже видели это при рассмотрении `PlusResultT`, где вызывали `HasPlusT<>`, чтобы выяснить, корректен ли вызов `operator+` в `PlusResultImpl<>`.

Давайте применим этот принцип к `ElementT`, представленному в разделе 19.3.1: он создает тип элемента из типа контейнера. И вновь, поскольку ответ зависит от того, имеет ли тип (контейнера) тип-член `value_type`, первичный шаблон должен пытаться определить член `Type` только тогда, когда тип контейнера имеет член `value_type`:

```
template<typename C, bool = HasMemberT_value_type<C>::value>
struct ElementT
{
    using Type = typename C::value_type;
};
```

```
template<typename C>
struct ElementT<C, false>
{
};
```

Третий пример создания свойства, дружественного по отношению к SFINAE, показан в разделе 19.7.2, где `IsNothrowMoveConstructibleT` сначала должен проверить, существует ли перемещающий конструктор перед тем, как выяснить, объявлен ли он как `noexcept`.

19.5. `IsConvertibleT`

Детали имеют значение. Поэтому общий подход к свойствам на основе SFINAE на практике может оказаться существенно более сложным. Проиллюстрируем это путем определения свойства, которое может выяснить, является ли данный тип преобразуемым в другой тип — например, если мы ожидаем определенный базовый класс или один из его производных классов. Свойство `IsConvertibleT` сообщает, можно ли преобразовать переданный первый тип в переданный второй тип:

`traits/isconvertible.hpp`

```
#include <type_traits> // Для true_type и false_type
#include <utility> // Для declval

template<typename FROM, typename TO>
struct IsConvertibleHelper
{
    private:
        // test() пытается вызвать вспомогательную функцию aux(TO)
        // для FROM, переданного как F:
        static void aux(TO);
        template<typename F, typename,
                 typename = decltype(aux(std::declval<F>()))>
        static std::true_type test(void*);

        // Резервная функция test():
        template<typename, typename>
        static std::false_type test(...);

    public:
        using Type = decltype(test<FROM>(nullptr));
};

template<typename FROM, typename TO>
struct IsConvertibleT : IsConvertibleHelper<FROM, TO>::Type
{
};

template<typename FROM, typename TO>
using IsConvertible = typename IsConvertibleT<FROM, TO>::Type;

template<typename FROM, typename TO>
constexpr bool isConvertible = IsConvertibleT<FROM, TO>::value;
```

Здесь мы используем подход с перегрузкой функций, как и в разделе 19.4.1. То есть внутри вспомогательного класса мы объявляем два перегруженных шаблона функций с именем `test()` и с различными возвращаемыми типами и объявляем член `Type` в качестве базового класса результирующего свойства:

```
template<...> static std::true_type test(void*);  
  
template<...> static std::false_type test(...);  
  
...  
  
using Type = decltype(test<FROM>(nullptr));  
  
...  
  
template<typename FROM, typename TO>  
struct IsConvertibleT : IsConvertibleHelper<FROM, TO>::Type  
{  
};
```

Как обычно, первая перегрузка `test()` предназначена для того, чтобы быть соответствующей аргументам только в том случае, если запрашиваемая проверка прошла успешно, в то время как вторая перегрузка является резервной. Таким образом, цель заключается в том, чтобы сделать первую перегрузку `test()` корректной тогда и только тогда, когда тип `FROM` может быть преобразован в тип `TO`. Для достижения этой цели мы вновь даем первому объявлению `test` фиктивный (безымянный) аргумент шаблона, инициализируемый конструкцией, которая является корректной тогда и только тогда, когда преобразование является допустимым. Этот параметр шаблона не может быть выведен, и мы не будем предоставлять для него явный аргумент шаблона. Таким образом, будет выполнена соответствующая подстановка, и, если эта подстановка окажется некорректной, данное объявление `test()` будут проигнорировано.

Еще раз обратите внимание, что следующий код неработоспособен:

```
static void aux(TO);  
template<typename = decltype(aux(std::declval<FROM>()))>  
static char test(void*);
```

Здесь `FROM` и `TO` полностью определены при синтаксическом анализе этого шаблона функции-члена, а потому пара типов, для которых это преобразование не является допустимым (например, `double*` и `int*`), приводит к немедленному выводу сообщения об ошибке, до любого вызова `test()` (а следовательно, вне любого контекста SFINAE).

По этой причине мы вводим `F` как определенный параметр шаблона функции-члена

```
static void aux(TO);  
template<typename F, typename = decltype(aux(std::declval<F>()))>  
static char test(void*);
```

и предоставляем тип `FROM` как явный аргумент шаблона в вызове `test()` в инициализации `value`:

```
static constexpr bool value
= isSame<decltype(test<FROM>(nullptr)), char>;
```

Обратите внимание на то, как для генерации значения без вызова какого бы то ни было конструктора используется шаблон `std::declval`, представленный в разделе 19.3.4. Если это значение преобразуемо в ТО, вызов `aux()` является допустимым, и это объявление `test()` соответствует аргументам. В противном случае происходит сбой SFINAE, и соответствовать аргументам будет только резервное объявление.

В результате мы можем использовать это свойство следующим образом:

```
IsConvertibleT<int, int>::value           // Дает true
IsConvertibleT<int, std::string>::value     // Дает false
IsConvertibleT<char const*, std::string>::value // Дает true
IsConvertibleT<std::string, char const*>::value // Дает false
```

Обработка особых случаев

Есть три случая, которые `IsConvertibleT` все еще обрабатывает некорректно.

1. Преобразование в тип массива всегда должно давать значение `false`, но в нашем коде параметр типа ТО в объявлении `aux()` будет просто низведен к типу указателя, и поэтому мы получим результат `true` для некоторых типов FROM.
2. Преобразования в типы функций всегда должны давать `false`, но, как и в случае массивов, наша реализация просто рассматривает их как низведенные типы.
3. Преобразование в (квалифицированные как `const/volatile`) типы `void` должно давать значение `true`. К сожалению, наша реализация даже не в состоянии выполнить успешное инстанцирование, когда тип ТО представляет собой тип `void`, потому что параметры не могут иметь тип `void` (а `aux()` быть объявленной с таким параметром).

Для корректной обработки всех этих случаев нам понадобятся дополнительные частичные специализации. Однако добавление такой специализации для каждой возможной комбинации квалификаторов `const` и `volatile` быстро становится слишком громоздким. Вместо этого мы можем добавить дополнительный параметр шаблона к шаблону нашего вспомогательного класса следующим образом:

```
template<typename FROM, typename TO, bool = IsVoidT<TO>::value
         || IsArrayT<TO>::value
         || IsFunctionT<TO>::value>
struct IsConvertibleHelper
{
    using Type = std::integral_constant<bool,
                                         IsVoidT<TO>::value
                                         && IsVoidT<FROM>::value>;
};

template<typename FROM, typename TO>
struct IsConvertibleHelper<FROM, TO, false>
```

```
{
    ... // Предыдущая реализация IsConvertibleHelper
};
```

Дополнительный логический параметр шаблона гарантирует, что для всех указанных особых случаев используется реализация основного вспомогательного свойства. Он дает `false_type`, если мы преобразуем типы в массивы или функции (потому что тогда `IsVoidT<TO>` имеет значение `false`), либо если `FROM` представляет собой `void`, а `TO` — нет, но для двух типов `void` он будет давать `true_type`. Все остальные случаи дают аргумент `false` в качестве третьего параметра, а потому выбирается частичная специализация, которая соответствует уже рассмотренной ранее реализации.

Реализация `IsArrayT` обсуждается в разделе 19.8.2, а реализация `IsFunctionT` — в разделе 19.8.3.

Стандартная библиотека C++ предоставляет соответствующее свойство типа `std::is_convertible<>`, которое описывается в разделе Г.3.3.

19.6. Обнаружение членов

Еще один пример свойств на основе SFINAE включает создание свойства (или, скорее, набора свойств), позволяющего определить, содержит ли данный тип `T` в качестве члена некоторое имя `X` (как члена, являющегося типом, так и не являющегося таковым).

19.6.1. Обнаружение членов-типов

Давайте сначала определим свойство, которое выясняет, имеет ли данный тип `T` член-тип `size_type`:

traits/hassizetype.hpp

```
#include <type_traits> // Определения true_type и false_type

// Вспомогательный шаблон для игнорирования
// любого количества параметров шаблона:
template<typename...> using VoidT = void;

// Первичный шаблон:
template<typename, typename = VoidT<>>
struct HasSizeTypeT : std::false_type
{
};

// Частичная инициализация (может быть отброшена SFINAE):
template<typename T>
struct HasSizeTypeT<T, VoidT<typename T::size_type>> : std::true_type
{
```

Здесь использован подход с отбрасыванием с помощью SFINAE частичных специализаций, представленный в разделе 19.4.2.

Как обычно для свойств-предикатов, мы определяем общий случай, производный от `std::false_type`, поскольку по умолчанию тип не имеет члена `size_type`.

В этом случае нам нужна только одна конструкция:

```
typename T::size_type
```

Эта конструкция корректна тогда и только тогда, когда тип `T` имеет член-тип `size_type`, а именно это мы и пытаемся определить. Если для конкретного `T` конструкция недопустима (т.е. тип `T` не содержит член-тип `size_type`), SFINAE приводит к отбрасыванию частичной специализации, и мы возвращаемся к первичному шаблону. В противном случае частичная специализация является корректной и предпочтительной.

Свойство можно использовать следующим образом:

```
std::cout << HasSizeTypeT<int>::value; // false
struct CX
{
    using size_type = std::size_t;
};
std::cout << HasSizeType<CX>::value; // true
```

Обратите внимание: если член-тип `size_type` объявлен как `private`, то `HasSizeTypeT` дает значение `false`, потому что наши шаблоны свойств не имеют особого доступа к типам аргументов, так что конструкция `typename T::size_type` является некорректной (т.е. запускает SFINAE). Другими словами, свойство проверяет, имеется ли *доступный* тип-член `size_type`.

Работа со ссылочными типами

Программисты знают о сюрпризах, которые могут возникать “на стыках” рассматриваемых предметных областей. Что касается шаблона свойства `HasSizeTypeT`, то здесь интересные проблемы возможны при работе со ссылочными типами. Например, в то время как следующий код отлично работает:

```
struct CXR
{
    using size_type = char&; // Обратите внимание:
}; // size_type - ссылочный тип
std::cout << HasSizeTypeT<CXR>::value; // OK: выводит true
```

приведенный далее код сбоят:

```
std::cout << HasSizeTypeT<CX&>::value; // Сбой: выводит false
std::cout << HasSizeTypeT<CXR&>::value; // Сбой: выводит false
```

Это потенциально удивительно. Да, ссылочный тип не имеет членов сам по себе, но когда мы используем ссылки, получающиеся выражения имеют базовый тип, так что, пожалуй, и в этом случае было бы предпочтительнее рассмотреть базовый тип. В данной ситуации этого можно добиться, используя свойство `RemoveReference` в частичной специализации `HasSizeTypeT`:

```
template<typename T>
struct HasSizeTypeT<T, VoidT<RemoveReference<T>::size_type>>
    : std::true_type
{
};
```

Внедренные имена классов

Стоит также отметить, что наша техника для обнаружения членов-типов также дает значение `true` для внедренных имен классов (см. раздел 13.2.3). Например:

```
struct size_type
{
};

struct Sizeable : size_type
{
};
static_assert(HasSizeTypeT<Sizeable>::value,
    "Ошибка компилятора: отсутствует внедренное имя класса");
```

`static_assert` в последней строке проходит успешно, поскольку `size_type` вводит свое собственное имя как член-тип, и это имя наследуется. Неудача означала бы, что мы нашли ошибку в компиляторе.

19.6.2. Обнаружение произвольных членов-типов

Определение такого свойства, как `HasSizeTypeT`, ставит вопрос о том, как параметризовать свойство, чтобы иметь возможность проверки *любого* имени члена-типа.

К сожалению, в настоящее время это можно сделать только с помощью макросов, поскольку не существует механизма для описания “потенциального” имени²¹. Лучшее, что мы можем получить в данный момент без использования макросов, — применение обобщенных лямбда-выражений, как показано в разделе 19.6.4.

Следующий макрос вполне работоспособен:

```
traits/hastype.hpp

#include <type_traits>           // true_type, false_type и void_t

#define DEFINE_HAS_TYPE(MemType) \
    template<typename, typename = std::void_t<>> \
    struct HasTypeT_##MemType \
        : std::false_type { }; \
    template<typename T> \
    struct HasTypeT_##MemType<T, std::void_t<typename T::MemType>> \
        : std::true_type { }     // ; Преднамеренно опущена
```

²¹ На момент написания книги Комитет по стандартизации C++ изучал пути “рефлексии” различных программных сущностей (наподобие типов классов и их членов) таким образом, чтобы программа могла их использовать (см. раздел 17.9).

Каждое применение `DEFINE_HAS_TYPE(MemberType)` определяет новое свойство `HasTypeT<MemberType>`. Например, чтобы обнаружить, имеет ли тип члены-типы `value_type` или `char_type`, его можно использовать следующим образом:

traits/hastype.cpp

```
#include "hastype.hpp"
#include <iostream>
#include <vector>

DEFINE_HAS_TYPE(value_type);
DEFINE_HAS_TYPE(char_type);

int main()
{
    std::cout << "int::value_type: "
        << HasTypeT<value_type>::value << '\n';
    std::cout << "std::vector<int>::value_type: "
        << HasTypeT<value_type><std::vector<int>>::value << '\n';
    std::cout << "std::iostream::value_type: "
        << HasTypeT<value_type><std::iostream>::value << '\n';
    std::cout << "std::iostream::char_type: "
        << HasTypeT<char_type><std::iostream>::value << '\n';
}
```

19.6.3. Обнаружение членов, не являющихся типами

Мы можем изменить свойства так, чтобы они также проверяли наличие членов-данных и (одиночных) функций-членов:

traits/hasmember.hpp

```
#include <type_traits>                                // true_type, false_type и void_t

#define DEFINE_HAS_MEMBER(Member) \
    template<typename, typename = std::void_t<>> \
    struct HasMemberT_ ## Member \
        : std::false_type { }; \
    template<typename T> \
    struct HasMemberT_ ## Member<T, std::void_t<decltype(&T::Member)>> \
        : std::true_type { }                                // ; Преднамеренно опущена
```

Здесь мы используем SFINAE для отключения частичной специализации, когда выражение `&T::Member` некорректно. Чтобы эта конструкция была корректной, должны выполняться следующие условия:

- `Member` должен однозначно идентифицировать член `T` (например, он не может быть именем перегруженной функции или именем нескольких унаследованных членов с одинаковым именем);
- этот член должен быть доступным;
- член не должен быть типом или перечислением (в противном случае префикс `&` будет некорректным);

- если `T::Member` является статическим членом-данным, его тип не должен предоставлять `operator&`, который делает `&T::Member` некорректным выражением (например потому, что этот оператор недоступен).

Этот шаблон можно использовать следующим образом:

`traits/hasmember.cpp`

```
#include "hasmember.hpp"
#include <iostream>
#include <vector>
#include <utility>

DEFINE_HAS_MEMBER(size);
DEFINE_HAS_MEMBER(first);

int main()
{
    std::cout << "int::size: "
        << HasMemberT_size<int>::value << '\n';
    std::cout << "std::vector<int>::size: "
        << HasMemberT_size<std::vector<int>>::value << '\n';
    std::cout << "std::pair<int,int>::first: "
        << HasMemberT_first<std::pair<int, int>>::value << '\n';
}
```

Не должно быть сложным изменение частичной специализации для исключения случаев, когда `&T::Member` не является типом указателя на член (тем самым исключая статические члены-данные). Аналогично можно исключить указатель на функцию-член либо потребовать ограничения свойства членами-данными или функциями-членами.

Обнаружение функций-членов

Обратите внимание на то, что свойство `HasMember` лишь проверяет, существует ли *единственный* элемент с соответствующим именем. Свойство будет неработоспособным при обнаружении двух членов, что возможно во время проверки перегруженных функций-членов. Например:

```
DEFINE_HAS_MEMBER(begin);
std::cout << HasMemberT_begin<std::vector<int>>::value; // false
```

Однако, как поясняется в разделе 8.4.1, принцип SFINAE защищает от попыток создать недопустимые типы и выражения в объявлении шаблона функции, позволяя расширить методику перегрузки для проверки корректности произвольных выражений.

То есть можно просто проверить, возможно ли вызвать интересующую функцию определенным образом, и будет ли этот вызов успешным, даже если функция перегружена. Как и в случае со свойством `IsConvertibleT` в разделе 19.5, трюк заключается в создании выражения, которое проверяет, можно ли вызвать `begin()` внутри выражения `decltype` для значения по умолчанию дополнительного параметра шаблона функции:

traits/hasbegin.hpp

```
#include <utility>      // Для declval
#include <type_traits> // Для true_type, false_type и void_t

// Первичный шаблон:
template<typename, typename = std::void_t<>>
struct HasBeginT : std::false_type
{
};

// Частичная инициализация (может быть отброшена SFINAE):
template<typename T>
struct HasBeginT<T, std::void_t<decltype(std::declval<T>().begin())>>
    : std::true_type
{
};
```

Здесь мы используем

`decltype(std::declval<T>().begin())`

чтобы для данного значения/объекта типа `T` проверить (с помощью `std::declval`, чтобы избежать необходимости наличия любого конструктора) корректность вызова члена `begin()`²².

Обнаружение других выражений

Эту методику можно использовать и для выражений других видов и даже для комбинации нескольких выражений. Например, для данных типов `T1` и `T2` можно проверить, имеется ли оператор `<`, определенный для значений этих типов:

traits/hasless.hpp

```
#include <utility>      // Для declval
#include <type_traits> // Для true_type, false_type и void_t

// Первичный шаблон:
template<typename, typename, typename = std::void_t<>>
struct HasLessT : std::false_type
{
};

// Частичная инициализация (может быть отброшена SFINAE):
template<typename T1, typename T2>
struct HasLessT < T1, T2, std::void_t<decltype(std::declval<T1>()
                                              < std::declval<T2>())>>
    : std::true_type
{
};
```

²² За исключением того, что `decltype(выражение_вызыва)` не требует несырьочного, отличного от `void` полного возвращаемого типа, в отличие от выражения вызова в других контекстах. Применение вместо этого `decltype(std::declval<T>().begin(), 0)` добавляет требование, чтобы тип возвращаемого значения вызова был полным, потому что возвращаемое значение больше не является результатом операнда `decltype`.

Как всегда, возникает задача определить допустимое выражение для проверяемого условия и использовать decltype для его размещения в контексте SFINAE, где это приведет к выбору первичного шаблона, если выражение оказывается недопустимым:

```
decltype(std::declval<T1>() < std::declval<T2>())
```

Свойства, которые обнаруживают допустимые выражения таким образом, достаточно надежны: они будут давать значение true, только когда выражение является корректным, и совершенно верно будут возвращать значение false, когда оператор < оказывается неоднозначным, удаленным или недоступным²³.

Это свойство можно использовать следующим образом:

HasLessT<int, char>::value	// true
HasLessT<std::string, std::string>::value	// true
HasLessT<std::string, int>::value	// false
HasLessT<std::string, char*>::value	// true
HasLessT<std::complex<double>, std::complex<double>>::value	// false

Как говорилось в разделе 2.3.1, можно использовать это свойство для того, чтобы потребовать от параметра шаблона T поддержки оператора <:

```
template<typename T>
class C
{
    static_assert(HasLessT<T>::value,
                  "Класс C требует сравнимых элементов");

    ...
};
```

Обратите внимание на то, что благодаря природе std::void_t можно комбинировать ограничения в свойствах:

traits/hasvarious.hpp

```
#include <utility>      // Для declval
#include <type_traits> // Для true_type, false_type и void_t

// Первичный шаблон:
template<typename, typename = std::void_t<>>
struct HasVariousT : std::false_type
{
};

// Частичная инициализация (может быть отброшена SFINAE):
template<typename T>
struct HasVariousT<T, std::void_t<decltype(std::declval<T>().begin()), typename T::difference_type, typename T::iterator>>
```

²³ До распространения в C++11 SFINAE на произвольные недопустимые выражения методики обнаружения корректности определенных выражений были сосредоточены на введении новой перегрузки для тестируемой функции (например, <), которая имела излишне разрешительную сигнатуру и возвращаемый тип с необычным размером в качестве резервного варианта. Однако такие подходы склонны к неоднозначностям и приводят к ошибкам из-за нарушения управления доступом.

```
: std::true_type
{
};
```

Свойства, которые обнаруживают корректность определенного синтаксиса, довольно мощные и позволяют настраивать поведение шаблонов на основе наличия или отсутствия конкретной операции. Эти свойства используются как часть определения свойств, дружественных по отношению к SFINAE (раздел 19.4.4) и для помощи в перегрузке на основе свойств типов (глава 20, “Перегрузка свойств типов”).

19.6.4. Использование обобщенных лямбда-выражений для обнаружения членов

Лямбда-выражение `isValid`, введенное в разделе 19.4.3, обеспечивает более компактную методику определения свойств для проверки членов, помогая избежать применения макросов для обработки членов в случае произвольных имен.

Приведенный ниже пример показывает, как определять свойства, проверяющие наличие таких членов-данных или членов-типов, как `first` или `size_type`, или выясняющие, определен ли оператор `operator<` для двух объектов разных типов:

`traits/isvalid1.cpp`

```
#include "isvalid.hpp"
#include<iostream>
#include<string>
#include<utility>

int main()
{
    using namespace std;
    cout << boolalpha;

    // Определение для проверки наличия члена-данных first:
    constexpr auto hasFirst
        = isValid([](auto x) -> decltype((void)valueT(x).first)
    {});

    cout << "hasFirst: " << hasFirst(type<pair<int, int>>) << '\n';

    // Определение для проверки наличия члена-типа size_type:
    constexpr auto hasSizeType
        = isValid([](auto x) -> typename decltype(valueT(x))::size_type
    {});

    struct CX
    {
        using size_type = std::size_t;
    };

    cout << "hasSizeType: " << hasSizeType(type<CX>) << '\n';
}
```

```
if constexpr(!hasSizeType(type<int>))
{
    cout << "int не имеет size_type\n";
    ***
}

// Определение для проверки наличия оператора <:
constexpr auto hasLess
= isValid([](auto x, auto y) -> decltype(valueT(x) < valueT(y))
{});

cout << hasLess(42, type<char>) << '\n'; // Дает true
cout << hasLess(type<string>, type<string>) << '\n'; // Дает true
cout << hasLess(type<string>, type<int>) << '\n'; // Дает false
cout << hasLess(type<string>, "hello") << '\n'; // Дает true
```

И вновь обратите внимание: `hasSizeType` использует `std::decay` для удаления ссылок из переданного `x`, потому что нельзя получить доступ к члену-типу из ссылки. Если вы пропустите этот шаблон, свойство всегда будет давать `false`, потому что будет использоваться вторая перегрузка `isValidImpl<>()`.

Чтобы иметь возможность использовать обычный обобщенный синтаксис, принимающий типы в качестве параметров шаблона, мы можем, как и ранее, определить дополнительные вспомогательные классы. Например:

traits/isvalid2.cpp

```
#include "isValid.hpp"
#include<iostream>
#include<string>
#include<utility>

constexpr auto hasFirst
= isValid([](auto && x) -> decltype((void)&x.first)
{});

template<typename T>
using HasFirstT = decltype(hasFirst(std::declval<T>()));

constexpr auto hasSizeType
= isValid([](auto && x)
-> typename std::decay_t<decltype(x)>::size_type
{});

template<typename T>
using HasSizeTypeT = decltype(hasSizeType(std::declval<T>()));

constexpr auto hasLess
= isValid([](auto && x, auto && y) -> decltype(x < y)
{});

template<typename T1, typename T2>
using HasLessT = decltype(hasLess(std::declval<T1>(),
std::declval<T2>()));
```

```

int main()
{
    using namespace std;
    cout << "first: " << HasFirstT<pair<int, int>>::value << '\n';

    struct CX
    {
        using size_type = std::size_t;
    };

    cout << "size_type: "<< HasSizeTypeT<CX>::value << '\n'; // true
    cout << "size_type: "<< HasSizeTypeT<int>::value << '\n'; // false
    cout << HasLessT<int, char>::value << '\n'; // true
    cout << HasLessT<string, string>::value << '\n'; // true
    cout << HasLessT<string, int>::value << '\n'; // false
    cout << HasLessT<string, char*>::value << '\n'; // true
}

```

Теперь

```
template<typename T>
using HasFirstT = decltype(hasFirst(std::declval<T>()));
```

позволяет осуществить вызов

```
HasFirstT<std::pair<int, int>>::value
```

который в результате выполняет вызов `hasFirst` для пары из двух `int`, который вычисляется так, как описано выше.

19.7. Прочие методы работы со свойствами

Давайте напоследок рассмотрим и обсудим иные возможные подходы к определению свойств.

19.7.1. If-Then-Else

В предыдущем разделе окончательное определение свойства `PlusResultT` имело совершенно иную реализацию в зависимости от результата работы другого свойства типа, `HasPlusT`. Такое поведение в стиле “if-then-else” можно сформулировать с помощью специального шаблона типа `IfThenElse`, который принимает логический параметр шаблона для выбора одного из двух параметров типа:

`traits/ifthenelse.hpp`

```

#ifndef IFTHENELSE_HPP
#define IFTHENELSE_HPP

// Первичный шаблон: по умолчанию дает второй аргумент и
// использует частичную специализацию для получения
// третьего аргумента при COND равном false
template<bool COND, typename TrueType, typename FalseType>
struct IfThenElseT

```

```

{
    using Type = TrueType;
};

// Частичная специализация: false дает третий аргумент
template<typename TrueType, typename FalseType>
struct IfThenElseT<false, TrueType, FalseType>
{
    using Type = FalseType;
};

template<bool COND, typename TrueType, typename FalseType>
using IfThenElse = typename
    IfThenElseT<COND, TrueType, FalseType>::Type;

#endif // IFTHENELSE_HPP

```

Приведенный ниже пример демонстрирует применение этого шаблона путем определения функции типа, которая выявляет для данного значения целочисленный тип наименьшего размера:

traits/smallestint.hpp

```

#include <limits>
#include "ifthenelse.hpp"

template<auto N>
struct SmallestIntT
{
    using Type =
        typename IfThenElseT<N<=std::numeric_limits<char>::max(), char,
        typename IfThenElseT<N<=std::numeric_limits<short>::max(), short,
        typename IfThenElseT<N<=std::numeric_limits<int>::max(), int,
        typename IfThenElseT<N<=std::numeric_limits<long>::max(), long,
        typename IfThenElseT<N<=std::numeric_limits<long long>::max(),
            long long,                                // иначе
            void                                     // резервный вариант
        >::Type
        >::Type
        >::Type
        >::Type;
};

```

Обратите внимание: в отличие от обычной конструкции “if-then-else” в C++ здесь до выполнения выбора вычисляются аргументы шаблона для обеих ветвей — и “then”, и “else”, так что ни одна ветвь не может содержать некорректный код, иначе программа будет ошибочной. Рассмотрим, например, свойство, которое дает беззнаковый тип, соответствующий данному знаковому типу. Существует стандартное свойство `std::make_unsigned`, которое делает такое преобразование, но оно требует, чтобы переданный тип был знаковым целочисленным типом и не был типом `bool`; в противном случае его использование

приводит к неопределенному поведению (см. раздел Г.4). Поэтому может быть хорошей идеей реализация свойства, которое возвращает соответствующий беззнаковый тип, если это возможно, и переданный тип в противном случае (избегая тем самым неопределенного поведения при передаче неподходящего типа). Простейшая реализация не срабатывает:

```
// Ошибка: неопределенное поведение, если T представляет
// собой bool или не является целочисленным типом:
template<typename T>
struct UnsignedT
{
    using Type = IfThenElse < std::is_integral<T>::value
                    && !std::is_same<T, bool>::value,
                    typename std::make_unsigned<T>::type, T >;
};

Инстанцирование UnsignedT<bool> демонстрирует неопределенное поведение, поскольку компилятор все равно будет пытаться образовать тип из
```

```
typename std::make_unsigned<T>::type
```

Для решения этой проблемы нам следует добавить дополнительный уровень косвенности, чтобы аргументы IfThenElse сами использовали функции типов для “завертывания” результата:

```
// Дает T при использовании члена Type:
template<typename T>
struct IdentityT
{
    using Type = T;
};

// Для получения unsigned после вычисления IfThenElse:
template<typename T>
struct MakeUnsignedT
{
    using Type = typename std::make_unsigned<T>::type;
};

template<typename T>
struct UnsignedT
{
    using Type = typename IfThenElse < std::is_integral<T>::value
                    && !std::is_same<T, bool>::value,
                    MakeUnsignedT<T>, IdentityT<T>
                    ::Type;
};

В этом определении UnsignedT аргументами типа IfThenElse являются сами экземпляры функций типов. Однако фактически функции типов не вычисляются до того, как IfThenElse выберет нужную. Шаблон IfThenElse выбирает экземпляр функции типа (MakeUnsignedT или IdentityT). И только затем ::Type вычисляет экземпляр выбранной функции типа для создания Type.
```

Стоит подчеркнуть, что здесь мы полностью полагаемся на тот факт, что отвергнутый тип оболочки в конструкции `IfThenElse` никогда не будет полностью инстанцирован. В частности, следующее решение *не* работает:

```
template<typename T>
struct UnsignedT
{
    using Type = typename IfThenElse < std::is_integral<T>::value
                                && !std::is_same<T, bool>::value,
                                MakeUnsignedT<T>::Type, T
    >;
};
```

Вместо этого мы должны применить `::Type` к `MakeUnsignedT<T>` позже, что означает, что нам нужен вспомогательный шаблон `IdentityT`, чтобы затем применить `::Type` в ветви `else`.

Это также означает, что в этом контексте мы не можем использовать код наподобие

```
template<typename T>
using Identity = typename IdentityT<T>::Type;
```

Мы можем объявить такой шаблон псевдонима, и это может быть полезным в других местах, но мы не можем эффективно использовать его в определении `IfThenElse`, потому что любое использование `Identity<T>` немедленно приводит к полному инстанцированию `IdentityT<T>` для получения его члена `Type`.

Шаблон `IfThenElseT` содержится в стандартной библиотеке C++ под именем `std::conditional` (см. раздел Г.5). С его помощью свойство `UnsignedT` может быть определено следующим образом:

```
template<typename T>
struct UnsignedT
{
    using Type
        = typename std::conditional_t < std::is_integral<T>::value
                                    && !std::is_same<T, bool>::value,
                                    MakeUnsignedT<T>, IdentityT<T>
        >::Type;
};
```

19.7.2. Обнаружение операций, не генерирующих исключений

Иногда оказывается полезно определить, может ли некоторая конкретная операция привести к исключению. Например, перемещающие конструкторы по возможности должны быть помечены как `noexcept` в качестве указания, что они не генерируют исключений. Однако, генерирует перемещающий конструктор конкретное исключение или нет, часто зависит от того, генерируют ли исключения конструкторы его членов и базовых классов. Например, рассмотрим перемещающий конструктор для простого шаблона класса `Pair`:

```
template<typename T1, typename T2>
class Pair
{
    T1 first;
    T2 second;
public:
    Pair(Pair&& other)
        : first(std::forward<T1>(other.first)),
          second(std::forward<T2>(other.second))
    {
    }
};
```

Перемещающий конструктор `Pair` может генерировать исключения, когда это могут делать операции перемещения `T1` или `T2`. Имея свойство `IsNothrowMoveConstructibleT`, мы могли бы выразить этот факт с помощью вычислённой спецификации исключений `noexcept` в перемещающем конструкторе `Pair`. Например:

```
Pair(Pair&& other) noexcept(IsNothrowMoveConstructibleT<T1>::value&&
                             IsNothrowMoveConstructibleT<T2>::value)
    : first(std::forward<T1>(other.first)),
      second(std::forward<T2>(other.second))
{}
```

Нам остается лишь реализовать свойство `IsNothrowMoveConstructibleT`. Это можно сделать непосредственно с помощью оператора `noexcept`, который определяет, является ли данное выражение гарантированно не генерирующими исключений:

traits/isnothrowmoveconstructible1.hpp

```
#include <utility>      // Для declval
#include <type_traits> // Для bool_constant

template<typename T>
struct IsNothrowMoveConstructibleT
: std::bool_constant<noexcept(T(std::declval<T>()))>
{};

};
```

Здесь используется операторная версия `noexcept`, которая определяет, является ли выражение не генерирующим исключений. Поскольку результатом является логическое значение, мы можем передать его непосредственно в определение базового класса, `std::bool_constant<>`, которое используется для определения `std::true_type` и `std::false_type` (см. раздел 19.3.3)²⁴.

Однако эта реализация должна быть улучшена, поскольку она не является дружественной по отношению к SFINAE (см. раздел 19.4.4): если это свойство инстанцируется с типом, который не имеет копирующего конструктора или перемещающего конструктора (делая выражение `T(std::declval<T&&>())` недействительным), то вся программа оказывается некорректной:

²⁴ В C++11 и C++14 мы должны указать базовый класс как `std::integral_constant<bool,...>`, а не как `std::bool_constant<...>`.

```

class E
{
public:
    E(E&&) = delete;
};

*** std::cout << // Ошибка времени компиляции:
IsNothrowMoveConstructibleT<E>::value;

```

Вместо аварийного завершения компиляции свойство типа должно давать `value` со значением `false`.

Как было описано в разделе 19.4.4, мы должны проверить, допустимо ли выражение, вычисляющее результат, до его вычисления. Здесь мы должны выяснить, корректно ли перемещающее создание, до проверки, является ли оно поexcept. Таким образом, мы пересматриваем первый вариант свойства, добавляя параметр шаблона, который по умолчанию равен `void`, и частичную специализацию, которая использует `std::void_t` для параметра с аргументом, который действителен только в том случае, когда перемещающее конструирование является корректным:

`traits/isnothrowmoveconstructible2.hpp`

```

#include <utility>      // Для declval
#include <type_traits> // для true_type, false_type и bool_constant<>

// Первичный шаблон:
template<typename T, typename = std::void_t<>>
struct IsNothrowMoveConstructibleT : std::false_type
{
};

// Частичная инициализация (может быть отброшена SFINAE):
template<typename T>
struct IsNothrowMoveConstructibleT
<T, std::void_t<decltype(T(std::declval<T>()))>>
    : std::bool_constant<noexcept(T(std::declval<T>()))>
{
};

```

Если подстановка `std::void_t<...>` в частичной специализации является допустимой, выбирается эта специализация, и выражение `noexcept(...)` в спецификаторе базового класса может быть безопасно вычислено. В противном случае частичная специализация отбрасывается без ее инстанцирования, а вместо этого инстанцируется первичный шаблон (давая в результате `std::false_type`).

Обратите внимание на то, что нет никакого способа проверить, генерирует ли перемещающий конструктор исключение, без того, чтобы вызывать его напрямую. То есть недостаточно, чтобы перемещающий конструктор был открытым и не удаленным, требуется также, чтобы соответствующий тип не был абстрактным классом (ссылки или указатели на абстрактные классы отлично работают). По этой причине свойство типа называется `IsNothrowMoveConstructible` вместо `HasNothrowMoveConstructor`. Для чего-либо иного нам потребуется поддержка компилятора.

Стандартная библиотека C++ предоставляет соответствующее свойство типа `std::is_move_constructible<>`, которое описано в разделе Г.3.2.

19.7.3. Повышение удобства свойств

Одной из распространенных жалоб, связанных со свойствами типов, является жалоба на их относительную многословность, поскольку каждое свойство типа обычно использует завершающую конструкцию `::Type`, а в зависимом контексте — ведущее ключевое слово `typename`; обе конструкции, по сути, являются стандартными вставками. Когда объединяются несколько свойств типа, это может привести к некрасивому и запутанному форматированию — как в нашем примере с оператором сложения для массивов, если мы бы корректно его реализовали, гарантуя, что он не возвращает ни константный, ни ссылочный типы:

```
template<typename T1, typename T2>
Array <
    typename RemoveCVT <
        typename RemoveReferenceT <
            typename PlusResultT<T1, T2>::Type
        ::::Type
    ::::Type
>
operator+ (Array<T1> const&, Array<T2> const&);
```

С помощью шаблонов псевдонимов и переменных можно сделать использование свойств, дающих соответственно типы и значения, более удобным. Однако обратите внимание: в некоторых контекстах эти сокращения недоступны, и мы вынуждены использовать исходный шаблон класса. Мы уже обсуждали одну такую ситуацию в нашем примере `MemberPointerToIntT`, но ниже этот вопрос будет обсуждаться в более общем контексте.

Шаблоны псевдонимов и свойства

Как говорилось в разделе 2.8, шаблоны псевдонимов предлагают способ для уменьшения многословия. Вместо того чтобы выразить свойство типа как шаблон класса с членом-типом `Type`, можно использовать шаблон псевдонима. Например, три следующие шаблона псевдонимов используются как “обертки” для рассматривавшихся выше свойств типов:

```
template<typename T>
using RemoveCV = typename RemoveCVT<T>::Type;

template<typename T>
using RemoveReference = typename RemoveReferenceT<T>::Type;

template<typename T1, typename T2>
using PlusResult = typename PlusResultT<T1, T2>::Type;
```

При наличии этих шаблонов псевдонимов можно упростить объявление нашего оператора `operator+` до

```
template<typename T1, typename T2>
Array<RemoveCV<RemoveReference<PlusResultT<T1, T2>>>
operator+ (Array<T1> const&, Array<T2> const&);
```

Эта версия заметно короче предыдущей и яснее показывает состав свойства. Возможность таких усовершенствований делает шаблоны псевдонимов более подходящими для некоторых применений свойств типов.

Однако у применения шаблонов псевдонимов для свойств типов есть и недостатки.

1. Шаблоны псевдонимов нельзя специализировать (как указывалось в разделе 16.3); а поскольку многие методы написания свойств зависят от специализации, шаблон псевдонима, скорее всего, в любом случае будет необходимо перенаправлять шаблону класса.
2. Некоторые свойства предназначены для специализации пользователями, например свойство, указывающее, является ли некоторая конкретная операция сложения коммутативной. Применение шаблонов псевдонимов может усложнять и запутывать специализацию шаблонов классов.
3. Использование шаблона псевдонима всегда будет инстанцировать тип (например, базовую специализацию шаблона класса), что усложняет устранение инстанцирования тех свойств, которые не имеют смысла для данного типа (см. раздел 19.7.1).

Еще один способ сформулировать последний пункт — шаблоны псевдонимов не могут использоваться с метафункциональной передачей (см. раздел 19.3.2).

Поскольку использование шаблонов псевдонимов для свойств типов имеет как положительные, так и отрицательные аспекты, мы рекомендуем применять их так, как показано в данном разделе и как это делается в стандартной библиотеке C++: предоставляя как шаблон класса с определенным соглашением по именованию (мы выбрали суффикс `T` и член-тип `Type`), так и шаблон псевдонима с несколько иным именованием (мы выбрали суффикс `T`); каждый шаблон псевдонима определяется через соответствующий базовый шаблон класса. Таким образом, мы можем использовать шаблоны псевдонимов везде, где они обеспечивают более понятный код, но всегда можем вернуться к шаблонам классов для более сложного применения.

Обратите внимание на то, что в силу исторических причин стандартная библиотека C++ использует различные соглашения. Шаблоны классов свойств типов указывают тип в члене `type` и не имеют конкретных суффиксов (многие из них были введены в C++11). Соответствующие шаблоны псевдонимов (которые дают тип непосредственно) начали появляться в C++14 и получили суффикс `_t`, так как бессуффиксные имена уже были стандартизированы (см. раздел Г.1).

Шаблоны переменных и свойства

Свойства, возвращающие значение, для получения результата требуют завершающую конструкцию `::value` (или иной выбор члена). В этом случае снизить многословность можно с помощью `constexpr` шаблонов переменных (см. раздел 5.6).

Например, приведенные ниже шаблоны переменных служат в качестве оболочек свойства `IsSameT`, определенного в разделе 19.3.3, и свойства `IsConvertibleT`, определенного в разделе 19.5:

```
template<typename T1, typename T2>
constexpr bool IsSame = IsSameT<T1, T2>::value;

template<typename FROM, typename TO>
constexpr bool IsConvertible = IsConvertibleT<FROM, TO>::value;
```

Теперь мы можем написать просто

```
if (IsSame<T,int> || IsConvertible<T,char>) ...
```

вместо

```
if (IsSameT<T,int>::value || IsConvertibleT<T,char>::value) ...
```

По историческим причинам стандартная библиотека C++ использует различные соглашения. Шаблоны классов свойств возвращают результат в члене `value` без определенного суффикса, и многие из них были введены в стандарте C++11. В стандарте C++17 введены соответствующие шаблоны переменных с суффиксом `_v`, которые дают результатирующее значение непосредственно (см. раздел Г.1)²⁵.

19.8. Классификация типов

Иногда полезно иметь возможность знать, является ли параметр шаблона встроенным типом, типом указателя, класса и так далее. В следующих разделах мы разработаем набор свойств типов, которые позволяют определять различные свойства конкретного типа. В результате мы сможем писать код, специфичный для некоторых конкретных типов:

```
if (IsClassT<T>::value)
{
    ...
}
```

Или при использовании конструкции `if` времени компиляции, доступной начиная с C++17 (см. раздел 8.5) и описанных выше средств повышения удобства свойств (см. раздел 19.7.3):

```
if constexpr(IsClass<T>)
{
    ...
}
```

Или с использованием частичных специализаций:

²⁵ Комитет по стандартизации C++ руководствуется давней традицией, согласно которой все стандартные имена состоят из символов в нижнем регистре и необязательного символа подчеркивания в качестве разделителя. Следовательно, такое имя, как `isSame` или `IsSame`, вряд ли когда-либо будет всерьез рассматриваться в качестве стандартного (за исключением концептов, использующих этот орфографический стиль).

```

template<typename T, bool = IsClass<T>>
class C           // Первичный шаблон для общего случая
{
    ...
};

template<typename T>
class C<T, true> // Частичная специализация для типов классов
{
    ...
};

```

Кроме того, такие выражения, как `IsPointerT<T>::value`, будут логическими константами, которые могут быть корректными аргументами шаблонов, не являющимися типами. Это позволяет создавать более сложные и мощные шаблоны, которые специализируют свое поведение на основе свойств их типовых аргументов.

Стандартная библиотека C++ содержит несколько подобных свойств для определения основных и составных категорий типов²⁶. Подробная информация представлена в разделах Г.2.1 и Г.2.2.

19.8.1. Определение фундаментальных типов

Для начала давайте разработаем шаблон, позволяющий определить, является ли данный тип фундаментальным. По умолчанию полагаем, что тип не является фундаментальным, и специализируем шаблон для фундаментальных типов:

`traits/isfunda.hpp`

```

#include <cstddef>      // Для nullptr_t
#include <type_traits> // Для true_type, false_type и bool_constant>

// Первичный шаблон: в общем случае Т фундаментальным типом не является
template<typename T>
struct IsFundaT : std::false_type
{
};

// Макрос для специализации для фундаментальных типов
#define MK_FUNDA_TYPE(T) \
    template<> struct IsFundaT<T> : std::true_type { \ \
    };

MK_FUNDA_TYPE(void)

MK_FUNDA_TYPE(bool)
MK_FUNDA_TYPE(char)
MK_FUNDA_TYPE(signed char)

```

²⁶ Использование основных (primary) и составных (composite) категорий типов не следует путать с различием между фундаментальными и составными типами. Стандарт описывает фундаментальные (такие как `int` или `std::nullptr_t`) и составные типы (такие как типы указателей или классов). Это понятие отличается от категории составного типа (например, арифметического), которая представляет собой категорию, объединяющую категории основных типов (как, например, числа с плавающей точкой).

```

MK_FUNDA_TYPE(unsigned char)
MK_FUNDA_TYPE(wchar_t)
MK_FUNDA_TYPE(char16_t)
MK_FUNDA_TYPE(char32_t)

MK_FUNDA_TYPE(signed short)
MK_FUNDA_TYPE(unsigned short)
MK_FUNDA_TYPE(signed int)
MK_FUNDA_TYPE(unsigned int)
MK_FUNDA_TYPE(signed long)
MK_FUNDA_TYPE(unsigned long)
MK_FUNDA_TYPE(signed long long)
MK_FUNDA_TYPE(unsigned long long)

MK_FUNDA_TYPE(float)
MK_FUNDA_TYPE(double)
MK_FUNDA_TYPE(long double)

MK_FUNDA_TYPE(std::nullptr_t)

#define MK_FUNDA_TYPE

```

Первичный шаблон определяет общий случай, в котором `IsFundat<T>::value` имеет значение `false`:

```

template<typename T>
struct IsFundat : std::false_type
{
    static constexpr bool value = false;
};

```

Для каждого фундаментального типа специализация определена так, что `IsFundat<T>::value` равно `true`. Для удобства мы определили макрос, который раскрывается в необходимый код. Например,

```
MK_FUNDA_TYPE(bool)
```

раскрывается в следующий код:

```

template<> struct IsFundat<bool> : std::true_type
{
    static constexpr bool value = true;
};

```

Приведенная ниже программа демонстрирует возможное использование этого шаблона:

traits/isfundatest.cpp

```

#include "isfundat.hpp"
#include <iostream>

template<typename T>
void test(T const&)
{
    if (IsFundat<T>::value)

```

```

    {
        std::cout << "T - фундаментальный тип" << '\n';
    }
    else
    {
        std::cout << "T - не фундаментальный тип" << '\n';
    }
}
int main()
{
    test(7);
    test("hello");
}

```

Вывод этой программы имеет следующий вид:

```

T - фундаментальный тип
T - не фундаментальный тип

```

Аналогично можно определить функции типов `IsIntegralT` и `IsFloatingT`, чтобы выяснить, какие типы являются целочисленными скалярными, а какие — скалярными типами с плавающей точкой.

Стандартная библиотека C++ использует более мелкозернистый подход по сравнению с проверкой, является ли тип фундаментальным. Она сначала определяет основные категории типов, где каждый тип соответствует ровно одной категории типа (см. раздел Г.2.1), а затем — составные категории типов, такие как `std::is_integral` или `std::is_fundamental` (см. раздел Г.2.2).

19.8.2. Определение составных типов

Составные типы являются типами, построенными из других типов. Простые составные типы включают типы указателей, ссылок на l- и на r-значение, указателей на члены и массивов. Они построены из одного или двух базовых типов. Типы классов и типы функций также являются составными типами, но в их состав может входить произвольное количество типов (в виде параметров или членов). В этой классификации типы перечислений также рассматриваются как составные типы, не являющиеся простыми, несмотря на то, что они не являются составными из нескольких базовых типов. Простые составные типы могут быть классифицированы с использованием частичной специализации.

Указатели

Начнем с одной простой классификации типов указателей:

`traits/ispointer.hpp`

```

template<typename T>
struct IsPointerT :
    std::false_type // Первичный шаблон: по умолчанию не указатель
{
};

```

```
template<typename T>
struct IsPointerT<T*> :
    std::true_type // Частичная специализация для указателей
{
    using BaseT = T; // Тип, на который указывает указатель
};
```

Первичный шаблон представляет собой “ловушку” для всех типов, не являющихся указателем; как обычно, он предоставляет константу `value`, равную `false` (указывающую, что тип не является указателем), с использованием базового класса `std::false_type`. Частичная специализация перехватывает указатели любого вида (`T*`) и предоставляет константу `value`, равную `true`, говорящую о том, что данный тип является указателем. Кроме того, он предоставляет член-тип `BaseT`, который описывает тип, на который указывает рассматриваемый указатель. Обратите внимание: этот член-тип доступен только тогда, когда исходный тип является указателем, что делает это свойство типа дружественным по отношению к SFINAE (см. раздел 19.4.4).

Стандартная библиотека C++ предоставляет соответствующее свойство типа `std::is_pointer<>`, в котором, однако, не предусмотрен член для типа, на который указывает указатель. Оно описано в разделе Г.2.1.

Ссылки

Аналогично можно идентифицировать тип ссылки на l-значение:

`traits/islvaluerefERENCE.hpp`

```
template<typename T>
struct IsLValueReferenceT :
    std::false_type // По умолчанию не ссылка на l-значение
{};

template<typename T>
struct IsLValueReferenceT<T&> :
    std::true_type // Ссылка на l-значение
{
    using BaseT = T; // Тип, на который указывает ссылка
};
```

и тип ссылки на r-значение:

`traits/isrvaluerefERENCE.hpp`

```
template<typename T>
struct IsRValueReferenceT :
    std::false_type // По умолчанию не ссылка на r-значение
{;
```

```
template<typename T>
struct IsRValueReferenceT < T&& > :
    std::true_type // Ссылка на r-значение
{
    using BaseT = T; // Тип, на который указывает ссылка
};
```

Эти шаблоны можно объединить в свойство `IsReferenceT<>`:

`traits/isreference.hpp`

```
#include "islvaluerefERENCE.hpp"
#include "isrvaluerefERENCE.hpp"
#include "ifthenelse.hpp"

template<typename T>
class IsReferenceT
    : public IfThenElseT<IsLValueReferenceT<T>::value,
        IsLValueReferenceT<T>,
        IsRValueReferenceT<T>
    >::Type
{
};
```

В этой реализации мы используем для выбора в качестве базового класса `IsLValueReference<T>` или `IsRValueReference<T>` шаблон `IfThenElseT` (из раздела 19.7.1) с помощью метафункциональной передачи (рассматривается в разделе 19.3.2). Если `T` представляет собой ссылку на `r`-значение, выполняется наследование от `IsLValueReference<T>`, чтобы получить соответствующее значение `value` и член `BaseT`. В противном случае выполняется наследование от `IsRValueReference<T>`, который определяет, является тип ссылкой на `r`-значение или нет (и в любом случае предоставляет соответствующие члены).

Стандартная библиотека C++ предоставляет соответствующие свойства `std::is_lvalue_reference<>` и `std::is_rvalue_reference<>`, описанные в разделе Г.2.1, и свойство `std::is_reference<>`, которое описано в разделе Г.2.2. Эти свойства не предоставляют член-тип, указывающий тип, на который указывает ссылка.

Массивы

При определении свойств для обнаружения массивов может оказаться неожиданным то, что частичные специализации используют больше параметров, чем первичный шаблон:

`traits/isarray.hpp`

```
#include <cstddef>

template<typename T>
struct IsArrayT : std::false_type // Первичный шаблон: не массив
{
};
```

```

template<typename T, std::size_t N>
struct IsArrayT<T[N]> :
    std::true_type // Частичная специализация для массивов
{
    using BaseT = T;
    static constexpr std::size_t size = N;
};

template<typename T>
struct IsArrayT<T[]> :
    std::true_type // Частичная специализация
{               // для массивов без границ
    using BaseT = T;
    static constexpr std::size_t size = 0;
};

```

Здесь несколько дополнительных членов предоставляют сведения о классифицируемых массивах: их базовый тип и размер (с нулем, используемым для обозначения неизвестного размера).

Стандартная библиотека C++ для проверки, является ли тип массивом, предоставляет соответствующее свойство `std::is_array<>`, которое описано в разделе Г.2.1. Кроме того, такие свойства, как `std::rank<>` и `std::extent<>`, позволяют запрашивать количество измерений и размер конкретного измерения (см. раздел Г.3.1).

Указатели на члены

Обработка указателей на члены использует ту же технологию:

traits/ispointertomember.hpp

```

template<typename T>
struct IsPointerToMemberT :
    std::false_type // По умолчанию - не указатель на член
{
};

template<typename T, typename C>
struct IsPointerToMemberT<T C::*> :
    std::true_type // Частичная специализация
{
    using MemberT = T;
    using ClassT = C;
};

```

Здесь дополнительные члены предоставляют как тип члена, так и тип класса, в котором содержится рассматриваемый член.

Стандартная библиотека C++ предоставляет более конкретные свойства, `std::is_member_object_pointer<>` и `std::is_member_function_pointer<>`, описанные в разделе Г.2.1, а также свойство `std::is_member_pointer<>`, описанное в разделе Г.2.2.

19.8.3. Идентификация типов функций

Типы функций интересны тем, что имеют произвольное количество параметров в дополнение к возвращаемому типу. Таким образом, в частичной специализации, соответствующей типу функции, мы используем пакет параметров для охвата всех типов параметров, как мы делали это в свойстве DecayT в разделе 19.3.2:

traits/isfunction.hpp

```
#include "../typelist/typelist.hpp"

template<typename T>
struct IsFunctionT :
    std::false_type // Первичный шаблон: не функция
{
};

template<typename R, typename... Params>
struct IsFunctionT<R(Params...)> : std::true_type // Функции
{
    using Type = R;
    using ParamsT = Typelist<Params...>;
    static constexpr bool variadic = false;
};

template<typename R, typename... Params>
struct IsFunctionT<R(Params..., ...)> :
    std::true_type // Вариативные функции
{
    using Type = R;
    using ParamsT = Typelist<Params...>;
    static constexpr bool variadic = true;
};
```

Обратите внимание на то, что представлена каждая часть типа функции: Type представляет тип результата, в то время как все параметры охватываются одним списком типов ParamsT (списки типов (typelist) рассматриваются в главе 24, “Списки типов”), а член и variadic указывает, является ли рассматриваемый тип типом функции с переменным количеством аргументов в стиле языка C.

К сожалению, данная формулировка IsFunctionT не обрабатывает все функциональные типы, потому что типы функций могут иметь квалификаторы `const` и `volatile`, а также квалификаторы ссылок на l-значения (`&`) и r-значения (`&&`) (описаны в разделе B.2.1) и, начиная с C++17, квалификаторы `noexcept`. Например:

```
using MyFuncType = void (int&) const;
```

Такие типы функций могут осмысленно использоваться только для нестатических функций-членов, но, тем не менее, остаются типами функций. Кроме того, типы функций, помеченные как `const`, на самом деле не являются константными

типами²⁷, так что RemoveConst не может удалить `const` из типа функции. Таким образом, чтобы распознать типы функций, которые имеют квалификаторы, нам необходимо ввести большое количество дополнительных частичных специализаций, охватывающих все комбинации квалификаторов (как для функций с переменным количеством аргументов в стиле языка C, так и для функций, не являющихся таковыми). Здесь мы покажем только пять из множества²⁸ необходимых частичных специализаций:

```
template<typename R, typename... Params>
struct IsFunctionT<R(Params...) const> : std::true_type
{
    using Type = R;
    using ParamsT = Typelist<Params...>;
    static constexpr bool variadic = false;
};

template<typename R, typename... Params>
struct IsFunctionT<R(Params..., ...) volatile> : std::true_type
{
    using Type = R;
    using ParamsT = Typelist<Params...>;
    static constexpr bool variadic = true;
};

template<typename R, typename... Params>
struct IsFunctionT<R(Params..., ...) const volatile> : std::true_type
{
    using Type = R;
    using ParamsT = Typelist<Params...>;
    static constexpr bool variadic = true;
};

template<typename R, typename... Params>
struct IsFunctionT<R(Params..., ...)> : std::true_type
{
    using Type = R;
    using ParamsT = Typelist<Params...>;
    static constexpr bool variadic = true;
};

template<typename R, typename... Params>
struct IsFunctionT<R(Params..., ...) const&> : std::true_type
{
    using Type = R;
    using ParamsT = Typelist<Params...>;
    static constexpr bool variadic = true;
};

***
```

²⁷ В частности, когда тип функции помечен как `const`, это относится к квалифиликатору объекта, на который указывает ее неявный параметр, тогда как `const` в константном типе ссылается на объект фактического типа.

²⁸ В соответствии с последним стандартом их общее количество — 48.

Собрав все эти свойства, мы теперь можем классифицировать все типы, за исключением типов классов и типов перечислений. Мы рассмотрим эти случаи в следующих разделах.

Стандартная библиотека C++ предоставляет свойство `std::is_function<>`, описанное в разделе Г.2.1.

19.8.4. Обнаружение типов классов

В отличие от других составных типов, с которыми мы имели дело до сих пор, частичной специализации шаблонов, которые соответствуют конкретно типам классов, не существует. Не имеет смысла и перечисление всех типов классов, как это было сделано для фундаментальных типов. Вместо этого мы должны использовать косвенный метод для идентификации типов классов, применяя некоторый тип или выражение, которое является действительным для всех типов классов (но не для прочих типов). С помощью такого типа или выражения можно применить методы свойств с использованием SFINAE, рассматривавшиеся в разделе 19.4.

В этом случае наиболее удобным свойством типов классов является то, что только типы класса могут использоваться в качестве базовых типов в указателях на член. Таким образом, в конструкции типа вида `X Y::* Y` может быть только типом класса. Следующая формулировка `IsClassT<>` использует это свойство (и выбирает `int` для типа `X` произвольным образом):

`traits/isclass.hpp`

```
#include <type_traits>

template<typename T, typename = std::void_t<>>
struct IsClassT
{
    : std::false_type // Первичный шаблон: по умолчанию не класс
};

template<typename T>
struct IsClassT<T, std::void_t<int T::*>> // Иметь указатель на
{
    : std::true_type // член могут только классы
};
```

Язык C++ определяет, что типом лямбда-выражения является “*уникальный неименованный тип класса, не являющийся объединением*”. По этой причине лямбда-выражения дают `true` при выяснении, являются ли они объектами типа класса:

```
auto l = [] {};
static_assert<IsClassT<decltype(l)>::value, "">; // Успешно
```

Заметим, что выражение `int T::*` является корректным для типов `union` (которые согласно стандарту C++ также являются типами классов).

Стандартная библиотека C++ предоставляет свойства `std::is_class<>` и `std::is_union<>`, которые описаны в разделе Г.2.1. Однако эти свойства требуют специальной поддержки компилятора, чтобы отличать типы `class` и `struct`.

от типов `union`, что в настоящее время не может быть сделано с использованием стандартных методов языка²⁹.

19.8.5. Обнаружение типов перечислений

Единственные типы, которые остались не классифицированными никакими из рассмотренных выше свойств, — это типы перечислений. Тестирование типов перечисления может быть выполнено непосредственно путем написания на основе SFINAE свойства, которое проверяет наличие явного преобразования в целочисленный тип (например, `int`) и явно исключает фундаментальные типы, типы классов, ссылочные типы, типы указателей и указателей на члены, которые могут быть преобразованы в целочисленный тип, но не являются перечислениями³⁰. Вместо этого мы просто заметим, что любой тип, который не попадает ни в одну из прочих категорий, должен быть типом перечисления, что мы можем реализовать следующим образом:

`traits/isenum.hpp`

```
template<typename T>
struct IsEnumT
{
    static constexpr bool value = !IsFundamentalT<T>::value &&
        !IsPointerT<T>::value &&
        !IsReferenceT<T>::value &&
        !IsArrayT<T>::value &&
        !IsPointerToMemberT<T>::value &&
        !IsFunctionT<T>::value &&
        !IsClassT<T>::value;
};
```

Стандартная библиотека C++ предоставляет свойство `std::is_enum<>`, которое описано в разделе Г.2.1. Обычно для улучшения производительности компиляции компиляторы обеспечивают непосредственную поддержку такого свойства вместо реализации его в качестве “чего-то иного”, как описано здесь.

19.9. Свойства стратегий

До сих пор наши примеры шаблонов свойств использовались для определения свойств параметров шаблона: какой тип они представляют, тип результата

²⁹ Большинство компиляторов поддерживают встроенные операторы, такие как `_is_union`, чтобы помочь реализовать различные шаблоны свойств стандартных библиотек. Это верно даже для некоторых свойств, которые технически могут быть реализованы с использованием методов, описанных в этой главе, потому что применение встроенных функций может улучшить производительность компиляции.

³⁰ Первое издание этой книги описывало обнаружение типа перечисления таким способом. Однако он проверяет неявное преобразование в целочисленный тип, которого было достаточно для стандарта C++98. Введение в язык перечислимых типов с областями видимости, которые не имеют такого неявного преобразования, усложняет обнаружение типов перечислений.

выполнения оператора, примененного к значениям этого типа, и так далее. Такие свойства называются *свойствами свойств* (property traits).

Хотя такие свойства часто могут быть реализованы как функции типа, свойства стратегий обычно инкапсулируют стратегии в функциях-членах. Чтобы проиллюстрировать это понятие, взглянем на функцию типа, которая определяет стратегию передачи параметров только для чтения.

19.9.1. Типы параметров только для чтения

В Си С++ аргументы вызова функции по умолчанию передаются по значению. Это значит, что значения аргументов, вычисленные вызывающим кодом, копируются в местоположения, контролируемые вызываемым кодом. Программисты знают, что для больших структур это может быть слишком дорого и для таких структур уместно передавать аргументы как ссылки на `const` (или указатель на `const` в С). Для более мелких структур картина не всегда ясна, и наилучший с точки зрения производительности механизм зависит от конкретной архитектуры, для которой пишется код. В большинстве случаев это не столь важно, но иногда даже небольшие структуры следуют обрабатывать с осторожностью.

При использовании шаблонов вопрос становится еще более деликатным: ведь мы не знаем заранее, насколько большим будет подставляемый вместо параметра шаблона тип. Кроме того, решение зависит не только от размера: маленькая структура может иметь дорогой копирующий конструктор, что оправдывало бы применение передачи параметров с помощью ссылки на `const`.

Как уже говорилось ранее, эту задачу удобно решать с помощью шаблона свойства стратегии, который является функцией типа: эта функция сопоставляет предполагаемый тип аргумента `T` с оптимальным параметром типа `T` или `T const&`. В первом приближении первичный шаблон может использовать передачу по значению для типов, не превышающих два указателя, и по константной ссылке — для всего остального:

```
template<typename T>
struct RParam
{
    using Type = typename IfThenElseT<sizeof(T) <= 2 * sizeof(void*),
                                              T, T const & >::Type;
};
```

С другой стороны, типы контейнеров, для которых `sizeof` возвращает малое значение, могут включать дорогие копирующие конструкторы, поэтому может понадобиться много специализаций и частичных специализаций, например таких:

```
template<typename T>
struct RParam<Array<T>>
{
    using Type = Array<T> const &;
};
```

Поскольку такие типы распространены в C++, может быть безопаснее маркировать как передаваемые по значению только малые типы с тривиальными копирующими и перемещающими конструкторами³¹, а затем выборочно добавлять другие типы классов, когда соображения производительности заставляют поступать таким образом (свойства типов `std::is_trivially_copy_constructible` и `std::is_trivially_move_constructible` являются частью стандартной библиотеки C++).

traits/rparam.hpp

```
#ifndef RPARAM_HPP
#define RPARAM_HPP

#include "ifthenelse.hpp"
#include <type_traits>

template<typename T>
struct RParam
{
    using Type
        = IfThenElse<(sizeof(T) <= 2 * sizeof(void*)
                      && std::is_trivially_copy_constructible<T>::value
                      && std::is_trivially_move_constructible<T>::value),
                      T, T const& >;
};

#endif // RPARAM_HPP
```

В любом случае теперь стратегия может быть сосредоточена в определении шаблона свойства, и клиенты могут использовать его, чтобы получить соответствующий результат. Например, предположим, что у нас есть два класса, причем об одном классе известно, что для аргументов только для чтения предпочтительнее передача по значению:

traits/rparamcls.hpp

```
#include "rparam.hpp"
#include <iostream>

class MyClass1
{
public:
    MyClass1()
    {
    }
    MyClass1(MyClass1 const&)
    {
        std::cout << "Копирующий конструктор MyClass1\n";
    }
};
```

³¹ Копирующий или перемещающий конструктор называется *тривиальным*, если его вызов фактически заменяется простым копированием байтов.

```

class MyClass2
{
public:
    MyClass2()
    {
    }
    MyClass2(MyClass2 const&)
    {
        std::cout << "Копирующий конструктор MyClass2\n";
    }
};

// Передача объектов MyClass2 с RParam<> по значению
template<>
class RParam<MyClass2>
{
public:
    using Type = MyClass2;
};

```

Теперь можно объявить функции, использующие для параметров только для чтения RParam<>, и вызвать их:

traits/rparam1.cpp

```

#include "rparam.hpp"
#include "rparamcls.hpp"

// Функция, разрешающая передачу параметров по значению и по ссылке
template<typename T1, typename T2>
void foo(typename RParam<T1>::Type p1,
          typename RParam<T2>::Type p2)
{
    ...
}

int main()
{
    MyClass1 mcl;
    MyClass2 mc2;
    foo<MyClass1, MyClass2>(mcl, mc2);
}

```

К сожалению, у использования RParam имеются некоторые существенные недостатки. Во-первых, объявление функции достаточно запутанное. Во-вторых, функция наподобие `foo()` не может вызываться с помощью вывода аргумента, потому что параметр шаблона появляется только в квалификаторах параметров функции (и это, возможно, главный недостаток). Поэтому при вызове необходимо указывать явные аргументы шаблона.

Громоздкий обходной путь заключается в использовании встроенного шаблона оболочки функции, который обеспечивает прямую передачу (раздел 15.6.3), но этот путь предполагает, что встроенная функция будет удалена компилятором. Например:

traits/rparam2.cpp

```
#include "rparam.hpp"
#include "rparamcls.hpp"

// Функция, позволяющая передачу параметров по значению и по ссылке
template<typename T1, typename T2>
void foo_core(typename RParam<T1>::Type p1,
              typename RParam<T2>::Type p2)
{
    ...
}

// Оболочка для устранения явной передачи параметров шаблона
template<typename T1, typename T2>
void foo(T1& p1, T2& p2)
{
    foo_core<T1, T2>(std::forward<T1>(p1), std::forward<T2>(p2));
}

int main()
{
    MyClass1 mc1;
    MyClass2 mc2;
    foo(mc1, mc2); // Эквивалентно foo_core<MyClass1, MyClass2>(mc1, mc2)
}
```

19.10. В стандартной библиотеке

В C++11 свойства типов стали неотъемлемой частью стандартной библиотеки C++. Они включают практически все функции и свойства типов, описанные в этой главе. Однако для, например, обнаружения тривиальных операций или обсуждавшегося свойства `std::is_union` не существует известных решений, основанных только на возможностях языка. Обычно для таких свойств предоставляет поддержку компилятор. Кроме того, для сокращения времени компиляции компиляторы начинают поддерживать даже те свойства, для которых в языке имеется возможность их реализации.

Поэтому, если вам нужны свойства типов, мы рекомендуем использовать свойства из стандартной библиотеки C++ (при наличии таковых). Все они подробно описаны в приложении Г, “Стандартные утилиты для работы с типами”.

Обратите внимание на то, что (как обсуждалось) некоторые свойства имеют потенциально удивительно поведение (по крайней мере для начинающего программиста). В дополнение к общим подсказкам, которые мы предоставляем в разделах 11.2.1 и Г.1.2, следует также рассмотреть конкретные описания, представленные в приложении Г, “Стандартные утилиты для работы с типами”.

Стандартная библиотека C++ определяет также некоторые свойства стратегий и свойств.

- Шаблон класса `std::char_traits` используется в качестве параметра свойств стратегии классами строк и потоков ввода-вывода.

- Для легкой адаптации алгоритмов к разным стандартным итераторам, для которых они используются, предоставляется (и используется в интерфейсах стандартной библиотеки) очень простой шаблон свойств `std::iterator_traits`.
- Шаблон `std::numeric_limits` также может быть полезен в качестве шаблона свойств.
- И, наконец, выделение памяти для типов стандартных контейнеров обрабатывается с помощью классов свойств стратегий. Начиная с C++98, для этой цели в качестве стандартного компонента предоставляется шаблон `std::allocator`. Начиная с C++11, чтобы иметь возможность изменять стратегию/поведение распределителей, был добавлен шаблон `std::allocator_traits` (переключение между классическим поведением и распределителями с областями видимости; последние не могут быть использованы в каркасах до стандарта C++11).

19.11. Заключительные замечания

Натан Майерс (Nathan Myers) был первым, кто формализовал идею параметров свойств. Он первоначально представил их Комитету по стандартизации C++ как средство для определения того, как символьные типы должны рассматриваться компонентами стандартной библиотеки (например, входными и выходными потоками). В то время он называл их *багажными шаблонами* (*baggage templates*) и отмечал, что они содержат свойства. Однако некоторым членам Комитета не понравился термин *багаж*, и вместо него стало использоваться название *свойство* (*trait*). С тех пор этот термин получил широкое распространение.

Обычно клиентский код не работает со свойствами: классы свойств по умолчанию удовлетворяют наиболее широко распространенные потребности, а поскольку они являются аргументами шаблона по умолчанию, они и не должны появляться в исходном тексте клиента вовсе. Это является доводом в пользу длинных описательных имен для шаблонов свойств по умолчанию. Когда клиентский код адаптирует поведение шаблона, предоставляемого в качестве аргумента пользовательское свойство, хорошей практикой является объявление имени псевдонима типа для результирующих специализаций, подходящих для пользовательского поведения. В этом случае класс свойств может получить длинное описательное имя без ущерба для исходных текстов.

Свойства можно использовать как разновидность *рефлексии*, в которой программа проверяет собственные свойства высокого уровня (такие, как структура типов). Такие свойства, как `IsClassT` и `PlusResultT`, как и многие другие свойства типов, исследующие типы в программе, реализуют формы *рефлексии времени компиляции*, которые оказываются мощным союзником *метапрограммирования* (см. главу 23, “Метапрограммирование”, и раздел 17.9).

Идея хранения свойств типов как членов специализаций шаблонов восходит по крайней мере к середине 1990-х годов. Среди ранних серьезных приложений

шаблонов для классификации типов был шаблон `__type_traits` из реализации STL, разработанной SGI (тогда известной как *Silicon Graphics*). Шаблон SGI предназначался для представления некоторых свойств его аргумента (например, был это *простой старый тип данных* (*plain old data – POD*) либо тривиален ли его деструктор). Затем эта информация использовалась для оптимизации определенных алгоритмов STL для данного типа. Интересной особенностью решения SGI было то, что некоторые компиляторы SGI распознавали специализации `__type_traits` и предоставляли информацию об аргументах, которая не могла быть получена с помощью стандартных методов. (Обобщенная реализация шаблона `__type_traits` была безопасна для использования, хотя и субоптимальна.)

Boost обеспечивает достаточно полный набор шаблонов для классификации типов ([22]), который лег в основу заголовочного файла `<type_traits>` стандартной библиотеки C++ в 2011 году. Хотя многие из этих свойств могут быть реализованы с использованием приемов, описанных в этой главе, другие (такие как `std::is_pod` для обнаружения POD) требуют поддержки компилятора, как и специализации `__type_traits`, предоставляемые компиляторами SGI.

Использование принципа SFNAE для целей классификации типов было замечено при уточнении правил вывода и подстановки в ходе первых усилий по стандартизации. Однако он никогда не был формально документирован, и в результате позднее пришлось затратить много усилий, пытаясь воссоздать некоторые из методов, описанных в этой главе. Первое издание данной книги было одним из наиболее ранних источников этой технологии, и именно в нем был введен термин *SFINAE*. Одним из первых известных разработчиков в этой области был Андрей Александреску (Andrei Alexandrescu), который сделал популярным использование оператора `sizeof` для определения результата разрешения перегрузки. Этот метод стал достаточно популярным, и стандарт 2011 года дал возможность распространить действие SFNAE на произвольные ошибки в непосредственном контексте шаблонов функций ([62]). Это расширение действия SFNAE в сочетании с введением `decltype`, ссылок на *г*-значения и вариативных шаблонов значительно увеличило возможности проверки определенных свойств в пределах свойств.

Применение обобщенных лямбда-выражений наподобие `isValid`, которые выявляют суть условия SFNAE, представляет собой технологию, представленную Луи Дионном (Louis Dionne) в 2015 году, которая используется в Boost.Hana [14], библиотеке метапрограммирования, предназначеннной для вычислений времени компиляции, выполняемых над типами и значениями.

Классы стратегий, по-видимому, были разработаны многими программистами и несколькими авторами книг. Андрей Александреску сделал популярным термин *классы стратегий* (policy classes), а его книга *Современное проектирование на C++* охватывает эту тему куда более подробно, чем наш краткий раздел ([3]).

Глава 20

Перегрузка свойств типов

Перегрузка функций позволяет использовать одно и то же имя для нескольких различных функций, лишь бы они отличались типами своих параметров. Например:

```
void f(int);  
void f(char const*);
```

При использовании шаблонов функций перегрузка выполняется с учетом схемы типов, как, например, для указателя на T или $\text{Array} < \text{T} >$:

```
template<typename T> void f(T*);  
template<typename T> void f(Array<T>);
```

Учитывая распространенность свойств типов (рассматривавшихся в главе 19, “Реализация свойств типов”), вполне естественным оказывается желание перегрузки шаблонов функций на основе свойств аргументов шаблона. Например:

```
template<typename Number> void f(Number); // Только для чисел  
template<typename Container> void f(Container); // Только для контейнеров
```

Однако в настоящее время C++ не обеспечивает никакого способа непосредственного выражения перегрузки на основе свойств типов. Два шаблона функций f, показанные выше, фактически являются объявлением одного и того же шаблона функции, а не различных перегрузок, потому что при сравнении двух шаблонов функций имена параметров шаблона игнорируются.

К счастью, есть целый ряд методов, которые могут использоваться для эмуляции перегрузки шаблонов функций на основе свойств типов. Именно эти методы, а также общие мотивы такой перегрузки рассматриваются в данной главе.

20.1. Специализация алгоритма

Одним из распространенных мотивов перегрузок шаблонов функций является предоставление более специализированных версий алгоритмов, основанное на знании особенностей используемых типов. Рассмотрим простую операцию swap() для обмена двух значений:

```
template<typename T>  
void swap(T& x, T& y)  
{  
    T tmp(x);  
    x = y;  
    y = tmp;  
}
```

Данная реализация включает в себя три операции копирования. Однако для некоторых типов можно предоставить более эффективную операцию swap(), как,

например, для типа `Array<T>`, который хранит свои данные в виде указателя на содержимое массива и его длину:

```
template<typename T>
void swap(Array<T>& x, Array<T>& y)
{
    swap(x.ptr, y.ptr);
    swap(x.len, y.len);
}
```

Обе реализации `swap()` корректно обменивают содержимое двух объектов `Array<T>`. Однако последняя реализация более эффективна, так как она использует дополнительные свойства `Array<T>` (в частности, информацию о наличии и смысле `ptr` и `len`), которые не доступны для произвольного типа¹. Последний шаблон функции (концептуально) более специализирован, чем первый, потому что он выполняет ту же операцию для подмножества типов, принимаемых первым шаблоном функции. К счастью, второй шаблон функции является также более специализированным на основе правил частичного упорядочения шаблонов функций (см. раздел 16.2.2), поэтому компилятор выберет более специализированный (а следовательно, и более эффективный) шаблон функции при его применимости для данного типа (т.е. для аргументов `Array<T>`) и вернется к более общему (и потенциально менее эффективному) алгоритму, когда более специализированная версия окажется неприменима.

Подход проектирования и оптимизации более специализированных вариантов универсального алгоритма называется *специализацией алгоритма*. Более специализированные варианты применяются к подмножеству допустимых входных данных универсального алгоритма, которое задается определенными типами или свойствами типов, и обычно являются более эффективными, чем обобщенные реализации универсального алгоритма.

Решающим значением для реализации специализации алгоритма является то свойство, что более специализированные варианты, когда они применимы, выбираются автоматически, так что вызывающему коду даже не нужно знать об их существовании. В нашем примере с шаблоном `swap()` это было достигнуто путем перегрузки (концептуально) более специализированным шаблоном функции (второй `swap()`) более общего шаблона функции (первый `swap()`), и гарантии того, что более специализированный шаблон функции является также более специализированным на основе правил частичного упорядочения C++.

Не все концептуально более специализированные варианты алгоритмов могут непосредственно транслироваться в шаблоны функций, которые обеспечивают правильное поведение частичного упорядочения. Рассмотрим в нашем следующем примере функцию `advanceIter()` (аналог `std::advance()` из стандартной библиотеки C++), которая перемещает итератор `x` вперед на `n` шагов. Этот общий алгоритм может работать с любым *входным итератором* (*input iterator*):

¹Наилучшим вариантом для `swap()` является использование `std::move()` для того, чтобы избежать копирования в первичном шаблоне. Однако представленная здесь альтернатива применима более широко.

```
template<typename InputIterator, typename Distance>
void advanceIter(InputIterator& x, Distance n)
{
    while (n > 0) // Линейное время
    {
        ++x;
        --n;
    }
}
```

Для определенного класса итераторов (которые предоставляют возможность произвольного доступа) можно предложить более эффективную реализацию:

```
template<typename RandomAccessIterator, typename Distance>
void advanceIter(RandomAccessIterator& x, Distance n)
{
    x += n; // Константное время
}
```

К сожалению, определение обоих шаблонов функций приведет к ошибке компилятора, поскольку, как отмечалось во введении, шаблоны функций, отличающиеся только именами параметров шаблонов, не перегружаются. В остальной части этой главы мы рассмотрим методы, имитирующие желаемый эффект перегрузки этих шаблонов функций.

20.2. Диспетчеризация дескрипторов

Одним из подходов к специализации алгоритмов является присваивание “дескрипторов” (“тегов”) различным вариантам реализации алгоритма в виде уникального типа, который идентифицирует конкретный вариант. Например, чтобы справиться с только что представленной проблемой `advanceIter()`, можно использовать типы дескрипторов категорий итераторов стандартной библиотеки (определенны ниже) для идентификации двух вариантов реализации алгоритма `advanceIter()`:

```
template<typename Iterator, typename Distance>
void advanceIterImpl(Iterator& x, Distance n, std::input_iterator_tag)
{
    while (n > 0) // Линейное время
    {
        ++x;
        --n;
    }
}
template<typename Iterator, typename Distance>
void advanceIterImpl(Iterator& x, Distance n,
                     std::random_access_iterator_tag)
{
    x += n; // Константное время
}
```

Тогда шаблон функции `advanceIter()` просто передает свои аргументы и соответствующий дескриптор:

```
template<typename Iterator, typename Distance>
void advanceIter(Iterator& x, Distance n)
{
    advanceIterImpl(x, n,
                    typename
                    std::iterator_traits<Iterator>::iterator_category());
}
```

Класс свойств `std::iterator_traits` предоставляет категорию итератора через его член-тип `iterator_category`. Категория итератора является одним из типов `_tag`, упоминавшихся ранее, который указывает, с какой разновидностью итератора мы имеем дело. В стандартной библиотеке C++ доступные дескрипторы определены с помощью наследования, отражающего ситуацию, когда один дескриптор описывает категорию, которая является производной от другой²:

```
namespace std
{
    struct input_iterator_tag{};
    struct output_iterator_tag{};
    struct forward_iterator_tag : public input_iterator_tag{};
    struct bidirectional_iterator_tag:public forward_iterator_tag{};
    struct random_access_iterator_tag:public bidirectional_iterator_tag{};
}
```

Ключ к эффективному использованию диспетчеризации дескрипторов заключается в отношениях между дескрипторами. Наши два варианта `advanceIterImpl()` отмечены дескрипторами `std::input_iterator_tag` и `std::random_access_iterator_tag`, а поскольку дескриптор `std::random_access_iterator_tag` наследуется от `std::input_iterator_tag`, обычная перегрузка функций при вызове `advanceIterImpl()` с итератором произвольного доступа предпочтет более специализированный вариант алгоритма (который использует `std::random_access_iterator_tag`). Таким образом, диспетчеризация дескрипторов основана на делегировании функциональности из единственного, первичного шаблона функции набору вариантов `_impl`, помеченных таким образом, что обычная перегрузка функций выберет наиболее специализированный алгоритм, применимый к данным аргументам шаблона.

Диспетчеризация хорошо работает, когда имеется естественная иерархическая структура характеристик, используемых алгоритмом, и существующий набор классов свойств, которые предоставляют эти значения дескрипторов. Данный способ не столь удобен, когда специализация алгоритма зависит от произвольных свойств типов, например от того, имеет ли тип `T` тривиальный оператор копирующего присваивания. Для этого нам нужна более мощная методика.

20.3. Включение/отключение шаблонов функций

Специализация алгоритмов включает в себя предоставление различных шаблонов функций, которые выбираются на основе свойств аргументов шаблона.

² В данном случае категории отражают *концепты*, а наследование концептов известно как *уточнение*. Концепты и уточнения детально описаны в приложении Д, “Концепты”.

К сожалению, ни частичное упорядочение шаблонов функции (раздел 16.2.2), ни разрешение перегрузки (приложение В, “Разрешение перегрузки”) не являются достаточно мощными для того, чтобы выразить более сложные формы специализации алгоритмов.

Одним из помощников для решения этих вопросов, которые предоставляет стандартная библиотека C++, является шаблон `std::enable_if`, описанный в разделе 6.3. В данном разделе рассматривается, как можно реализовать этот вспомогательный шаблон путем введения соответствующего шаблона псевдонима, который мы назовем `EnableIf` во избежание коллизии имен.

Так же, как и `std::enable_if`, шаблон псевдонима `EnableIf` можно использовать для того, чтобы включать (или отключать) конкретные шаблоны функций при определенных условиях. Например, версия с произвольным доступом алгоритма `advanceIter()` может быть реализована следующим образом:

```
template<typename Iterator>
constexpr bool IsRandomAccessIterator =
    IsConvertible<
        typename std::iterator_traits<Iterator>::iterator_category,
        std::random_access_iterator_tag>;
```



```
template<typename Iterator, typename Distance>
EnableIf<IsRandomAccessIterator<Iterator>>
    advanceIter(Iterator& x, Distance n)
{
    x += n; // Константное время
}
```

Здесь специализация `EnableIf` используется для включения данного варианта `advanceIter()` только тогда, когда итератор на самом деле является итератором произвольного доступа. Эти два аргумента `EnableIf` являются логическим условием, указывающим, следует ли включить этот шаблон, и типом, получаемым при раскрытии `EnableIf`, когда условие имеет значение `true`. В нашем примере выше мы использовали свойство типа `IsConvertible`, представленное в разделах 19.5 и 19.7.3, в качестве условия для определения свойства типа `IsRandomAccessIterator`. Таким образом, эта конкретная версия нашей реализации `advanceIter()` рассматривается только тогда, когда конкретный тип, подставляемый вместо `Iterator`, может использоваться в качестве итератора с произвольным доступом (т.е. он связан с дескриптором, преобразуемым в `std::random_access_iterator_tag`).

Шаблон `EnableIf` имеет очень простую реализацию:

`typeoverload/enableif.hpp`

```
template<bool, typename T = void>
struct EnableIfT
{
};
```

```

template<typename T>
struct EnableIfT<true, T>
{
    using Type = T;
};

template<bool Cond, typename T = void>
using EnableIf = typename EnableIfT<Cond, T>::Type;

```

EnableIf раскрывается в тип и, следовательно, реализован как шаблон псевдонима. Для его реализации мы хотим использовать частичную специализацию (см. главу 16, “Специализация и перегрузка”), но шаблоны псевдонимов не могут быть частично специализированы. К счастью, можно ввести шаблон вспомогательного класса EnableIfT, который делает всю необходимую фактическую работу, нужную нам, и тогда шаблон псевдонима EnableIf просто выбирает результирующий тип из вспомогательного шаблона. Если условие истинно, EnableIfT<...>::Type (а значит, и EnableIf<...>) просто вычисляется как второй аргумент шаблона T. Когда условие имеет значение false, EnableIf не производит допустимый тип, потому что первичный шаблон класса EnableIfT не имеет члена с именем Type. Обычно это рассматривается как ошибка, но в контексте SFINAE (см. раздел 15.7) в качестве возвращаемого типа шаблона функции это приводит к сбою вывода аргумента шаблона, так что данный шаблон функции удаляется из рассмотрения³.

В advanceIter() использование EnableIf означает, что шаблон функции будет доступен (и иметь возвращаемый тип void), когда аргумент Iterator будет итератором произвольного доступа, а если Iterator не является таковым — этот шаблон будет полностью удален из рассмотрения. Мы можем рассматривать EnableIf как способ “охранять” шаблоны от инстанцирования с аргументами шаблона, которые не соответствуют требованиям его реализации (поскольку этот advanceIter() может быть инстанцирован только с итератором произвольного доступа, так как он требует операции, которые доступны только для итераторов с произвольным доступом). Хотя защита с помощью EnableIf не является совершенно “пуленепробиваемой” (пользователь может утверждать, что предоставляемый им тип является итератором с произвольным доступом, без предоставления необходимых операций), данный метод может помочь ранней диагностике распространенных ошибок.

Теперь мы знаем, как явно “активировать” более специализированный шаблон для типов, к которым он применим. Однако этого недостаточно: мы должны также “деактивировать” менее специализированный шаблон, поскольку компилятор не имеет возможности “заказать” два шаблона, так что при применимости обеих версий он сообщит об ошибке неоднозначности. К счастью, решить эту проблему легко: мы просто используем тот же шаблон EnableIf для менее

³ EnableIf можно также поместить в качестве значения параметра шаблона по умолчанию, что имеет определенные преимущества по сравнению с размещением в качестве типа результата (см. вопрос о размещении шаблона EnableIf в разделе 20.3.2).

специализированного шаблона, но только с отрицанием условного выражения. Это гарантирует, что для любого конкретного типа итератора будет активирован только один из двух шаблонов. Таким образом, наша версия `advanceIter()` для итератора, который не является итератором с произвольным доступом, принимает следующий вид:

```
template<typename Iterator, typename Distance>
EnableIf < !IsRandomAccessIterator<Iterator>>
    advanceIter(Iterator& x, Distance n)
{
    while (n > 0) // Линейное время
    {
        ++x;
        --n;
    }
}
```

20.3.1. Предоставление нескольких специализаций

Предыдущая схема обобщается для случаев, где требуется более двух альтернативных реализаций. Мы просто оснащаем все альтернативы конструкциями `EnableIf`, условия которых являются взаимоисключающими для определенного набора конкретных аргументов шаблона. Эти условия обычно используют различные характеристики типов, которые могут быть выражены через их свойства.

Рассмотрим, например, добавление третьего варианта алгоритма `advanceIter()`: на этот раз мы хотим разрешить перемещение “назад”, указывая отрицательное расстояние⁴. Это явно недопустимо для входного итератора и вполне корректно для итератора произвольного доступа. Однако стандартная библиотека включает в себя еще и понятие двунаправленного итератора, который также допускает передвижение назад без необходимости произвольного доступа. Реализация этого варианта требует несколько более сложной логики: каждый шаблон функции должен использовать `EnableIf` с условием, которое является взаимоисключающим с условиями всех прочих шаблонов функций, представляющих различные варианты алгоритма. Это приводит нас к следующему набору условий.

- Итератор произвольного доступа: случай произвольного доступа (контактное время; перемещение и вперед, и назад).
- Двунаправленный итератор без произвольного доступа: двунаправленный случай (линейное время; перемещение и вперед, и назад).
- Входной итератор, не являющийся двунаправленным: общий случай (линейное время; перемещение только вперед).

⁴ Обычно специализация алгоритма используется только для достижения большей эффективности — либо в смысле времени вычисления, либо в смысле использования ресурсов. Однако некоторые специализации алгоритмов обеспечивают большую функциональность, как, например, в данном случае — возможность перемещения назад по последовательности.

Описанную ситуацию реализует следующий набор шаблонов функций:

typeoverload/advance2.hpp

```
#include <iterator>

// Реализация для итераторов произвольного доступа:
template<typename Iterator, typename Distance>
EnableIf<IsRandomAccessIterator<Iterator>>
advanceIter(Iterator& x, Distance n)
{
    x += n; // Константное время
}

template<typename Iterator>
constexpr bool IsBidirectionalIterator =
    IsConvertible <
        typename std::iterator_traits<Iterator>::iterator_category,
        std::bidirectional_iterator_tag >

// Реализация для двунаправленных итераторов:
template<typename Iterator, typename Distance>
EnableIf < IsBidirectionalIterator<Iterator>&&
           !IsRandomAccessIterator<Iterator> >
advanceIter(Iterator& x, Distance n)
{
    if (n > 0)
    {
        for (; n > 0; ++x, --n)      // Линейное время
        {
        }
    }
    else
    {
        for (; n < 0; --x, ++n)      // Линейное время
        {
        }
    }
}

// Реализация для всех прочих итераторов:
template<typename Iterator, typename Distance>
EnableIf < !IsBidirectionalIterator<Iterator> >
advanceIter(Iterator& x, Distance n)
{
    if (n < 0)
    {
        throw "advanceIter(): неверная категория итератора";
    }

    while (n > 0)      // Линейное время
    {
        ++x;
        --n;
    }
}
```

Делая условие `EnableIf` для каждого шаблона функции взаимоисключающим с условиями `EnableIf` для всех других шаблонов функций, мы гарантируем, что для каждого данного набора аргументов успешно выводится не более одного шаблона функции.

Наш пример иллюстрирует один из недостатков использования `EnableIf` для специализации алгоритмов: каждый раз, когда вводится новый вариант алгоритма, необходимо пересмотреть условия всех вариантов алгоритма для обеспечения того, чтобы все они были взаимно исключающими. В отличие от этого введение варианта для двунаправленного итератора с помощью диспетчеризации дескрипторов (раздел 20.2) требует добавления только новой перегрузки `advanceImpl()` с использованием дескриптора `std::bidirectional_iterator_tag`.

Обе методики — диспетчеризация дескрипторов и `EnableIf` — оказываются полезными в различных контекстах. В общем случае диспетчеризация дескрипторов поддерживает простую диспетчеризацию на основе иерархии дескрипторов, в то время как `EnableIf` поддерживает более сложную диспетчеризацию на основе произвольного набора характеристик, определяемых свойствами типов.

20.3.2. Откуда берется `EnableIf`?

`EnableIf` обычно используется в качестве типа возвращаемого значения шаблона функции. Однако этот подход не работает для шаблонов конструкторов или шаблонов функций преобразования типов, потому что в них не задается возвращаемый тип⁵. Кроме того, использование `EnableIf` может сделать тип возвращаемого значения очень трудным для чтения и понимания. В таких случаях мы можем вместо этого встроить `EnableIf` в аргумент шаблона по умолчанию следующим образом:

`typeoverload/container1.hpp`

```
#include <iterator>
#include "enableif.hpp"
#include "isconvertible.hpp"

template<typename Iterator>
constexpr bool IsInputIterator =
    IsConvertible <
        typename std::iterator_traits<Iterator>::iterator_category,
        std::input_iterator_tag >;

template<typename T>
class Container
{
public:
    // Конструирование из последовательности входных итераторов:
    template<typename Iterator,
             typename = EnableIf<IsInputIterator<Iterator>>>
    Container(Iterator first, Iterator last);
```

⁵ Вообще говоря, шаблон функции преобразования имеет возвращаемый тип, а именно — тип, в который выполняется преобразование, поэтому параметры шаблона в таком типе должны быть выводимы (см. главу 15, “Вывод аргументов шаблона”) для того, чтобы функция преобразования была корректной.

```
// Преобразование в контейнер, если типы значений преобразуемы:
template<typename U, typename = EnableIf<IsConvertible<T, U>>>
operator Container<U>() const;
};
```

Однако тут есть проблема. Если мы попытаемся добавить еще одну перегрузку (например, более эффективную версию конструктора `Container` для итераторов произвольного доступа), это приведет к ошибке:

```
// Конструирование из последовательности входных итераторов:
template<typename Iterator,
         typename = EnableIf<IsInputIterator<Iterator> &&
                           !IsRandomAccessIterator<Iterator >>>
Container(Iterator first, Iterator last);

template<typename Iterator,
         typename = EnableIf<IsRandomAccessIterator<Iterator>>>
Container(Iterator first, Iterator last); // Ошибка: повторное
                                            // объявление шаблона конструктора
```

Проблема заключается в том, что два шаблона конструкторов идентичны, за исключением аргумента шаблона по умолчанию, но аргументы шаблона по умолчанию не учитываются при определении эквивалентности двух шаблонов.

Мы можем решить эту проблему, добавив еще один параметр шаблона по умолчанию, так что два шаблона конструкторов будут иметь различное количество параметров шаблона:

```
// Конструирование из последовательности входных итераторов:
template<typename Iterator,
         typename = EnableIf<IsInputIterator<Iterator> &&
                           !IsRandomAccessIterator<Iterator >>>
Container(Iterator first, Iterator last);

template<typename Iterator,
         typename = EnableIf<IsRandomAccessIterator<Iterator>>>
Container(Iterator first, Iterator last); // Теперь работает
```

20.3.3. if времени компиляции

Стоит отметить, что конструкция C++17 `if constexpr` (см. раздел 8.5) во многих случаях позволяет избежать необходимости в `EnableIf`. Например, в C++17 наш пример `advanceIter()` можно переписать следующим образом:

`typeoverload/advance3.hpp`

```
template<typename Iterator, typename Distance>
void advanceIter(Iterator& x, Distance n)
{
    if constexpr(IsRandomAccessIterator<Iterator>)
    {
        // Реализация для итераторов произвольного доступа:
        x += n; // Константное время
    }
}
```

```

else if constexpr(IsBidirectionalIterator<Iterator>)
{
    // Реализация для двунаправленных итераторов:
    if (n > 0)
    {
        for(;n>0;++x,--n) {} // Линейное время для положительных n
    }
    else
    {
        for(;n<0;--x,++n) {} // Линейное время для отрицательных n
    }
}
else
{
    // Реализация для прочих (как минимум входных) итераторов
    if (n < 0)
    {
        throw "advanceIter(): неверная категория итератора";
    }

    while(n>0) { ++x; --n; } // Линейное время для положительных n
}
}

```

Этот код гораздо яснее. Более специализированные пути выполнения (например, для итераторов произвольного доступа) будут создаваться только для типов, которые могут их поддерживать. Таким образом, код может безопасно включать операции, присущие не всем итераторам (например, `=`), при условии, что в исходном тексте они надлежащим образом охраняются конструкцией `if constexpr`.

Однако и здесь есть свои недостатки. Такое применение `if constexpr` возможно только тогда, когда разница в обобщенном компоненте может быть полностью выражена в теле шаблона функции. Нам все еще нужен `EnableIf`, если:

- используются различные “интерфейсы”;
- необходимы разные определения классов;
- не должно существовать корректных инстанцирований для определенных списков аргументов шаблонов.

Последнюю ситуацию облазнительно обработать с использованием следующей схемы:

```

template<typename T>
void f(T p)
{
    if constexpr(condition<T>::value)
    {
        // Что-то делаем...
    }
    else
    {
        // Для этого T f() не имеет смысла:
        static_assert(condition<T>::value,

```

```

        "Вызов f() для этого T не имеет смысла");
    }
}

```

Но поступать так не рекомендуется, потому что этот способ плохо согласуется со SFNAE: функция `f<T>()` не удаляется из списка кандидатов, а потому может мешать другим результатам разрешения перегрузки. В качестве альтернативы при использовании `EnableIf` вызов `f<T>()` приведет к удалению при сборке подстановки `EnableIf<...>`.

20.3.4. Концепты

Методы, представленные до настоящего времени, работают хорошо, но они зачастую неуклюжи, могут требовать большого количества ресурсов компилятора и в случае ошибки выдавать громоздкие диагностические сообщения. Поэтому многие авторы обобщенных библиотек были бы рады видеть в языке средство, позволяющее достичь того же результата более эффективно и непосредственно. По этой причине такое языковое средство под названием *концепты*, скорее всего, будет добавлено к языку в ближайшем будущем; см разделы 6.5 и 18.4, а также приложение Д, “Концепты”.

Например, ожидается, что наши перегруженные конструкторы класса `Container` будут выглядеть следующим образом:

`typeoverload/container4.hpp`

```

template<typename T>
class Container
{
public:
    // Конструирование из последовательности входных итераторов:
    template<typename Iterator>
    requires IsInputIterator<Iterator>
    Container(Iterator first, Iterator last);

    // Конструирование из последовательности
    // итераторов произвольного доступа:
    template<typename Iterator>
    requires IsRandomAccessIterator<Iterator>
    Container(Iterator first, Iterator last);

    // Преобразование в контейнер, если типы значений преобразуемы:
    template<typename U>
    requires IsConvertible<T, U>
    operator Container<U>() const;
};

```

Конструкция *требований* (*requires clause*) (рассматривается в разделе Д.1) описывает требования шаблона. Если какое-либо из требований не выполнено, шаблон не рассматривается в качестве кандидата. Поэтому концепты являются более непосредственным выражением идеи, формулируемой с помощью `EnableIf`, при этом поддерживаемым самим языком.

Конструкция требований имеет дополнительные преимущества перед `Enable If`. Категоризация ограничений (описанная в разделе Д.3.1) предоставляет возможность упорядочения шаблонов, которые отличаются только требованиями, устранив гем самым необходимость в диспетчеризации дескрипторов. Кроме того, требование может быть прикреплено к сущности, не являющейся шаблоном. Например, для того, чтобы предоставить функцию-член `sort()` только тогда, когда объекты типа `T` можно сравнивать с помощью оператора `<`:

```
template<typename T>
class Container
{
public:
    ...
    requires HasLess<T>
    void sort()
    {
        ...
    }
};
```

20.4. Специализация класса

Частичные специализации шаблона классов могут использоваться для предоставления альтернативных, специализированных реализаций шаблонов классов для определенных аргументов шаблонов так же, как мы использовали перегрузки для шаблонов функций. Подобно перегруженным шаблонам функций может иметь смысл дифференциация этих частичных специализаций на основе свойств аргументов шаблона. Рассмотрим обобщенный шаблон класса `Dictionary` с типами ключа и значения в качестве параметров шаблона. Простой (но неэффективный) словарь может быть реализован при условии, что тип ключа обеспечивает один лишь оператор сравнения на равенство:

```
template<typename Key, typename Value>
class Dictionary
{
private:
    vector<pair<Key const, Value>> data;
public:
    // Индексированный доступ к данным:
    value& operator[](Key const& key)
    {
        // Поиск элемента с данным ключом:
        for (auto& element : data)
        {
            if (element.first == key)
            {
                return element.second;
            }
        }
    }

    // Элемента с указанным ключом нет – добавляем:
    data.push_back(pair<Key const, Value>(key, Value()));
}
```

```

        return data.back().second;
    }
    ...
};

}

```

Если тип ключа поддерживает оператор `<`, можно обеспечить более эффективную реализацию на основе контейнера `map` стандартной библиотеки. Аналогично, если тип ключа поддерживает операции хеширования, мы можем обеспечить еще более эффективную реализацию на основе `unordered_map` из стандартной библиотеки.

20.4.1. Включение/отключение шаблонов классов

Способ включения/отключения различных реализаций шаблонов классов заключается в использовании включения/отключения частичных специализаций шаблонов классов. Для использования `EnableIf` с частичной специализацией шаблона класса мы сначала введем в `Dictionary` неименованный параметр шаблона по умолчанию:

```

template<typename Key, typename Value, typename = void>
class Dictionary
{
    ... // Реализация с vector, как показано выше
};

```

Этот новый параметр шаблона служит якорем для `EnableIf`, который теперь может быть встроен в список аргументов шаблона частичной специализации для версии словаря с использованием контейнера `map`:

```

template<typename Key, typename Value>
class Dictionary<Key, Value,
                 EnableIf<HasLess<Key>>>
{
private:
    map<Key, Value> data;
public:
    value& operator[](Key const& key)
    {
        return data[key];
    }
    ...
};

```

В отличие от перегруженных шаблонов функций, нам не нужно отключать никакие условия в первичном шаблоне, потому что любая частичная специализация имеет приоритет над первичным шаблоном. Однако, добавив еще одну реализацию для ключей с поддержкой операций хеширования, мы должны будем обеспечить взаимоисключаемость условий частичных специализаций:

```

template<typename Key, typename Value, typename = void>
class Dictionary
{
    ... // Реализация с vector, как показано выше
};

```

```

template<typename Key, typename Value>
class Dictionary<Key, Value,
    EnableIf < HasLess<Key>&& !HasHash<Key >>>
{
{
    ... // Реализация с map, как показано выше
};

template<typename Key, typename Value>
class Dictionary<Key, Value,
    EnableIf<HasHash<Key>>>
{
private:
    unordered_map<Key, Value> data;
public:
    value& operator[](Key const& key)
    {
        return data[key];
    }
    ...
};

```

20.4.2. Диспетчеризация дескрипторов для шаблонов классов

Диспетчеризация дескрипторов может использоваться также и для выбора среди частичных специализаций шаблона класса. Чтобы это проиллюстрировать, определим тип функционального объекта `Advance<Iterator>`, аналога алгоритма `advanceIter()`, который перемещает итератор на некоторое количество шагов (этот алгоритм уже использовался в предыдущих разделах). Мы предоставляем как общую реализацию (для входных итераторов), так и специализированные реализации для двунаправленных итераторов и итераторов произвольного доступа, полагаясь на вспомогательное свойство `BestMatchInSet` (описано ниже) для выбора наилучшего соответствия дескриптору категории итератора:

```

// Первичный шаблон (преднамеренно не определен):
template<
    typename Iterator,
    typename Tag =
    BestMatchInSet<
        typename std::iterator_traits<Iterator>::iterator_category,
        std::input_iterator_tag,
        std::bidirectional_iterator_tag,
        std::random_access_iterator_tag>>
class Advance;

// Общая линейная реализация для входных итераторов:
template<typename Iterator>
class Advance<Iterator, std::input_iterator_tag>
{
public:
    using DifferenceType =
        typename std::iterator_traits<Iterator>::difference_type;

```

```

void operator()(Iterator& x, DifferenceType n) const
{
    while (n > 0)
    {
        ++x;
        --n;
    }
}
};

// Двунаправленный линейный алгоритм для двунаправленных итераторов:
template<typename Iterator>
class Advance<Iterator, std::bidirectional_iterator_tag>
{
public:
    using DifferenceType =
        typename std::iterator_traits<Iterator>::difference_type;
    void operator()(Iterator& x, DifferenceType n) const
    {
        if (n > 0)
        {
            while (n > 0)
            {
                ++x;
                --n;
            }
        }
        else
        {
            while (n < 0)
            {
                --x;
                ++n;
            }
        }
    }
};

// Двунаправленный с константным временем алгоритм
// для итераторов произвольного доступа:
template<typename Iterator>
class Advance<Iterator, std::random_access_iterator_tag>
{
public:
    using DifferenceType =
        typename std::iterator_traits<Iterator>::difference_type;
    void operator()(Iterator& x, DifferenceType n) const
    {
        x += n;
    }
};

```

Эта формулировка аналогична таковой для диспетчеризации дескрипторов для шаблонов функций. Однако задача заключается в написании свойства `BestMatchInSet`, предназначенного для выбора наиболее подходящего дескриптора (среди дескрипторов входного, двунаправленного и итератора произвольного доступа) для данного итератора. В сущности, предназначение этого

свойства — выяснить, какая из следующих перегрузок была бы выбрана для данного значения дескриптора категории итератора, и сообщить его тип параметра:

```
void f(std::input_iterator_tag);
void f(std::bidirectional_iterator_tag);
void f(std::random_access_iterator_tag);
```

Самый простой способ эмулировать разрешение перегрузки — использовать ее на самом деле, как показано ниже:

```
// Строим набор перегрузок match() для типов в Types...:
template<typename... Types>
struct MatchOverloads;

// Базовый случай: соответствий нет:
template<>
struct MatchOverloads<>
{
    static void match(...);
};

// Рекурсивный случай: вводим новую перегрузку match():
template<typename T1, typename... Rest>
struct MatchOverloads<T1, Rest...> : public MatchOverloads<Rest...>
{
    static T1 match(T1); // Перегрузка для T1
    using MatchOverloads<Rest...>::match; // Сбор перегрузок
};

// Поиск лучшего соответствия для T из Types...:
template<typename T, typename... Types>
struct BestMatchInSetT
{
    using Type = decltype(MatchOverloads<Types...>::match(declval<T>()));
};

template<typename T, typename... Types>
using BestMatchInSet = typename BestMatchInSetT<T, Types...>::Type;
```

Шаблон MatchOverloads использует рекурсивное наследование для объявления функции match() с каждым типом из входного множества Types. Каждое инстанцирование рекурсивной частичной специализации MatchOverloads вводит новые функции match() для следующего типа в списке. Затем он использует объявление using для сбора функций match(), определенных в базовом классе, который обрабатывает оставшиеся в списке типы. При рекурсивном применении результат представляет собой полный набор перегрузок match(), соответствующий заданным типам, каждый из которых возвращает свой тип параметра. Затем шаблон BestMatchInSetT передает объект T этому множеству перегруженных функций match() и производит возвращаемый тип выбранной (наилучшей) функции match()⁶. Если ни одна из функций не является соответствующей,

⁶ В C++17 можно устраниТЬ рекурсию с помощью раскрытия пакета в списке базового класса и в объявлении using (см. раздел 4.4.5). Эта методика будет продемонстрирована в разделе 26.4.

возвращающий `void` базовый случай (который использует многоточие для соответствия любому аргументу) указывает на сбой⁷. Подытоживая, мы видим, что `BestMatchInSetT` транслирует результат перегрузки функции в свойство и обеспечивает относительно легкое использование диспетчеризации дескрипторов для выбора среди частичных специализаций шаблона класса.

20.5. Шаблоны, безопасные по отношению к инстанцированию

Суть метода `EnableIf` состоит во включении конкретного шаблона или частичной специализации только в случае соответствия аргументов шаблона некоторым определенным критериям. Например, наиболее эффективная разновидность алгоритма `advanceIter()` проверяет, что категория итератора-аргумента преобразуема в `std::random_access_iterator_tag`. Отсюда следует, что алгоритму доступны различные операции итераторов произвольного доступа.

Но что, если довести это понятие до предела и закодировать каждую операцию, выполняемую шаблоном над его аргументами, как часть условия `EnableIf`? Инстанцирование такого шаблона никогда не будет неудачным, потому что аргументы шаблона, не предоставляющие необходимые операции, приведут к сбою вывода типов (через `EnableIf`), не позволяя перейти к инстанцированию. Мы будем говорить о таких шаблонах как о “безопасных с точки зрения инстанцирования” и представим здесь набросок реализации таких шаблонов.

Начнем с очень простого шаблона `min()`, который вычисляет минимальное из двух значений. Обычно в виде шаблона этот алгоритм реализуется следующим образом:

```
template<typename T>
T const& min(T const& x, T const& y)
{
    if (y < x)
    {
        return y;
    }

    return x;
}
```

Этот шаблон требует, чтобы тип `T` имел оператор `<`, способный сравнивать два значения типа `T` (в частности, два константных l-значения `T`), а затем неявно преобразующий результат этого сравнения в `bool` для использования в инструкции `if`. Свойство, которое проверяет наличие оператора `<` и вычисляет тип его

⁷ Было бы немного лучше не предоставлять никакого результата в случае сбоя, тем самым добиваясь свойства, дружественного по отношению к SFINAЕ (см. раздел 19.4.4). Кроме того, надежная реализация использовала бы обертку возвращаемого типа во что-то вроде `Identity`, потому что существуют некоторые типы (такие как типы массивов и функций), которые могут быть типами параметров, но не возвращаемыми типами. Мы опускаем эти улучшения ради краткости и удобочитаемости.

результата, является аналогом дружественного по отношению к SFINAE свойства PlusResultT, рассматривавшегося в разделе 19.4.4; однако для удобства мы приведем код LessResultT здесь:

typeoverload/lessresult.hpp

```
#include <utility>      // Для declval()
#include <type_traits> // Для true_type и false_type

template<typename T1, typename T2>
class HasLess
{
    template<typename T> struct Identity;
    template<typename U1, typename U2> static std::true_type
    test(Identity<decltype(std::declval<U1>())<std::declval<U2>()>*);
    template<typename U1, typename U2> static std::false_type
    test(...);

public:
    static constexpr bool value =
        decltype(test<T1, T2>(nullptr))::value;
};

template<typename T1, typename T2, bool HasLess>
class LessResultImpl
{
public:
    using Type = decltype(std::declval<T1>() < std::declval<T2>());
};

template<typename T1, typename T2>
class LessResultImpl<T1, T2, false>
{};

template<typename T1, typename T2>
class LessResultT
    : public LessResultImpl<T1, T2, HasLess<T1, T2>::value>
{};

template<typename T1, typename T2>
using LessResult = typename LessResultT<T1, T2>::Type;
```

Затем это свойство может быть объединено со свойством IsConvertible, чтобы сделать `min()` безопасным с точки зрения инстанцирования:

typeoverload/min2.hpp

```
#include "isconvertible.hpp"
#include "lessresult.hpp"

template<typename T>
EnableIf<IsConvertible<LessResult<T const&, T const&>, bool>, T const&>
min(T const& x, T const& y)
```

```
{
    if (y < x)
    {
        return y;
    }

    return x;
}
```

Поучительно попытаться вызвать эту функцию `min()` с различными типами с разными операторами `<` (или с полностью отсутствующим данным оператором), как показано в следующем примере:

typeoverload/min.cpp

```
#include "min.hpp"

struct X1 { };
bool operator< (X1 const&, X1 const&)
{
    return true;
}

struct X2 { };
bool operator<(X2, X2)
{
    return true;
}

struct X3 { };
bool operator<(X3&, X3&)
{
    return true;
}

struct X4 { };
struct BoolConvertible
{
    operator bool() const
    {
        return true;           // неявное преобразование в bool
    }
};

struct X5 { };
BoolConvertible operator< (X5 const&, X5 const&)
{
    return BoolConvertible();
}

struct NotBoolConvertible // Нет преобразования в bool
{
```

```

struct X6 { };
NotBoolConvertible operator< (X6 const&, X6 const&)
{
    return NotBoolConvertible();
}

struct BoolLike
{
    explicit operator bool() const
    {
        return true;           // Явное преобразование в bool
    }
};

struct X7 { };
BoolLike operator< (X7 const&, X7 const&)
{
    return BoolLike();
}

int main()
{
    min(X1(), X1()); // X1 может быть передан в min()
    min(X2(), X2()); // X2 может быть передан в min()
    min(X3(), X3()); // Ошибка: X3 не может быть передан в min()
    min(X4(), X4()); // Ошибка: X4 не может быть передан min()
    min(X5(), X5()); // X5 может быть передан в min()
    min(X6(), X6()); // Ошибка: X6 не может быть передан min()
    min(X7(), X7()); // Неожиданная ошибка:
                      // X7 не может быть передан min()
}

```

При компиляции этой программы обратите внимание на то, что хотя для четырех различных вызовов `min()` (для `X3`, `X4`, `X6` и `X7`) есть ошибки, они генерируются не телом `min()`, как было бы в варианте, небезопасном с точки зрения инстанцирования. Вместо этого мы получаем сообщения о том, что не существует подходящей функции `min()`, потому что единственный вариант был исключен в соответствии с принципом SFNAE. Вот как выглядит диагностика Clang:

```

min.cpp:41:3: error: no matching function for call to 'min'
    min(X3(), X3()); // ERROR: X3 cannot be passed to min
    ^
./min.hpp:8:1: note: candidate template ignored: substitution failure
[with T = X3]: no type named 'Type' in
    'LessResultT<const X3 &, const X3 &>'
min(T const& x, T const& y)

```

Таким образом, `EnableIf` допускает инстанцирование только для тех аргументов шаблона, которые отвечают требованиям шаблона (`X1`, `X2` и `X5`), поэтому мы никогда не получим сообщения об ошибке из тела `min()`. Кроме того, если бы у нас была некоторая другая перегрузка `min()`, которая могла бы работать с этими типами, то процесс разрешения перегрузки мог бы выбрать ее вместо вывода сообщения об ошибках.

Последний тип в нашем примере, X7, иллюстрирует некоторые тонкости реализации безопасных с точки зрения инстанцирования шаблонов. В частности, если X7 передается в небезопасный с точки зрения инстанцирования шаблон `min()`, то инстанцирование будет успешным. Безопасный с точки зрения инстанцирования шаблон `min()` отвергает его, потому что `BoolLike` не является неявно преобразуемым в `bool`. Различие здесь особенно тонкое: `explicit`-преобразование в `bool` может в некоторых контекстах использоваться *неявно*, в том числе в логических условиях для управления потоком выполнения (`if`, `while`, `for` и `do`), во встроенных операторах `!`, `&&` и `||` и в тернарном операторе `??:`. В этих контекстах значение называется *контекстно преобразуемым* в `bool`⁸.

Однако наше упорство в требовании наличия общего, неявного преобразования в `bool` приводит к тому, что наш безопасный с точки зрения инстанцирования шаблон перегружен ограничениями, т.е. его требования (в `EnableIf`) оказываются сильнее, чем его фактические потребности (что необходимо для надлежащего инстанцирования шаблона). Если, с другой стороны, полностью забыть о требовании преобразования в `bool`, наш шаблон `min()` будет недоопределен, и это допускало бы некоторые аргументы шаблона, которые могут вызвать сбой при инстанцировании (например, X6).

Чтобы исправить безопасный с точки зрения инстанцирования шаблон `min()`, нам требуется свойство, способное определить, является ли тип T контекстно преобразуемым в `bool`. В определении этого свойства нам не помогут ни инструкции управления потоком выполнения (потому что инструкции не могут выполняться в контексте SFINAE), ни логические операции, которые могут быть перегружены для произвольного типа. К счастью, тернарный оператор `?:` является выражением, и он не перегружаем, поэтому он позволяет проверить, является ли данный тип контекстно преобразуемым в `bool`:

`typeoverload/iscontextualbool.hpp`

```
#include <utility>      // Для declval()
#include <type_traits> // Для true_type и false_type

template<typename T>
class IsContextualBoolT
{
private:
    template<typename T> struct Identity;
    template<typename U> static std::true_type
    test(Identity < decltype(declval<U>() ? 0 : 1) > *);
}
```

⁸ C++11 наряду с операторами явного преобразования вводит понятие контекстного преобразования в `bool`. Вместе они заменяют использование идиомы “безопасный `bool`” ([46]), которая обычно включает (неявные) пользовательские преобразования в указатель на член данных. Указатель на член данных используется потому, что его можно рассматривать как логическое значение, но при этом он не имеет других нежелательных преобразований, таких как `bool`, который может быть повышен до `int` в рамках арифметических операций. Например, `BoolConvertible() + 5` – это (к сожалению) вполне корректный код.

```

template<typename U> static std::false_type
test(...);
public:
    static constexpr bool value = decltype(test<T>(nullptr))::value;
};

template<typename T>
constexpr bool IsContextualBool = IsContextualBoolT<T>::value;

```

При наличии этого нового свойства мы можем предоставить безопасный с точки зрения инстанцирования шаблон `min()` с правильным набором требований в `EnableIf`:

`typeoverload/min3.hpp`

```

#include "iscontextualbool.hpp"
#include "lessresult.hpp"

template<typename T>
EnableIf<IsContextualBool<LessResult<T const&, T const&>>,
         T const&>
min(T const& x, T const& y)
{
    if (y < x)
    {
        return y;
    }

    return x;
}

```

Методы, используемые здесь для того, чтобы сделать шаблон `min()` безопасным с точки зрения инстанцирования, можно расширить для описания требований для нетривиальных шаблонов. Такое расширение можно обеспечить путем объединения проверок различных требований в свойства, которые описывают некоторые классы типов, как, например, односторонние итераторы, и комбинирования этих свойств в пределах `EnableIf`. Поступая так, мы получаем как преимущества лучшего поведения перегрузки, так и устранение “романов об ошибках”, которые так любят выводить компиляторы при ошибках в инстанцировании глубоко вложенного шаблона. С другой стороны, в этом случае сообщения об ошибках имеют тенденцию к неконкретности относительно конкретной неудачной операции. Кроме того, как мы показали на примере небольшого и несложного шаблона `min()`, аккуратное определение и кодирование точных требований шаблона может оказаться сложной задачей. Методы отладки, позволяющие использовать эти свойства, мы рассмотрим в разделе 28.2.

20.6. В стандартной библиотеке

Стандартная библиотека C++ предоставляет дескрипторы итераторов для различных итераторов — дескрипторы входного, выходного, одностороннего,

дву направленного и с произвольным доступом итераторов, которые мы использовали в нашей презентации. Эти дескрипторы итераторов являются частью стандартных свойств итераторов (`std::iterator_traits`) и требований, предъявляемых к итераторам, чтобы они могли безопасно использоваться для целей диспетчеризации дескрипторов.

Шаблон класса стандартной библиотеки C++11 `std::enable_if` демонстрирует то же поведение, что и шаблон класса `EnableIfT`, представленный в этой главе. Единственное отличие заключается в том, что стандарт использует имя члена-типа `type` в нижнем регистре вместо нашего `Type`.

Специализация алгоритмов используется в ряде мест стандартной библиотеки C++. Например, `std::advance()` и `std::distance()` имеют несколько вариантов, основанных на категории их аргумента-итератора. Большинство реализаций стандартной библиотеки, как правило, используют диспетчеризацию дескрипторов, хотя совсем недавно некоторые из них стали использовать для реализации специализации алгоритмов `std::enable_if`. Кроме того, ряд реализаций стандартной библиотеки C++ также используют эти методы внутренне, для реализации специализации различных стандартных алгоритмов. Например, `std::copy()` можно специализировать для вызова `std::memcpuy()` или `std::memmove()`, когда итераторы указывают на непрерывную память, а их типы значений имеют тривиальные операторы присваивания. Аналогично алгоритм `std::fill()` может быть оптимизирован для вызова `std::memset()`, а различные алгоритмы могут избежать вызова деструкторов, когда известно, что тип имеет тривиальный деструктор. Эти специализации алгоритмов не санкционированы стандартом так же, как, например, в случае `std::advance()` или `std::distance()`, но разработчики решили использовать их из соображений эффективности.

Как показано в разделе 8.4, стандартная библиотека C++ также намекает на использование `std::enable_if<>` или аналогичных методов, основанных на SFINAE, в своих требованиях. Так, `std::vector` имеет шаблон конструктора, который создает вектор из последовательности итераторов:

```
template<typename InputIterator>
vector(InputIterator first, InputIterator second,
       allocator_type const& alloc = allocator_type());
```

При этом выдвигается требование, что “если конструктор вызывается с типом `InputIterator`, который не квалифицирован как входной, то такой конструктор не участвует в разрешении перегрузки” (см. пункт x23.2.3, абзац 14 [27]). Эта формулировка слишком расплывчата, чтобы допустить использование наиболее эффективных современных методов для реализации, но в то время, когда это требование было добавлено к стандарту, предполагалось использование `std::enable_if<>`.

20.7. Заключительные замечания

Диспетчеризация дескрипторов давно известна в C++. Она использовалась в оригинальной реализации STL (см. [63]) и часто применяется вместе со

свойствами. Использование SFINAE и `EnableIf` намного новее: термин SFINAE появился в первом издании этой книги (см. [71]), где и было продемонстрировано его применение, в частности, для обнаружения наличия членов-типов.

Технология и терминология `EnableIf` была впервые опубликована Яакко Ярви (Jaakko Järvi), Иеремией Уиллкоком (Jeremiah Willcock), Говардом Хиннантом (Howard Hinnant) и Эндрю Ламсдейном (Andrew Lumsdaine) в [42], где описан шаблон `EnableIf`, как реализовать перегрузку функций с использованием пар `EnableIf` и `DisableIf` и использовать `EnableIf` с частичной специализацией шаблона класса. С тех пор `EnableIf` и аналогичные методы стали повсеместно применяться в реализации сложных библиотек шаблонов, включая стандартную библиотеку C++. Кроме того, популярность этих методов мотивирована расширенным поведением SFINAE в C++11 (см. раздел 15.7). Питер Димов (Peter Dimov) был первым, кто заметил, что аргументы шаблона по умолчанию для шаблонов функций (еще одна возможность C++11) сделали возможным использование `EnableIf` в шаблонах конструкторов без добавления другого параметра функции.

Как ожидается, *концепты* (описаны в приложении Д, “Концепты”) войдут в следующий стандарт C++, после C++17. Ожидается, что они сделают много методов разработки, включая применение `EnableIf`, в значительной степени устаревшими. Пока же наличие в стандарте C++17 конструкции `if constexpr` (см. разделы 8.5 и 20.3.3) также постепенно сокращает их присутствие в современных шаблонных библиотеках.

Глава 21

Шаблоны и наследование

Априори нет причины полагать, что шаблоны и наследование взаимодействуют каким-то особым образом. Следует отметить лишь тот факт (см. главу 13, “Имена в шаблонах”), что порождение от зависимых базовых классов требует особой тщательности при использовании неквалифицированных имен. Однако оказывается, что некоторые интересные технологии программирования объединяют эти две возможности языка, включая странно рекурсивный шаблон проектирования (Curiously Recurring Template Pattern – CRTP) и максины. В этой главе мы бегло рассмотрим эти две технологии.

21.1. Оптимизация пустого базового класса

Классы C++ часто бывают “пустыми”, т.е. их внутреннее представление не требует выделения памяти во время работы программы. Это типичное поведение классов, которые содержат только члены-типы, невиртуальные функции-члены и статические члены-данные. Нестатические члены-данные, виртуальные функции и виртуальные базовые классы требуют при работе программы выделения памяти.

Однако даже пустые классы имеют ненулевой размер. Если вы хотите это проверить, попробуйте запустить приведенную ниже программу.

inherit/empty.cpp

```
#include <iostream>

class EmptyClass
{
};

int main()
{
    std::cout << "sizeof(EmptyClass): " << sizeof(EmptyClass) << '\n';
}
```

На множестве платформ эта программа выведет 1 в качестве размера класса `EmptyClass`. Некоторые системы продемонстрируют более строгие требования к выравниванию и выведут иное небольшое значение (обычно 4).

21.1.1. Принципы размещения

Проектировщики C++ имели множество причин избегать классов с нулевым размером. Например, массив классов, имеющих нулевые размеры, также имел бы нулевой размер, но при этом арифметика указателей оказалась бы неприменима. Пусть, например, `ZeroSizedT` — тип с нулевым размером:

```
ZeroSizedT z[10];
...
&z[i] - &z[j] // Вычисление расстояния между указателями/адресами
```

Обычно разность из предыдущего примера получается путем деления числа байтов между двумя адресами на размер объекта указываемого типа. Однако, если этот размер нулевой, понятно, что такая операция не приведет к корректному результату.

Тем не менее, даже при том, что в C++ нет типов с нулевым размером, стандарт C++ устанавливает, что, когда пустой класс используется в качестве базового, память для него не выделяется *при условии, что это не приводит к размещению объекта по адресу, где уже расположен другой объект или подобъект того же самого типа*. Рассмотрим несколько примеров, чтобы разъяснить, что означает на практике так называемая *оптимизация пустого базового класса* (empty base class optimization – EBCO). Рассмотрим приведенную ниже программу:

inherit/ebc01.cpp

```
#include <iostream>

class Empty
{
    using Int = int; // Псевдоним типа не делает класс непустым
};

class EmptyToo : public Empty
{
};

class EmptyThree : public EmptyToo
{
};

int main()
{
    std::cout << "sizeof(Empty) : " << sizeof(Empty) << '\n';
    std::cout << "sizeof(EmptyToo) : " << sizeof(EmptyToo) << '\n';
    std::cout << "sizeof(EmptyThree) : " << sizeof(EmptyThree) << '\n';
}
```

Если ваш компилятор реализует оптимизацию пустого базового класса, то он выведет один и тот же размер для каждого класса (но ни один из этих классов не будет иметь нулевой размер (рис. 21.1)). Это означает, что внутри класса EmptyToo классу Empty не выделяется никакое пространство. Обратите внимание и на то, что пустой класс с оптимизированными пустыми базовыми классами (при отсутствии непустых базовых классов) также пуст. Это объясняет, почему класс EmptyThree может иметь тот же размер, что и класс Empty. Если же ваш компилятор не выполняет оптимизацию пустого базового класса, выведенные размеры будут разными (рис. 21.2).

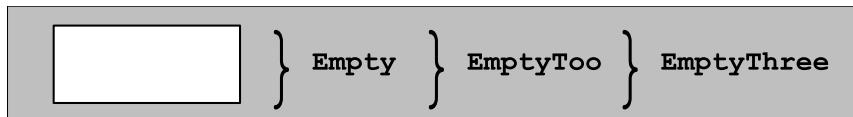


Рис. 21.1. Размещение EmptyThree компилятором, который реализует EBCO

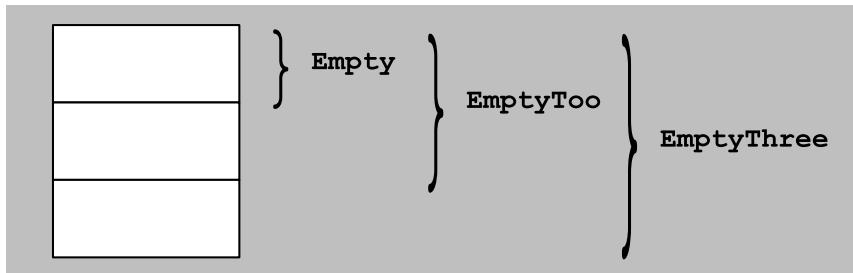


Рис. 21.2. Размещение EmptyThree компилятором, который не реализует EBCO

Рассмотрим пример, в котором оптимизация пустого базового класса запрещена:

inherit/ebc02.cpp

```
#include <iostream>

class Empty
{
    using Int = int; // Псевдоним типа не делает класс непустым
};

class EmptyToo : public Empty
{
};

class NonEmpty : public Empty, public EmptyToo
{
};

int main()
{
    std::cout << "sizeof(Empty): " << sizeof(Empty) << '\n';
    std::cout << "sizeof(EmptyToo): " << sizeof(EmptyToo) << '\n';
    std::cout << "sizeof(NonEmpty): " << sizeof(NonEmpty) << '\n';
}
```

Может показаться неожиданным, что класс NonEmpty не пустой. Ведь ни он, ни его базовые классы не содержат никаких членов. Но дело в том, что базовые классы Empty и EmptyToo класса NonEmpty не могут быть размещены по одному и тому же адресу, поскольку это привело бы к размещению объекта базового класса Empty, принадлежащего классу EmptyToo, по тому же адресу, что и объекта базового класса Empty, принадлежащего классу NonEmpty. Иными словами, два подобъекта одного и того же типа находились бы в одном месте, а это

не разрешено правилами размещения объектов языка C++. Можно решить, что один из базовых подобъектов `Empty` помещен со смещением 0 байт, а другой — со смещением 1 байт, но полный объект `NonEmpty` все равно не может иметь размер в один байт, так как в массиве из двух объектов `NonEmpty` подобъект `Empty` первого элемента не может находиться по тому же адресу, что и подобъект `Empty` второго элемента (рис. 21.3):

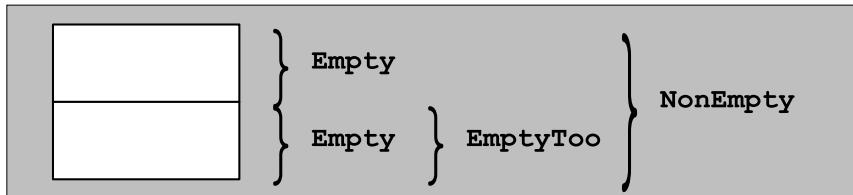


Рис. 21.3. Размещение объекта `NonEmpty` компилятором, реализующим ЕВСО

Ограничение на оптимизацию пустого базового класса можно объяснить необходимостью проверки, не указывают ли два указателя на один и тот же объект. Поскольку указатели почти всегда внутренне представлены как обычные адреса, необходимо гарантировать, что два различных адреса соответствуют двум различным объектам.

Это ограничение может показаться не очень существенным, однако с ним часто приходится сталкиваться на практике, поскольку многие классы наследуются из небольшого набора пустых классов, определяющих некоторое общее множество синонимов имен типов. Когда два подобъекта таких классов оказываются в одном и том же полном объекте, оптимизация запрещена.

Даже при этом ограничении ЕВСО является важной оптимизацией для библиотек шаблонов, потому что ряд методов основываются на введении базовых классов просто с целью введения нового псевдонима типа или предоставления дополнительной функциональности без добавления новых данных. В этой главе будет описано несколько таких методов.

21.1.2. Члены как базовые классы

Оптимизация пустого базового класса не имеет эквивалента для членов-данных, поскольку (помимо прочего) это создало бы ряд проблем с представлением указателей на члены. Поэтому иногда то, что реализовано как данные-члены, желательно реализовать в виде (закрытого) базового класса, который на первый взгляд выглядит как переменная-член. Однако и здесь не обходится без проблем.

Наиболее интересна эта задача в контексте шаблонов, поскольку параметры шаблона часто заменяются типами пустых классов (хотя, конечно, в общем случае полагаться на это нельзя). Если о типовом параметре шаблона ничего не известно, оптимизацию пустого базового класса осуществить не так-то легко. Рассмотрим тривиальный пример:

```
template<typename T1, typename T2>
class MyClass
{
    private:
        T1 a;
        T2 b;
        ...
};
```

Вполне возможно, что один или оба параметра шаблона заменяются типом пустого класса. В этом случае представление `MyClass<T1, T2>` может оказаться не оптимальным, что приведет к напрасной трате одного слова памяти для каждого экземпляра `MyClass<T1, T2>`.

Этого можно избежать, сделав аргументы шаблона базовыми классами:

```
template<typename T1, typename T2>
class MyClass : private T1, private T2
{
};
```

Однако этот простой вариант имеет свои недостатки.

- Он не работает, если `T1` или `T2` заменяются типом, не являющимся классом или типом объединения.
- Он также не работает, если два параметра заменяются одним и тем же типом (хотя можно легко решить эту проблему, добавив лишний уровень наследования, как было показано ранее в главе).
- Класс может быть объявлен как `final`, и в этом случае попытки его наследования будут приводить к ошибкам.

Но даже после решения этих проблем адресации останется еще одна очень серьезная неприятность: добавление базового класса может существенно изменить интерфейс данного класса. Для нашего класса `MyClass` эта проблема может показаться не очень значительной, поскольку здесь совсем немного элементов интерфейса, на которые может воздействовать добавление базового класса, но, как будет показано далее в главе, наличие виртуальных функций-членов меняет картину. Понятно, что рассматриваемый подход к EBCO чреват всеми описанными видами проблем.

Более практическое решение может быть разработано для распространенного случая, когда известно, что параметр шаблона заменяется только типами классов и когда доступен другой член шаблона класса. Основная идея состоит в том, чтобы “слиять” потенциально пустой параметр типа с другим членом с использованием EBCO. Например, вместо записи

```
template<typename CustomClass>
class Optimizable
{
    private:
        CustomClass info; // Может быть пустым
        void* storage;
        ...
};
```

МОЖНО написать:

```
template<typename CustomClass>
class Optimizable
{
    private:
        BaseMemberPair<CustomClass, void*> info_and_storage;
    ...
};
```

Даже беглого взгляда достаточно, чтобы понять, что использование шаблона `BaseMemberPair` делает реализацию `Optimizable` более многословной. Однако некоторые разработчики библиотек шаблонов отмечают, что повышение производительности (для клиентов их библиотек) стоит этой дополнительной сложности. Мы рассмотрим эту идиому позже при изучении кортежей в разделе 25.1.1.

Реализация `BaseMemberPair` может быть достаточно компактной:

inherit/basememberpair.hpp

```
#ifndef BASE_MEMBER_PAIR_HPP
#define BASE_MEMBER_PAIR_HPP
template<typename Base, typename Member>
class BaseMemberPair : private Base
{
    private:
        Member mem;
    public:
        // Конструктор
        BaseMemberPair(Base const& b, Member const& m)
            : Base(b), mem(m)
        {
        }
        // Доступ к данным базового класса через base()
        Base const& base() const
        {
            return static_cast<Base const&>(*this);
        }
        Base& base()
        {
            return static_cast<Base&>(*this);
        }
        // Доступ к членам-данным через member()
        Member const& member() const
        {
            return this->mem;
        }
        Member& member()
        {
            return this->mem;
        }
    };
#endif // BASE_MEMBER_PAIR_HPP
```

Для доступа к инкапсулированным (и, возможно, оптимизированным с точки зрения расхода памяти) элементам данных реализация должна использовать функции-члены `base()` и `member()`.

21.2. Странно рекурсивный шаблон проектирования

Еще одна идиома, которую стоит рассмотреть — *Странно рекурсивный шаблон проектирования* (Curiously Recurring Template Pattern — CRTP). Это странное название обозначает общий класс методов, которые состоят в передаче класса-наследника в качестве аргумента шаблона одному из собственных базовых классов. В самой простой форме код C++ такой модели выглядит, как показано ниже.

```
template<typename Derived>
class CuriousBase
{
    ...
};

class Curious : public CuriousBase<Curious>
{
    ...
};
```

Наш первый набросок CRTP имеет независимый базовый класс: `Curious` не является шаблоном и, следовательно, защищен от проблем видимости имен зависимых базовых классов. Однако это не главная характеристика CRTP. Точно так же можно было использовать альтернативную схему:

```
template<typename Derived>
class CuriousBase
{
    ...
};

template<typename T>
class CuriousTemplate : public CuriousBase<CuriousTemplate<T>>
{
    ...
};
```

Передавая производный класс базовому в качестве параметра, базовый класс может настроить собственное поведение для данного производного класса так, что не возникнет необходимости в использовании виртуальных функций. Это делает CRTP полезным методом для исключения реализаций, которые могут быть только функциями-членами (например, конструкторы, деструкторы или операторы индексов) или зависят от производного класса.

Простейшее применение CRTP — отслеживание количества созданных объектов некоторого типа класса. Этого легко достичь посредством увеличения целого статического члена-данного в каждом конструкторе и его уменьшения в деструкторе. Однако необходимость обеспечить соответствующий код в каждом классе весьма утомительна, а реализация этой функциональности посредством одного (не-CRTP) базового класса будет спутываться со счетчиками объектов

для различных производных классов. Но вместо этого можно просто написать следующий шаблон:

inherit/objectcounter.hpp

```
#include <cstddef>

template<typename CountedType>
class ObjectCounter
{
private:
    inline static           // Количество
    std::size_t count = 0;   // существующих объектов
protected:
    // Конструктор по умолчанию
    ObjectCounter()
    {
        ++count;
    }
    // Копирующий конструктор
    ObjectCounter(ObjectCounter<CountedType> const&)
    {
        ++count;
    }
    // Перемещающий конструктор
    ObjectCounter(ObjectCounter<CountedType>&&)
    {
        ++count;
    }
    // Деструктор
    ~ObjectCounter()
    {
        --count;
    }
public:
    // Возврат количества имеющихся объектов:
    static std::size_t live()
    {
        return count;
    }
};
```

Обратите внимание на использование `inline`, чтобы иметь возможность определить и инициализировать член `count` внутри структуры класса. До C++17 мы должны были определять его вне шаблона класса:

```
template<typename CountedType>
class ObjectCounter
{
private:
    static std::size_t count; // Количество существующих объектов
    ...
};

// Инициализация счетчика нулем:
template<typename CountedType>
std::size_t ObjectCounter<CountedType>::count = 0;
```

Если требуется подсчитать количество активных (не уничтоженных) объектов некоторого типа класса, достаточно породить класс из шаблона `ObjectCounter`. Например, можно определить и использовать класс строк с подсчетом объектов.

inherit/countertest.cpp

```
#include "objectcounter.hpp"
#include <iostream>

template<typename CharT>
class MyString : public ObjectCounter<MyString<CharT>>
{
    ...
};

int main()
{
    MyString<char> s1, s2;
    MyString<wchar_t> ws;
    std::cout << "Количество MyString<char>: "
        << MyString<char>::live() << '\n';
    std::cout << "Количество MyString<wchar_t>: "
        << ws.live() << '\n';
}
```

21.2.1. Метод Бартона–Нэкмана

В 1994 году Джон Бартон (John J. Barton) и Ли Нэкман (Lee R. Nackman) представили метод применения шаблонов, названный ими *ограниченным расширением шаблонов* (restricted template expansion [7]). Частично причиной разработки этого метода послужил тот факт, что в то время шаблоны функций нельзя было перегружать¹, а пространства имен в большинстве компиляторов были недоступны.

Чтобы проиллюстрировать указанный метод, предположим, что у нас есть шаблон класса `Array`, в котором требуется определить оператор равенства `==`. Одна из возможностей — объявить этот оператор членом класса. Однако недостаток этого подхода состоит в том, что первый аргумент (связанный с указателем `this`) подчиняется правилам преобразования типов, отличным от тех, которые применимы ко второму аргументу. Поскольку удобнее, чтобы оператор `==` был симметричным относительно своих аргументов, лучше объявить его как функцию в области видимости пространства имен. Общая схема такого подхода к реализации оператора `==` может выглядеть следующим образом:

```
template<typename T>
class Array
{
public:
    ...
};
```

¹ Возможно, вам стоит прочитать раздел 16.2, чтобы понять, как в современном C++ работает перегрузка шаблонов функций.

```
template<typename T>
bool operator==(Array<T> const& a, Array<T> const& b)
{
    ...
}
```

Однако если перегрузка шаблонов функций не допускается, возникает проблема: в этом пространстве имен другие шаблоны операторов == объявлять нельзя, хотя они могут понадобиться для иных шаблонов классов. Бартону и Нэкману удалось решить эту проблему путем определения в классе оператора равенства в виде обычной функции-друга.

```
template<typename T>
class Array
{
    static bool areEqual(Array<T> const& a, Array<T> const& b);
public:
    ...
    friend bool operator==(Array<T> const& a, Array<T> const& b)
    {
        return areEqual(a, b);
    }
};
```

Предположим, что эта версия шаблона `Array` инстанцируется для типа `float`. Тогда в процессе инстанцирования объявляется функция-друг, с помощью которой реализован оператор равенства. Заметим, что сама по себе эта функция не является результатом инстанцирования шаблона функции. Это обычная функция (а не шаблон), *введенная* в глобальную область видимости как побочный эффект процесса инстанцирования. Поскольку это нешаблонная функция, ее можно перегружать с другими объявлениями оператора ==, причем это можно было делать еще до того, как в языке появились шаблоны функций. Бартон и Нэкман дали этому методу название *ограниченного расширения шаблонов*, поскольку в нем не используется шаблон `operator==(T, T)`, применимый для всех типов `T` (другими словами, *неограниченное расширение*).

Поскольку

```
operator==(Array<T> const&, Array<T> const&)
```

определен в теле класса, он автоматически является встраиваемой (`inline`) функцией, поэтому мы решили делегировать его реализацию статической функции-члену `areEqual`, которая не обязательно должна быть встроенной.

Поиск имен для определений дружественных функций с 1994 года изменился, поэтому метод Бартона–Нэкмана в стандартном C++ не так полезен. Во время его изобретения объявление друзей были видимы в охватывающей области видимости шаблона класса, когда этот шаблон инстанцировался с помощью процесса, который называется *внесение имени друга* (friend name injection). Нынешний стандарт C++ вместо этого находит объявления дружественных функций с помощью поиска, зависящего от аргумента (см. детали в разделе 13.2.2). Это означает, что по крайней мере один из аргументов вызова функции уже должен иметь класс, содержащий дружественную функцию, в качестве связанного. Дружественную

функцию не удастся найти, если аргументы имеют тип несвязанного класса, который может быть преобразован в класс, содержащий друга. Например:

inherit/wrapper.cpp

```
class S
{
};

template<typename T>
class Wrapper
{
private:
    T object;
public:
    Wrapper(T obj) // Неявное преобразование T в Wrapper<T>
        : object(obj){}
    friend void foo(Wrapper<T> const&)
    {}
};

int main()
{
    S s;
    Wrapper<S> w(s);
    foo(w); // OK: Wrapper<S> – класс, связанный с w
    foo(s); // ERROR: Wrapper<S> не связан с s
}
```

В данном примере вызов функции `foo (w)` корректен, поскольку функция `foo ()` является другом, объявленным в классе `Wrapper<S>`, с которым связана переменная `w`². Однако при вызове `foo (s)` объявление функции-друга `foo (Wrapper<S> const&)` невидимо, поскольку класс `Wrapper<S>`, в котором определена функция `foo ()`, не связан с аргументом `s` типа `S`. Поэтому, несмотря на допустимость неявного преобразования типа `S` в тип `Wrapper<S>` (с помощью конструктора класса `Wrapper<S>`), такое преобразование не рассматривается, поскольку функция-кандидат `foo ()` не найдена в первую очередь. Во времена изобретения Бартоном и Нэкманом своего метода введение имени друга делало друга `foo ()` видимым, а вызов `foo (s)` – успешным.

В современном C++ единственное преимущество при определении дружественной функции в шаблоне класса перед простым определением шаблона обычной функции является сугубо синтаксическим: определения дружественных функций имеют доступ к закрытым и защищенным членам включающего их класса, и не нужно повторять все параметры шаблона охватывающего шаблона класса. Однако определения дружественных функций могут быть полезными при сочетании их со странно рекурсивным шаблоном проектирования, как показано в реализации операторов, описанной в следующем разделе.

² Заметим, что эта переменная также связана с классом `S`, поскольку этот класс представляет собой аргумент шаблона типа переменной `w`. Конкретные правила ADL рассматриваются в разделе 13.2.1.

21.2.2. Реализации операторов

При реализации класса, который предоставляет перегруженные операторы, обычно предоставляются перегрузки для целого ряда различных (но взаимосвязанных) операторов. Например, класс, который реализует оператор равенства (`==`), скорее всего, реализует также оператор неравенства (`!=`), а класс, реализующий оператор меньше (`<`), скорее всего, реализует и другие реляционные операторы (`>`, `<=`, `>=`). Во многих случаях фактически интересно только определение одного из этих операторов, в то время как другие просто могут быть определены в терминах этого одного оператора. Например, оператор неравенства для класса `X` может быть определен с помощью оператора равенства:

```
bool operator!= (X const& x1, X const& x2)
{
    return !(x1 == x2);
}
```

Учитывая большое количество типов с аналогичными определениями оператора `!=`, возникает соблазн обобщить его в шаблоне:

```
template<typename T>
bool operator!= (T const& x1, T const& x2)
{
    return !(x1 == x2);
}
```

Стандартная библиотека языка C++ и в самом деле содержит такие определения в заголовочном файле `<utility>`. Однако эти определения (для `!=`, `>`, `<=` и `>=`) были отнесены к пространству имен `std::rel_ops` во время стандартизации, когда было установлено, что, будучи доступны в пространстве имен `std`, они вызывали проблемы. Когда эти определения видимы, получается, что *любой* тип имеет оператор `!=` (который может не инстанцироваться), и что этот оператор всегда будет иметь точное соответствие для обоих своих аргументов. В то время как первая проблема может быть решена с использованием методов SFINAE (см. раздел 19.4), так что это определение оператора `!=` будет инстанцироваться только для типов с подходящим оператором `==`, вторая проблема остается нерешенной: общее определение оператора `!=`, показанное выше, будет предпочтительнее пользовательского определения, которое требует, например, преобразовать производный класс в базовый, что может оказаться неприятным сюрпризом.

Альтернативная формулировка этих шаблонов операторов на основе CRTP позволяет классам прибегать к определениям общих операторов, предоставляя преимущества повышения повторного использования кода без побочных эффектов наличия слишком общего оператора:

inherit/equalitycomparable.cpp

```
template<typename Derived>
class EqualityComparable
{
public:
```

```
friend bool operator!= (Derived const& x1, Derived const& x2)
{
    return !(x1 == x2);
}

};

class X : public EqualityComparable<X>
{
public:
    friend bool operator== (X const& x1, X const& x2)
    {
        // Реализация логики сравнения двух объектов типа X
    }
};

int main()
{
    X x1, x2;
    if (x1 != x2) { }
}
```

Здесь мы объединили CRTP с методом Бартона–Нэкмана. Шаблон `Equality Comparable<>` использует CRTP для предоставления оператора `!=` для производного класса на основе определения оператора `==` производного класса. Это определение фактически предоставлено через определение дружественной функции (метод Бартона–Нэкмана), которая обеспечивает одинаковое поведение обоих параметров оператора `!=` при преобразовании типов.

Применение CRTP может быть полезным при переносе поведения в базовый класс при сохранении идентичности возможного производного класса. Наряду с методом Бартона–Нэкмана идиома CRTP может обеспечить общие определения для ряда операторов на основе некоторых канонических операторов. Эти свойства сделали идиому CRTP с методом Бартона–Нэкмана любимой технологией авторов шаблонных библиотек C++.

21.2.3. Фасады

Использование идиомы CRTP и метода Бартона–Нэкмана для определения некоторых операторов представляет собой удобное сокращение. Мы можем развить эту идею дальше, так, чтобы базовый класс CRTP определял большую часть или весь открытый интерфейс класса в терминах гораздо меньшего (но легче реализуемого) интерфейса, предоставляемого производным классом CRTP. Этот проектный шаблон, именуемый *фасадом*, особенно полезен при определении новых типов, которые должны отвечать требованиям некоторого существующего интерфейса — числовых типов, итераторов, контейнеров и так далее.

Чтобы проиллюстрировать проектный шаблон фасада, мы реализуем фасад для итераторов, который существенно упрощает процесс написания итераторов, соответствующих требованиям стандартной библиотеки. Требуемый интерфейс для типа итератора (особенно *итератора с произвольным доступом*) довольно велик. Приведенная ниже схема для шаблона класса `IteratorFacade` демонстрирует требования к интерфейсу итератора.

inherit/iteratorfacadeskel.hpp

```
template<typename Derived, typename Value, typename Category,
         typename Reference = Value&,
         typename Distance = std::ptrdiff_t>
class IteratorFacade
{
public:
    using value_type = typename std::remove_const<Value>::type;
    using reference = Reference;
    using pointer = Value*;
    using difference_type = Distance;
    using iterator_category = Category;

    // Интерфейс входного итератора:
    reference operator *() const
    { ... }
    pointer operator ->() const
    { ... }
    Derived& operator ++()
    { ... }
    Derived operator ++(int)
    { ... }
    friend bool operator==(IteratorFacade const& lhs,
                           IteratorFacade const& rhs)
    { ... }
    ...

    // Интерфейс двунаправленного итератора:
    Derived& operator --()
    { ... }
    Derived operator --(int)
    { ... }

    // Интерфейс итератора с произвольным доступом:
    reference operator [](difference_type n) const
    { ... }
    Derived& operator +=(difference_type n)
    { ... }
    ...
    friend difference_type operator -(IteratorFacade const& lhs,
                                      IteratorFacade const& rhs)
    { ... }
    friend bool operator <(IteratorFacade const& lhs,
                           IteratorFacade const& rhs)
    { ... }
    ...
};

};
```

Мы полностью опустили некоторые объявления для краткости, но даже реализация всех перечисленных функций для каждого нового итератора является довольно трудоемкой работой. К счастью, этот интерфейс можно свести к нескольким основным операциям.

- Для всех итераторов:
 - `dereference()`: доступ к значению, на которое указывает итератор (обычно используется операторами `*` и `->`);
 - `increment()`: перемещение итератора так, чтобы он указывал на следующий элемент последовательности;
 - `equals()`: определение, указывают ли два итератора на один и тот же элемент последовательности.
- Для двунаправленных итераторов:
 - `decrement()`: перемещение итератора так, чтобы он указывал на предыдущий элемент в последовательности.
- Для итераторов с произвольным доступом:
 - `advance()`: перемещение итератора на n шагов вперед (или назад);
 - `measureDistance()`: определение количества шагов для перехода от одного итератора к другому в последовательности.

Фасад призван адаптировать тип, который реализует только указанные основные операции, для предоставления полного интерфейса итератора. Реализация `IteratorFacade` в основном состоит из отображения синтаксиса итератора на этот минимальный интерфейс. В следующих примерах мы используем функции-члены `asDerived()` для доступа к производному классу `CRTP`:

```
Derived& asDerived()
{
    return *static_cast<Derived*>(this);
}

Derived const& asDerived() const
{
    return *static_cast<Derived const*>(this);
}
```

При наличии такого определения реализация большей части фасада является простой задачей³. Мы проиллюстрируем только определения для некоторых требований входных итераторов; прочие определения создаются аналогично.

```
reference operator*() const
{
    return asDerived().dereference();
}

Derived& operator++()
{
    asDerived().increment();
    return asDerived();
}
```

³Для упрощения представления мы игнорируем наличие прокси-итераторов, операция разыменования которых не возвращает истинную ссылку. Полная реализация фасада итератора, как в [15], должна корректировать возвращаемые типы `operator->` и `operator[]` для учета наличия прокси.

```
Derived operator++(int)
{
    Derived result(asDerived());
    asDerived().increment();
    return result;
}

friend bool operator== (IteratorFacade const& lhs,
                        IteratorFacade const& rhs)
{
    return lhs.asDerived().equals(rhs.asDerived());
}
```

Определение итератора связанныго списка

При наличии нашего определения `IteratorFacade` мы можем легко определить итератор для простого класса связанныго списка. Представим, например, что мы определили узел в связанным списке следующим образом:

inherit/listnode.hpp

```
template<typename T>
class ListNode
{
public:
    T value;
    ListNode<T>* next = nullptr;
    ~ListNode()
    {
        delete next;
    }
};
```

Используя `IteratorFacade`, можно легко определить итератор, работающий с таким списком:

inherit/listnodeiterator0.hpp

```
template<typename T>
class ListNodeIterator
    : public IteratorFacade<ListNodeIterator<T>, T,
                           std::forward_iterator_tag>
{
    ListNode<T>* current = nullptr;
public:
    T& dereference() const
    {
        return current->value;
    }
    void increment()
    {
        current = current->next;
    }
};
```

```

bool equals(ListNodeIterator const& other) const
{
    return current == other.current;
}
ListNodeIterator(ListNode<T>* current = nullptr) : current(
    current) { }
};

```

`ListNodeIterator` предоставляет все корректные операторы и вложенные типы, необходимые для работы в качестве итератора одностороннего итератора, и требует очень мало кода для реализации. Как мы увидим позже, определение более сложных итераторов (например, итераторов произвольного доступа) требует лишь небольшого количества дополнительной работы.

Скрытие интерфейса

Один из недостатков нашей реализации `ListNodeIterator` заключается в том, что мы обязаны предоставить в качестве открытого интерфейса операции `dereference()`, `advance()` и `equals()`. Чтобы исключить это требование, можно переделать `IteratorFacade` так, чтобы он выполнял все свои операции над производным классом `CRTP` через отдельный класс *доступа*, который мы назовем `IteratorFacadeAccess`:

inherit/iteratorfacadeaccessskel.hpp

```

// Делаем этот класс другом, чтобы обеспечить для
// IteratorFacade доступ к основным операциям итератора:
class IteratorFacadeAccess
{
    // Эти определения может использовать только IteratorFacade
    template<typename Derived, typename Value, typename Category,
        typename Reference, typename Distance>
    friend class IteratorFacade;

    // Требуется для всех итераторов:
    template<typename Reference, typename Iterator>
    static Reference dereference(Iterator const& i)
    {
        return i.dereference();
    }

    ...
    // Требуется для двунаправленных итераторов:
    template<typename Iterator>
    static void decrement(Iterator& i)
    {
        return i.decrement();
    }

    // Требуется для итераторов произвольного доступа:
    template<typename Iterator, typename Distance>
    static void advance(Iterator& i, Distance n)
    {
        return i.advance(n);
    }

    ...
};

};


```

Этот класс предоставляет статические функции-члены для каждой из основных операций итератора, вызывая соответствующие (нестатические) функции-члены предоставленного итератора. Все статические функции-члены являются закрытыми, с доступом только для самого `IteratorFacade`. Таким образом, наш `ListNodeIterator` может сделать `IteratorFacadeAccess` другом и оставить закрытым интерфейс, необходимый для фасада:

```
friend class IteratorFacadeAccess;
```

Адаптеры итераторов

Наш `IteratorFacade` позволяет легко создать *адаптер* итератора, который принимает существующий итератор и возвращает новый итератор, предоставляющий некоторое преобразованное представление базовой последовательности. Например, у нас мог бы существовать контейнер значений `Person`:

inherit/person.hpp

```
struct Person
{
    std::string firstName;
    std::string lastName;
    friend std::ostream& operator<<(std::ostream& strm,
                                         Person const& p)
    {
        return strm << p.lastName << ", " << p.firstName;
    }
};
```

Однако вместо того, чтобы проходить по всем значениям `Person` в контейнере, мы хотим видеть только имена. В этом разделе мы разрабатываем адаптер итератора `ProjectionIterator`, который позволяет нам “спроектировать” значения базового итератора на некоторый указатель на член-данное, например `Person::firstName`.

`ProjectionIterator` представляет собой итератор, определенный в терминах базового итератора (`Iterator`) и типа значения, доступного через итератор (`T`):

inherit/projectioniteratorskel.hpp

```
template<typename Iterator, typename T>
class ProjectionIterator
: public IteratorFacade <
    ProjectionIterator<Iterator, T>,
    T,
    typename std::iterator_traits<Iterator>::iterator_category,
    T&,
    typename std::iterator_traits<Iterator>::difference_type >
{
    using Base = typename std::iterator_traits<Iterator>::value_type;
    using Distance =
        typename std::iterator_traits<Iterator>::difference_type;
```

```

Iterator iter;
T Base::* member;

friend class IteratorFacadeAccess;
... // Реализация основных итераторных
    // операций для IteratorFacade
public:
    ProjectionIterator(Iterator iter, T Base::* member)
        : iter(iter), member(member) { }
};

template<typename Iterator, typename Base, typename T>
auto project(Iterator iter, T Base::* member)
{
    return ProjectionIterator<Iterator, T>(iter, member);
}

```

Каждый итератор проекции хранит два значения: `iter` — итератор базовой последовательности (значений `Base`) и `member` — указатель на член-данное, описывающий, какой именно член проецируется. С учетом этого мы рассмотрим аргументы шаблона, предоставляемые базовому классу `IteratorFacade`. Первым является сам `ProjectionIterator` (для возможности применения идиомы CRTP). Второй (`T`) и четвертый (`T&`) аргументы представляют собой типы значения и ссылки нашего итератора проекции, определяя его как последовательность значений ⁴. Третий и пятый аргументы просто передают категорию и тип разности базового итератора. Таким образом, наш итератор проекции будет входным итератором, когда `Iterator` является входным итератором, двунаправленным итератором, когда `Iterator` является двунаправленным итератором, и так далее. Функция `project()` позволяет легко создать итератор проекции.

Единственной недостающей частью является реализация основных требований к `IteratorFacade`. Наиболее интересна функция `dereference()`, разыменовывающая базовый итератор, а затем выполняющая проекцию:

```

T& dereference() const
{
    return (*iter).*member;
}

```

Остальные операции реализуются в терминах базового итератора:

```

void increment()
{
    ++iter;
}

bool equals(ProjectionIterator const& other) const
{
    return iter == other.iter;
}

```

⁴ Для простоты мы вновь считаем, что базовый итератор возвращает ссылку, а не прокси.

```
void decrement()
{
    --iter;
}
```

Для краткости мы опустили определения итераторов произвольного доступа, которые создаются аналогично.

Вот и все! С нашим итератором проекции мы можем вывести имена из вектора, содержащего значения Person:

inherit/projectioniterator.cpp

```
#include <vector>
#include <algorithm>
#include <iterator>

int main()
{
    std::vector<Person> authors = {
        {"David", "Vandevoorde"},
        {"Nicolai", "Josuttis"},
        {"Douglas", "Gregor"}
    };
    std::copy(std::project(authors.begin(), &Person::firstName),
              std::project(authors.end(), &Person::firstName),
              std::ostream_iterator<std::string>(std::cout, "\n"));
}
```

Вывод программы имеет следующий вид:

```
David
Nicolai
Douglas
```

Проектный шаблон “Фасад” особенно полезен для создания новых типов, которые соответствуют некоторому определенному интерфейсу. Новые типы должны предоставлять фасаду только несколько основных операций (от 3 до 6 для нашего фасада итератора), а он заботится о предоставлении полного и корректного открытого интерфейса, используя комбинацию CRTP и метода Бартона–Нэкмана.

21.3. Миксины

Рассмотрим простой класс Polygon, состоящий из последовательности точек:

```
class Point
{
public:
    double x, y;
    Point() : x(0.0), y(0.0) { }
    Point(double x, double y) : x(x), y(y) { }
};
```

```
class Polygon
{
    private:
        std::vector<Point> points;
    public:
        ... // Открытые операции
};
```

Этот класс `Polygon` будет более полезным, если пользователь сможет расширить набор сведений, связанных с каждой точкой `Point`, чтобы иметь возможность указать информацию приложения, такую как цвет каждой точки, или, возможно, связать с каждой точкой метку. Один из вариантов сделать такое расширение возможным — параметризовать `Polygon` типом точки:

```
template<typename P>
class Polygon
{
    private:
        std::vector<P> points;
    public:
        ... // Открытые операции
};
```

Пользователи могут создавать свой собственный “точкообразный” тип данных, предоставляющий тот же интерфейс, что и `Point`, но включающий и другие данные приложения с помощью наследования:

```
class LabeledPoint : public Point
{
public:
    std::string label;
    LabeledPoint() : Point(), label("") { }
    LabeledPoint(double x, double y) : Point(x, y), label("") { }
};
```

Эта реализация имеет свои недостатки. С одной стороны она требует, чтобы тип `Point` был доступен пользователю, так, чтобы пользователь мог его наследовать. Кроме того, автор `LabeledPoint` должен быть осторожным и предоставить точно такой же интерфейс, как и `Point` (например, наследованием или предоставлением всех тех же конструкторов, что и у `Point`), иначе `LabeledPoint` не сможет работать с `Polygon`. Это ограничение становится более проблематичным, если `Point` изменяется от одной версии шаблона `Polygon` к другой: добавление нового конструктора `Point` может потребовать обновления каждого производного класса.

Миксины (*mixins*) обеспечивают альтернативный способ настроить поведение типа без наследования от него. Миксины, по существу, инвертируют нормальное направление наследования, потому что новые классы являются “подмешанными” (*mixed in*) в иерархию наследования как базовые классы шаблона класса, а не создаются как новые производные классы. Этот подход разрешает введение новых членов-данных и других операций, не требуя какого-либо дублирования интерфейса.

Шаблон класса, который поддерживает миксины, обычно принимает произвольное количество дополнительных классов, от которых он наследуется:

```
template<typename... Mixins>
class Point : public Mixins...
{
public:
    double x, y;
    Point() : Mixins()..., x(0.0), y(0.0) { }
    Point(double x, double y) : Mixins()..., x(x), y(y) { }
};
```

Теперь мы можем “подмешать” базовый класс с меткой, чтобы получить Labeled Point:

```
class Label
{
public:
    std::string label;
    Label() : label("") { }
};
using LabeledPoint = Point<Label>;
```

или даже добавить несколько базовых классов:

```
class Color
{
public:
    unsigned char red = 0, green = 0, blue = 0;
};
using MyPoint = Point<Label, Color>;
```

Этот основанный на миксинах класс Point позволяет легко предоставить дополнительную информацию к классу Point без изменения его интерфейса, так что Polygon становится проще использовать и развивать. Пользователи должны только применять неявное преобразование из специализации Point в соответствующий класс миксина (такой как Label или Color, показанные выше) для доступа к данным или интерфейсу. Кроме того, класс Point может быть даже полностью скрытым с помощью предоставления миксинов шаблону класса Polygon:

```
template<typename... Mixins>
class Polygon
{
private:
    std::vector<Point<Mixins...>> points;
public:
    ... // Открытые операции
};
```

Миксины полезны в тех случаях, когда шаблон требует некоторого небольшого уровня настройки — например, декорирование внутренне хранимых объектов пользовательскими данными — без необходимости требовать от библиотеки открытия и документирования этих внутренних типов данных и их интерфейсов.

21.3.1. Странные миксины

Миксины могут быть еще более мощным средством при объединении их со странно рекурсивным шаблоном проектирования (CRTP), описанным в разделе 21.2. Здесь каждый из миксинов на самом деле является шаблоном класса,

который будет предоставлен с типом производного класса, что обеспечивает дополнительные настройки этого производного класса. Версия класса `Point` на основе миксина с применением идиомы CRTP будет выглядеть следующим образом:

```
template<template<typename>... Mixins>
class Point : public Mixins<Point>...
{
public:
    double x, y;
    Point() : Mixins<Point>()..., x(0.0), y(0.0) { }
    Point(double x, double y) : Mixins<Point>()..., x(x), y(y) { }
};
```

Эта формулировка требует немного больше работы для каждого класса, который будет “подмешан”, поэтому такие классы, как `Label` и `Color`, должны стать шаблонами классов. Однако теперь миксины могут адаптировать свое поведение для определенного экземпляра производного класса, к которому они “подмешиваются”. Например, можно смешать рассматривавшийся ранее шаблон `ObjectCounter` с `Point`, чтобы подсчитывать количество точек, создаваемых `Polygon`, или объединять миксины с другими миксинами, специфичными для данного приложения.

21.3.2. Параметризованная виртуальность

Миксины также позволяют косвенно параметризовать такие атрибуты производного класса, как виртуальность функции-члена. Простой пример демонстрирует эту весьма удивительную технику:

inherit/virtual.cpp

```
#include <iostream>

class NotVirtual
{
};

class Virtual
{
public:
    virtual void foo()
    {
    }
};

template<typename... Mixins>
class Base : public Mixins...
{
public:
    // Виртуальность foo() зависит от ее объявления
    // (если таковое имеется) в миксинах базовых классов...
    void foo()
    {
        std::cout << "Base::foo()" << '\n';
    }
};
```

```

template<typename... Mixins>
class Derived : public Base<Mixins...>
{
public:
    void foo()
    {
        std::cout << "Derived::foo()" << '\n';
    }
};

int main()
{
    Base<NotVirtual>* p1 = new Derived<NotVirtual>;
    p1->foo();           // Вызов Base::foo()
    Base<Virtual>* p2 = new Derived<Virtual>;
    p2->foo();           // Вызов Derived::foo()
}

```

Эта методика может служить инструментом для проектирования шаблона класса, используемого как для инстанцирования конкретных классов, так и для расширения с применением наследования. Однако для получения класса, который оказывается хорошим базовым классом для более специализированной функциональности, редко бывает достаточно всего лишь придать виртуальность некоторым функциям-членам. Метод разработки такого вида требует более фундаментальных проектных решений. Поэтому обычно более практичной оказывается разработка двух различных инструментов (иерархии классов или шаблонов классов), чем попытки интегрировать их в единую иерархию шаблонов.

21.4. Именованные аргументы шаблона

Иногда различные методы работы с шаблонами приводят к тому, что шаблон содержит весьма значительное число разных типовых параметров. Конечно, как правило, многие из них имеют вполне приемлемые значения по умолчанию. Естественный способ определения такого шаблона класса может выглядеть так, как показано ниже.

```

template<typename Policy1 = DefaultPolicy1,
         typename Policy2 = DefaultPolicy2,
         typename Policy3 = DefaultPolicy3,
         typename Policy4 = DefaultPolicy4>
class BreadSlicer
{
    ...
};

```

Вероятно, такой шаблон чаще всего будет использоваться с аргументами по умолчанию с применением синтаксиса `BreadSlicer<>`. Однако, если некоторый аргумент имеет значение не по умолчанию, все предшествующие ему аргументы должны быть явно указаны (даже если они используют значения по умолчанию).

Понятно, что было бы гораздо привлекательнее использовать конструкцию `BreadSlicer<Policy3 = Custom>`, чем стандартную, имеющую вид `BreadSlicer<DefaultPolicy1, DefaultPolicy2, Custom>`. Ниже рассматривается методика, позволяющая использовать синтаксис, очень похожий на описанный⁵.

Наш метод заключается в размещении значений типа по умолчанию в базовом классе и в переопределении некоторых из них в процессе наследования. Вместо непосредственного указания аргументов типа предоставим их через вспомогательные классы. Например, можно написать `BreadSlicer<Policy3_is<Custom>>`. Поскольку каждый аргумент шаблона может описывать любую из стратегий, значения по умолчанию не могут быть различными. Иными словами, на верхнем уровне все параметры шаблона эквивалентны:

```
template<typename PolicySetter1 = DefaultPolicyArgs,
         typename PolicySetter2 = DefaultPolicyArgs,
         typename PolicySetter3 = DefaultPolicyArgs,
         typename PolicySetter4 = DefaultPolicyArgs>
class BreadSlicer
{
    using Policies = PolicySelector<PolicySetter1, PolicySetter2,
                                    PolicySetter3, PolicySetter4>;
    // Используем Policies::P1, Policies::P2, ...
    // для обращения к различным стратегиям
    ...
};
```

После этого остается только одна проблема — написать шаблон `PolicySelector`. Он должен объединить различные аргументы шаблона в единый тип, который перекрывает используемые по умолчанию члены-псевдонимы типа. Такое объединение может быть достигнуто с использованием наследования:

```
// PolicySelector<A,B,C,D> создает A,B,C,D в качестве
// базовых классов. Discriminator<> даже позволяет иметь
// несколько одинаковых базовых классов
template<typename Base, int D>
class Discriminator : public Base
{
};
template<typename Setter1, typename Setter2,
         typename Setter3, typename Setter4>
class PolicySelector : public Discriminator<Setter1, 1>,
                      public Discriminator<Setter2, 2>,
                      public Discriminator<Setter3, 3>,
                      public Discriminator<Setter4, 4>
{
```

Обратите внимание на использование промежуточного шаблона `Discriminator`. Он необходим для того, чтобы можно было иметь одинаковые типы `Setter`.

⁵ Обратите внимание на то, что подобное расширение языка для аргументов вызова функции было предложено (и отклонено) в процессе стандартизации языка C++ еще раньше (более подробно об этом говорится в разделе 17.4).

(Иметь несколько непосредственных базовых классов одного и того же типа нельзя; обойти это ограничение можно с помощью опосредованного наследования.)

Как обещали ранее, соберем в базовом классе все значения по умолчанию.

```
// Именуем стратегии по умолчанию как P1, P2, P3, P4
class DefaultPolicies
{
    public:
        using P1 = DefaultPolicy1;
        using P2 = DefaultPolicy2;
        using P3 = DefaultPolicy3;
        using P4 = DefaultPolicy4;
};
```

Однако мы должны быть внимательны и избегать неоднозначности при много-кратном наследовании от этого базового класса, т.е. базовый класс должен наследоваться виртуально:

```
// Класс для стратегий по умолчанию позволяет избежать
// неоднозначности при помощи виртуального наследования
class DefaultPolicyArgs : virtual public DefaultPolicies
{};

};
```

Наконец, нужно написать ряд шаблонов для перекрытия значений стратегий, заданных по умолчанию:

```
template<typename Policy>
class Policy1_is : virtual public DefaultPolicies
{
    public:
        using P1 = Policy; // Перекрытие шаблона типа
};

template<typename Policy>
class Policy2_is : virtual public DefaultPolicies
{
    public:
        using P2 = Policy; // Перекрытие шаблона типа
};

template<typename Policy>
class Policy3_is : virtual public DefaultPolicies
{
    public:
        using P3 = Policy; // Перекрытие шаблона типа
};

template<typename Policy>
class Policy4_is : virtual public DefaultPolicies
{
    public:
        using P4 = Policy; // Перекрытие шаблона типа
};
```

Теперь мы можем достичь нашей цели. Рассмотрим конкретный пример и инстанцируем `BreadSlicer<>` следующим образом:

```
BreadSlicer<Policy3_is<CustomPolicy>> bc;
```

Для этого `BreadSlicer<>` тип `Policies` определен как

```
PolicySelector<Policy3_is<CustomPolicy>,
               DefaultPolicyArgs,
               DefaultPolicyArgs,
               DefaultPolicyArgs>
```

С помощью шаблона класса `Discriminator<>` в результате будет получена иерархия, в которой все аргументы шаблона являются базовыми классами (рис. 21.4). Важное замечание: все эти базовые классы имеют один и тот же виртуальный базовый класс `DefaultPolicies`, который определяет заданные по умолчанию типы для `P1`, `P2`, `P3` и `P4`. Однако `P3` переопределен в одном из порожденных классов, а именно — в классе `Policy3_is<>`. Согласно так называемому *правилу доминирования* (domination rule), это определение скрывает определение базового класса. Таким образом, здесь *нет* неоднозначности⁶.

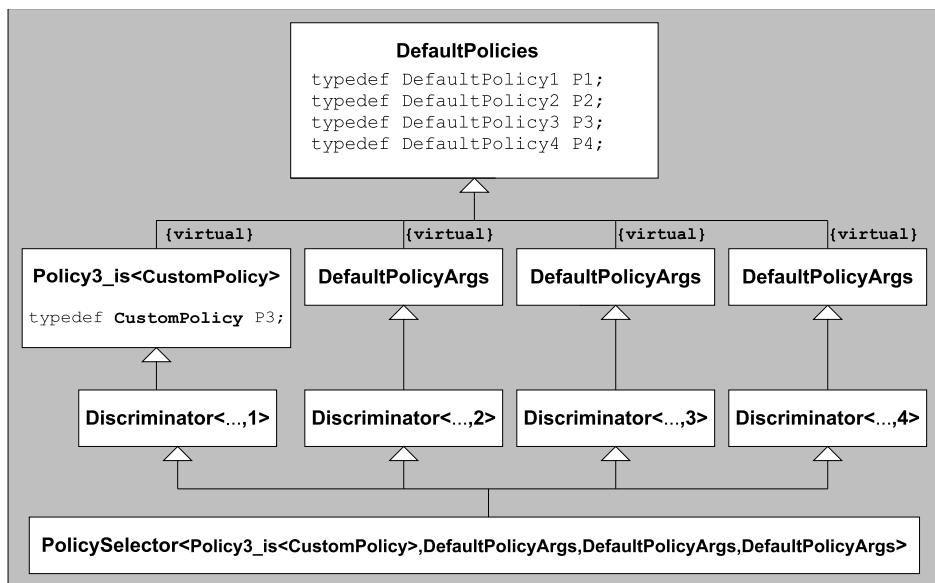


Рис. 21.4. Иерархия типов `BreadSlicer<>::Policies`

Внутри шаблона `BreadSlicer` можно обращаться к четырем стратегиям, используя для этого квалифицированные имена наподобие `Policies::P3`. Например:

⁶ Определение правила доминирования можно найти в разделе 10.2/6 первого Стандарта C++ [25], а также в [39], раздел 10.1.1.

```

template<...>
class BreadSlicer
{
    ...
public:
    void print()
    {
        Policies::P3::doPrint();
    }
    ...
};

}

```

Полный исходный текст можно найти в файле `inherit/namedtmpl.cpp`.

Здесь разработана методика для четырех параметров шаблона, но очевидно, что эта методика применима для любого разумного количества таких параметров. Обратите внимание на то, что при этом нигде не было реального инстанцирования объекта вспомогательного класса, содержащего виртуальные базовые классы. Следовательно, тот факт, что они являются виртуальными базовыми классами, не влияет на производительность или потребление памяти.

21.5. Заключительные замечания

Билл Гиббонс (Bill Gibbons) был основным инициатором введения оптимизации пустого базового класса в язык программирования C++, а Натан Майерс (Nathan Myers) предложил шаблон, подобный нашему `BaseMemberPair`, для получения максимальной пользы от применения данной оптимизации. В библиотеке Boost имеется значительно более сложный шаблон `compressed_pair`, который решает ряд упомянутых в данной главе проблем для шаблона `MyClass` и может использоваться вместо разработанного нами шаблона `BaseMemberPair`.

Метод CRTP используется еще с 1991 года. Первым этот метод формально описал Джеймс Коплин (James Coplien) [32]. С тех пор было опубликовано множество разнообразных применений CRTP. Однако иногда CRTP ошибочно применяют к *параметризованному наследованию*. Как было показано, CRTP не требует, чтобы наследование было параметризовано, а многие формы параметризованного наследования не согласуются с CRTP. Кроме того, CRTP иногда путают с методом Бартона–Нэкмана (см. раздел 21.2.1), поскольку Бартон и Нэкман часто использовали CRTP в комбинации с введением дружественных имен (каковое является важным компонентом метода Бартона–Нэкмана). Наше использование CRTP с методом Бартона–Нэкмана для реализации операторов следует тому же основному подходу, что и библиотека `Boost.Operators` [17], которая предоставляет обширный набор определений операторов. Аналогично наше рассмотрение фасадов итераторов следует библиотеке `Boost.Iterator` [15], которая обеспечивает богатый, соответствующий требованиям стандартной библиотеки интерфейс итераторов для производного типа, который предоставляет несколько основных операций итератора (равенство, разыменование, перемещение), а также решает сложные вопросы с участием прокси-итераторов (которые в книге не

рассматривались). Наш пример `ObjectCounter` практически идентичен методу, разработанному Скоттом Мейерсом (Scott Meyers) [51].

Понятие миксинов в объектно-ориентированном программировании появилось не позже 1986 года [54] как способ внести небольшие фрагменты функциональности в объектно-ориентированный класс. Использование шаблонов C++ для миксинов приобрело популярность вскоре после публикации первого стандарта C++, когда в статьях [61] и [38] были описаны подходы, широко используемые сегодня для миксинов. С тех пор они стали популярным методом в проектировании библиотек C++.

Именованные аргументы шаблона используются для упрощения некоторых шаблонов классов в библиотеке Boost, которая применяет метапрограммирование для создания типа со свойствами, схожими с нашим `PolicySelector` (но без использования виртуального наследования). Более простая альтернатива, представленная здесь, была разработана одним из авторов книги — Дэвидом Вандевурдом.

Глава 22

Статический и динамический полиморфизм

В главе 18, “Полиморфная мощь шаблонов”, описана природа статического (через шаблоны) и динамического (через наследование и виртуальные функции) полиморфизма в C++. Оба вида полиморфизма предоставляют мощные абстракции для написания программ, но каждый имеет свои недостатки: статический полиморфизм обеспечивает такую же производительность, как и неполиморфный код, но набор типов, которые могут быть использованы во время выполнения, фиксируется во время компиляции. С другой стороны, динамический полиморфизм посредством наследования позволяет одной версии полиморфной функции работать с типами, не известными во время компиляции, но при этом оказывается менее гибким, потому что типы должны наследоваться от общего базового класса.

В этой главе описывается, как устранить разрыв между статическим и динамическим полиморфизмом в C++, предоставляя некоторые из преимуществ каждой модели, обсуждающихся в разделе 18.3: меньший размер выполнимого кода и (почти) полностью компилируемый характер динамического полиморфизма, наряду с гибкостью интерфейса статического полиморфизма, обеспечивающей возможность работы без всякого наследования, например со встроенными типами. В качестве примера построим упрощенную версию шаблона `function<>` стандартной библиотеки.

22.1. Функциональные объекты, указатели и `std::function<>`

Функциональные объекты полезны для предоставления настраиваемого поведения для шаблонов. Например, приведенный ниже шаблон функции перечисляет целочисленные значения от 0 до некоторого значения, предоставляя каждое значение заданному функциональному объекту `f`:

bridge/forupto1.cpp

```
#include <vector>
#include <iostream>

template<typename F>
void forUpTo(int n, F f)
{
    for (int i = 0; i != n; ++i)
    {
        f(i); // Вызов переданной функции f с аргументом i
    }
}
```

```

void printInt(int i)
{
    std::cout << i << ' ';
}

int main()
{
    std::vector<int> values;
    // Вставка значений от 0 до 4:
    forUpTo(5, [&values](int i)
    {
        values.push_back(i);
    });

    // Вывод элементов:
    forUpTo(5, printInt); // Вывод 0 1 2 3 4
    std::cout << '\n';
}

```

Шаблон функции `forUpTo()` может использоваться с любым функциональным объектом, включая лямбда-выражения, указатели на функции или любой класс, который реализует либо соответствующий `operator()`, либо преобразование в указатель на функцию или ссылку. Каждое использование `forUpTo()`, скорее всего, будет приводить к другому инстанцированию шаблона функции. Наш пример шаблона функции достаточно мал, но при достаточно большом шаблоне вполне возможно, что эти инстанцирования приведут к увеличению размера кода.

Один из способов ограничить увеличение размера кода — превратить шаблон функции в нешаблонную функцию, не требующую инстанцирования. Например, можно попытаться сделать это с помощью указателя на функцию:

bridge/forupto2.hpp

```

void forUpTo(int n, void (*f)(int))
{
    for (int i = 0; i != n; ++i)
    {
        f(i); // Вызов переданной функции f для i
    }
}

```

Однако, хотя эта реализация будет работать при передаче функции `printInt()`, при передаче лямбда-выражения мы получим ошибку:

```

forUpTo(5, printInt); // OK: Вывод 0 1 2 3 4
forUpTo(5,
       [&values](int i) // Ошибка: лямбда-выражение не
                      { // преобразуется в указатель на функцию
                          values.push_back(i);
                      });

```

Шаблон класса стандартной библиотеки `std::function<>` позволяет записать функцию `forUpTo()` иначе:

bridge/forupto3.hpp

```
#include <functional>

void forUpTo(int n, std::function<void(int)> f)
{
    for (int i = 0; i != n; ++i)
    {
        f(i); // Вызов переданной функции f для i
    }
}
```

Аргументом шаблона `std::function<>` является тип функции, который описывает типы параметров функционального объекта и тип возвращаемого значения, подобно тому как указатель на функцию описывает типы параметров и результата.

Такая запись `forUpTo()` предоставляет некоторые аспекты статического полиморфизма — способность работать с неограниченным набором типов, включая указатели на функции, лямбда-выражения и произвольные классы с подходящим оператором `operator()` — в то время как сама функция остается ненешаблонной, с единственной реализацией. Это достигается с помощью метода под названием *стирание типа* (type erasure), который ликвидирует разрыв между статическим и динамическим полиморфизмом.

22.2. Обобщение указателей на функции

Тип `std::function<>` по сути представляет собой обобщенный указатель на функцию C++, обеспечивая те же основные операции.

- Он используется для вызова функции так, что вызывающий код может не иметь никакой информации о вызываемой функции.
- Он может быть скопирован, перемещен или присвоен.
- Он может быть инициализирован или ему может быть присвоена другая функция (с совместимой сигнатурой).
- Он имеет “нулевое” состояние, которое указывает, что с ним не связана никакая функция.

Однако в отличие от указателей на функции C++ `std::function<>` может также хранить лямбда-выражения или любой иной функциональный объект с подходящим оператором `operator()`, типы которых могут быть различными.

В оставшейся части этой главы мы будем создавать наш собственный шаблон класса обобщенного указателя на функцию, `FunctionPtr`, обеспечивающий те же основные операции и возможности, и могущий использоваться вместо `std::function<>`:

bridge/forupto4.cpp

```
#include "functionptr.hpp"
#include <vector>
#include <iostream>
```

```

void forUpTo(int n, FunctionPtr<void(int)> f)
{
    for (int i = 0; i != n; ++i)
    {
        f(i); // Вызов переданной функции f для i
    }
}

void printInt(int i)
{
    std::cout << i << ' ';
}

int main()
{
    std::vector<int> values;

    // Вставка значений от 0 до 4:
    forUpTo(5, [&values](int i)
    {
        values.push_back(i);
    });

    // Вывод элементов:
    forUpTo(5, printInt); // Вывод 0 1 2 3 4
    std::cout << '\n';
}

```

Интерфейс `FunctionPtr` довольно простой, обеспечивающий создание, копирование, перемещение, уничтожение, инициализацию, присваивание произвольных функциональных объектов, а также вызов базового функционального объекта. Наиболее интересной частью интерфейса является то, как все это описано полностью в рамках частичной специализации шаблона класса, служащей для разделения аргумента шаблона (типа функции) на его составные части (типы результата и аргументов):

bridge/functionptr.hpp

```

// Первичный шаблон:
template<typename Signature>
class FunctionPtr;

// Частичная специализация:
template<typename R, typename... Args>
class FunctionPtr<R(Args...)>
{
private:
    FunctorBridge<R, Args...>* bridge;
public:
    // Конструкторы:
    FunctionPtr() : bridge(nullptr)
    {
    }
    FunctionPtr(FunctionPtr const& other); // См.functionptr-cpinv.hpp
    FunctionPtr(FunctionPtr& other)
        : FunctionPtr(static_cast<FunctionPtr const&>(other))

```

```

{
}
FunctionPtr(FunctionPtr& other) : bridge(other.bridge)
{
    other.bridge = nullptr;
}
// Построение из произвольного функционального объекта:
template<typename F> FunctionPtr(F&& f); // См. functionptr-init.hpp

// Операторы присваивания:
FunctionPtr& operator=(FunctionPtr const& other)
{
    FunctionPtr tmp(other);
    swap(*this, tmp);
    return *this;
}
FunctionPtr& operator=(FunctionPtr&& other)
{
    delete bridge;
    bridge = other.bridge;
    other.bridge = nullptr;
    return *this;
}
// Построение и присваивание произвольного функционального объекта:
template<typename F> FunctionPtr& operator=(F&& f)
{
    FunctionPtr tmp(std::forward<F>(f));
    swap(*this, tmp);
    return *this;
}

// Деструктор:
~FunctionPtr()
{
    delete bridge;
}

friend void swap(FunctionPtr& fp1, FunctionPtr& fp2)
{
    std::swap(fp1.bridge, fp2.bridge);
}
explicit operator bool() const
{
    return bridge == nullptr;
}

// Вызов:
R operator()(Args... args) const; // См. functionptr-cpinv.hpp
};

}

```

Реализация содержит единственную нестатическую переменную-член `bridge`, которая будет отвечать за хранение и работу с хранимым функциональным объектом. Владение этим указателем передано объекту `FunctionPtr`, поэтому большая часть реализации просто управляет этим указателем. Нереализованные функции содержат интересный код, который будет описан в следующих подразделах.

22.3. Интерфейс моста

Шаблон класса `FunctorBridge` отвечает за владение базовым функциональным объектом и манипуляции с ним. Он реализован как абстрактный базовый класс, формирующий основу для динамического полиморфизма `FunctionPtr`:

bridge/functorbridge.hpp

```
template<typename R, typename... Args>
class FunctorBridge
{
public:
    virtual ~FunctorBridge() = 0;
    virtual FunctorBridge* clone() const = 0;
    virtual R invoke(Args... args) const = 0;
};
```

`FunctorBridge` предоставляет основные операции, необходимые для работы с хранимым функциональным объектом посредством виртуальных функций: деструктор, операция `clone()` для копирования и `invoke()` для вызова базового функционального объекта. Не забудьте определить `clone()` и `invoke()` как константные функции-члены¹.

С помощью этих виртуальных функций можно реализовать копирующий конструктор и оператор вызова функции `FunctionPtr`:

bridge/functionptr-cpinv.hpp

```
template<typename R, typename... Args>
FunctionPtr<R(Args...)>::FunctionPtr(FunctionPtr const& other)
    : bridge(nullptr)
{
    if (other.bridge)
    {
        bridge = other.bridge->clone();
    }
}

template<typename R, typename... Args>
R FunctionPtr<R(Args...)>::operator()(Args... args) const
{
    return bridge->invoke(std::forward<Args>(args)...);
}
```

22.4. Стирание типа

Каждое инстанцирование `FunctorBridge` является абстрактным классом, поэтому его производные классы ответственны за предоставление фактических

¹ Константность `invoke()` — это ремень безопасности против вызова неконстантных перегрузок `operator()` посредством константных объектов `FunctionPtr`, что нарушает ожидания программистов.

реализаций своих виртуальных функций. Для поддержки полного спектра потенциальных функциональных объектов (т.е. неограниченного их множества) нам потребуется неограниченное количество производных классов. К счастью, мы можем достичь этого путем параметризации производного класса типом хранимого функционального объекта:

bridge/specifcfunctorbridge.hpp

```
template<typename Functor, typename R, typename... Args>
class SpecificFunctorBridge : public FunctorBridge<R, Args...>
{
    Functor functor;
public:
    template<typename FunctorFwd>
    SpecificFunctorBridge(FunctorFwd&& functor)
        : functor(std::forward<FunctorFwd>(functor))
    {
    }
    virtual SpecificFunctorBridge* clone() const override
    {
        return new SpecificFunctorBridge(functor);
    }
    virtual R invoke(Args... args) const override
    {
        return functor(std::forward<Args>(args)...);
    }
};
```

Каждый экземпляр `SpecificFunctorBridge` хранит копию функционального объекта (типа которого является `Functor`), который может быть вызван, скопирован (путем клонирования `SpecificFunctorBridge`) или уничтожен (невно в деструкторе). Экземпляры `SpecificFunctorBridge` создаются каждый раз, когда `FunctionPtr` инициализируется новым функциональным объектом (что завершает наш пример `FunctionPtr`):

bridge/functionptr-init.hpp

```
template<typename R, typename... Args>
template<typename F>
FunctionPtr<R(Args...)>::FunctionPtr(F&& f)
    : bridge(nullptr)
{
    using Functor = std::decay_t<F>;
    using Bridge = SpecificFunctorBridge<Functor, R, Args...>;
    bridge = new Bridge(std::forward<F>(f));
}
```

Обратите внимание: в то время как сам конструктор `FunctionPtr` шаблонизируется функциональным объектом типа `F`, этот тип известен только определенной специализации `SpecificFunctorBridge` (описанной как псевдоним типа `Bridge`). После того как вновь выделенный экземпляр `Bridge` присваивается члену-данному `bridge`, дополнительная информация о конкретном типе `F`

становится потерянной из-за преобразования производного типа в базовый — из `Bridge*` в `FunctorBridge<R, Args...>*`². Эта потеря информации о типе объясняет, почему для описания методики преодоления пропасти между статическим и динамическим полиморфизмом часто используется термин *стирание типа* (*type erasure*).

Одной из особенностей реализации является использование `std::decay` (см. раздел Г.4) для получения типа `Functor`, который делает выведенный тип `F` подходящим для хранения, например, преобразуя ссылки на функциональные типы в типы указателей на функции и удаляя `const`, `volatile` и ссылочные типы верхнего уровня.

22.5. Условная передача владения

Шаблон `FunctionPtr` почти полностью заменяет указатель на функцию. Именно “почти” — он пока что не поддерживает одну операцию, предоставляемую указателями на функции: проверку, будут ли два объекта `FunctionPtr` вызывать одну и ту же функцию. Добавление такой операции требует оснащения `FunctorBridge` операцией `equals`:

```
virtual bool equals(FunctorBridge const* fb) const = 0;
```

вместе с реализацией в `SpecificFunctorBridge`, которая сравнивает хранимые функциональные объекты, если они имеют одинаковые типы:

```
virtual bool equals(FunctorBridge<R, Args...> const* fb) const
override
{
    if (auto specFb = dynamic_cast<SpecificFunctorBridge const*>(fb))
    {
        return functor == specFb->functor;
    }

    // Функторы с разными типами всегда различны:
    return false;
}
```

Наконец, мы реализуем оператор `operator==` для `FunctionPtr`, который сначала проверяет функторы на нулевое значение, а затем выполняет делегирование в `FunctorBridge`:

```
friend bool
operator==(FunctionPtr const& f1, FunctionPtr const& f2)
{
    if (!f1 || !f2)
    {
        return !f1 && !f2;
    }
```

² Хотя этот тип можно запросить (среди прочего) с помощью `dynamic_cast`, класс `FunctionPtr` делает указатель `bridge` закрытым, так что клиенты `FunctionPtr` не имеют доступа к самому типу.

```

        return f1.bridge->equals(f2.bridge);
    }

friend bool
operator!=(FunctionPtr const& f1, FunctionPtr const& f2)
{
    return !(f1 == f2);
}

```

Эта реализация корректна. Однако она имеет достойный сожаления недостаток: если FunctionPtr инициализирован функциональным объектом (или этот объект ему присвоен позже), который не имеет подходящего оператора operator== (например, сюда входят лямбда-выражения), то программа не будет компилироваться. Это может быть сюрпризом, потому что operator== из FunctionPtr еще даже не был использован. Многие другие шаблоны классов — например, std::vector — могут быть созданы с типами, которые не имеют оператора operator==, лишь бы этот оператор не был использован в программе для данных типов.

Данная проблема с operator== связана со стиранием типа: поскольку мы, по сути, теряем тип функционального объекта, при инициализации или присваивании FunctionPtr нужно захватить всю информацию, которую нам необходимо знать о типе, до того, как это присваивание или инициализация завершается. Эта информация включает вызов operator== функционального объекта, потому что мы не можем быть уверены, потребуется ли он³.

К счастью, чтобы выяснить, доступен ли operator==, перед тем как его вызывать, можно воспользоваться (довольно сложным) свойством (trait) на основе SFINAE (см. раздел 19.4):

bridge/isequalitycomparable.hpp

```

#include <utility>      // Для declval()
#include <type_traits> // Для true_type и false_type

template<typename T>
class IsEqualityComparable
{
private:
    // Проверка преобразуемости == и != в bool:
    static void* conv(bool); // Для проверки конвертируемости в bool
    template<typename U>
    static std::true_type test(
        decltype(conv(std::declval<U const&>() == std::declval<U const&>())),
        decltype(conv(!(std::declval<U const&>() == std::declval<U const&>())))
    );

```

³ Код вызова operator== инстанцируется потому, что все виртуальные функции шаблона класса (в данном случае SpecificFunctorBridge) обычно инстанцируются при инстанцировании самого шаблона класса.

```
// Резервный вариант:
template<typename U>
static std::false_type test(...);
public:
    static constexpr bool value = decltype(test<T>(nullptr,
                                                       nullptr))::value;
};
```

Свойство `IsEqualityComparable` применяет типичный подход для проверки выражений, представленный в разделе 19.4.1: две перегрузки `test()`, одна из которых содержит тестируемое выражение, обернутое в `decltype`, и другая, которая содержит многоточие и потому принимает произвольные аргументы. Первая функция `test()` пытается сравнивать два объекта типа `T const` с использованием оператора `==`, а затем убеждается, что результат может быть неявно преобразован в `bool` (для первого параметра) и передан оператору логического отрицания `operator!` с результатом, преобразуемым в `bool`. Если обе операции корректны, оба типа параметров будут `void*`.

С помощью свойства `IsEqualityComparable` можно построить шаблон класса `TryEquals`, который может либо вызывать `==` для данного типа (если он доступен), либо вызвать исключение, если подходящего оператора `==` не существует:

`bridge/tryequals.hpp`

```
#include <exception>
#include "isequalitycomparable.hpp"

template<typename T,
         bool EqComparable = IsEqualityComparable<T>::value>
struct TryEquals
{
    static bool equals(T const& x1, T const& x2)
    {
        return x1 == x2;
    }
};

class NotEqualityComparable : public std::exception
{
};

template<typename T>
struct TryEquals<T, false>
{
    static bool equals(T const& x1, T const& x2)
    {
        throw NotEqualityComparable();
    }
};
```

Наконец, с помощью `TryEquals` в нашей реализации `SpecificFunctorBridge` мы можем обеспечить поддержку для оператора `==` в `FunctionPtr` всякий раз,

когда типы сохраненных функциональных объектов совпадают и функциональный объект поддерживает оператор ==:

```
virtual bool equals(FunctorBridge<R, Args...> const* fb) const
override
{
    if (auto specFb = dynamic_cast<SpecificFunctorBridge const*>(fb))
    {
        return TryEquals<Functor>::equals(functor, specFb->functor);
    }

    // Функторы с разными типами не равны:
    return false;
}
```

22.6. Вопросы производительности

Стирание типа предоставляет некоторые преимущества как статического, так и динамического полиморфизма, но не все. В частности, производительность сгенерированного с помощью стирания типа кода оказывается более близкой к динамическому полиморфизму, потому что они оба используют динамическую диспетчеризацию посредством виртуальных функций. Следовательно, такие традиционные преимущества статического полиморфизма, как способность компилятора к встраиванию вызовов, могут быть потеряны. Будет ли эта потеря производительности существенной — зависит от конкретного приложения, но зачастую ее легко оценить, рассматривая, какая работа выполняется в вызываемой функции по отношению к стоимости вызова виртуальной функции. Если эти два значения близки (например, если с помощью FunctionPtr просто суммировать два целых числа), то, вероятно, стирание типа будет гораздо более медленным, чем версия со статическим полиморфизмом. С другой стороны, если вызываемая функция выполняет значительный объем работы — запрос базы данных, сортировку контейнера или обновление пользовательского интерфейса, — то накладные расходы стирания типа вряд ли будут вообще измеримыми.

22.7. Заключительные замечания

Кевлин Хенни (Kevlin Henney) популяризировал стирание типа в C++ введением типа any [41], который позднее стал популярной библиотекой Boost [11] и частью стандартной библиотеки C++, начиная с C++17. Эта методика была усовершенствована в библиотеке Boost.Function [12], в которой применяются различные оптимизации производительности и размера кода, и в конечном итоге превратилась в шаблон стандартной библиотеки std::function<>. Однако каждая из ранних библиотек рассматривала только один набор операций: any был типом простого значения, с одними лишь операциями приведения и копирования; function добавил к нему еще возможность вызова.

Более поздние проекты, такие как библиотеки Boost.TypeErasure [19] и Adobe's Poly [2], применяли методы шаблонного метапрограммирования, чтобы позволить пользователям создавать значения стертого типа с некоторым заданным списком возможностей. Например, следующий тип (созданный с использованием библиотеки Boost.TypeErasure) обрабатывает копирующее конструирование, операцию typeid и потоковый вывод:

```
using AnyPrintable = any<mpl::vector<copy_constructible<>,
    typeid_<>,
    ostreamable<>
>>;
```

Глава 23

Метапрограммирование

Метапрограммирование – это “программирование программ”. Другими словами, в процессе метапрограммирования создается код, который, будучи выполнен системой программирования, в свою очередь генерирует новый код, реализующий необходимую функциональность. Обычно понятие *метапрограммирование* подразумевает рефлексивность: метапрограммный компонент – это часть программы, для которой он генерирует фрагмент программного кода (дополнительный или отличающийся от кода программы).

Зачем нужно метапрограммирование? Как и большинство других технологий программирования, оно применяется, чтобы достичь больших функциональных возможностей ценой меньших затрат, измеряемых объемом кода, усилиями на сопровождение и т.п. Характерной особенностью метапрограммирования является то, что определенная часть необходимых пользователю вычислений выполняется на этапе трансляции программы. Его применение нередко объясняется повышением производительности (вычисление, которое выполняется во время трансляции, зачастую может быть полностью удалено из программы) или упрощением интерфейса (в общем случае метапрограмма короче, чем программа, в которую она раскрывается), а иногда и обеими этими причинами.

Нередко метапрограммирование основано на концепции свойств и функций типов, описанных в главе 19, “Реализация свойств типов”. Поэтому рекомендуется ознакомиться с указанной главой до изучения материала настоящей главы.

23.1. Состояние современного метапрограммирования

Метапрограммирование на C++ существенно эволюционировало со временем (заключительные замечания в конце этой главы рассказывают о некоторых вехах в данной области). Давайте рассмотрим и классифицируем различные подходы к метапрограммированию, широко используемые в современном C++.

23.1.1. Метапрограммирование значений

В первом издании этой книги мы были ограничены возможностями языка из исходного стандарта C++ (опубликован в 1998 году, с незначительными исправлениями в 2003 году). В том мире написание простых вычислений времени компиляции (“мета-”) было проблематично. Поэтому ему был посвящена значительная часть главы, включая довольно сложный пример вычисления квадратного корня из целочисленного значения во время компиляции с помощью рекурсивного инстанцирования. Как говорилось в разделе 8.2, стандарт C++11 и, в особенности, C++14, устранил массу сложностей, введя в язык

`constexpr`-функции¹. Например, начиная с C++14, функция времени компиляции для вычисления квадратного корня легко записывается следующим образом:

meta/sqrtconstexpr.hpp

```
template<typename T>
constexpr T sqrt(T x)
{
    // Обработка случаев, когда x и его квадратный корень
    // равны — как частный случай для упрощения критерия
    // прекращения итераций для больших x:
    if (x <= 1)
    {
        return x;
    }

    // Многократное определение, в какой половине интервала
    // [lo, hi] находится квадратный корень их x, пока интервал
    // не уменьшится до единственного значения:
    T lo = 0, hi = x;

    for (;;)
    {
        auto mid = (hi + lo) / 2, midSquared = mid * mid;

        if (lo + 1 >= hi || midSquared == x)
        {
            // Значение mid должно быть квадратным корнем:
            return mid;
        }

        // Продолжение с соответствующим полуинтервалом:
        if (midSquared < x)
        {
            lo = mid;
        }
        else
        {
            hi = mid;
        }
    }
}
```

Этот алгоритм ищет ответ, многократно деля пополам интервал, который содержит квадратный корень из x (корни 0 и 1 рассматриваются как особые случаи для упрощения критерия сходимости). Эта функция `sqrt()` может быть вычислена как во время компиляции, так и во время выполнения:

¹ Возможностей `constexpr` в C++11 было достаточно для решения многих распространенных задач, но эта модель программирования была далеко не всегда приятной (например, не были доступны инструкции цикла, поэтому итеративные вычисления приходилось заменять рекурсивными вызовами функций; см. раздел 23.2). В стандарт C++14 включены как инструкции цикла, так и многие другие языковые конструкции.

```

static_assert(sqrt(25) == 5,""); // OK (вычисление времени компиляции)
static_assert(sqrt(40) == 6,""); // OK (вычисление времени компиляции)

std::array<int,sqrt(40)+1> arr; // Объявление массива из 7 элементов
                                // (вычисление времени компиляции)
long long l = 53478;
std::cout << sqrt(l) << '\n';    // Вывод 231
                                // (вычисление времени выполнения)

```

Реализация этой функции может не быть наиболее эффективным способом вычисления квадратного корня во время выполнения (где часто используются особенности целевой машины), но, поскольку она предназначена для выполнения вычислений во время компиляции, абсолютная эффективность важна меньше, чем переносимость. Обратите внимание на то, что в этом примере нет никакой “шаблонной магии” для вычисления квадратного корня, только вывод аргумента шаблона, обычный для шаблона функции. Это “простой код C++”, который к тому же не является особенно сложным для чтения.

Метапрограммирование значений (т.е. программирование вычисления значений во время компиляции), пример которого показан выше, иногда очень полезно, но есть еще две дополнительные разновидности метапрограммирования, которые могут быть выполнены с использованием современного C++ (скажем, C++14 и C++17), — метапрограммирование типов и гибридное метапрограммирование.

23.1.2. Метапрограммирование типов

Мы уже сталкивались с вычислениями типов, рассматривая шаблоны свойств в главе 19, “Реализация свойств типов”, которые принимают в качестве входных данных тип и производят из него новый тип. Например, наш шаблон класса `RemoveReferenceT` вычисляет базовый тип для входного ссылочного типа. Однако примеры, которые мы разработали в упомянутой главе, выполняют только самые элементарные операции над типами. С помощью рекурсивного инстремирования шаблона — основы метапрограммирования шаблонов — мы можем выполнить значительно более сложные вычисления типов.

Рассмотрим следующий небольшой пример:

`meta/removeallextents.hpp`

```

// Первичный шаблон: в общем случае дает переданный тип:
template<typename T>
struct RemoveAllExtentsT
{
    using Type = T;
};

// Частичная специализация для типов массивов
// (с границами и без таковых):
template<typename T, std::size_t SZ>
struct RemoveAllExtentsT<T[SZ]>

```

```

{
    using Type = typename RemoveAllExtentsT<T>::Type;
};

template<typename T>
struct RemoveAllExtentsT<T[]>
{
    using Type = typename RemoveAllExtentsT<T>::Type;
};

template<typename T>
using RemoveAllExtents = typename RemoveAllExtentsT<T>::Type;

```

Здесь `RemoveAllExtents` представляет собой метафункцию типа (т.е. вычислительное устройство, производящее тип результата), которая будет удалять из типа произвольное количество “слоев массива”². Его можно использовать следующим образом:

```

RemoveAllExtents<int []>      // Дает int
RemoveAllExtents<int [5][10]> // Дает int
RemoveAllExtents<int [][10]>  // Дает int
RemoveAllExtents<int (*)[5]>  // Дает int (*)[5]

```

Эта метафункция выполняет свою задачу путем частичной специализации, которая соответствует случаю массива верхнего уровня и рекурсивно вызывает саму эту метафункцию.

Вычисления со значениями будут весьма ограниченными, если все, что нам доступно – это скалярные значения. К счастью, почти любой язык программирования имеет по крайней мере одну конструкцию контейнера значений, что значительно увеличивает мощность этого языка (а большинство языков имеют целый ряд разных контейнеров, таких как массивы/векторы, хеш-таблицы и т.д.). То же верно и для метапрограммирования типов: добавление конструкции “контейнер типов” значительно увеличивает применимость метапрограммирования. К счастью, современный C++ включает в себя механизмы, позволяющие разработать такой контейнер. В главе 24, “Списки типов”, подробно рассматривается разработка шаблона класса `TypeList<...>`, который представляет собой именно такой контейнер типов.

23.1.3. Гибридное метапрограммирование

Метапрограммирование значений и метапрограммирование типов позволяет вычислять значения и типы во время компиляции. Однако в конечном счете мы заинтересованы в результатах во время выполнения, поэтому используем эти метапрограммы в коде времени выполнения в местах, где ожидаются типы и константы. Но метапрограммирование способно на большее: во время компиляции можно программно собрать фрагменты кода с результатом, получаемым во время выполнения. Мы называем эту деятельность *гибридным метапрограммированием*.

² Стандартная библиотека C++ предоставляет соответствующее свойство типа `std::remove_all_extents` (см. раздел Г.4).

Чтобы проиллюстрировать этот принцип, начнем с простого примера: скалярного произведения двух значений `std::array`. Вспомним, что `std::array` является шаблоном контейнера фиксированной длины, объявленным следующим образом:

```
namespace std
{
    template<typename T, size_t N> struct array;
```

где `N` — количество элементов (типа `T`) в массиве. Для двух объектов одного и того же типа массива их скалярное произведение можно вычислить следующим образом:

```
template<typename T, std::size_t N>
auto dotProduct(std::array<T, N> const& x, std::array<T, N> const& y)
{
    T result{};

    for (std::size_t k = 0; k < N; ++k)
    {
        result += x[k] * y[k];
    }

    return result;
}
```

Прямая компиляция цикла `for` будет генерировать инструкции ветвления, которые на некоторых машинах могут привести к определенным накладным расходам по сравнению с линейным вычислением:

```
result += x[0] * y[0];
result += x[1] * y[1];
result += x[2] * y[2];
result += x[3] * y[3];
***
```

К счастью, современные компиляторы оптимизируют циклы в ту форму вычислений, которая оказывается наиболее эффективной для целевой платформы. Однако с didактическими целями перепишем реализацию нашего шаблона `dotProduct()` так, чтобы избежать цикла³:

```
template<typename T, std::size_t N>
struct DotProductT
{
    static inline T result(T* a, T* b)
    {
        return *a * *b + DotProduct<T,N-1>::result(a+1, b+1);
    }
};
```

³Этот метод известен как разворачивание цикла (*loop unrolling*). В общем случае мы не рекомендуем явно разворачивать циклы в переносимом коде, так как наилучшая стратегия разворачивания цикла сильно зависит от целевой платформы и тела цикла; компилятор обычно гораздо лучше принимает во внимание эту информацию.

```
// Частичная специализация в качестве критерия остановки
template<typename T>
struct DotProductT<T, 0>
{
    static inline T result(T*, T*)
    {
        return T{};
    }
};

template<typename T, std::size_t N>
auto dotProduct(std::array<T, N> const& x,
                std::array<T, N> const& y)
{
    return DotProductT<T, N>::result(x.begin(), y.begin());
}
```

Новая реализация делегирует работу шаблону класса `DotProductT`. Это позволяет нам использовать рекурсивное инстанцирование шаблона с частичной специализацией шаблона класса для прекращения рекурсии. Обратите внимание на то, как каждый экземпляр `DotProductT` выполняет суммирование одного члена скалярного произведения и скалярное произведение остальных компонентов массива. Поэтому для значений типа `std::array<T, N>` будет существовать N экземпляров первичного шаблона и один экземпляр завершающей частичной специализации. Это решение будет эффективным, если компилятор будет встраивать каждый вызов статической функции-члена `result()`. К счастью, обычно это так и происходит даже при включенном умеренном уровне оптимизации компилятора⁴.

Главным в этом коде является то, что он смешивает вычисления времени компиляции (достигнутые здесь с помощью рекурсивного инстанцирования шаблона), определяющие общую структуру кода, с вычислениями времени выполнения (вызов `result()`), которые и определяют конкретный результат вычислений времени выполнения.

Ранее мы упоминали, что возможности метапрограммирования типов значительно усиливаются при наличии “контейнеров типов”. Мы уже видели, что в гибридном метапрограммировании может быть полезен тип массива фиксированной длины. Тем не менее истинным “героем” гибридного метапрограммирования является кортеж. *Кортеж* — это последовательность значений, каждое с выбираемым типом. Стандартная библиотека языка C++ содержит шаблон класса `std::tuple`, который поддерживает это понятие. Например, код

```
std::tuple<int, std::string, bool> tVal{42, "Answer", true};
```

определяет переменную `tVal`, которая объединяет три значения с типами `int`, `std::string` и `bool` (в этом определенном порядке). Ввиду огромной важности таких контейнеров, как кортежи, для современного программирования C++,

⁴ Мы явно указали ключевое слово `inline`, потому что некоторые компиляторы (в особенности Clang) принимают его как подсказку, что выполнить встраивание в данном случае крайне желательно. С точки зрения языка эти функции и так неявно являются `inline`, потому что они определены в теле включающего их класса.

мы подробно рассматриваем один из них в главе 25, “Кортежи”. Тип `tVal` выше очень похож на простой тип `struct`:

```
struct MyTriple
{
    int v1;
    std::string v2;
    bool v3;
};
```

При наличии `std::array` и `std::tuple`, которые обеспечивают нас гибкими двойниками типов массивов и (простых) `struct`, естественно задуматься над аналогом простых типов объединений `union`, которые также могут быть полезными для гибридных вычислений. Для этой цели стандартная библиотека C++ в стандарте C++17 представляет шаблон `std::variant` (мы разработаем аналогичный компонент в главе 26, “Контролируемые объединения”).

Поскольку `std::tuple` и `std::variant`, как и типы `struct`, представляют собой гетерогенные типы, гибридное метапрограммирование, использующее такие типы, иногда называют *гетерогенным метапрограммированием*.

23.1.4. Гибридное метапрограммирование с типами единиц

Еще один пример, демонстрирующий возможности гибридных вычислений, — это библиотеки, которые в состоянии вычислять результаты значений типов различных единиц. Вычисление значений происходит во время выполнения, а результирующие единицы вычисляются во время компиляции.

Проиллюстрируем сказанное на весьма упрощенном примере. Мы хотим отслеживать единицы в терминах долей основной единицы. Например, если основная единица времени — секунда, то миллисекунда может быть представлена с помощью соотношения $1/1000$, а минута — с помощью соотношения $60/1$. Ключевым моментом является определение типа отношения, в котором каждое значение имеет свой собственный тип:

`meta/ratio.hpp`

```
template<unsigned N, unsigned D = 1>
struct Ratio
{
    static constexpr unsigned num = N; // Числитель
    static constexpr unsigned den = D; // Знаменатель
    using Type = Ratio<num, den>;
};
```

Теперь мы можем определить вычисления времени компиляции, такие как сложение двух единиц:

`meta/ratioadd.hpp`

```
// Реализация сложения двух дробей:
template<typename R1, typename R2>
struct RatioAddImpl
```

```

{
private:
    static constexpr unsigned den = R1::den*R2::den;
    static constexpr unsigned num = R1::num*R2::den+R2::num*R1::den;
public:
    typedef Ratio<num, den> Type;
};

// Используем объявление для удобства работы:
template<typename R1, typename R2>
using RatioAdd = typename RatioAddImpl<R1, R2>::Type;

```

Это позволяет нам вычислять сумму двух рациональных значений (дробей) во время компиляции:

```

using R1 = Ratio<1, 1000>;
using R2 = Ratio<2, 3>;
using RS = RatioAdd<R1, R2>;           // RS имеет тип Ratio<2003,2000>
std::cout << RS::num << '/'
    << RS::den << '\n';                // Выводит 2003/3000
using RA =
    RatioAdd<Ratio<2,3>,Ratio<5,7>>; // RA имеет тип type Ratio<29,21>
std::cout << RA::num << '/'
    << RA::den << '\n';                // Выводит 29/21

```

Теперь можно определить шаблон класса для единицы измерения длительностей, параметризованный произвольным типом значения и типом единицы, который является экземпляром `Ratio<>`:

meta/duration.hpp

```

// Тип длительности для значений типа T и типа единиц U:
template<typename T, typename U = Ratio<1>>
class Duration
{
public:
    using ValueType = T;
    using UnitType = typename U::Type;
private:
    ValueType val;
public:
    constexpr Duration(ValueType v = 0)
        : val(v)
    {
    }
    constexpr ValueType value() const
    {
        return val;
    }
};

```

Наиболее интересной частью является определение оператора `operator+` для сложения двух `Duration`:

meta/durationadd.hpp

```
// Сложение двух длительностей, единицы которых могут быть различны:
template<typename T1, typename U1, typename T2, typename U2>
auto constexpr operator+(Duration<T1, U1> const& lhs,
                         Duration<T2, U2> const& rhs)
{
    // Результатирующий тип представляет собой единицу с 1 в качестве
    // числителя и результатирующим знаменателем суммы двух дробей
    // с типами единиц
    using VT = Ratio<1, RatioAdd<U1, U2>::den>;
    // Результатирующее значение представляет собой сумму обоих
    // значений, преобразованных в тип результатирующих единиц:
    auto val = lhs.value() * VT::den / U1::den * U1::num +
               rhs.value() * VT::den / U2::den * U2::num;
    return Duration<decltype(val), VT>(val);
}
```

Мы разрешаем аргументам иметь различные типы единиц измерения, U1 и U2. И мы используем эти типы единиц для вычисления результатирующей длительности, имеющей тип единиц, который представляет собой соответствующую *единичную дробь* (дробь, у которой числитель равен 1). С учетом всего сказанного можно скомпилировать следующий код:

```
int x = 42;
int y = 77;
auto a = Duration<int, Ratio<1, 1000>>(x); // x миллисекунд
auto b = Duration<int, Ratio<2, 3>>(y); // y 2/3 секунд
auto c = a + b; // Вычисление результатирующего типа единиц измерения,
                // как 1/3000 секунды, и генерация кода для вычис-
                // ления времени выполнения c = a*3 + b*2000
```

Ключевым “гибридным” результатом является то, что для суммы с результатирующим тип определяется компилятором как тип единицы `Ratio<1, 3000>` во время компиляции, и генерируется код для вычисления во время выполнения результтирующего значения, которое измеряется в соответствующих типах единиц.

Поскольку тип значения представляет собой параметр шаблона, мы можем использовать класс `Duration` с типами значений, отличными от `int`, или даже применить гетерогенные типы значений (лишь бы была определена операция суммирования значений этих типов):

```
auto d = Duration<double, Ratio<1, 3>>(7.5); // 7.5 раз по 1/3 секунды
auto e = Duration<int, Ratio<1>>(4); // 4 секунды
auto f = d + e; // Результатирующий тип единицы вычисляется как
                // 1/3 секунды, и генерируется код для вычисления
                // f = d + e*3
```

Кроме того, компилятор может даже вычислить значения во время компиляции, если, конечно, они известны ему на этом этапе, поскольку `operator+` для длительностей является `constexpr`-оператором.

Этот подход использован в шаблоне класса `std::chrono` стандартной библиотеки C++ (с рядом усовершенствований, таких как применение предопределенных единиц (например, `std::chrono::milliseconds`), поддержка литералов, представляющих длительность (например, `10ms`) и обработка переполнений).

23.2. Размерности рефлексивного метапрограммирования

Ранее мы описывали метапрограммирование значений, основанное на вычислениях `constexpr`, и метапрограммирование типов, использующее рекурсивные инстанцирования шаблона. Очевидно, что оба эти варианта, доступные в современном C++, используют различные методы управления вычислениями. Однако оказывается, что метапрограммирование значений также может управляться с использованием рекурсивных инстанцирований шаблона (а до введения `constexpr`-функций в стандарте C++11 это был единственный доступный механизм управления). Например, приведенный ниже код вычисляет квадратный корень из целого числа с использованием рекурсивных инстанцирований:

meta/sqrt1.hpp

```
// Первичный шаблон для вычисления sqrt(N)
template<int N, int LO = 1, int HI = N>
struct Sqrt
{
    // Вычисление округленного вверх значения средней точки
    static constexpr auto mid = (LO + HI + 1) / 2;
    // Поиск не слишком большого значения в половине интервала
    static constexpr auto value = (N < mid * mid)
        ? Sqrt<N, LO, mid - 1>::value
        : Sqrt<N, mid, HI>::value;
};

// Частичная специализация для LO, равного HI
template<int N, int M>
struct Sqrt<N, M, M>
{
    static constexpr auto value = M;
};
```

Эта метапрограмма использует, по сути, тот же алгоритм, что и наше вычисление целого квадратного корня в `constexpr`-функции в разделе 23.1.1, последовательно деляя пополам интервал, который содержит квадратный корень. Входными данными для метафункции служат нетиповые аргументы шаблона; таковыми же оказываются и “локальные переменные”, отслеживающие границы интервала. Очевидно, что это гораздо менее дружественный подход, чем использование `constexpr`-функции. Позже мы проанализируем этот код, чтобы понять, насколько сильно он потребляет ресурсы компилятора.

В любом случае, мы видим, что вычислительный механизм метапрограммирования потенциально может предлагать много вариантов выбора. Кроме того, эти варианты могут рассматриваться не в единственном измерении — мы предпочтет думать, что всеобъемлющее метапрограммное решение в C++ должно делать выбор в *трех* измерениях:

- вычисления;
- рефлексия;
- генерация.

Рефлексия представляет собой возможность программно проверять функциональные возможности программы. *Генерация* относится к возможности создания дополнительного кода программы.

Мы уже видели два возможных варианта для выполнения вычислений: рекурсивное инстанцирование и `constexpr`-вычисление. Для рефлексии мы нашли частичное решение в свойствах типов (см. раздел 19.6.1). Хотя имеющиеся признаки делают доступными некоторое количество расширенных методов работы с шаблонами, они далеки от того, чтобы охватить все, что требуется в языке программирования от рефлексии. Например, для данного классового типа многие приложения хотели бы иметь возможность программно изучить члены этого класса. Имеющиеся свойства (*traits*) основаны на инстанцировании шаблонов, и C++, по-видимому, мог бы обеспечить дополнительные языковые средства или “встроенные” библиотечные компоненты⁵ для создания экземпляров шаблонов классов, которые содержат рефлексивную информацию времени компиляции. Такой подход хорошо использовать для вычислений, основанных на рекурсивном инстанцировании шаблонов. Но, к сожалению, экземпляры шаблонов классов потребляют много памяти компилятора, которая не может быть освобождена до конца компиляции (в противном случае потребуется значительно больше времени на компиляцию). Альтернативный вариант заключается в том, чтобы ввести новый стандартный тип для представления *рефлексивной информации*. Этот вопрос обсуждается в разделе 17.9 (и в Комитете по стандартизации C++).

В разделе 17.9 также показан потенциальный подход к *генерации кода*, который может стать актуален в будущем. Создание гибкого, обобщенного и дружественного к программисту механизма генерации кода в рамках существующего языка C++ остается активно исследуемой проблемой. Однако верно и то, что инстанцирование шаблонов всегда было разновидностью механизма генерации кода. Кроме того, компиляторы стали достаточно надежными в плане встраивания небольших функций в код вместо их вызовов для того, чтобы этот механизм мог использоваться как средство генерации кода. Эти наблюдения лежат в основе рассмотренного выше примера `DotProductT`. Таким образом, существующие методы, будучи скомбинированы с более мощными возможностями рефлексии, позволяют добиться замечательных успехов в метапрограммировании.

23.3. Стоимость рекурсивного инстанцирования

Проанализируем шаблон `Sqrt<>`, представленный в разделе 23.2. Первичный шаблон представляет собой общее рекурсивное вычисление, вызываемое с параметром шаблона `N` (значение, для которого вычисляется квадратный корень) и двумя другими необязательными параметрами. Эти необязательные параметры представляют собой минимальное и максимальное значения, которые может

⁵ Некоторые из свойств, предоставляемых стандартной библиотекой C++, уже полагаются на определенное сотрудничество с компилятором (с использованием нестандартных “встроенных” операторов) — см. раздел 19.10.

иметь результат. Если шаблон вызывается только с одним аргументом, мы знаем, что квадратный корень лежит между 1 и самим значением N.

Затем выполняется рекурсия с использованием метода *бинарного поиска* (часто именуемого *методом бисекции*). Внутри шаблона мы выясняем, находится ли значение в первой или второй половине диапазона между LO и HI. Это делается с помощью условного оператора ?: . Если значение mid² больше, чем N, мы продолжаем поиск в первой половине. Если же значение mid² меньше или равно N, мы используем тот же шаблон для второй половины.

Частичная специализация завершает рекурсивный процесс, когда HI и LO имеют одинаковое значение M, которое и является нашим окончательным значением квадратного корня value.

Инстанцирование шаблона — процесс не из дешевых: даже сравнительно скромные шаблоны класса могут потребовать более килобайта памяти для каждого инстанцирования, и эта память не может быть освобождена до завершения компиляции. Рассмотрим детали простой программы, которая использует наш шаблон Sqrt:

meta/sqrt1.cpp

```
#include <iostream>
#include "sqrt1.hpp"

int main()
{
    std::cout << "Sqrt<16>::value = " << Sqrt<16>::value << '\n';
    std::cout << "Sqrt<25>::value = " << Sqrt<25>::value << '\n';
    std::cout << "Sqrt<42>::value = " << Sqrt<42>::value << '\n';
    std::cout << "Sqrt<1>::value = " << Sqrt<1>::value << '\n';
}
```

Выражение

Sqrt<16>::value

раскрывается в

Sqrt<16,1,16>::value

Внутри шаблона метапрограмма вычисляет Sqrt<16,1,16>::value следующим образом:

```
mid = (1+16+1)/2
      = 9
value = (16<9*9) ? Sqrt<16,1,8>::value
                  : Sqrt<16,9,16>::value
      = (16<81) ? Sqrt<16,1,8>::value
                  : Sqrt<16,9,16>::value
      = Sqrt<16,1,8>::value
```

Таким образом, результат вычисляется как значение Sqrt<16,1,8>::value, которое раскрывается следующим образом:

```

mid = (1+8+1)/2
      = 5
value = (16<5*5) ? Sqrt<16,1,4>::value
                  : Sqrt<16,5,8>::value
= (16<25) ? Sqrt<16,1,4>::value
            : Sqrt<16,5,8>::value
= Sqrt<16,1,4>::value

```

Аналогично `Sqrt<16,1,4>::value` раскрывается следующим образом:

```

mid = (1+4+1)/2
      = 3
value = (16<3*3) ? Sqrt<16,1,2>::value
                  : Sqrt<16,3,4>::value
= (16<9) ? Sqrt<16,1,2>::value
          : Sqrt<16,3,4>::value
= Sqrt<16,3,4>::value

```

И, наконец, `Sqrt<16,3,4>::value` приводит к следующему:

```

mid = (3+4+1)/2
      = 4
value = (16<4*4) ? Sqrt<16,3,3>::value
                  : Sqrt<16,4,4>::value
= (16<16) ? Sqrt<16,3,3>::value
          : Sqrt<16,4,4>::value
= Sqrt<16,4,4>::value

```

Инициализация `Sqrt<16,4,4>::value` завершает рекурсивный процесс из-за соответствия явной специализации для равных значений нижней и верхней границ. Таким образом, окончательный результат оказывается равен

```
value = 4
```

23.3.1. Отслеживание всех инстанцирований

Наш анализ, приведенный выше, следует за всеми значительными инстанцированиями, которые вычисляют квадратный корень из 16. Однако, когда компилятор вычисляет выражение

```
(16 <= 8 * 8) ? Sqrt<16, 1, 8>::value
                  : Sqrt<16, 9, 16>::value
```

он инстанцирует шаблоны не только в положительной ветви, но и в ветви отрицательной (`Sqrt<16, 9, 16>`). Кроме того, поскольку код пытается обратиться к члену результирующего типа класса с использованием оператора `::`, инстанцируются также все члены внутри этого типа класса. Это означает, что полное инстанцирование `Sqrt<16, 9, 16>` приводит к полному инстанцированию `Sqrt<16, 9, 12>` и `Sqrt<16, 13, 16>`. Если подробно рассмотреть весь процесс, то мы обнаружим, что в конечном итоге будут выполнены десятки инстанцирований. Общее их количество почти вдвое превышает значение N.

К счастью, есть способы уменьшить этот взрывной рост количества инстанцирований. Чтобы проиллюстрировать один такой важный метод, мы перепишем нашу метапрограмму `Sqrt` следующим образом:

meta/sqrt2.hpp

```
#include "ifthenelse.hpp"

// Первичный шаблон для выполнения рекурсии
template<int N, int LO = 1, int HI = N>
struct Sqrt
{
    // Вычисление округленного вверх значения средней точки
    static constexpr auto mid = (LO + HI + 1) / 2;
    // Поиск не слишком большого значения в половине интервала
    using SubT = IfThenElse < (N < mid * mid),
        Sqrt < N, LO, mid - 1 >,
        Sqrt < N, mid, HI >>;
    static constexpr auto value = SubT::value;
};

// Частичная специализация для завершения рекурсии
template<int N, int S>
struct Sqrt<N, S, S>
{
    static constexpr auto value = S;
};
```

Ключевым изменением здесь является использование шаблона `IfThenElse`, который был представлен в разделе 19.7.1. Вспомните, что шаблон `IfThenElse` представляет собой механизм, который на основании данной логической константы выбирает один из двух типов. Если константа имеет значение `true`, в качестве псевдонима типа выбирается первый тип; в противном случае — второй. На данном этапе важно помнить, что определение псевдонима типа для экземпляра шаблона класса не приводит к инстанцированию тела этого экземпляра компилятором C++. Поэтому, когда мы пишем

```
using SubT = IfThenElse < (N < mid * mid),
    Sqrt < N, LO, mid - 1 >,
    Sqrt<N, mid, HI >>;
```

то не инстанцируются ни `Sqrt<N, LO, mid-1>`, ни `Sqrt<N, mid, HI>`. Какой бы из этих двух типов ни стал в конечном итоге синонимом `SubT`, он полностью инстанцируется только при поиске `SubT::value`. В отличие от нашего первого подхода эта стратегия приводит к количеству инстанцирований, пропорциональному $\log N$: весьма значительное снижение стоимости метапрограммирования при умеренно больших N .

23.4. Вычислительная полнота

Наш пример `Sqrt<>` демонстрирует, что метапрограммы могут содержать:

- переменные состояния: параметры шаблонов;
- конструкции циклов: посредством рекурсии;

- выбор пути выполнения: с помощью условных выражений или специализаций;
- целочисленную арифметику.

Если нет никаких ограничений на количество рекурсивных инстанцирований и количество разрешенных переменных состояния, можно показать, что это достаточно, чтобы вычислить все, что является вычислимым. Однако делать это с помощью шаблонов может быть не слишком удобно. Кроме того, поскольку экземпляр шаблона требует существенного количества ресурсов компилятора, обширные рекурсивные инстанцирования быстро замедлят работу компилятора или даже полностью истощат его ресурсы. Стандарт C++ рекомендует (но не предписывает) разрешить как минимум 1024 уровня рекурсивного инстанцирования, что достаточно для большинства (но, конечно, не для всех) задач шаблонного метапрограммирования.

Таким образом, на практике шаблонные метапрограммы следует использовать с осторожностью. Однако есть несколько ситуаций, когда они незаменимы как инструмент для реализации удобных шаблонов. В частности, иногда они могут быть скрыты внутри обычных, несложных шаблонов, “выжимая” максимальную производительность из реализаций критических алгоритмов.

23.5. Рекурсивное инстанцирование и рекурсивные аргументы шаблонов

Рассмотрим следующий рекурсивный шаблон:

```
template<typename T, typename U>
struct Doublify
{
};

template<int N>
struct Trouble
{
    using LongType = Doublify<typename Trouble<N-1>::LongType,
                                typename Trouble<N-1>::LongType>;
};

template<>
struct Trouble<0>
{
    using LongType = double;
};
Trouble<10>::LongType ouch;
```

Использование `Trouble<10>::LongType` не только запускает рекурсивное инстанцирование `Trouble<9>, Trouble<8>, ..., Trouble<0>`, но и инстанцирует `Doublify` для всех более сложных типов. В табл. 23.1 проиллюстрировано, насколько быстро растет эта сложность.

Таблица 23.1. Рост Trouble<N>::LongType

Псевдоним типа	Базовый тип
Trouble<0>::LongType	double
Trouble<1>::LongType	Doublify<double,double>
Trouble<2>::LongType	Doublify<Doublify<double,double>, Doublify<double,double>>
Trouble<3>::LongType	Doublify<Doublify<Doublify<double,double>, Doublify<double,double>>, <Doublify<double,double>, Doublify<double,double>>>

Как показано в табл. 23.1, сложность описания типа выражения Trouble<N>::LongType с ростом N растет экспоненциально. В общем случае такая ситуация нагружает компилятор C++ даже больше, чем выполнение рекурсивных инстанцирований, которые не предполагают рекурсивных аргументов шаблона. Одной из проблем является то, что компилятор сохраняет представление декорированного (mangled) имени типа. Это декорированное имя некоторым способом кодирует точную специализацию шаблона, и в ранних реализациях C++ использовалось декорирование, примерно пропорциональное длине идентификатора шаблона. Эти компиляторы используют для Trouble<10>::LongType свыше 10000 символов.

Новейшие реализации C++ принимают во внимание тот факт, что вложенные идентификаторы шаблонов довольно распространены в современных программах на C++ и используют интеллектуальные методы декорирования, чтобы значительно уменьшить рост длин декорированных имен (например, всего несколько сотен символов для Trouble<10>::LongType). Эти новые компиляторы также позволяют избегать возникновения декорированных имен, если они на самом деле не нужны для генерации низкоуровневого кода экземпляра шаблона. Тем не менее при прочих равных условиях, вероятно, предпочтительнее организовывать рекурсивные инстанцирования таким образом, чтобы аргументы шаблона не оказывались рекурсивно вложенными.

23.6. Значения перечислений и статические константы

Во времена, когда шаблоны только начинали активно применяться в C++, значения перечислений были единственным механизмом для создания “истинных констант” (так называемых *константных выражений*) в виде именованных членов в объявлениях классов. С их использованием мы могли бы, например, определить метапрограмму Pow3 для вычисления степеней числа 3 следующим образом:

meta/pow3epmt.hpp

```
// Первичный шаблон для вычисления 3 в N-й степени
template<int N>
struct Pow3
```

```
{
    enum { value = 3 * Pow3 < N - 1 >::value };
};

// Полная специализация для завершения рекурсии
template<>
struct Pow3<0>
{
    enum { value = 1 };
};
```

Стандарт C++98 ввел концепцию статических константных инициализаторов, с использованием которой наша метапрограмма Pow3 может выглядеть так:

meta/pow3const.hpp

```
// Первичный шаблон для вычисления 3 в N-й степени
template<int N>
struct Pow3
{
    static int const value = 3 * Pow3 < N - 1 >::value;
};

// Полная специализация для завершения рекурсии
template<>
struct Pow3<0>
{
    static int const value = 1;
};
```

Однако у данной версии имеется недостаток: статические константные члены являются l-значениями (см. приложение Б, “Категории значений”). Так что, если мы имеем такое объявление функции, как

`void foo(int const&);`

и передаем ей результат метапрограммы:

`foo(Pow3<7>::value);`

то компилятор должен передать *адрес* Pow3<7>::value, что заставляет компилятор инстанцировать и выделять память для определения статического члена. В результате вычисления при использовании статических константных членов не ограничиваются только вычислением “времени компиляции”.

Значения перечислений не являются l-значениями (т.е. они не имеют адреса). Поэтому, когда мы передаем их по ссылке, статическая память не используется. Это почти то же самое, как если бы вы передали вычисленное значение как литерал. Поэтому в первом издании этой книги для такого рода приложений мы предпочтитали использовать константы перечислений.

Однако в C++11 введены `constexpr`-статические члены-данные, и они не ограничиваются целыми типами. Они не решают упомянутый выше вопрос с адресом, но, несмотря на этот недостаток, в настоящее время они представляют

собой распространенный способ получения результатов метапрограмм. Они имеют преимущество использования корректного типа (в отличие от искусственного типа перечисления), и этот тип может быть выведен при объявлении статического члена со спецификатором типа `auto`. C++17 добавил встраиваемые (`inline`) статические члены данных, которые решают вопрос с адресом и могут быть использованы в сочетании с `constexpr`.

23.7. Заключительные замечания

Автором первой опубликованной метапрограммы был Эрвин Анрух, представлявший в Комитете по стандартизации C++ компанию Siemens. Он заметил, что процесс инстанцирования шаблонов обладает вычислительной полнотой, и продемонстрировал это положение, разработав свою первую метапрограмму. При этом был использован компилятор Metaware, выдающий в процессе трансляции данной программы сообщения об ошибках, номера которых образуют последовательность простых чисел. Ниже приведен код упомянутой программы, обсуждавшийся на заседании Комитета по стандартизации C++ в 1994 году (модифицированный для работы со стандартными современными компиляторами)⁶.

meta/unruh.cpp

```
// Вычисление простых чисел
// (модификация исходного кода Эрвина Анруха 1994 года)
template<int p, int i>
struct is_prime
{
    enum { pri = (p==2) || ((p%i)&&is_prime<(i>2?p:0),i-1>::pri) };
};

template<>
struct is_prime<0, 0>
{
    enum {pri = 1};
};

template<>
struct is_prime<0, 1>
{
    enum {pri = 1};
};

template<int i>
struct D
{
    D(void*);
};

template<int i>
struct CondNull
```

⁶ Выражаем благодарность Эрвину Анруху за предоставление этого кода для данной книги. Оригинальный вариант кода можно найти в [70].

```

{
    static int const value = i;
};

template<>
struct CondNull<0>
{
    static void* value;
};
void* CondNull<0>::value = 0;

template<int i>
struct Prime_print // Первичный шаблон - цикл вывода простых чисел
{
    Prime_print < i - 1 > a;
    enum { pri = is_prime < i, i - 1 >::pri };
    void f()
    {
        D<i> d = CondNull<pri?1:0>::value; // 1 - ошибка, 0 - OK
        a.f();
    }
};
template<>
struct Prime_print<1> // Полная специализация для завершения цикла
{
    enum {pri = 0};
    void f()
    {
        D<1> d = 0;
    }
};

#ifndef LAST
#define LAST 18
#endif

int main()
{
    Prime_print<LAST> a;
    a.f();
}

```

В процессе компиляции этой программы компилятор выводит сообщения об ошибках, если в функции `Prime_print::f()` не удается инициализировать переменную `d`. Это происходит, когда начальное значение равно 1, поскольку конструктор определен только для типа `void*`, а этот тип корректно преобразуется лишь значение 0. Например, один из компиляторов среди других сообщений выдает такие⁷:

```
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<17>'
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<13>'
```

⁷Обработка ошибок в разных компиляторах может быть разной; некоторые компиляторы прекращают работу после первого же сообщения.

```
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<1>'  
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<7>'  
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<5>'  
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<3>'  
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<2>'
```

Концепция шаблонного метапрограммирования на C++ стала популярным инструментом программистов благодаря статье Тодда Вельдхаузена (Todd Veldhuizen) *Using C++ Template Metaprograms* (Использование шаблонных метапрограмм в C++) [73]. Работа Тодда над библиотекой Blitz++ (библиотека численных функций C++, выполняющих действия с массивами [9]) также внесла многие усовершенствования и дополнения в метапрограммирование (в частности, в методы применения шаблонов выражений, которые рассматриваются в главе 27, “Шаблоны выражений”).

Первое издание этой книги и книги Андрея Александреску (Andrei Alexandrescu) *Современное проектирование на C++* [3] способствовали взрывному росту количества библиотек C++, использующих шаблонное метапрограммирование с применением ряда методов, которые используются и сегодня. Проект Boost [10] сыграл важную роль в наведении порядка в последствиях этого взрыва. На раннем этапе он представил MPL (библиотеку метапрограммирования), которая заложила основы для дальнейшего развития метапрограммирования типов [16]. Свою роль в его популяризации сыграла книга Абрахамса (Abrahams) и Гуртового (Gurtovoy) *C++ Template Metaprogramming* [1].

Дополнительный важный вклад в большую синтаксическую доступность метапрограммирования был сделан Луи Дионном (Louis Dionne), особенно в его библиотеке Boost.Hana [14]. Луи вместе с Эндрю Саттоном (Andrew Sutton), Гербом Саттером (Herb Sutter), Дэвидом Вандевурдом и другими теперь возглавляет усилия Комитета по стандартизации по поддержке метапрограммирования в языке C++. Важной основой для этой работы стало выяснение, какие свойства программ должны быть доступны через рефлексию; здесь следует отметить особый вклад Матуша Хохлика (Matúš Chochlík), Алекса Науманна (Axel Naumann) и Дэвида Санкеля (David Sankel).

В [7] Джон Бартон (John J. Barton) и Ли Нэкман (Lee R. Nackman) показали, как отслеживать единицы размерностей при выполнении вычислений. Библиотека *SUnits*, разработанная Уолтером Брауном (Walter Brown) [24], стала всеобъемлющей библиотекой для работы с физическими единицами измерений. Компонент стандартной библиотеки std::chrono, который мы использовали в качестве мотивации в разделе 23.1.4, был разработан Говардом Хиннантом (Howard Hinnant).

Глава 24

Списки типов

Эффективное программирование обычно требует использования различных структур данных, и метaprogramмирование в этом смысле ничем не отличается. Для метапрограммирования типов центральной структурой данных является *список типов* (typelist), который, как следует из его имени, представляет собой список, содержащий типы. Шаблонные метапрограммы могут работать с этими списками типов, в конечном итоге производя часть выполнимой программы. В этой главе мы обсуждаем методы работы со списками типов. Поскольку большинство операций, связанных со списками типов, используются в шаблонном метапрограммировании, мы рекомендуем вам сначала познакомиться с метапрограммированием, описанным в главе 23, “Метапрограммирование”.

24.1. Анатомия списков типов

Список типов – это тип, который содержит типы и с которым может работать шаблонная метапрограмма. Он обеспечивает операции, обычно предоставляемые списками: перебор элементов (типов), содержащихся в списке, добавление или удаление элементов. Однако списки типов отличаются от большинства схожих структур данных времени выполнения, таких как `std::list`, тем, что они не допускают изменения. Например, добавление элемента к `std::list` изменяет сам список, и эти изменения можно наблюдать в любой другой части программы, которая имеет доступ к данному списку. Добавление же элемента в список типов не изменяет исходный тип: вместо этого добавление элемента в существующий список создает новый тип, не изменяя оригинал. Читатели, знакомые с языками функционального программирования, такими как Scheme, ML и Haskell, скорее всего, увидят параллели между работой со списками типов в C++ и списками на этих языках.

Список типов обычно реализуется как специализация шаблона класса, который кодирует содержимое списка типов (т.е. содержащиеся в нем типы и их порядок) в аргументах его шаблона. Непосредственная реализация списка типов кодирует элементы в пакет параметров:

`typelist/typelist.hpp`

```
template<typename... Elements>
class Typelist
{
};
```

Элементы `Typelist` записываются непосредственно как его аргументы шаблона. Пустой список типов записывается как `Typelist<>`, список, содержащий только `int`, записывается как `Typelist<int>`, и так далее. Вот список типов, содержащий знаковые целочисленные типы:

```
using SignedIntegralTypes =
    Typelist<signed char, short, int, long, long long>;
```

Работа с таким списком типов обычно требует разделения списка на части, как правило, путем отделения первого элемента списка (головы списка) от остальных элементов в списке (хвоста). Например, метафункция `Front` извлекает первый элемент из списка типов:

`typelist/typelistfront.hpp`

```
template<typename List>
class FrontT;

template<typename Head, typename... Tail>
class FrontT<Typelist<Head, Tail...>>
{
public:
    using Type = Head;
};

template<typename List>
using Front = typename FrontT<List>::Type;
```

Таким образом, `FrontT<SignedIntegralTypes>::Type` (более лаконично записываемый как `Front<SignedIntegralTypes>`) производит `signed char`. Аналогично, метафункция `PopFront` удаляет первый элемент из списка типов. Ее реализация делит элементы списка типов на голову и хвост, а затем формирует новую специализацию `Typelist` из элементов хвоста.

`typelist/typelistpopfront.hpp`

```
template<typename List>
class PopFrontT;

template<typename Head, typename... Tail>
class PopFrontT<Typelist<Head, Tail...>> {
public:
    using Type = Typelist<Tail...>;
};

template<typename List>
using PopFront = typename PopFrontT<List>::Type;
```

`PopFront<SignedIntegralTypes>` производит список типов:

`Typelist<short, int, long, long long>`

Можно также вставлять элементы в начало списка типов путем захвата всех существующих элементов в пакет параметров шаблона с последующим созданием новой специализации `Typelist`, содержащей все элементы:

`typelist/typelistpushfront.hpp`

```
template<typename List, typename NewElement>
class PushFrontT;
```

```

template<typename... Elements, typename NewElement>
class PushFrontT<Typelist<Elements...>, NewElement>
{
public:
    using Type = Typelist<NewElement, Elements...>;
};

template<typename List, typename NewElement>
using PushFront = typename PushFrontT<List, NewElement>::Type;

```

Как можно ожидать,

```
PushFront<SignedIntegralTypes, bool>
```

дает

```
Typelist<bool, signed char, short, int, long, long long>
```

24.2. Алгоритмы над списками типов

Фундаментальные операции над списками типов Front, PopFront и PushFront могут быть объединены для более интересных действий со списками типов. Например, можно заменить первый элемент списка типа путем применения PushFront к результату PopFront:

```
using Type = PushFront<PopFront<SignedIntegralTypes>, bool>;
// Эквивалентно Typelist<bool, short, int, long, long long>
```

Идя дальше, мы можем реализовать алгоритмы — поиск, преобразования, обращения — как шаблонные метафункции, работающие со списками типов.

24.2.1. Индексация

Одна из наиболее фундаментальных операций над списком типов должна извлекать определенный элемент из списка. В разделе 24.1 показано, как реализовать операцию, которая извлекает первый элемент. Здесь мы обобщим эту операцию для извлечения N-го элемента. Например, чтобы извлечь тип с индексом 2 из данного списка типов, можно написать:

```
using TL = NthElement<Typelist<short, int, long>, 2>;
```

что делает TL псевдонимом для long. Операция NthElement реализуется рекурсивной метапрограммой, которая проходит по списку до тех пор, пока не найдет запрашиваемый элемент:

typelist/nthelement.hpp

```

// Рекурсивный случай:
template<typename List, unsigned N>
class NthElementT : public NthElementT < PopFront<List>, N-1 >
{
};

```

```
// Базовый случай:
template<typename List>
class NthElementT<List, 0> : public FrontT<List>
{
};

template<typename List, unsigned N>
using NthElement = typename NthElementT<List, N>::Type;
```

Сначала рассмотрим базовый случай, обрабатываемый частичной специализацией, для случая, когда N равно 0. Данная специализация завершает рекурсию, предоставляя элемент из начала списка. Это делается путем открытого (public) наследования от `FrontT<List>`, который (косвенно) обеспечивает псевдоним типа `Type`, представляющий первый элемент этого списка, а следовательно, и результат метафункции `NthElement`, с использованием метафункциональной передачи (рассматривается в разделе 19.3.2).

Рекурсивный случай, который является также определением первичного шаблона, проходит по списку. Поскольку частичная специализация гарантирует, что $N > 0$, в рекурсивном случае из списка удаляется первый элемент и запрашивается $(N-1)$ -й элемент из оставшихся в списке. В нашем примере

`NthElementT<Typelist<short, int, long>, 2>`

наследуется от

`NthElementT<Typelist<int, long>, 1>`

который наследуется от

`NthElementT<Typelist<long>, 0>`

Здесь мы достигаем базового случая, и наследование от `FrontT<Typelist<long>>` предоставляет окончательный результат через вложенный тип `Type`.

24.2.2. Поиск наилучшего соответствия

Многие алгоритмы над списками типов выполняют поиск данных в этих списках. Например, может потребоваться найти наибольший из содержащихся в списке типов (скажем, для выделения достаточного количества памяти для любого из имеющихся типов). Такой поиск также может быть выполнен с помощью рекурсивной шаблонной метапрограммы:

`typelist/largesttype.hpp`

```
template<typename List>
class LargestTypeT;

// Рекурсивный случай:
template<typename List>
class LargestTypeT
{
private:
    using First = Front<List>;
    using Rest = typename LargestTypeT<PopFront<List>>::Type;
```

```

public:
    using Type = IfThenElse<(sizeof(First)>=sizeof(Rest)),First,Rest>;
};

// Базовый случай:
template<>
class LargestTypeT<Typelist<>>
{
public:
    using Type = char;
};

template<typename List>
using LargestType = typename LargestTypeT<List>::Type;

```

Алгоритм `LargestType` вернет первый из наибольших типов в списке. Например, для списка `Typelist<bool,int,long,short>` этот алгоритм вернет первый тип, имеющий тот же размер, что и тип `long`, — которым может быть `int` или `long`, в зависимости от вашей платформы¹.

Первичный шаблон для `LargestTypeT` использует распространенную *идиому первый/остальные*, которая состоит из трех этапов. На первом этапе вычисляется частичный результат на основе первого элемента, который в данном случае является передним элементом списка, и помещается в первую `First`. Далее выполняется рекурсия для вычисления результата для остальных элементов в списке, и этот результат помещается в `Rest`. Например, для списка типов `Typelist<bool,int,long,short>` на первом этапе рекурсии `First` представляет собой `bool`, в то время как `Rest` является результатом применения алгоритма к `Typelist<int,long,short>`. Наконец, третий этап объединяет результаты `First` и `Rest` для получения решения. Здесь `IfThenElse` выбирает больший элемент из первого элемента в списке `First`, или лучшего из кандидатов в `Rest`, и возвращает победителя². Оператор `>=` разрешает неопределенность равных элементов в пользу элемента, который в исходном списке находится раньше.

Рекурсия прекращается, когда список пуст. По умолчанию мы используем `char` как тип-страж для инициализации алгоритма, потому что любой тип имеет размер не меньше, чем у `char`.

Обратите внимание на то, что в базовом случае явно упоминается пустой список типов `Typelist<>`. Это не совсем неудачно, так как мешает применению других форм списков типов, к которым мы вернемся в последующих разделах (включая разделы 24.3, 24.5 и главу 25, “Кортежи”). Для решения этой проблемы мы вводим метафункцию `IsEmpty`, которая определяет, не является ли данный список типов пустым:

¹Существуют даже такие экзотические платформы, где тип `bool` имеет тот же размер, что и тип `long`!

²Обратите внимание на то, что список типов может содержать типы, к которым оператор `sizeof` неприменим, такие как `void`. В этом случае при попытке вычислить наибольший тип в списке компилятор выдаст сообщение об ошибке.

typelist/typelistisempty.hpp

```
template<typename List>
class IsEmpty
{
public:
    static constexpr bool value = false;
};

template<>
class IsEmpty<Typelist<>>
{
public:
    static constexpr bool value = true;
};
```

Используя `IsEmpty`, мы можем реализовать `LargestType` так, что он будет однаково хорошо работать для любого списка типов, который реализует `Front`, `PopFront` и `IsEmpty`, как показано ниже:

typelist/genericlargesttype.hpp

```
template<typename List, bool Empty = IsEmpty<List>::value>
class LargestTypeT;

// Рекурсивный случай:
template<typename List>
class LargestTypeT<List, false>
{
private:
    using Contender = Front<List>;
    using Best = typename LargestTypeT<PopFront<List>>::Type;
public:
    using Type = IfThenElse<(sizeof(Contender)>=sizeof(Best)),
                           Contender, Best>;
};

// Базовый случай:
template<typename List>
class LargestTypeT<List, true>
{
public:
    using Type = char;
};

template<typename List>
using LargestType = typename LargestTypeT<List>::Type;
```

Имеющий значение по умолчанию второй параметр шаблона `LargestTypeT`, `Empty`, проверяет, пуст ли список. Если нет, в рекурсивном случае (у которого этот аргумент равен `false`) продолжается исследование списка. В противном случае базовый случай (для которого этот аргумент имеет значение `true`) прекращает рекурсию и обеспечивает исходный результат (`char`).

24.2.3. Добавление в список типов

Примитивная операция `PushFront` позволяет нам добавить новый элемент в начало списка типов, производя новый список. Предположим, что вместо этого мы хотим добавить новый элемент в конец списка, как мы часто поступаем с такими контейнерами, как `std::list` и `std::vector`, во время выполнения, чтобы получить операцию `PushBack`. Для нашего шаблона `Typelist` требуется только небольшая модификация реализации `PushFront` из раздела 24.1:

`typelist/typelistpushback.hpp`

```
template<typename List, typename NewElement>
class PushBackT;

template<typename... Elements, typename NewElement>
class PushBackT<Typelist<Elements..., NewElement>>
{
public:
    using Type = Typelist<Elements..., NewElement>;
};

template<typename List, typename NewElement>
using PushBack = typename PushBackT<List, NewElement>::Type;
```

Однако, как и в случае алгоритма `LargestType`, мы можем реализовать общий алгоритм `PushBack`, который использует только примитивные операции `Front`, `PushFront`, `PopFront` и `IsEmpty`³:

`typelist/genericpushback.hpp`

```
template<typename List, typename NewElement, bool = IsEmpty<List>::value>
class PushBackRecT;

// Рекурсивный случай:
template<typename List, typename NewElement>
class PushBackRecT<List, NewElement, false>
{
using Head = Front<List>;
using Tail = PopFront<List>;
using NewTail = typename PushBackRecT<Tail, NewElement>::Type;
public:
    using Type = PushFront<Head, NewTail>;
};

// Базовый случай:
template<typename List, typename NewElement>
class PushBackRecT<List, NewElement, true>
{
public:
    using Type = PushFront<List, NewElement>;
};
```

³ При экспериментах с этой версией алгоритма обратите внимание на то, что будет нужно удалить частичную специализацию `PushBack` для `Typelist`, иначе она будет использоваться вместо универсальной версии.

```
// Обобщенная операция PushBack:
template<typename List, typename NewElement>
class PushBackT : public PushBackRect<List, NewElement> { };

template<typename List, typename NewElement>
using PushBack = typename PushBackT<List, NewElement>::Type;
```

Шаблон PushBackRect управляет рекурсией. В базовом случае мы используем PushFront для добавления NewElement в пустой список, поскольку для пустого списка PushFront эквивалентен PushBack. Рекурсивный случай гораздо более интересен: он разбивает список на первый элемент (Head) и список типов, содержащий остальные элементы (Tail). Затем новый элемент добавляется к Tail, рекурсивно производя NewTail. Затем мы снова используем PushFront, чтобы добавить Head в начало списка NewTail для формирования окончательного списка.

Давайте раскроем рекурсию для простого примера:

```
PushBackRect<Typelist<short, int>, long>
```

На первом этапе Head представляет собой short, а Tail – Typelist<int>. Мы рекурсивно переходим к

```
PushBackRect<Typelist<int>, long>
```

где Head представляет собой int, а Tail – Typelist<>.

Еще один шаг рекурсии состоит в вычислении

```
PushBackRect<Typelist<>, long>
```

что представляет собой базовый случай, и возвращает PushFront<Typelist<>, long>. Эта конструкция вычисляется как Typelist<long>. Затем рекурсия сворачивается, внося предыдущий Head в начало списка:

```
PushFront<int, Typelist<long>>
```

Так получается Typelist<int, long>. Дальнейшее сворачивание рекурсии приводит к внесению первого (внешнего) Head (short) в список:

```
PushFront<short, Typelist<int, long>>
```

Так получается окончательный результат:

```
Typelist<short, int, long>
```

Обобщенная реализация PushBackRect работает со списками типов любых разновидностей подобно предыдущим алгоритмам, разработанным в этом разделе. Для вычисления ей требуется линейное количество инстанцирований шаблонов, поскольку для списка типов длины N будет выполняться N+1 инстанцирований PushBackRect и PushFrontT, а также N инстанцирований FrontT и PopFrontT. Подсчет количества инстанцирований шаблонов может дать грубую оценку времени, которое потребуется для *компиляции* конкретной метапрограммы, потому что инстанцирование шаблона само по себе является довольно сложным процессом для компилятора.

Время компиляции может оказаться проблемой для больших шаблонных метапрограмм, поэтому имеет смысл попытаться уменьшить количество инстанцирований шаблонов, выполняемых этими алгоритмами⁴. В действительности наша первая реализация PushBack с использованием частичной специализации Typelist требует только константного количества инстанцирований шаблонов, что делает его гораздо более эффективным (во время компиляции), чем обобщенная версия. Кроме того, будучи описанной как частичная специализация PushBackT, эта эффективная реализация будет автоматически выбрана при применении PushBack к экземпляру Typelist, перенося понятие специализации алгоритма (описанного в разделе 20.1) на шаблонные метапрограммы. Многие из способов, представленных в этом разделе, могут быть применены к шаблонным метапрограммам для уменьшения количества инстанцирований шаблонов, выполняемых алгоритмом.

24.2.4. Обращение порядка типов в списке

Когда списки типов содержат элементы в некотором порядке, удобно иметь возможность изменить порядок элементов в списке с помощью некоторого алгоритма. Например, список SignedIntegralTypes, представленный в разделе 24.1, содержит типы в порядке увеличения их целочисленного ранга. Однако может быть более полезным обратить этот список для получения списка Typelist<long long, long, int, short, signed char>. Алгоритм Reverse реализуется следующей мetaфункцией:

typelist/typelistreverse.hpp

```
template<typename List, bool Empty = IsEmpty<List>::value>
class ReverseT;

template<typename List>
using Reverse = typename ReverseT<List>::Type;

// Рекурсивный случай:
template<typename List>
class ReverseT<List, false>
    : public PushBackT<Reverse<PopFront<List>>, Front<List>> { };

// Базовый случай:
template<typename List>
class ReverseT<List, true>
{
public:
    using Type = List;
};
```

Базовый случай для рекурсии в этой мetaфункции представляет собой тождественную функцию для пустого списка. Рекурсивный случай разделяет список на его первый элемент и оставшиеся элементы в списке. Например, для типа

⁴ В книге Абрахамса и Гуртового [1] представлено гораздо более подробное обсуждение времени компиляции шаблонных метапрограмм, включая ряд методик его уменьшения. Здесь мы лишь кратко касаемся этого вопроса.

`TypeList<short, int, long>` рекурсивный шаг отделяет первый элемент (`short`) от остальных элементов (`TypeList<int, long>`). Затем выполняется рекурсивное обращение этого списка остальных элементов (что дает тип `TypeList<long, int>`) и, наконец, к обращенному списку добавляется первый элемент с помощью `PushBackT` (и мы получаем тип `TypeList<long, int, short>`).

Алгоритм `Reverse` делает для списков типов возможной реализацию операции `PopBackT`, которая удаляет последний элемент из списка:

`typelist/typelistpopback.hpp`

```
template<typename List>
class PopBackT
{
public:
    using Type = Reverse<PopFront<Reverse<List>>>;
};

template<typename List>
using PopBack = typename PopBackT<List>::Type;
```

Этот алгоритм обращает список, удаляет из обращенного списка первый элемент с использованием `PopFront`, а затем еще раз обращает порядок элементов в получившемся списке.

24.2.5. Преобразование списка типов

Наши предыдущие алгоритмы для работы со списками типов позволили нам извлекать из списка произвольные элементы, выполнять поиск в списке, создавать новые и изменять существующие списки. Однако нам еще предстоит выполнение любых операций с элементами внутри списка типов. Например, мы хотим, “преобразовать” все типы в списке определенным образом⁵, например, добавив к каждому типу квалификатор `const` с использованием метафункции `AddConst`:

`typelist/addconst.hpp`

```
template<typename T>
struct AddConstT
{
    using Type = T const;
};
template<typename T>
using AddConst = typename AddConstT<T>::Type;
```

С этой целью мы реализуем алгоритм `Transform`, который принимает список типов и метафункцию, и производит другой список типов, содержащий результат применения метафункции к каждому типу. Например, тип

`Transform<SignedIntegralTypes, AddConstT>`

⁵ В сообществе функциональных программистов такая операция обычно известна как *map*. Однако мы используем термин *преобразование* (*transform*) для лучшего согласования имен алгоритмов со стандартной библиотекой C++.

будет представлять собой список типов, содержащий `signed char const, short const, int const, long const и long long const`. Метафункция предоставляется как шаблонный параметр шаблона, который отображает входной тип на выходной. Алгоритм `Transform` сам по себе является, как и ожидалось, рекурсивным алгоритмом:

typelist/transform.hpp

```
template<typename List, template<typename T> class MetaFun,
         bool Empty = IsEmpty<List>::value>
class TransformT;

// Рекурсивный случай:
template<typename List, template<typename T> class MetaFun>
class TransformT<List, MetaFun, false>
    : public PushFrontT<typename TransformT<PopFront<List>,
                           MetaFun>::Type,
                           typename MetaFun<Front<List>>::Type>
{
};

// Базовый случай:
template<typename List, template<typename T> class MetaFun>
class TransformT<List, MetaFun, true>
{
public:
    using Type = List;
};

template<typename List, template<typename T> class MetaFun>
using Transform = typename TransformT<List, MetaFun>::Type;
```

Рекурсивный случай прост, хотя синтаксически и тяжеловесен. Результатом преобразования является преобразование первого элемента в списке (второй аргумент `PushFront`) и добавление его в начало последовательности, генерируемой рекурсивным преобразованием остальных элементов в списке типов (первый аргумент `PushFront`).

В разделе 24.4 показано, как можно получить более эффективную реализацию алгоритма `Transform`.

24.2.6. Накопление списков типов

`Transform` является полезным алгоритмом для преобразования каждого элемента в списке. Он часто используется в сочетании с алгоритмом `Accumulate`, который объединяет все элементы последовательности в единое результирующее значение⁶. Алгоритм `Accumulate` получает список типов `T` с элементами T_1, T_2, \dots, T_N , исходный тип `I` и метафункцию `F`, которая принимает два типа и возвращает

⁶ В сообществе функциональных программистов такая операция обычно известна как *reduce*. Однако мы используем термин *накопление* (*accumulate*) для лучшего согласования имен алгоритмов со стандартной библиотекой C++.

типа. Алгоритм возвращает $F(F(F(\dots F(I; T_1), T_2), \dots, T_{N-1}), T_N)$, где на i -м шаге накопления F применяется к результату предыдущих $i-1$ шагов и T_i .

В зависимости от списка типов, выбора F и первоначального типа, мы можем использовать `Accumulate` для получения целого ряда различных результатов. Например, если F выбирает больший из двух типов, `Accumulate` будет вести себя как алгоритм `LargestType`. С другой стороны, если F принимает список типов и тип и добавляет этот тип в конец списка типов, `Accumulate` будет вести себя как алгоритм `Reverse`.

Реализация алгоритма `Accumulate` следует нашим стандартным методам рекурсивного метaproграммирования:

`typelist/accumulate.hpp`

```
template<typename List,
         template<typename X, typename Y> class F,
         typename I,
         bool = IsEmpty<List>::value>
class AccumulateT;

// Рекурсивный случай:
template<typename List,
         template<typename X, typename Y> class F,
         typename I>
class AccumulateT<List, F, I, false>
: public AccumulateT<PopFront<List>, F,
  typename F<I, Front<List>>::Type>
{
};

// Базовый случай:
template<typename List,
         template<typename X, typename Y> class F,
         typename I>
class AccumulateT<List, F, I, true>
{
public:
    using Type = I;
};

template<typename List,
         template<typename X, typename Y> class F,
         typename I>
using Accumulate = typename AccumulateT<List, F, I>::Type;
```

Здесь исходный тип I также используется как аккумулятор, захватывая текущий результат. Таким образом, базовый случай возвращает этот результат, когда достигает конца списка типов⁷. В рекурсивном случае алгоритм применяет F к предыдущему результату (I) и первому элементу списка, передавая результат применения F как начальный тип для накопления оставшейся части списка.

⁷Это также гарантирует, что результатом накопления пустого списка будет начальное значение.

Имея `Accumulate`, можно обратить список типов, используя `PushFrontT` в качестве метафункции `F` и пустой список `TypeList<T>` в качестве начального типа `I`:

```
using Result =
    Accumulate<SignedIntegralTypes, PushFrontT, TypeList<>>;
// Создает TypeList<long long, long, int, short, signed char>
```

Реализация `LargestTypeAcc` версии `LargestType` на основе `Accumulator` требует несколько большего количества усилий, так как нам нужна метафункция, возвращающая больший из двух типов:

`typelist/largesttypeacc0.hpp`

```
template<typename T, typename U>
class LargerTypeT
    : public IfThenElseT<sizeof(T) >= sizeof(U), T, U >
{
};

template<typename Typelist>
class LargestTypeAccT
    : public AccumulateT<PopFront<Typelist>, LargerTypeT,
        Front<Typelist>>
{
};

template<typename Typelist>
using LargestTypeAcc = typename LargestTypeAccT<Typelist>::Type;
```

Обратите внимание та то, что эта формулировка `LargestType` требует непустой список типов, потому что первый элемент списка используется в качестве начального типа. Мы могли бы обрабатывать случай пустого списка явно, либо возвращая некоторый ограничивающий тип (`char` или `void`), либо делая сам алгоритм дружественным по отношению к SFINAE, как описано в разделе 19.4.4:

`typelist/largesttypeacc.hpp`

```
template<typename T, typename U>
class LargerTypeT
    : public IfThenElseT<sizeof(T) >= sizeof(U), T, U >
{
};

template<typename Typelist, bool = IsEmpty<Typelist>::value>
class LargestTypeAccT;
template<typename Typelist>
class LargestTypeAccT<Typelist, false>
    : public AccumulateT<PopFront<Typelist>, LargerTypeT,
        Front<Typelist>>
{
};
```

```

template<typename Typelist>
class LargestTypeAccT<Typelist, true>
{
};

template<typename Typelist>
using LargestTypeAcc = typename LargestTypeAccT<Typelist>::Type;

```

Accumulate является мощным алгоритмом для работы со списками типов, потому что он позволяет выразить множество различных операций, так что его можно считать одним из основополагающих алгоритмов для работы со списками типов.

24.2.7. Сортировка вставками

В качестве последнего алгоритма для работы со списками типов реализуем сортировку вставками. Как и в прочих алгоритмах, рекурсивный шаг разбивает список на его “голову” (первый элемент списка) и “хвост”, состоящий из остальных элементов. Затем “хвост” рекурсивно сортируется, а “голова” вставляется в корректную позицию в отсортированном списке. Оболочка этого алгоритма выражена как алгоритм для работы со списками типов:

typelist/insertionsort.hpp

```

template<typename List,
         template<typename T, typename U> class Compare,
         bool = IsEmpty<List>::value>
class InsertionSortT;

template<typename List,
         template<typename T, typename U> class Compare>
using InsertionSort = typename InsertionSortT<List, Compare>::Type;

// Рекурсивный случай (вставка первого элемента
// в отсортированный список):
template<typename List,
         template<typename T, typename U> class Compare>
class InsertionSortT<List, Compare, false>
: public InsertSortedT<InsertionSort<PopFront<List>, Compare>,
  Front<List>, Compare>
{
};

// Базовый случай (пустой список является отсортированным):
template<typename List,
         template<typename T, typename U> class Compare>
class InsertionSortT<List, Compare, true>
{
public:
    using Type = List;
};

```

Параметр Compare представляет собой компаратор, используемый для упорядочения элементов в списке типов. Он принимает два типа и вычисляет логическое

значение, возвращаемое через значение члена `value`. Базовый случай — пустой список типов — тривиален.

Ядром сортировки вставками является метафункция `InsertSortedT`, которая вставляет значение в отсортированный список в первой точке, где список после вставки останется отсортированным:

typelist/insertsorted.hpp

```
#include "identity.hpp"

template<typename List, typename Element,
         template<typename T, typename U> class Compare,
         bool = IsEmpty<List>::value>
class InsertSortedT;

// Рекурсивный случай:
template<typename List, typename Element,
         template<typename T, typename U> class Compare>
class InsertSortedT<List, Element, Compare, false>
{
    // Вычисление хвоста результирующего списка:
    using NewTail =
        typename IfThenElse<Compare<Element, Front<List>>::value,
        IdentityT<List>,
        InsertSortedT<PopFront<List>, Element, Compare>
    >::Type;
    // Вычисление головы результирующего списка:
    using NewHead = IfThenElse<Compare<Element, Front<List>>::value,
        Element,
        Front<List>>;
public:
    using Type = PushFront<NewTail, NewHead>;
};

// Базовый случай:
template<typename List, typename Element,
         template<typename T, typename U> class Compare>
class InsertSortedT<List, Element, Compare, true>
    : public PushFrontT<List, Element>
{
};

template<typename List, typename Element,
         template<typename T, typename U> class Compare>
using InsertSorted = typename
    InsertSortedT<List, Element, Compare>::Type;
```

И снова базовый случай тривиален, потому что список из одного элемента всегда отсортирован. Рекурсивный случай отличается от базового вставкой элемента на корректное место в списке. Если вставляемый элемент предшествует первому элементу в (уже отсортированном) списке, результатом является добавление элемента в список с помощью `PushFront`. В противном случае список разделяется на “голову” и “хвост”, выполняется рекурсивная вставка элемента в “хвост”, а затем “голова” добавляется к списку, получающемуся в результате вставки элемента в “хвост”.

Данная реализация включает в себя оптимизацию времени компиляции, позволяющую избежать инстанцирования типов, которые не будут использоваться, — методику, рассмотренную в разделе 19.7.1. Приведенная ниже реализация также является технически правильной:

```
template<typename List, typename Element,
         template<typename T, typename U> class Compare>
class InsertSortedT<List, Element, Compare, false>
    : public IfThenElseT<Compare<Element, Front<List>>::value,
                           PushFront<List, Element>,
                           PushFront<InsertSorted<PopFront<List>,
                                         Element, Compare>,
                           Front<List>>>
{
};
```

Однако такая формулировка рекурсивного случая чересчур неэффективна, поскольку вычисляет аргументы шаблона в обеих ветвях `IfThenElseT`, хотя используется будет только одна ветвь. В нашем случае `PushFront` в ветви *then*, как правило — довольно дешевая конструкция, но этого нельзя сказать о рекурсивном вызове `InsertSorted` в ветви *else*.

В оптимизированной реализации первый `IfThenElse` вычисляет хвост полученного списка, `NewTail`. Вторым и третьим аргументами `IfThenElse` являются две метафункции, которые будут вычислять результаты для своих ветвей. Второй аргумент (ветвь *then*) использует `IdentityT` (показана в разделе 19.7.1), чтобы оставить список неизмененным. Третий аргумент (ветвь *else*) использует `InsertSortedT` для вычисления результата вставки элемента в отсортированный список. На верхнем уровне будет инстанцирована только одна из метафункций `IdentityT` и `InsertSortedT`, так что будет выполнена только очень небольшая работа (в наихудшем случае — `PopFront`). Затем второй `IfThenElse` вычисляет голову результирующего списка; ветви выполняются немедленно, потому что они обе считаются дешевыми. Окончательный список строится из вычисленных `NewHead` и `NewTail`. Эта формулировка обладает тем же жалательным свойством, что количество инстанцирований, требующихся для вставки элемента в отсортированный список, пропорционально его позиции в результирующем списке. Это проявляется в виде свойства сортировки вставками более высокого уровня, состоящего в том, что количество инстанцирований для сортировки уже отсортированного списка линейно зависит от длины списка. (Сортировка вставками остается квадратичной в смысле количества инстанцирований для входных данных, отсортированных в обратном порядке.)

Приведенная ниже программа демонстрирует использование сортировки вставками для упорядочения списка типов на основании их размеров. Операция сравнения использует оператор `sizeof` и сравнивает результаты:

`typelist/insertionsorttest.hpp`

```
template<typename T, typename U>
struct SmallerThanT
```

```

    {
        static constexpr bool value = sizeof(T) < sizeof(U);
    };

void testInsertionSort()
{
    using Types = Typelist<int, char, short, double>;
    using ST = InsertionSort<Types, SmallerThanT>;
    std::cout <<
        std::is_same<ST, Typelist<char, short, int, double>>::value
        << '\n';
}

```

24.3. Списки нетиповых параметров

Списки типов обеспечивают возможность описания и работы с последовательностями типов с помощью богатого набора алгоритмов и операций. В некоторых случаях полезно также иметь возможность работать с последовательностями значений времени компиляции, таких как границы многомерных массивов или индексы другого списка типов.

Существует несколько способов получения списков значений времени компиляции. Один простой подход включает определение шаблона класса `CTValue`, который представляет значение определенного типа в списке типов⁸:

typelist/ctvalue.hpp

```

template<typename T, T Value>
struct CTValue
{
    static constexpr T value = Value;
};

```

Используя шаблон `CTValue`, можно записать список типов, содержащий несколько первых простых чисел, следующим образом:

```

using Primes = Typelist<CTValue<int, 2>, CTValue<int, 3>,
    CTValue<int, 5>, CTValue<int, 7>,
    CTValue<int, 11>>;

```

При таком представлении можно выполнять числовые вычисления над списком значений, например вычисление произведения этих чисел.

Начнем с шаблона `MultiplyT`, который принимает два значения времени компиляции одинакового типа и производит новое значение времени компиляции, представляющее собой произведение входных значений:

typelist/multiply.hpp

```

template<typename T, typename U>
struct Multiply;

```

⁸Стандартная библиотека определяет шаблон `std::integral_constant`, который представляет собой версию `CTValue` с большими функциональными возможностями.

```
template<typename T, T Value1, T Value2>
struct MultiplyT<CTValue<T, Value1>, CTValue<T, Value2>>
{
public:
    using Type = CTValue<T, Value1* Value2>;
};

template<typename T, typename U>
using Multiply = typename MultiplyT<T, U>::Type;
```

Теперь, используя `MultiplyT`, можно записать следующее выражение, которое дает произведение всех простых чисел из списка:

```
Accumulate<Primes, MultiplyT, CTValue<int, 1>>::value
```

К сожалению, это использование `TypeList` и `CTValue` довольно многословное, в особенности если все значения принадлежат к одному и тому же типу. Мы можем оптимизировать этот частный случай путем введения шаблона псевдонима `CTTypeList`, который предоставляет гомогенный список значений, т.е. `TypeList`, содержащий `CTValue`:

```
typelist/cttypelist.hpp
```

```
template<typename T, T... Values>
using CTTypelist = TypeList<CTValue<T, Values>...>;
```

Теперь можно написать эквивалентное (но гораздо более краткое) определение `Primes` с использованием `CTTypeList` следующим образом:

```
using Primes = CTTypelist<int, 2, 3, 5, 7, 11>;
```

Единственный недостаток этого подхода заключается в том, что шаблоны псевдонимов являются всего лишь псевдонимами, так что сообщения об ошибках могут в конечном итоге использовать имя базового типа списка `TypeList` типов `CTValueType`, и быть куда более подробными, чем хотелось бы. Для решения этой проблемы можно создать совершенно новый класс `ValueList`, аналогичный списку типов, но который хранит значения непосредственно:

```
typelist/valueList.hpp
```

```
template<typename T, T... Values>
struct ValueList
{
};

template<typename T, T... Values>
struct IsEmpty<ValueList<T, Values...>>
{
    static constexpr bool value = sizeof...(Values) == 0;
};

template<typename T, T Head, T... Tail>
struct FrontT<ValueList<T, Head, Tail...>>
```

```

{
    using Type = CTValue<T, Head>;
    static constexpr T value = Head;
};

template<typename T, T Head, T... Tail>
struct PopFrontT<Valuelist<T, Head, Tail...>>
{
    using Type = Valuelist<T, Tail...>;
};

template<typename T, T... Values, T New>
struct PushFrontT<Valuelist<T, Values...>, CTValue<T, New>>
{
    using Type = Valuelist<T, New, Values...>;
};

template<typename T, T... Values, T New>
struct PushBackT<Valuelist<T, Values...>, CTValue<T, New>>
{
    using Type = Valuelist<T, Values..., New>;
};

```

Предоставляя IsEmpty, FrontT, PopFrontT и PushFrontT, мы делаем Valuelist корректным списком типов, который может использоваться с алгоритмами, определенными в данной главе. PushBackT предоставляется как специализация алгоритма для снижения стоимости этой операции во время компиляции. Например, Valuelist можно использовать с алгоритмом InsertionSort, определенным ранее:

typelist/valuelisttest.hpp

```

template<typename T, typename U>
struct GreaterThanT;

template<typename T, T First, T Second>
struct GreaterThanT<CTValue<T, First>, CTValue<T, Second>>
{
    static constexpr bool value = First > Second;
};

void valuelisttest()
{
    using Integers = Valuelist<int, 6, 2, 4, 9, 5, 2, 1, 7>;
    using SortedIntegers = InsertionSort<Integers, GreaterThanT>;
    static_assert(std::is_same_v<SortedIntegers,
                      Valuelist<int, 9, 7, 6, 5, 4, 2, 2, 1>,
                      "insertion sort failed");
}

```

Обратите внимание на возможность инициализировать CTValue с помощью оператора литерала (literal operator), например, как

auto a = 42_c; // Инициализация a как CTValue<int, 42>

Подробности представлены в разделе 25.6.

24.3.1. Выводимые нетиповые параметры

В C++17 `CTValue` можно усовершенствовать путем использования единственного выводимого нетипового параметра (записывая его как `auto`):

`typelist/ctvalue17.hpp`

```
template<auto Value>
struct CTValue
{
    static constexpr auto value = Value;
};
```

Это устраняет необходимость указывать тип для каждого применения `CTValue`, что делает его проще в использовании:

```
using Primes = Typelist<CTValue<2>, CTValue<3>, CTValue<5>,
                    CTValue<7>, CTValue<11>>;
```

То же самое может быть сделано для `Valuelist`, соответствующего стандарту C++17, но результат не обязательно будет лучше. Как отмечалось в разделе 15.10.1, пакет параметров с выведенным типом позволяет типам каждого аргумента быть различными:

```
template<auto... Values>
class Valuelist { };
int x;
using MyValueList = Valuelist<1, 'a', true, &x>;
```

Такой гетерогенный список значений может быть полезным, но это не то же самое, что наш предыдущий список `Valuelist`, который требует, чтобы все элементы имели один и тот же тип. Хотя может потребоваться, чтобы все элементы имели одинаковый тип (что обсуждается в разделе 15.10.1), пустой `Valuelist<>` в обязательном порядке будет иметь не известный тип элемента.

24.4. Оптимизация алгоритмов с помощью раскрытий пакетов

Раскрытия пакетов (pack expansions, подробно описанные в разделе 12.4.1) могут оказаться полезным механизмом для уменьшения количества работы компилятора со списками типов. Алгоритм `Transform`, разработанный в разделе 24.2.5, естественным образом приводит к использованию раскрытия пакета, потому что применяет одну и ту же операцию для каждого из элементов списка. Это позволяет использовать специализацию алгоритма (путем частичной специализации) для `Transform` при работе с `Typelist`:

`typelist/variadictransform.hpp`

```
template<typename... Elements, template<typename T> class MetaFun>
class TransformT<Typelist<Elements...>, MetaFun, false>
{
public:
    using Type = Typelist<typename MetaFun<Elements>::Type...>;
};
```

Эта реализация захватывает элементы списка в пакет параметров `Elements`. Затем она использует раскрытие пакета со схемой `typename MetaFun<Elements>::Type` для применения метафункции к каждому из типов в `Elements` и формирует список типов из получаемых результатов. Эта реализация проще, потому что не требует рекурсии и довольно простым способом использует возможности языка. Кроме того, она требует меньше инстанцирований шаблонов, потому что нужно инстанцировать только один экземпляр шаблона `Transform`. Алгоритм по-прежнему требует линейного количества инстанцирований `MetaFun`, но эти инстанцирования являются фундаментальными для данного алгоритма.

Другие алгоритмы получают выгоду от использования раскрытия пакета косвенно. Например, алгоритм `Reverse`, описанный в разделе 24.2.4, требует линейного количества инстанцирований `PushBack`. При применении `PushBack` с раскрытием пакета для `Typelist`, описанного в разделе 24.2.3 (который требует одного инстанцирования), алгоритм `Reverse` будет линейным. Однако более общая рекурсивная реализация `Reverse`, также описанная в этом разделе, сама по себе является линейной в смысле количества инстанцирований, что делает алгоритм `Reverse` квадратичным!

Раскрытие пакета может быть полезным и для выбора элементов в данном списке индексов для получения нового списка типов. Метафункция `Select` принимает список типов и `Valuelist`, содержащий индексы в этом списке, а затем создает новый список типов, содержащий элементы, указанные в `Valuelist`:

```
typelist/select.hpp

template<typename Types, typename Indices>
class SelectT;

template<typename Types, unsigned... Indices>
class SelectT<Types, Valuelist<unsigned, Indices...>>
{
public:
    using Type = Typelist<NthElement<Types, Indices...>...>;
};

template<typename Types, typename Indices>
using Select = typename SelectT<Types, Indices>::Type;
```

Индексы захватываются в пакет параметров `Indices`, который раскрывается для создания последовательности типов `NthElement`, индексирующих данный список типов, и создает новый список типов `Typelist`. В следующем примере показано, как можно использовать `Select` для обращения списка типов:

```
using SignedIntegralTypes =
    Typelist<signed char, short, int, long, long long>;
using ReversedSignedIntegralTypes =
    Select<SignedIntegralTypes, Valuelist<unsigned, 4, 3, 2, 1, 0>>;
// Создает Typelist<long long, long, int, short, signed char>
```

Список нетиповых параметров, содержащий индексы элементов в другом списке, часто называется *списком индексов* (*index list*) (или *последовательностью*)

индексов (index sequence)) и может упростить или вовсе устраниТЬ рекурсивные вычисления. Списки индексов подробно описаны в разделе 25.3.4.

24.5. Списки типов в стиле LISP

До введения вариативных шаблонов списки типов обычно создавались с рекурсивной структурой данных, моделирующей ячейки cons в LISP. Каждая такая ячейка содержит значение (голова списка) и вложенный список, который может быть либо другой ячейкой, либо пустым списком nil. Это понятие можно выразить непосредственно в C++:

typelist/cons.hpp

```
class Nil { };
template<typename HeadT, typename TailT = Nil>
class Cons
{
public:
    using Head = HeadT;
    using Tail = TailT;
};
```

Пустой список типов записывается как Nil, список из одного элемента int записывается как Cons<int, Nil> (или более лаконично как Cons<int>). Более длинные списки требуют вложенности:

using TwoShort = Cons<short, Cons<unsigned short>>;

Произвольно длинные списки типов могут быть построены с помощью глубокой рекурсивной вложенности, хотя писать такие длинные списки вручную может быть довольно неудобно:

```
using SignedIntegralTypes =
    Cons<signed char,
        Cons<short,
            Cons<int,
                Cons<long,
                    Cons<long long, Nil>>>>;
```

Извлечение первого элемента из такого списка ссылается непосредственно на голову списка:

typelist/consfront.hpp

```
template<typename List>
class FrontT
{
public:
    using Type = typename List::Head;
};

template<typename List>
using Front = typename FrontT<List>::Type;
```

Добавление элемента в начало списка “оборачивает” еще одну ячейку Cons вокруг существующего списка:

typelist/conspushfront.hpp

```
template<typename List, typename Element>
class PushFrontT
{
public:
    using Type = Cons<Element, List>;
};

template<typename List, typename Element>
using PushFront = typename PushFrontT<List, Element>::Type;
```

Наконец, удаление первого элемента из рекурсивного списка типов извлекает хвост списка:

typelist/conspopfront.hpp

```
template<typename List>
class PopFrontT
{
public:
    using Type = typename List::Tail;
};

template<typename List>
using PopFront = typename PopFrontT<List>::Type;
```

Специализация `IsEmpty` для `Nil` завершает множество фундаментальных операций со списками типов:

typelist/consisempty.hpp

```
template<typename List>
struct IsEmpty
{
    static constexpr bool value = false;
};

template<>
struct IsEmpty<Nil>
{
    static constexpr bool value = true;
};
```

Обладая этими операциями со списками типов, мы можем использовать алгоритм `InsertionSort`, определенный в разделе 24.2.7, для списков в стиле LISP:

typelist/conslisttest.hpp

```
template<typename T, typename U>
struct SmallerThanT
```

```

{
    static constexpr bool value = sizeof(T) < sizeof(U);
};

void conslisttest()
{
    using ConsList = Cons<int, Cons<char, Cons<short, Cons<double>>>>;
    using SortedTypes = InsertionSort<ConsList, SmallerThanT>;
    using Expected = Cons<char, Cons<short, Cons<int, Cons<double>>>>;
    std::cout << std::is_same<SortedTypes, Expected>::value << '\n';
}

```

Как мы видели на примере сортировки вставками, LISP-стиль списков типов позволяет выразить все те же алгоритмы, описанные в этой главе, которые разработаны для вариативных списков типов. Многие из описанных алгоритмов на самом деле написаны точно в том же стиле, что и для списков в стиле LISP. Однако они имеют несколько недостатков, которые заставляют нас выбирать вариативные версии. Во-первых, вложенность затрудняет написание и чтение — как исходных текстов, так и диагностики компилятора. Во-вторых, несколько алгоритмов (включая `PushBack` и `Transform`) можно специализировать для вариативных списков типов, чтобы добиться более эффективной реализации (измеряемой в количествах инстанцирований). Наконец, использование вариативных шаблонов для списков типов хорошо сочетается с использованием вариативных шаблонов для гетерогенных контейнеров, рассматриваемых в главах 25, “Кортежи”, и 26, “Контролируемые объединения”.

24.6. Заключительные замечания

Списки типов, по-видимому, появились вскоре после того, как в 1998 году был опубликован первый стандарт C++. Кшиштоф Чарнецки (Krysztof Czarnecki) и Ульрих Айзенекер (Ulrich Eisenecker) представили списки целочисленных констант в стиле LISP в [34]; впрочем, они так и не выполнили переход к обобщенным спискам типов.

Александреску популяризовал списки типов в своей замечательной книге *Современное проектирование на C++* ([3]). Прежде всего Александреску продемонстрировал разносторонность списков типов и возможность решения интересных задач проектирования с применением шаблонного метапрограммирования и списков типов, что сделало эти методы доступными для программистов на C++.

Абрахамс и Гуртовой предоставили многие необходимые для метапрограммирования структуры в [1], где описали абстракции для списков типов, алгоритмы для работы с ними и связанные с ними компоненты в знакомых терминах стандартной библиотеки C++: последовательности, итераторы, алгоритмы и (мета)функции. Соответствующая библиотека, Boost.MPL ([16]), широко используется для работы со списками типов.

Глава 25

Кортежи

В этой книге для иллюстрации возможностей шаблонов часто используются однородные (гомогенные) контейнеры и типы, подобные массивам. Такие однородные структуры расширяют концепцию массива в языках программирования C и C++ и часто используются во многих приложениях. В этих языках также есть гетерогенные образования, способные хранить данные — классы или структуры. В данной главе рассматриваются *кортежи* (tuple), агрегирующие данные подобно классам или структурам. Например, кортеж, содержащий int, double и std::string, подобен структуре с членами int, double и std::string, с тем отличием, что к элементам кортежа можно обращаться по их индексам (0, 1, 2), а не с помощью имен. Этот позиционный интерфейс и возможность легко создавать кортежи из списков типов делают кортежи более подходящими для использования в шаблонном метапрограммировании, чем структуры.

Существует альтернативный взгляд на кортежи как на проявление списков типов в выполнимой программе. Например, в то время как список типов Typelist<int, double, std::string> описывает последовательность типов, содержащую int, double и std::string, с которыми можно работать во время компиляции, Tuple<int, double, std::string> описывает хранилище для int, double и std::string, с которым можно работать во время выполнения. Например, в следующей программе создается экземпляр такого кортежа:

```
template<typename... Types>
class Tuple
{
    ... // Реализация обсуждается ниже
};

Tuple<int, double, std::string> t(17, 3.14, "Hello, World!");
```

В шаблонном метапрограммировании распространено использование списков типов для создания кортежей, которые могут применяться для хранения данных. Например, несмотря на то, что мы произвольно выбрали int, double и std::string как типы элементов в приведенном выше примере, мы могли бы создать набор типов, хранящихся в кортеже, с помощью метaproограммы.

В оставшейся части этой главы мы будем изучать реализацию и работу с шаблоном класса Tuple, который представляет собой упрощенную версию шаблона класса std::tuple.

25.1. Базовый дизайн Tuple

25.1.1. Хранилище

Кортежи содержат хранилища для каждого из типов в списке аргументов шаблона. Эти места хранения могут быть доступны с использованием шаблонной

функции `get`, синтаксис применения которой `get<I>(t)` для получения I-го элемента кортежа `t`. Например, `get<0>(t)` для `t` из предыдущего примера возвращает ссылку на `int 17`, в то время как `get<1>(t)` возвращает ссылку на `double 3.14`.

Рекурсивная формулировка хранилища в кортеже основана на той идее, что кортеж, содержащий $N > 0$ элементов, может храниться как единственный элемент (первый элемент, или голова списка) и кортеж, содержащий $N-1$ элемент (хвост), с отдельным частным случаем кортежа без элементов. Таким образом, трехэлементный кортеж `Tuple<int, double, std::string>` может храниться как `int` и кортеж `Tuple<double, std::string>`. Этот двухэлементный кортеж сам может храниться как `double` и `Tuple<std::string>`, причем последний может храниться как `std::string` и `Tuple<>`. Фактически это такое же рекурсивное разложение, как и используемое в обобщенных версиях алгоритмов для списков типов, и фактическая реализация рекурсивного хранения кортежа разворачивается аналогично тому, как это делалось со списками типов:

`tuples/tuple0.hpp`

```
template<typename... Types>
class Tuple;
// Рекурсивный случай:
template<typename Head, typename... Tail>
class Tuple<Head, Tail...>
{
private:
    Head head;
    Tuple<Tail...> tail;
public:
    // Конструкторы:
    Tuple()
    {
    }
    Tuple(Head const& head, Tuple<Tail...> const& tail)
        : head(head), tail(tail)
    {
    }
    ***
    Head& getHead()
    {
        return head;
    }
    Head const& getHead() const
    {
        return head;
    }
    Tuple<Tail...>& getTail()
    {
        return tail;
    }
    Tuple<Tail...> const& getTail() const
    {
        return tail;
    }
};
```

```
// Базовый случай:
template<>
class Tuple<>
{
    // Хранилище не требуется
};
```

В рекурсивном случае каждый экземпляр класса `Tuple` содержит член-данное `head`, который хранит первый элемент в списке, а также член-данное `tail`, который хранит оставшиеся элементы в списке. Базовый случай является просто пустым кортежем, который не имеет связанного с ним хранилища для данных.

Шаблон функции `get` проходит по этой рекурсивной структуре для извлечения запрошенного элемента¹:

`tuples/tupleget.hpp`

```
// Рекурсивный случай:
template<unsigned N>
struct TupleGet
{
    template<typename Head, typename... Tail>
    static auto apply(Tuple<Head, Tail...> const& t)
    {
        return TupleGet < N - 1 >::apply(t.getTail());
    }
};

// Базовый случай:
template<>
struct TupleGet<0>
{
    template<typename Head, typename... Tail>
    static Head const& apply(Tuple<Head, Tail...> const& t)
    {
        return t.getHead();
    }
};

template<unsigned N, typename... Types>
auto get(Tuple<Types...> const& t)
{
    return TupleGet<N>::apply(t);
}
```

Обратите внимание на то, что шаблон функции `get` — это просто тонкая оболочка над вызовом статической функции-члена `TupleGet`. Этот метод фактически является обходным путем для отсутствующей частичной специализации шаблонов функций (данный вопрос обсуждается в разделе 17.3), который мы используем для специализации значения `N`. В рекурсивном случае

¹ Полная реализация `get()` должна соответствующим образом обрабатывать неконстантные кортежи и кортежи, являющиеся ссылками на `г`-значения.

($N > 0$) статическая функция-член `apply()` извлекает хвост текущего кортежа и уменьшает N для продолжения поиска запрошенного элемента далее в кортеже. Базовый случай ($N = 0$) возвращает голову текущего кортежа, завершая нашу реализацию.

25.1.2. Создание кортежа

Помимо определенных к настоящему времени конструкторов:

```
Tuple()
{
}
Tuple(Head const& head, Tuple<Tail...> const& tail)
    : head(head), tail(tail)
{
}
```

мы должны иметь возможность создавать его как из набора независимых значений (по одному для каждого элемента), так и из другого кортежа. Копирующее конструирование из набора независимых значений берет первое из этих значений для инициализации элемента `head` (через его базовый класс), а затем передает оставшиеся значения базовому классу, представляющему хвост кортежа:

```
Tuple(Head const& head, Tail const& ... tail)
    : head(head), tail(tail...)
{
}
```

Этот код обеспечивает корректность нашего исходного примера `Tuple`:

```
Tuple<int, double, std::string> t(17, 3.14, "Hello, World!");
```

Однако это не самый обобщенный интерфейс: пользователи могут пожелать применить перемещающее конструирование для инициализации некоторых (но не всех) элементов или иметь элемент, созданный из значения иного типа. Таким образом, для инициализации кортежа мы должны использовать прямую передачу (раздел 15.6.3):

```
template<typename VHead, typename... VTail>
Tuple(VHead&& vhead, VTail&& ... vtail)
    : head(std::forward<VHead>(vhead)),
      tail(std::forward<VTail>(vtail)...)
```

Затем мы реализуем поддержку построения кортежа из другого кортежа:

```
template<typename VHead, typename... VTail>
Tuple(Tuple<VHead, VTail...> const& other)
    : head(other.getHead()), tail(other.getTail()) { }
```

Однако введение этого конструктора недостаточно, чтобы разрешить преобразования кортежей: для показанного выше кортежа `t` попытка создать еще один кортеж с совместимыми типами завершится ошибкой:

```
// Ошибка: нет преобразования Tuple<int, double, string> в long
Tuple<long int, long double, std::string> t2(t);
```

Проблема заключается в том, что шаблон конструктора, предназначенный для инициализации из набора независимых значений, оказывается лучшим соотвествием, чем шаблон конструктора, который принимает кортеж. Для решения этой проблемы мы должны использовать `std::enable_if<>` (см. разделы 6.3 и 20.3) для отключения обоих шаблонов функций-членов, когда длина хвоста не соответствует ожидаемой.

```
template<typename VHead, typename... VTail,
         typename = std::enable_if_t<sizeof...(VTail) == sizeof...(Tail)>>
Tuple(VHead && vhead, VTail && ... vtail)
    : head(std::forward<VHead>(vhead)),
      tail(std::forward<VTail>(vtail)) {}
```



```
template<typename VHead, typename... VTail,
         typename = std::enable_if_t<sizeof...(VTail) == sizeof...(Tail)>>
Tuple(Tuple<VHead, VTail...> const& other)
    : head(other.getHead()), tail(other.getTail()) {}
```

Вы можете найти все объявления конструкторов в файле `tuples/tuple.hpp`.

Шаблон функции `makeTuple()` использует вывод для определения типов элементов возвращаемого кортежа `Tuple`, что облегчает создание кортежа из заданного набора элементов:

`tuples/maketuple.hpp`

```
template<typename... Types>
auto makeTuple(Types&& ... elems)
{
    return Tuple<std::decay_t<Types>...>(std::forward<Types>(elems)...);
```

Здесь вновь используется прямая передача в сочетании со свойством `std::decay<>` для преобразования строковых литералов и других обычных массивов в указатели и удаления `const` и ссылок. Например:

`makeTuple(17, 3.14, "Hello, World!")`

инициализирует

`Tuple<int, double, char const*>`

25.2. Базовые операции кортежей

25.2.1. Сравнение

Кортежи представляют собой структурные типы, содержащие другие значения. Для сравнения двух кортежей достаточно сравнить их элементы. Таким образом, можно написать определение оператора `operator==` для поэлементного сравнения двух определений:

tuples/tupleeq.hpp

```
// Базовый случай:
bool operator==(Tuple<> const&, Tuple<> const&)
{
    // Пустые кортежи всегда эквивалентны
    return true;
}

// Рекурсивный случай:
template<typename Head1, typename... Tail1,
         typename Head2, typename... Tail2,
         typename = std::enable_if_t<sizeof...(Tail1) == sizeof...(Tail2)>>
bool operator==(Tuple<Head1, Tail1...> const& lhs,
                  Tuple<Head2, Tail2...> const& rhs)
{
    return lhs.getHead() == rhs.getHead() &&
           lhs.getTail() == rhs.getTail();
}
```

Подобно многим алгоритмам для списков типов и кортежей поэлементное сравнение посещает элемент в голове списка, а затем рекурсивно проходит по хвосту, в конечном итоге доходя до базового случая. Операторы !=, <, >, <= и >= реализуются аналогично.

25.2.2. Вывод

В этой главе мы будем создавать новые типы кортежей, поэтому полезно иметь возможность видеть эти кортежи в выполняемой программе. Оператор `operator<<` выводит любой кортеж, типы элементов которого могут быть выведены:

tuples/tupleio.hpp

```
#include <iostream>
void printTuple(std::ostream& strm, Tuple<> const&,
                bool isFirst = true)
{
    strm << (isFirst ? '(' : ')';
}

template<typename Head, typename... Tail>
void printTuple(std::ostream& strm, Tuple<Head, Tail...> const& t,
                bool isFirst = true)
{
    strm << (isFirst ? "(" : ", ");
    strm << t.getHead();
    printTuple(strm, t.getTail(), false);
}

template<typename... Types>
std::ostream& operator<<(std::ostream& strm, Tuple<Types...> const& t)
{
    printTuple(strm, t);
    return strm;
}
```

Теперь легко как создавать кортежи, так и выводить их. Например, фрагмент

```
std::cout << makeTuple(1, 2.5, std::string("hello")) << '\n';
```

выводит

```
(1, 2.5, hello)
```

25.3. Алгоритмы для работы с кортежами

Кортежи представляют собой контейнеры, которые обеспечивают возможность доступа и изменения каждого из их элементов (через `get`), а также создания новых кортежей (непосредственно или с помощью `makeTuple()`) и разделения кортежей на голову и хвост (`getHead()` и `getTail()`). Этих фундаментальных строительных блоков достаточно для создания набора алгоритмов для работы с кортежами, таких как добавление элементов в кортеж или их удаление из него, изменение порядка элементов в кортеже или выборка некоторого подмножества элементов из кортежа.

Алгоритмы для работы с кортежами особенно интересны, потому что они требуются и во время компиляции, и во время выполнения. Подобно работе со списками типов в главе 24, “Списки типов”, применение алгоритма к кортежу может привести к кортежу совершенно иного типа, который требуется для выполнения вычислений времени компиляции. Например, обращение кортежа `Tuple<int, double, string>` дает кортеж `Tuple<string, double, int>`. Однако подобно алгоритмам для гомогенного контейнера (например, `std::reverse()` для `std::vector`) алгоритмы для кортежей фактически требуют выполнения кода во время выполнения программы, так что нам нужно не забывать об эффективности генерируемого кода.

25.3.1. Кортежи как списки типов

Если игнорировать конкретные компоненты времени выполнения нашего шаблона `Tuple`, то мы увидим, что он имеет точно такую же структуру, что и шаблон `TypeList`, разработанный в главе 24, “Списки типов”: он принимает любое количество параметров типа шаблона. Фактически с использованием небольшой частичной специализации можно превратить кортеж в полнофункциональный список типов:

```
tuples/tupletypelist.hpp
```

```
// Определение, пуст ли контейнер:
template<>
struct IsEmpty<Tuple<>>
{
    static constexpr bool value = true;
};

// Извлечение первого элемента:
template<typename Head, typename... Tail>
class FrontT<Tuple<Head, Tail...>>
```

```

{
    public:
        using Type = Head;
};

// Удаление первого элемента:
template<typename Head, typename... Tail>
class PopFrontT<Tuple<Head, Tail...>>
{
    public:
        using Type = Tuple<Tail...>;
};

// Добавление элемента в начало:
template<typename... Types, typename Element>
class PushFrontT<Tuple<Types...>, Element>
{
    public:
        using Type = Tuple<Element, Types...>;
};

// Добавление элемента в конец:
template<typename... Types, typename Element>
class PushBackT<Tuple<Types...>, Element>
{
    public:
        using Type = Tuple<Types..., Element>;
};

```

Теперь все алгоритмы, разработанные в главе 24, “Списки типов”, для списков типов, одинаково хорошо работают как с Tuple, так и с Typelist, так что мы легко можем работать с типом кортежей. Например:

```

Tuple<int, double, std::string> t1(17, 3.14, "Hello, World!");
using T2 = PopFront<PushBack<decltype(t1), bool>>;
T2 t2(get<1>(t1), get<2>(t1), true);
std::cout << t2;

```

Этот фрагмент кода выводит:

```
(3.14, Hello, World!, 1)
```

Как мы вскоре увидим, алгоритмы для списков типов, применяемые к типам кортежей, часто используются для того, чтобы определить тип результата алгоритма для работы с кортежами.

25.3.2. Добавление элементов в кортеж и удаление их оттуда

Возможность добавления элемента в начало или конец кортежа имеет важное значение для построения более сложных алгоритмов. Как и в случае со списками типов, вставка элемента в начало кортежа гораздо легче, чем вставка в его конец, так что мы начнем с pushFront:

tuples/pushfront.hpp

```
template<typename... Types, typename V>
PushFront<Tuple<Types...>, V>
pushFront(Tuple<Types...> const& tuple, V const& value)
{
    return PushFront<Tuple<Types...>, V>(value, tuple);
}
```

Добавление нового элемента (с именем *value*) в начало существующего кортежа требует от нас сформировать новый кортеж со значением *value* в качестве его головы и существующего кортежа в качестве хвоста. Получающийся в результате кортеж имеет тип *Tuple<V, Types...>*. Однако мы решили использовать алгоритм *PushFront* для списков типов, чтобы продемонстрировать тесную связь между аспектами времени компиляции и времени выполнения алгоритмов для кортежей: *PushFront* времени компиляции вычисляет тип, который нам нужно построить, чтобы получить соответствующее значение времени выполнения.

Добавление нового элемента в конец существующего кортежа является более сложным, потому что оно требует рекурсивного обхода кортежа с построением по мере обхода модифицированного кортежа. Обратите внимание на то, как структура реализации *pushBack()* следует рекурсивной формулировке *PushBack()* для списка типов из раздела 24.2.3:

tuples/pushback.hpp

```
// Базовый случай
template<typename V>
Tuple<V> pushBack(Tuple<> const&, V const& value)
{
    return Tuple<V>(value);
}

// Рекурсивный случай
template<typename Head, typename... Tail, typename V>
Tuple<Head, Tail..., V>
pushBack(Tuple<Head, Tail...> const& tuple, V const& value)
{
    return Tuple<Head, Tail..., V>(tuple.getHead(),
                                    pushBack(tuple.getTail(), value));
}
```

Базовый случай, как и ожидалось, добавляет значение к кортежу нулевой длины, производя кортеж, содержащий только одно это значение. В рекурсивном случае мы формируем новый кортеж из текущего элемента в начале списка (*tuple.getHead()*) и результата добавления нового элемента к хвосту списка (рекурсивный вызов *pushBack*). Хотя мы решили выразить сконструированный тип как *Tuple<Head, Tail..., V>*, следует отметить, что это эквивалентно использованию *PushBack<Tuple<Head, Tail...>, V>* времени компиляции.

Реализовать *popFront()* очень легко:

tuples/popfront.hpp

```
template<typename... Types>
PopFront<Tuple<Types...>> popFront(Tuple<Types...> const& tuple)
{
    return tuple.getTail();
}
```

Теперь пример из раздела 25.3.1 можно переписать следующим образом:

```
Tuple<int, double, std::string> t1(17, 3.14, "Hello, World!");
auto t2 = popFront(pushBack(t1, true));
std::cout << std::boolalpha << t2 << '\n';
```

Этот код выводит

```
(3.14, Hello, World!, true)
```

25.3.3. Обращение порядка элементов кортежа

Порядок элементов кортежа может быть обращен с помощью другого рекурсивного алгоритма для кортежей, структура которого соответствует обращению списка типов в разделе 24.2.4:

tuples/reverse.hpp

```
// Базовый случай
Tuple<> reverse(Tuple<> const& t)
{
    return t;
}

// Рекурсивный случай
template<typename Head, typename... Tail>
Reverse<Tuple<Head, Tail...>> reverse(Tuple<Head, Tail...> const& t)
{
    return pushBack(reverse(t.getTail()), t.getHead());
}
```

Базовый случай тривиален, в то время как рекурсивный случай обращает хвост списка и добавляет текущую голову списка к обращенному списку. Это означает, например, что

```
reverse(makeTuple(1, 2.5, std::string("hello")))
```

даст *Tuple<string, double, int>* со значениями *string("hello")*, 2.5 и 1 соответственно.

Как и в случае со списками типов, теперь таким образом можно легко реализовать *popBack()* путем вызова *popFront()* для временно обращенного списка с использованием *PopBack* из раздела 24.2.4:

tuples/popback.hpp

```
template<typename... Types>
PopBack<Tuple<Types...>> popBack(Tuple<Types...> const& tuple)
{
    return reverse(popFront(reverse(tuple)));
}
```

25.3.4. Индексные списки

Рекурсивная формулировка обращения порядка элементов кортежа в предыдущем разделе корректна, но излишне неэффективна во время выполнения. Чтобы понять, в чем заключается проблема, рассмотрим простой класс, который подсчитывает количество своих копирований²:

tuples/copycounter.hpp

```
template<int N>
struct CopyCounter
{
    inline static unsigned numCopies = 0;
    CopyCounter()
    {
    }
    CopyCounter(CopyCounter const&)
    {
        ++numCopies;
    }
};
```

Затем создадим и обратим кортеж экземпляров CopyCounter:

tuples/copycountertest.hpp

```
void copycountertest()
{
    Tuple<CopyCounter<0>, CopyCounter<1>, CopyCounter<2>,
          CopyCounter<3>, CopyCounter<4>> copies;
    auto reversed = reverse(copies);
    std::cout << "0: " << CopyCounter<0>::numCopies << " copies\n";
    std::cout << "1: " << CopyCounter<1>::numCopies << " copies\n";
    std::cout << "2: " << CopyCounter<2>::numCopies << " copies\n";
    std::cout << "3: " << CopyCounter<3>::numCopies << " copies\n";
    std::cout << "4: " << CopyCounter<4>::numCopies << " copies\n";
}
```

Вывод данной программы имеет следующий вид:

```
0: 5 copies
1: 8 copies
2: 9 copies
3: 8 copies
4: 5 copies
```

Очень большое количество копирований! В идеальной реализации обращения порядка элементов кортежа каждый элемент будет скопирован только один раз, из исходной позиции непосредственно в правильную позицию в результирующем кортеже. Мы могли бы достичь этой цели с помощью тщательного использования ссылок, включая применение ссылок для типов промежуточных аргументов, но это значительно усложнит нашу реализацию.

² До C++17 не поддерживались встраиваемые статические члены. Поэтому мы должны инициализировать numCopies вне структуры класса в одной единице трансляции.

Для устранения излишних копирований при обращении порядка элементов в кортеже рассмотрим, как можно было бы реализовать эту операцию для одного кортежа известной длины (скажем, из 5 элементов, как в нашем примере). Мы могли бы просто использовать `makeTuple()` и `get()`:

```
auto reversed = makeTuple(get<4>(copies), get<3>(copies),
                          get<2>(copies), get<1>(copies),
                          get<0>(copies));
```

Эта программа дает тот результат, который нам нужен, при единственном копировании каждого элемента кортежа:

```
0: 1 copies
1: 1 copies
2: 1 copies
3: 1 copies
4: 1 copies
```

Индексные списки (index lists, именуемые также *списками индексов* или *последовательностями индексов* (index sequences); см. раздел 24.4) обобщают это понятие, захватывая множество индексов кортежа — в данном случае 4, 3, 2, 1, 0 — в пакет параметров, который позволяет получить последовательность вызовов `get` с помощью раскрытия пакета. Это дает возможность отделить вычисление индексов, которое может быть сколь угодно сложной шаблонной метапрограммой, от фактического применения этого индексного списка, где очень важна эффективность времени выполнения. Для представления индексных списков часто используется стандартный тип `std::integer_sequence` (введен в C++14).

25.3.5. Обращение с использованием индексных списков

Чтобы выполнить обращение порядка элементов в кортеже с помощью индексных списков, сначала нам нужно иметь представление этих индексных списков. Такой список является типом, содержащим значения, используемые в качестве индексов в списке типов или гетерогенной структуре данных (см. раздел 24.4). Для нашего индексного списка мы будем использовать тип `ValueList`, разработанный в разделе 24.3. Список индексов, соответствующий приведенному выше примеру обращения элементов кортежа, имеет вид

```
ValueList<unsigned, 4, 3, 2, 1, 0>
```

Как получить этот список индексов? Один из подходов заключается в том, чтобы начать с создания списка с индексами от 0 до $N-1$ включительно, где N — длина кортежа, используя простую шаблонную метапрограмму `MakeIndexList`³:

```
tuples/makeindexlist.hpp
```

```
// Рекурсивный случай
template<unsigned N, typename Result = ValueList<unsigned>>
struct MakeIndexListT
```

³ Стандарт C++14 предоставляет подобный шаблон `make_index_sequence`, который создает список индексов типа `std::size_t`, а также более общий шаблон `make_integer_sequence`, который позволяет выбрать определенный тип индексов.

```

        : MakeIndexListT<N-1,
                      PushFront<Result, CTValue<unsigned, N-1>>>
    };

// Базовый случай
template<typename Result>
struct MakeIndexListT<0, Result>
{
    using Type = Result;
};

template<unsigned N>
using MakeIndexList = typename MakeIndexListT<N>::Type;

```

Затем можно объединить эту операцию с Reverse для списков типов, чтобы создавать соответствующий список индексов:

```

using MyIndexList = Reverse<MakeIndexList<5>>;
// Эквивалентно Valuelist<unsigned, 4, 3, 2, 1, 0>

```

Для фактического обращения индексы в списке должны быть захвачены пакетом нетиповых параметров. Это выполняется путем разделения реализации алгоритма reverse() для кортежа с индексами на две части:

tuples/indexlistreverse.hpp

```

template<typename... Elements, unsigned... Indices>
auto reverseImpl(Tuple<Elements...> const& t,
                  Valuelist<unsigned, Indices...>)
{
    return makeTuple(get<Indices>(t)...);
}

template<typename... Elements>
auto reverse(Tuple<Elements...> const& t)
{
    return reverseImpl(t,
                       Reverse<MakeIndexList<sizeof...(Elements)>>());
}

```

В C++11 возвращаемые типы должны быть объявлены как

```
->decltype(makeTuple(get<Indices>(t)...))
```

и

```
->decltype(reverseImpl(t, Reverse<MakeIndexList<sizeof...(Elements)>>()))
```

Шаблон функции reverseImpl() захватывает индексы из своего параметра Valuelist в пакет параметров Indices. Затем он возвращает результат вызова makeTuple() с аргументами, сформированными вызовом get() для кортежа со множеством захваченных индексов.

Сам по себе алгоритм reverse(), как обсуждалось ранее, просто формирует соответствующее множество индексов и предоставляет его алгоритму reverseImpl.

Работа с индексами выполняется с помощью шаблонного метапрограммирования, а потому не производит кода времени выполнения. Единственный код времени выполнения находится в шаблоне `reverseImpl`, который использует `makeTuple()` для построения результирующего кортежа за один шаг, и поэтому копирование элементов кортежа выполняется только однократно.

25.3.6. Тасование и выбор

Шаблон функции `reverseImpl()`, используемый в предыдущем разделе для формирования кортежа с обращенным порядком элементов, фактически не содержит кода, специфичного для операции `reverse()`. Вместо этого он просто выбирает определенное множество индексов из существующего кортежа и использует его для формирования нового кортежа. `reverse()` обеспечивает набор индексов в обратном порядке, но многие алгоритмы могут быть построены на базовом алгоритме `select()`⁴:

tuples/select.hpp

```
template<typename... Elements, unsigned... Indices>
auto select(Tuple<Elements...> const& t,
            Valuelist<unsigned, Indices...>)
{
    return makeTuple(get<Indices>(t)...);
}
```

Один простой алгоритм, который основан на `select()`, представляет собой операцию “размножения” над кортежем, принимая один элемент кортежа и реплицируя его для создания другого кортежа с некоторым количеством копий этого элемента. Например, код

```
Tuple<int, double, std::string> t1(42, 7.7, "hello");
auto a = splat<1, 4>(t);
std::cout << a << '\n';
```

генерирует кортеж `Tuple<double, double, double, double>`, в котором каждое значение является копией `get<1>(t)`, так что вывод этого фрагмента кода имеет вид
`(7.7, 7.7, 7.7, 7.7)`

В приведенной ниже метапрограмме, производящей репликованное множество индексов, состоящее из N копий значения I , `splat()` представляет собой непосредственное применение `select()`⁵:

tuples/splat.hpp

```
template<unsigned I, unsigned N,
        typename IndexList = Valuelist<unsigned>>
class ReplicatedIndexListT;
```

⁴ В C++11 возвращаемый тип должен быть объявлен как `->decltype(makeTuple(get<Indices>(t)...))`.

⁵ В C++11 возвращаемым типом `splat()` должен быть `->decltype(возвращаемое выражение)`.

```

template<unsigned I, unsigned N, unsigned... Indices>
class ReplicatedIndexListT<I, N, Valuelist<unsigned, Indices...>>
    : public ReplicatedIndexListT < I, N - 1,
      Valuelist<unsigned, Indices..., I >>
{
};

template<unsigned I, unsigned... Indices>
class ReplicatedIndexListT<I, 0, Valuelist<unsigned, Indices...>>
{
public:
    using Type = Valuelist<unsigned, Indices...>;
};

template<unsigned I, unsigned N>
using ReplicatedIndexList = typename ReplicatedIndexListT<I, N>::Type;

template<unsigned I, unsigned N, typename... Elements>
auto splat(Tuple<Elements...> const& t)
{
    return select(t, ReplicatedIndexList<I, N>());
}

```

Даже сложные алгоритмы для работы с кортежами могут быть реализованы в терминах шаблонных метапрограмм для списка индексов с последующим применением `select()`. Например, можно использовать сортировку вставками, разработанную в разделе 24.2.7, для сортировки кортежа на основе размеров типов элементов. При наличии такой функции `sort()`, которая принимает шаблонную метафункцию, сравнивающую типы элементов кортежа, в качестве компаратора, мы могли бы сортировать элементы кортежа по размеру с помощью кода наподобие следующего:

`tuples/tuplesorttest.hpp`

```

#include <complex>

template<typename T, typename U>
class SmallerThanT
{
public:
    static constexpr bool value = sizeof(T) < sizeof(U);
};

void testTupleSort()
{
    auto t1 = makeTuple(17LL, std::complex<double>(42, 77), 'c', 42, 7.7);
    std::cout << t1 << '\n';
    auto t2 = sort<SmallerThanT>
        (t1); // t2 является Tuple<int, long, std::string>
    std::cout << "Сортировка по размеру: " << t2 << '\n';
}

```

Вывод этого кода может иметь следующий вид⁶:

```
(17, (42,77), c, 42, 7.7)
Сортировка по размеру: (c, 42, 7.7, 17, (42,77))
```

Фактическая реализация `sort()` включает применение `InsertionSort` в `select()`⁷:

tuples/tuplesort.hpp

```
// Оболочка-методфункция, сравнивающая элементы кортежа:
template<typename List, template<typename T, typename U> class F>
class MetafunOfNthElementT
{
public:
    template<typename T, typename U> class Apply;
    template<unsigned N, unsigned M>
    class Apply<CTValue<unsigned, M>, CTValue<unsigned, N>>
        : public F<NthElement<List, M>, NthElement<List, N>>{};
};

// Сортировка кортежа, основанная на сравнении типов элементов:
template<template<typename T, typename U> class Compare,
         typename... Elements>
auto sort(Tuple<Elements...> const& t)
{
    return select(t,
                  InsertionSort<MakeIndexList<sizeof...(Elements)>,
                  MetafunOfNthElementT<
                      Tuple<Elements...>,
                      Compare>::template Apply>());
}
```

Взгляните на использование `InsertionSort` внимательно: фактический сортируемый список типов является списком индексов, указывающих на элементы списка типов, построенного с помощью `MakeIndexList<>`. Таким образом, результатом сортировки вставками является набор индексов, указывающих на элементы кортежа, который затем предоставается алгоритму `select()`. Однако, поскольку `InsertionSort` работает с индексами, он ожидает компаратор, который сравнивает два индекса. Принцип легче понять при рассмотрении сортировки индексов `std::vector`, выполняемой в следующем (не метапрограммном!) примере:

tuples/indexsort.cpp

```
#include <vector>
#include <algorithm>
#include <string>
```

⁶ Получающийся в результате порядок зависит от конкретных размеров типов на данной платформе. Например, размер `double` может быть меньше, таким же или больше, чем размер `long long`.

⁷ В C++11 возвращаемый тип `sort()` должен быть объявлен как `->decltype(возвращаемое выражение)`.

```

int main()
{
    std::vector<std::string> strings = {"banana", "apple", "cherry"};
    std::vector<unsigned> indices = { 0, 1, 2 };
    std::sort(indices.begin(), indices.end(),
              [&strings](unsigned i, unsigned j)
    {
        return strings[i] < strings[j];
    });
}

```

Здесь `indices` содержит индексы элементов вектора строк. Операция `sort()` сортирует индексы, так что лямбда-выражение компаратора принимает два значения типа `unsigned` (вместо строк). Однако тело лямбда-выражения использует эти `unsigned` значения в качестве индексов элементов вектора строк, поэтому порядок фактически соответствует содержимому строк. По окончании сортировки `indices` содержит индексы элементов вектора строк, отсортированные на основе значений этих строк.

Наше использование `InsertionSort` для `sort()` для кортежа применяет тот же подход. Шаблон адаптера `MetafunOfNthElementT` предоставляет шаблонную метафункцию (вложенный класс `Apply`), которая принимает два индекса (специализации `CTValue`) и использует `NthElement`, чтобы извлечь соответствующие элементы из его аргумента `TypeList`. В некотором смысле шаблон члена `Apply` “захватывает” список типов, предоставленный охватывающему его шаблону (`MetafunOfNthElementT`) таким же образом, как лямбда-выражение захватывает вектор строк из охватывающей области видимости. Затем `Apply` передает извлеченные типы базовой метафункции `F`, завершая адаптацию.

Обратите внимание на то, что все вычисления для сортировки выполняются во время компиляции, а результирующий кортеж формируется непосредственно, без излишнего копирования значений во время выполнения.

25.4. Распаковка кортежей

Кортежи полезны для хранения набора связанных значений вместе в одном объекте, независимо от типов этих значений или от их количества. В определенный момент может потребоваться распаковать такой кортеж, например, передать его элементы функции в качестве отдельных аргументов. В качестве простого примера мы хотим взять кортеж и передать его элементы вариативной операции `print()`, описанной в разделе 12.4:

```

Tuple<std::string,char const*,int,char> t("Pi","is roughly",3,'\n');
print(t...); // Ошибка: невозможно распаковать кортеж - он
            // не является пакетом параметров

```

Как показано в примере, “очевидная” попытка распаковать кортеж не удается, потому что это не пакет параметров. Того же можно добиться с помощью списка индексов. Показанный ниже шаблон функции `apply()` принимает функцию и кортеж, а затем вызывает функцию с распакованными элементами кортежа:

tuples/apply.hpp

```
template<typename F, typename... Elements, unsigned... Indices>
auto applyImpl(F f, Tuple<Elements...> const& t,
               ValueList<unsigned, Indices...>)
-> decltype(f(get<Indices>(t)...))
{
    return f(get<Indices>(t)...);
}

template<typename F, typename... Elements,
          unsigned N = sizeof...(Elements)>
auto apply(F f, Tuple<Elements...> const& t)
-> decltype(applyImpl(f, t, MakeIndexList<N>()))
{
    return applyImpl(f, t, MakeIndexList<N>());
}
```

Шаблон функции `applyImpl()` принимает список индексов и использует его для распаковки элементов кортежа в список аргументов для его аргумента `f`, который представляет собой функциональный объект. Функция `apply()`, с которой работает пользователь, отвечает только за построение начального списка индексов. Вместе они позволяют нам распаковать кортеж для передачи в качестве аргументов `print()`:

```
Tuple<std::string, char const*, int, char>
    t("Pi", "примерно равно", 3, '\n');
apply(print, t); // OK: выводит "Pi примерно равно 3"
```

C++17 предоставляет подобную функцию, которая работает с любым “кортежеобразным” типом.

25.5. Оптимизация кортежей

Кортеж является фундаментальным гетерогенным контейнером с большим количеством потенциальных способов использования. Поэтому имеет смысл обсудить, что можно сделать для оптимизации использования кортежей как во время выполнения (с точки зрения используемой памяти и времени работы), так и во время компиляции (количество инстанцирований шаблонов). В этом разделе рассматриваются несколько конкретных оптимизаций нашей реализации кортежа `Tuple`.

25.5.1. Кортежи и оптимизация пустого базового класса

Наш кортеж `Tuple` использует больше памяти, чем это строго необходимо. Одна из проблем заключается в том, что член `tail` в конечном итоге будет пустым кортежем, потому что каждый непустой кортеж завершается пустым кортежем, а члены-данные всегда должны иметь размер, составляющий хотя бы один байт.

Для повышения эффективности использования памяти кортежем `Tuple` можно применить *оптимизацию пустого базового класса* (empty base class optimization – EBCO), рассматривавшуюся в разделе 21.1, наследуя хвост вместо использования его в качестве члена. Например:

tuples/tuplestorage1.hpp

```
// Рекурсивный случай:
template<typename Head, typename... Tail>
class Tuple<Head, Tail...> : private Tuple<Tail...>
{
private:
    Head head;
public:
    Head& getHead() { return head; }
    Head const& getHead() const { return head; }
    Tuple<Tail...>& getTail() { return *this; }
    Tuple<Tail...> const& getTail() const { return *this; }
};
```

Это тот же подход, который мы применяли к `BaseMemberPair` в разделе 21.1.2. К сожалению, на практике он имеет побочный эффект, заключающийся в обратном порядке инициализации элементов кортежа в конструкторах. Ранее, поскольку член `head` предшествовал члену `tail`, `head` инициализировался первым. В новой формулировке `Tuple` хвост содержится в базовом классе, поэтому он будет инициализирован до члена `head`⁸.

Эта проблема может быть решена путем помещения члена `head` в собственный базовый класс, который предшествует в списке базовых классов хвосту. Непосредственная реализация этого решения вводит шаблон `TupleElt`, который используется для создания оболочки для каждого типа элемента, так что кортеж может его наследовать:

tuples/tuplestorage2.hpp

```
template<typename... Types>
class Tuple;

template<typename T>
class TupleElt
{
    T value;
public:
    TupleElt() = default;
    template<typename U>
    TupleElt(U&& other) : value(std::forward<U>(other)) { }
    T& get()
    {
        return value;
    }
    T const& get() const
    {
        return value;
    }
};
```

⁸ Еще одно практическое воздействие этого изменения состоит в том, что элементы кортежа в конечном итоге будут храниться в обратном порядке, поскольку базовые классы обычно хранятся до членов.

```

// Рекурсивный случай:
template<typename Head, typename... Tail>
class Tuple<Head, Tail...>
    : private TupleElt<Head>, private Tuple<Tail...>
{
public:
    Head& getHead()
    {
        // Потенциальная неоднозначность
        return static_cast<TupleElt<Head>*>(this)->get();
    }
    Head const& getHead() const
    {
        // Потенциальная неоднозначность
        return static_cast<TupleElt<Head> const*>(this)->get();
    }
    Tuple<Tail...>& getTail()
    {
        return *this;
    }
    Tuple<Tail...> const& getTail() const
    {
        return *this;
    }
};

// Базовый случай:
template<>
class Tuple<>
{
    // Хранилище не требуется
};

```

Несмотря на то что этот подход решает проблему порядка инициализации, он добавляет новую (худшую) проблему: мы больше не можем извлекать элементы из кортежа, содержащего два элемента одного и того же типа, такого как `Tuple<int, int>`, потому что преобразование производного типа кортежа в базовый тип `TupleElt` для этого типа (например, `TupleElt<int>`) оказывается неоднозначным.

Чтобы избежать неоднозначности, мы должны гарантировать, что каждый базовый класс `TupleElt` является уникальным в пределах данного `Tuple`. Один из подходов заключается в том, чтобы закодировать “высоту” этого значения в кортеже, т.е. длину хвоста кортежа. Последний элемент в кортеже будет храниться с высотой 0, предпоследний элемент будет храниться с высотой 1 и так далее⁹:

`tuples/tupleelt1.hpp`

```

template<unsigned Height, typename T>
class TupleElt

```

⁹Более интуитивно понятно было бы просто использовать индекс элемента кортежа, а не его высоту. Однако эта информация не так легко доступна в `Tuple`, потому что данный кортеж может быть и автономным кортежем, и хвостом другого кортежа. Однако данный `Tuple` знает, сколько элементов находятся в его собственном хвосте.

```
{
    T value;
public:
    TupleElt() = default;
    template<typename U>
    TupleElt(U&& other) : value(std::forward<U>(other)) { }
    T& get()
    {
        return value;
    }
    T const& get() const
    {
        return value;
    }
};
```

При наличии такого решения можно создавать Tuple, который использует оптимизацию пустого базового класса при сохранении порядка инициализации и поддержки нескольких элементов одного типа:

tuples/tuplестorage3.hpp

```
template<typename... Types>
class Tuple;
// Рекурсивный случай:
template<typename Head, typename... Tail>
class Tuple<Head, Tail...>
    : private TupleElt<sizeof...(Tail), Head>, private Tuple<Tail...>
{
    using HeadElt = TupleElt<sizeof...(Tail), Head>;
public:
    Head& getHead()
    {
        return static_cast<HeadElt*>(this)->get();
    }
    Head const& getHead() const
    {
        return static_cast<HeadElt const*>(this)->get();
    }
    Tuple<Tail...>& getTail()
    {
        return *this;
    }
    Tuple<Tail...> const& getTail() const
    {
        return *this;
    }
};

// Базовый случай:
template<>
class Tuple<>
{
    // Хранилище не требуется
};
```

При использовании такой реализации приведенная ниже программа

tuples/compressedtuple1.cpp

```
#include <algorithm>
#include "tupleelt1.hpp"
#include "tuplestorage3.hpp"
#include <iostream>

struct A
{
    A()
    {
        std::cout << "A()" << '\n';
    }
};

struct B
{
    B()
    {
        std::cout << "B()" << '\n';
    }
};

int main()
{
    Tuple<A, char, A, char, B> t1;
    std::cout << sizeof(t1) << " bytes" << '\n';
}
```

выводит на консоль следующее:

```
A()
A()
B()
5 bytes
```

Оптимизация пустого базового класса убрала один байт (для пустого кортежа `Tuple<>`). Однако обратите внимание на то, что и `A`, и `B` являются пустыми классами, что намекает на еще одну возможность для применения этой оптимизации в `Tuple`. `TupleElt` можно несколько расширить и наследовать от типа элемента в случаях, когда это можно сделать безопасно, без внесения изменений в `Tuple`:

tuples/tupleelt2.hpp

```
#include <type_traits>
template < unsigned Height, typename T,
          bool = std::is_class<T>::value && !std::is_final<T>::value >
class TupleElt;

template<unsigned Height, typename T>
class TupleElt<Height, T, false>
{
    T value;
public:
    TupleElt() = default;
```

```
template<typename U>
TupleElt(U&& other) : value(std::forward<U>(other)) { }

T& get()
{
    return value;
}

T const& get() const
{
    return value;
}

};

template<unsigned Height, typename T>
class TupleElt<Height, T, true> : private T
{
public:
    TupleElt() = default;
    template<typename U>
    TupleElt(U&& other) : T(std::forward<U>(other)) { }

    T& get()
    {
        return *this;
    }

    T const& get() const
    {
        return *this;
    }
};
```

Когда `TupleElt` предоставляется с классом, не являющимся `final`, он закрыто (`private`) наследует этот класс, чтобы обеспечить применение оптимизации пустого базового класса к сохраненному значению. При внесении этого изменения представленная ранее программа теперь выведет в консоль

```
A()
A()
B()
2 bytes
```

25.5.2. `get()` с константным временем работы

Операция `get()` исключительно широко используется при работе с кортежами, но ее рекурсивная реализация требует линейного количества инстанцирований шаблонов, что может повлиять на время компиляции. К счастью, оптимизация пустого базового класса, представленная в предыдущем разделе, обеспечивает более эффективную реализацию `get`, которую мы сейчас рассмотрим.

Ключевым является понимание того, что вывод аргумента шаблона (глава 15, “Вывод аргументов шаблона”) выводит аргументы шаблона для базового класса, когда параметр (тип базового класса) соответствует аргументу (типу производного класса). Таким образом, если мы можем вычислить высоту и элемента, который хотим извлечь, то для извлечения элемента без ручного перебора

всех индексов мы можем полагаться на преобразование специализации Tuple в TupleElt<H, T> (где тип T выводится):

tuples/constantget.hpp

```
template<unsigned H, typename T>
T& getHeight(TupleElt<H, T>& te)
{
    return te.get();
}

template<typename... Types>
class Tuple;

template<unsigned I, typename... Elements>
auto get(Tuple<Elements...>& t)
-> decltype(getHeight < sizeof...(Elements) - I - 1 > (t))
{
    return getHeight < sizeof...(Elements) - I - 1 > (t);
}
```

Поскольку `get<I>(t)` получает индекс I требуемого элемента (который отсчитывается с начала кортежа), в то время как фактически хранение данных в кортеже выполняется в терминах высоты H (которая отсчитывается от конца кортежа), мы вычисляем H из I. Фактический поиск выполняется выводом аргумента шаблона для вызова `getHeight()`: высота H фиксирована, потому что она явно предоставлена в вызове, так что соответствие при выводе будет только у одного базового класса TupleElt, из которого будет выведен тип T. Обратите внимание на то, что `getHeight()` должен быть объявлен другом Tuple, чтобы разрешить преобразование в закрытый базовый класс. Например:

```
// В рекурсивном случае шаблона класса Tuple:
template<unsigned I, typename... Elements>
friend auto get(Tuple<Elements...>& t)
-> decltype(getHeight < sizeof...(Elements) - I - 1 > (t));
```

Обратите внимание на то, что эта реализация требует только константного количества инстанцирований шаблонов, поскольку тяжелая работа по выявлению соответствия индекса передана механизму вывода аргумента шаблона компилятора.

25.6. Индексы кортежа

В принципе, можно также определить `operator[]` для доступа к элементам кортежа, аналогично тому, как `std::vector` определяет оператор `operator[]`¹⁰. Однако в отличие от `std::vector` каждый элемент кортежа может иметь свой тип, поэтому `operator[]` для кортежа должен быть шаблоном, тип результата которого отличается в зависимости от индекса элемента. Это, в свою очередь, требует, чтобы каждый индекс имел собственный тип, так что тип индекса может использоваться для определения типа элемента.

¹⁰ Мы выражаем признательность Луи Дионну (Louis Dionne) за указание на метод, использованный в данном разделе.

Шаблон класса `CTValue`, представленный в разделе 24.3, позволяет закодировать числовой индекс в типе. Мы можем использовать его для определения оператора индексации как члена `Tuple`:

```
template<typename T, T Index>
auto& operator[](CTValue<T, Index>)
{
    return get<Index>(*this);
}
```

Здесь мы используем значение переданного индекса в типе аргумента `CTValue`, чтобы выполнить соответствующий вызов `get<>()`.

Теперь можно использовать этот класс следующим образом:

```
auto t = makeTuple(0, '1', 2.2f, std::string("hello"));
auto a = t[CTValue<unsigned, 2> {}];
auto b = t[CTValue<unsigned, 3> {}];
```

Здесь `a` и `b` будут инициализированы типом и значением третьего и четвертого элементов кортежа `Tuple t`.

Чтобы сделать использование константных индексов более удобным, можно реализовать `constexpr`-оператор литерала (*literal operator with constexpr*) для вычисления числовых литералов времени компиляции непосредственно из обычных литералов с помощью суффикса `_c`:

tuples/literals.hpp

```
#include "ctvalue.hpp"
#include <cassert>
#include <cstddef>

// Преобразование отдельного символа в соответствующее
// значение типа int во время компиляции:
constexpr int.toInt(char c)
{
    // Шестнадцатеричные буквы:
    if (c >= 'A' && c <= 'F')
    {
        return static_cast<int>(c) - static_cast<int>('A') + 10;
    }

    if (c >= 'a' && c <= 'f')
    {
        return static_cast<int>(c) - static_cast<int>('a') + 10;
    }

    // Прочие (отключаем '.' для литералов чисел с плавающей точкой):
    assert(c >= '0' && c <= '9');
    return static_cast<int>(c) - static_cast<int>('0');
}

// Анализ массива символов для преобразования в соответствующее
// целочисленное значение во время компиляции:
template<std::size_t N>
constexpr int parseInt(char const(&arr)[N])
```

```

{
    int base = 10; // Обработка основания (10 по умолчанию)
    int offset = 0; // Для пропуска префиксов наподобие 0x

    if (N > 2 && arr[0] == '0')
    {
        switch (arr[1])
        {
            case 'x': // Префикс 0x или 0X, шестнадцатеричное
            case 'X': // значение
                base = 16;
                offset = 2;
                break;

            case 'b': // Префикс 0b или 0B (начиная с C++14)
            case 'B': // бинарное значение
                base = 2;
                offset = 2;
                break;

            default: // Префикс 0 – восьмеричное значение
                base = 8;
                offset = 1;
                break;
        }
    }

    // Итерация по всем цифрам и вычисление результата:
    int value = 0;
    int multiplier = 1;

    for (std::size_t i = 0; i < N - offset; ++i)
    {
        if (arr[N - 1 - i] != '\'') //忽視單引號
            // кавычками (например, в 1'000)
        {
            value += toInt(arr[N - 1 - i]) * multiplier;
            multiplier *= base;
        }
    }

    return value;
}
// Оператор литерала: анализ целочисленного литерала
// с суффиксом _c как последовательности символов:
template<char... cs>
constexpr auto operator"" _c()
{
    return CTValue<int, parseInt<sizeof...(cs)>({cs...})> {};
}

```

Здесь используется тот факт, что для числовых литералов можно применять оператор литерала для вывода каждого символа литерала как своего собственного параметра шаблона (см. подробнее в разделе 15.5.1). Мы передаем символы во

вспомогательную `constexpr`-функцию `parseInt()`, которая вычисляет значение последовательности символов во время компиляции и выдает его как `CTValue`. Например:

- `42_c` дает `CTValue<int, 42>`
- `0x815_c` дает `CTValue<int, 2069>`
- `0b1111'1111_c` дает `CTValue<int, 255>11`

Обратите внимание на то, что анализатор не обрабатывает литералы с плавающей точкой. Для них выдается ошибка времени компиляции, потому что функция времени выполнения не может использоваться в контексте времени компиляции.

Теперь мы можем использовать кортежи следующим образом:

```
auto t = makeTuple(0, '1', 2.2f, std::string("hello"));
auto c = t[2_c];
auto d = t[3_c];
```

Этот подход использован в Boost.Hana [14], библиотеке метапрограммирования, пригодной для вычислений как с типами, так и со значениями.

25.7. Заключительные замечания

По-видимому, построение кортежей является одним из тех применений шаблонов, которые многие программисты пытались написать самостоятельно. Библиотека Boost.Tuple [18] стала одной из самых популярных библиотек кортежей в C++, и в конечном итоге превратилась в стандартный шаблон C++11 `std::tuple`.

До C++11 многие реализации кортежей основывались на идее рекурсивных парных структур. В первом издании этой книги [71] один такой подход проиллюстрирован с применением “рекурсивных дуэтов”. Еще один интересный вариант был разработан Андреем Александреску в [3]. Он аккуратно отделил список типов от списка полей в кортеже, используя концепцию списков типов, описанную в главе 24, “Списки типов”, в качестве основы для кортежей.

C++11 добавил в язык вариативные шаблоны, пакеты параметров которых могли явно захватывать список типов для кортежа, устранив необходимость рекурсивных пар. Раскрытие пакетов и понятие индексных списков [40] привело к забвению рекурсивного инстанцирования шаблонов для создания кортежей в пользу более простого и эффективного инстанцирования вариативных шаблонов, делающего кортежи более широко применимыми на практике. Индексные списки стали столь критичны к производительности кортежей и алгоритмов для работы со списками типов, что компиляторы включают встроенный шаблон псевдонима, такой как `_make_integer_seq<S, T, N>`, который раскрывается до `S<T, 0, 1, ..., N>` без дополнительных инстанцирований шаблонов, тем самым ускоряя применение `std::make_index_sequence` и `make_integer_sequence`.

¹¹ Префикс `0b` для бинарных литералов и одинарная кавычка в качестве разделителя поддерживаются, начиная с C++14.

Кортеж — это один из наиболее широко используемых гетерогенных контейнеров, но он не единственный. Библиотека Boost.Fusion [13] предоставляет другие обобщенные гетерогенные контейнеры, такие как гетерогенные `list`, `deque`, `set` и `map`. Что еще более важно, она обеспечивает основу для написания алгоритмов для гетерогенных коллекций, используя те же виды абстракций и терминологии, что и стандартная библиотека C++ (например, итераторы, последовательности и контейнеры).

Boost.Hana [14] взяла многие из идей библиотек Boost.MPL [16] и Boost.Fusion, разработанных и реализованных задолго до реализации C++11 (в конечном итоге они оказались в новом стандарте языка C++11 (и C++14)). Результатом явилась элегантная библиотека, предоставляющая мощные компоненты для гетерогенных вычислений.

Глава 26

Контролируемые объединения

Кортежи, разработанные в предыдущей главе, объединяют значения некоторого списка типов в одно значение, обеспечивая примерно такую же функциональность, как и простая структура. С учетом этой аналогии естественно задаться вопросом, чему же соответствует объединение (*union*), которое содержит единственное значение, но это значение имеет тип, выбранный из некоторого множества возможных типов. Например, поле базы данных может содержать целое число, значение с плавающей точкой, строку или бинарный объект, но в любой момент времени оно может содержать значение только одного из этих типов.

В этой главе мы разработаем шаблон класса `Variant`, который динамически хранит значение одного из заданного набора возможных типов, аналогичный шаблону стандартной библиотеки C++17 `std::variant<>`. `Variant` — это *контролируемое объединение* (*discriminated union*), т.е. оно знает, какое из его возможных типов значений в настоящее время активно, и обеспечивает лучшую безопасность типов, чем эквивалентное объединение C++. Сам `Variant` является вариативным шаблоном, принимающим список типов, который может иметь активное значение. Например, переменная

```
Variant<int, double, string> field;
```

может хранить `int`, `double` или `string`, но в каждый момент времени — только одно из этих значений¹. Поведение `Variant` иллюстрируется следующей программой:

```
variant/variant.cpp
```

```
#include "variant.hpp"
#include <iostream>
#include <string>
int main()
{
    Variant<int, double, std::string> field(17);

    if (field.is<int>())
    {
        std::cout << "Field хранит целое значение "
              << field.get<int>() << '\n';
    }
}
```

¹ Обратите внимание: список возможных типов фиксируется в момент объявления `Variant`, следовательно, `Variant` является *закрытым* контролируемым объединением. Открытое контролируемое объединение позволяет хранить значения в объявлении значения дополнительных типов, не известных во время создания контролируемого объединения. Класс `FunctionPtr`, обсуждавшийся в главе 22, “Статический и динамический полиморфизм”, может рассматриваться как разновидность открытого контролируемого объединения.

```

    field = 42;      // Присваивание значения того же типа
    field = "hello"; // Присваивание значения иного типа
    std::cout << "Теперь field хранит строку '"
                  << field.get<std::string>() << "'\n";
}

```

Вывод этого кода имеет следующий вид:

```

Field хранит целое значение 17
Теперь field хранит строку 'hello'

```

Переменной типа `Variant` может быть присвоено значение любого из его типов. Мы можем проверить, содержит ли переменная в настоящее время значение типа `T` с помощью функции-члена `is<T>()`, а затем получить сохраненное значение с помощью функции-члена `get<T>()`.

26.1. Хранилище

Первый главный аспект дизайна нашего типа `Variant` представляет собой управление хранением *активного значения*, т.е. значения, которое в настоящий момент хранится в переменной. Различные типы, вероятно, имеют различные размеры и выравнивание. Кроме того, типу `Variant` необходимо хранить *дискриминатор* (*discriminator*), указывающий, какой из возможных типов является типом активного значения. Один простой (хотя и неэффективный) механизм хранения непосредственно использует кортеж (см. главу 25, “Кортежи”):

`variant/variantstorageastuple.hpp`

```

template<typename... Types>
class Variant
{
public:
    Tuple<Types...> storage;
    unsigned char discriminator;
};

```

Здесь дискриминатор действует как динамический индекс в кортеже. Допустимое значение имеет только тот элемент кортежа, статический индекс которого равен текущему значению дискриминатора, так что когда `discriminator` равен 0, `get<0>(storage)` обеспечивает доступ к активному значению; когда `discriminator` равен 1, доступ к активному значению обеспечивает `get<1>(storage)`, и т. д.

Мы могли бы построить базовый вариант операций `is<T>()` и `get<T>()` поверх кортежа. Однако это довольно неэффективное решение, поскольку сам `Variant` при этом требует количества памяти, равное сумме размеров типов всех возможных значений, даже если одновременно будет активен только один из них².

² При таком подходе имеется много других проблем, таких как вытекающее требование, чтобы все типы в `Types` имели конструктор по умолчанию.

Лучший подход использует перекрытие памяти для всех возможных типов. Мы могли бы осуществить его путем рекурсивного развертывания контролируемого объединения на голову и хвост, как мы делали это для кортежей в разделе 25.1.1, но с помощью объединения, а не класса:

`variant/variantstorageasunion.hpp`

```
template<typename... Types>
union VariantStorage;

template<typename Head, typename... Tail>
union VariantStorage<Head, Tail...>
{
    Head head;
    VariantStorage<Tail...> tail;
};

template<>
union VariantStorage<>
{
};
```

Здесь объединение гарантированно имеет достаточный размер и выравнивание, чтобы позволить хранение любого из типов в `Types` в любой момент времени. К сожалению, с таким объединением довольно трудно работать, потому что большинство методов, которые мы будем использовать для реализации `Variant`, будут применять наследование, которое для объединения является недопустимым.

Вместо этого мы предпочитаем низкоуровневое представление хранилища: массив символов, размер которого достаточен для хранения любых типов, и с выравниванием, подходящим для любого из типов, используемый нами как буфер для хранения активного значения. Шаблон класса `VariantStorage` реализует этот буфер вместе с дискриминатором:

`variant/variantstorage.hpp`

```
#include <new>      // Для std::launder()

template<typename... Types>
class VariantStorage
{
    using LargestT = LargestType<Typelist<Types...>>;
    alignas(Types...) unsigned char buffer[sizeof(LargestT)];
    unsigned char discriminator = 0;
public:
    unsigned char getDiscriminator() const
    {
        return discriminator;
    }
    void setDiscriminator(unsigned char d)
    {
        discriminator = d;
    }
};
```

```

void* getRawBuffer()
{
    return buffer;
}
const void* getRawBuffer() const
{
    return buffer;
}
template<typename T>
T* getBufferAs()
{
    return std::launder(reinterpret_cast<T*>(buffer));
}
template<typename T>
T const* getBufferAs() const
{
    return std::launder(reinterpret_cast<T const*>(buffer));
}
};

```

Здесь мы используем разработанную в разделе 24.2.2 метапрограмму `LargestType` для вычисления размера буфера, гарантируя достаточный его размер для любого из типов значений. Аналогично раскрытие пакета `alignas` гарантирует, что буфер будет иметь выравнивание, подходящее для любого из типов³. Вычисленный нами размер буфера, по сути, представляет собой машинное представление показанного выше объединения. Мы можем получить доступ к узателю на буфер с помощью `getBuffer()` и манипулировать с памятью путем явного приведения типов, размещающего `new` (для создания новых значений) и явного вызова деструкторов (для уничтожения созданных значений). Если вы не знакомы с `std::launder()`, используемым в `getBufferAs()`, то пока что достаточно знать, что он возвращает неизмененным свой аргумент. Мы поясним его роль, когда будем говорить об операторах присваивания для нашего шаблона `Variant` (см. раздел 26.4.3).

26.2. Дизайн

Теперь, когда у нас есть решение проблемы хранения для контролируемых объединений, мы проектируем сам тип `Variant`. Как и с типом `Tuple`, мы используем наследование для предоставления поведения для каждого типа в списке `Types`. Однако в отличие от `Types` эти базовые классы не имеют хранилища. Вместо этого каждой из базовых классов использует *Странно рекурсивный шаблон проектирования* (Curiously Recurring Template Pattern – CRTP), рассмотревшийся в разделе 21.2, для доступа к общему хранилищу через наиболее производный тип.

³Хотя мы и не выбрали этот вариант, но для вычисления максимального выравнивания мы могли бы использовать шаблонную метапрограмму, а не делать это с помощью раскрытия пакета `alignas`. Результат получается одинаковым в любом случае, но приведенная выше формулировка передает вычисление выравнивания компилятору.

Шаблон класса VariantChoice, определенный ниже, предоставляет базовые операции, необходимые для работы с буфером, когда активным значением объединения является (или будет) значение типа T:

variant/variantchoice.hpp

```
#include "findindexof.hpp"

template<typename T, typename... Types>
class VariantChoice
{
    using Derived = Variant<Types...>;
    Derived& getDerived()
    {
        return *static_cast<Derived*>(this);
    }
    Derived const& getDerived() const
    {
        return *static_cast<Derived const*>(this);
    }
protected:
    // Вычисление дискриминатора для данного типа
    constexpr static unsigned Discriminator =
        FindIndexOfT<Typelist<Types...>, T>::value + 1;
public:
    VariantChoice() { }
    VariantChoice(T const& value); // См. variantchoiceinit.hpp
    VariantChoice(T& value); // См. variantchoiceinit.hpp
    bool destroy(); // См. variantchoicedestroy.hpp
    Derived& operator=(T const& value); // См. variantchoiceassign.hpp
    Derived& operator=(T& value); // См. variantchoiceassign.hpp
};
```

Пакет параметров шаблона Types будет содержать все типы в Variant. Это позволяет нам сформировать тип Derived (для CTRP) и таким образом обеспечить операцию нисходящего приведения getDerived(). Второе интересное применение Types заключается в том, чтобы найти размещение конкретного типа T в списке Types, чего мы добиваемся с помощью метафункции FindIndexOfT:

variant/findindexof.hpp

```
template<typename List, typename T, unsigned N = 0,
         bool Empty = IsEmpty<List>::value>
struct FindIndexOfT;
// Рекурсивный случай:
template<typename List, typename T, unsigned N>
struct FindIndexOfT<List, T, N, false>
    : public IfThenElse < std::is_same<Front<List>, T>::value,
      std::integral_constant<unsigned, N>,
      FindIndexOfT < PopFront<List>, T, N + 1 >>
{};

};
```

```
// Базовый случай:
template<typename List, typename T, unsigned N>
struct FindIndexOfT<List, T, N, true>
{
};
```

Это значение индекса используется для вычисления значения дискриминатора, соответствующего `T`; мы вернемся к конкретным значениям дискриминатора позже.

Скелет `Variant`, приведенный ниже, иллюстрирует взаимоотношения `Variant`, `VariantStorage` и `VariantChoice`:

`variant/variant-skel.hpp`

```
template<typename... Types>
class Variant
    : private VariantStorage<Types...>,
    private VariantChoice<Types, Types...>...
{
    template<typename T, typename... OtherTypes>
    friend class VariantChoice; // Включение CRTP
    ...
};
```

Как упоминалось ранее, каждый `Variant` имеет единственный, общий базовый класс `VariantStorage`⁴. Кроме того, он имеет некоторое количество базовых классов `VariantChoice`, которые получаются из следующих раскрытий вложенных пакетов (см. раздел 12.4.4):

`VariantChoice<Types, Types...>...`

В этом случае у нас есть два раскрытия: внешнее раскрытие производит базовый класс `VariantChoice` для каждого типа `T` в `Types` путем раскрытия первой ссылки на `Types`. Внутреннее раскрытие второго вхождения `Types` дополнительно проходит все типы в `Types` для каждого базового класса `VariantChoice`. Для `Variant<int, double, std::string>`

этот дает следующее множество базовых классов `VariantChoice`⁵:

```
VariantChoice<int, int, double, std::string>,
VariantChoice<double, int, double, std::string>,
VariantChoice<std::string, int, double, std::string>
```

⁴ Базовые классы закрыты (private), так как их представление не является частью открытого интерфейса. Дружественный шаблон требуется для того, чтобы позволить функциям `asDerived()` в `VariantChoice` выполнять приведение к `Variant`.

⁵ Одним интересным результатом того, что базовые классы `VariantChoice` данного `Variant` отличаются только типом `T`, является предотвращение дублирования типов. `Variant<double,int,double>` приведет к ошибке компилятора, указывающей, что класс не может непосредственно наследовать один и тот же базовый класс (в данном случае `VariantChoice<double,double,int,double>`) дважды.

Для этих трех базовых классов значениями дискриминаторов будут 1, 2 и 3 соответственно. Когда член `discriminator` хранилища соответствует дискриминатору определенного базового класса `VariantChoice`, этот базовый класс отвечает за управление активным значением.

Нулевое значение дискриминатора зарезервировано для случаев, когда `Variant` не содержит значения, — странное состояние, которое может наблюдаться только при генерации исключения во время присваивания. На протяжении всего обсуждения `Variant` мы будем достаточно внимательны и аккуратны, чтобы справиться с нулевым дискриминатором (и устанавливать его при необходимости), но его обсуждение мы отложим до раздела 26.4.3.

Полное определение `Variant` приведено далее. В следующих разделах будут описаны реализации каждого из членов `Variant`.

`variant/variant.hpp`

```
template<typename... Types>
class Variant
    : private VariantStorage<Types...>,
      private VariantChoice<Types, Types...>...
{
    template<typename T, typename... OtherTypes>
    friend class VariantChoice;
public:
    template<typename T> bool is() const;           // См. variantis.hpp
    template<typename T> T& get() &;                  // См. variantget.hpp
    template<typename T> T const& get() const&; // См. variantget.hpp
    template<typename T> T&& get() &&;            // См. variantget.hpp

    // См. variantvisit.hpp:
    template<typename R = ComputedResultType, typename Visitor>
        VisitResult<R, Visitor, Types...> visit(Visitor && vis) &;
    template<typename R = ComputedResultType, typename Visitor>
        VisitResult<R, Visitor, Types const& ...>
            visit(Visitor && vis) const&;
    template<typename R = ComputedResultType, typename Visitor>
        VisitResult < R, Visitor, Types && ... >
            visit(Visitor && vis) &&;
    using VariantChoice<Types, Types...>::VariantChoice...;
    Variant();                                         // См. variantdefaultctor.hpp
    Variant(Variant const& source); // См. variantcopyctor.hpp
    Variant(Variant&& source); // См. variantmovector.hpp
    template<typename... SourceTypes>
        Variant(Variant<SourceTypes...> const&
                source); // См. variantcopyctortmpl.hpp
    template<typename... SourceTypes>
        Variant(Variant<SourceTypes...>&& source);
    using VariantChoice<Types, Types...>::operator=...;
    Variant& operator=(Variant const&
                        source); // См. variantcopyassign.hpp
    Variant& operator=(Variant&& source);
    template<typename... SourceTypes>
        Variant& operator=(Variant<SourceTypes...> const& source);
```

```

template<typename... SourceTypes>
    Variant& operator=(Variant<SourceTypes...>&& source);
bool empty() const;
~Variant()
{
    destroy();
}
void destroy();                                // См. variantdestroy.hpp
};

```

26.3. Запрос и извлечение значения

Фундаментальными запросами для типа Variant являются запрос, имеет ли его активное значение определенный тип T, и доступ к активному значению, когда его тип известен. Функция-член `is()`, определяемая ниже, выясняет, хранит ли Variant в настоящее время значение типа T:

variant/variantis.hpp

```

template<typename... Types>
template<typename T>
bool Variant<Types...>::is() const
{
    return this->getDiscriminator() ==
           VariantChoice<T, Types...>::Discriminator;
}

```

Для данной переменной v, значение `v.is<int>()` выясняет, имеет ли активное значение переменной тип int. Проверка реализована очень просто — она сравнивает значение дискриминатора со значением дискриминатора соответствующего базового класса VariantChoice.

Если искомый тип T не найден в списке, базовый класс VariantChoice не будет инстанцирован, поскольку `FindIndexOfT` не будет содержать член `value`, что приведет к (преднамеренному) сбою компиляции `is<T>()`. Это предотвращает ошибку пользователя, когда он запрашивает тип, который не может храниться в контролируемом объединении.

Функция-член `get()` извлекает ссылку на хранимое значение. Ей должен быть передан извлекаемый тип (например, `v.get<int>()`), и она корректна только тогда, когда активное значение имеет этот указанный тип:

variant/variantget.hpp

```

#include <exception>

class EmptyVariant : public std::exception
{
};

template<typename... Types>
template<typename T>

```

```

T& Variant<Types...>::get() &
{
    if (empty())
    {
        throw EmptyVariant();
    }

    assert(is<T>());
    return *this->template getBufferAs<T>();
}

```

Когда Variant не имеет значения (его дискриминатор равен 0), метод `get()` генерирует исключение `EmptyVariant`. Условия, при которых дискриминатор может быть нулевым, сами возникают из-за исключения (см. раздел 26.4.3). Другие попытки получить значение некорректного типа являются ошибками программиста, обнаруживаемые с помощью `assert()`.

26.4. Инициализация, присваивание и уничтожение элементов

Когда активное значение имеет тип `T`, за обработку его инициализации, присваивания и уничтожения отвечает базовый класс `VariantChoice`. В данном разделе эти фундаментальные операции рассматриваются при заполнении соответствующих пробелов шаблона класса `VariantChoice`.

26.4.1. Инстанцирования

Мы начинаем с инициализации `Variant` с помощью значения одного из типов, которые он может хранить (например, инициализация `Variant<int, double, string>` значением `double`). Это достигается с помощью конструкторов `VariantChoice`, которые принимают значение типа `T`:

`variant/variantchoiceinit.hpp`

```

#include <utility> // for std::move()

template<typename T, typename... Types>
VariantChoice<T, Types...>::VariantChoice(T const& value)
{
    // Размещение значения в буфере
    // и установка дискриминатора типа:
    new (getDerived().getRawBuffer()) T(value);
    getDerived().setDiscriminator(Discriminator);
}

template<typename T, typename... Types>
VariantChoice<T, Types...>::VariantChoice(T&& value)
{
    // Размещение перемещаемого значения в буфере
    // и установка дискриминатора типа:
    new (getDerived().getRawBuffer()) T(std::move(value));
    getDerived().setDiscriminator(Discriminator);
}

```

В каждом случае конструктор использует операцию CRTP `getDerived()` для доступа к совместно используемому буферу, а затем выполняет размещающий оператор `new` для инициализации хранилища новым значением типа `T`. Первый конструктор является копирующим конструктором, в то время как второй — перемещающим конструктором⁶. Затем конструкторы устанавливают значение дискриминатора для указания (динамического) типа хранимого значения.

Наша конечная цель состоит в получении возможности инициализировать объединение значением любого из его типов, даже учитывая неявные преобразования. Например:

```
// Неявное преобразование в string:
Variant<int, double, string> v("hello");
```

Для этого мы наследуем конструкторы `VariantChoice` в самом `Variant` с использованием объявлений `using`⁷:

```
using VariantChoice<Types, Types...>::VariantChoice...;
```

По сути, это объявление `using` создает конструкторы `Variant`, которые копируют или перемещают каждый тип `T` в `Types`. Для `Variant<int, double, string>` такими конструкторами являются:

```
Variant(int const&);
Variant(int&&);
Variant(double const&);
Variant(double&&);
Variant(string const&);
Variant(string&&);
```

26.4.2. Уничтожение

Когда `Variant` инициализируется, значение создается в его буфере. Операция `destroy` выполняет уничтожение этого значения:

`variant/variantchoicedestroy.hpp`

```
template<typename T, typename... Types>
bool VariantChoice<T, Types...>::destroy()
{
    if (getDerived().getDiscriminator() == Discriminator)
    {
        // Если типы совпадают, вызывается размещающее удаление:
        getDerived().template getBufferAs<T>() ->~T();
        return true;
    }

    return false;
}
```

⁶ Применение конструкторов запрещает использование ссылочных типов в нашем варианте `Variant`. Это ограничение можно обойти, оборачивая ссылки в класс наподобие `std::reference_wrapper`.

⁷ Использование раскрытия пакета в объявлении `using` (раздел 4.4.5) появилось в C++17. До этого наследование этих конструкторов требовало рекурсивной схемы наследования, подобной дизайну `Tuple` в главе 25, “Кортежи”.

При соответствии дискриминаторов мы явным образом уничтожаем содержимое буфера путем вызова соответствующего деструктора с использованием `->~T()`.

Операция `VariantChoice::destroy()` полезна только тогда, когда дискриминатор совпадает. Тем не менее в общем случае мы хотим уничтожить значение, хранящееся в `Variant`, независимо от того, какой тип в настоящее время активен. Поэтому `Variant::destroy()` вызывает *все* операции `VariantChoice::destroy()` в базовых классах:

`variant/variantdestroy.hpp`

```
template<typename... Types>
void Variant<Types...>::destroy()
{
    // Вызов destroy() для каждого базового класса VariantChoice;
    // успешным будет максимум один вызов:
    bool results[] =
    {
        VariantChoice<Types, Types...>::destroy()...
    };
    // Указывает, что объект не хранит никакого значения
    this->setDiscriminator(0);
}
```

Раскрытие пакета в инициализаторе `results` гарантирует, что `destroy` вызывается для каждого из базовых классов `VariantChoice`. Успешным будет максимум один из этих вызовов (с подходящим дискриминатором), после чего объект остается пустым. Пустое состояние указывается нулевым значением дискриминатора.

Массив `results` сам по себе присутствует только для того, чтобы обеспечить контекст для использования списка инициализации; его фактические значения игнорируются. В C++17 для устранения необходимости в этой лишней переменной можно использовать выражения свертки (рассматриваются в разделе 12.4.6):

`variant/variantdestroy17.hpp`

```
template<typename... Types>
void Variant<Types...>::destroy()
{
    // Вызов destroy() для каждого базового класса VariantChoice;
    // успешным будет максимум один вызов:
    (VariantChoice<Types, Types...>::destroy(), ...);
    // Указывает, что объект не хранит никакого значения
    this->setDiscriminator(0);
}
```

26.4.3. Присваивание

Присваивание основывается на инициализации и уничтожении, что иллюстрируется операторами присваивания:

variant/variantchoiceassign.hpp

```

template<typename T, typename... Types>
auto VariantChoice<T, Types...>::operator=(T const& value) -> Derived&
{
    if (getDerived().getDiscriminator() == Discriminator)
    {
        // Присваивание нового значения того же типа:
        *getDerived().template getBufferAs<T>() = value;
    }
    else
    {
        // Присваивание нового значения другого типа:
        getDerived().destroy();           // destroy() для всех типов
        new (getDerived().getRawBuffer())
            T(value);                  // Размещение нового значения
        getDerived().setDiscriminator(Discriminator);
    }

    return getDerived();
}

template<typename T, typename... Types>
auto VariantChoice<T, Types...>::operator= (T&& value) -> Derived&
{
    if (getDerived().getDiscriminator() == Discriminator)
    {
        // Присваивание нового значения того же типа:
        *getDerived().template getBufferAs<T>() = std::move(value);
    }
    else
    {
        // Присваивание нового значения другого типа:
        getDerived().destroy();           // destroy() для всех типов
        new (getDerived().getRawBuffer())
            T(std::move(value));        // Размещение нового значения
        getDerived().setDiscriminator(Discriminator);
    }

    return getDerived();
}

```

Как и в случае с инициализацией одним из типов хранимых значений, каждый VariantChoice предоставляет собственный оператор присваивания, который копирует (или перемещает) значение своего типа в хранилище Variant. Эти операторы присваивания наследуются типом Variant с помощью следующего объявления using:

```
using VariantChoice<Types, Types...>::operator=...;
```

Реализация оператора присваивания имеет два пути. Если объект Variant уже хранит значение данного типа T (идентифицируется путем соответствия дискриминатора), то оператор присваивания будет копировать (или перемещать) значение типа T непосредственно в буфер. Дискриминатор при этом не изменяется.

Если в объекте хранится значение типа, отличного от T, присваивание выполняется в два этапа: сначала текущее значение уничтожается с помощью вызова `Variant::destroy()`, а затем новое значение типа T инициализируется с помощью *размещающего new* с соответствующим изменением дискриминатора.

При таком двухэтапном присваивании с помощью размещающего new возникают три распространенные проблемы, которые следует принять во внимание:

- присваивание самому себе;
- исключения;
- `std::launder()`.

Присваивание самому себе

Присваивание самому себе может произойти из-за выражения наподобие следующего:

```
v = v.get<T>()
```

При такой реализации двухэтапного присваивания, как показано выше, исходное значение будет уничтожено, прежде чем оно будет скопировано, что может привести к повреждению памяти. К счастью, присваивание самому себе подразумевает соответствие дискриминатора, так что такой код будет просто вызывать оператор присваивания для T, а не выполнять оба этапа присваивания.

Исключения

Если уничтожение существующего значения завершается, но при инициализации нового значения генерируется исключение, — каким при этом будет состояние объекта Variant? В нашей реализации `Variant::destroy()` сбрасывает значение дискриминатора в нулевое. В случаях без генерации исключения дискриминатор будет должным образом установлен после завершения инициализации. Если же исключение генерируется во время инициализации нового значения, дискриминатор остается нулевым, указывая, что Variant не хранит никакого значения. В нашем дизайне это единственный способ получить Variant без значения.

Приведенная ниже программа показывает, как получить Variant без хранимого значения путем попытки копировать значение типа, копирующий конструктор которого генерирует исключение:

```
variant/variantexception.cpp
```

```
#include "variant.hpp"
#include <exception>
#include <iostream>
#include <string>

class CopiedNonCopyable : public std::exception
{};

class VariantTest
{
public:
    VariantTest() = default;
    VariantTest(CopiedNonCopyable const& value) : value_(value) {}

    void print() const
    {
        std::cout << value_.what();
    }

private:
    CopiedNonCopyable value_;
};
```

```

class NonCopyable
{
public:
    NonCopyable()
    {
    }
    NonCopyable(NonCopyable const&)
    {
        throw CopiedNonCopyable();
    }
    NonCopyable(NonCopyable&&) = default;
    NonCopyable& operator= (NonCopyable const&)
    {
        throw CopiedNonCopyable();
    }
    NonCopyable& operator= (NonCopyable&&) = default;
};

int main()
{
    Variant<int, NonCopyable> v(17);

    try
    {
        NonCopyable nc;
        v = nc;
    }
    catch (CopiedNonCopyable)
    {
        std::cout << "Копирующее присваивание "
        << "NonCopyable завершилось неудачно." << '\n';

        if (!v.is<int>() && !v.is<NonCopyable>())
        {
            std::cout << "Variant не имеет значения." << '\n';
        }
    }
}

```

Вывод этой программы имеет следующий вид:

Копирующее присваивание NonCopyable завершилось неудачно.
Variant не имеет значения.

Доступ к объекту Variant, не содержащему значения, независимо от способа (через метод `get()` или механизм посетителя (`visitor`), описанный в следующем разделе) генерирует исключение `EmptyVariant`, которое позволяет программам восстановиться после этой исключительной ситуации. Функция-член `empty()` проверяет, не является ли объект пустым:

`variant/variantempty.hpp`

```

template<typename... Types>
bool Variant<Types...>::empty() const
{
    return this->getDiscriminator() == 0;
}

```

Третья проблема, связанная с нашим двухэтапным присваиванием, заключается в тонкости, о которой комитету по стандартизации C++ стало известно только в конце процесса стандартизации C++17. Далее мы вкратце обсудим ее.

std::launder()

В общем случае компиляторы C++ направлены на создание высокопроизводительного кода, и, вероятно, основным механизмом повышения производительности генерированного кода является устранение многократного копирования данных из памяти в регистры. Для этого компилятор должен делать определенные предположения, в частности о том, что некоторые виды данных остаются неизменными все время их существования. Сюда включаются константные данные, ссылки (которые могут быть инициализированы, но не могут быть впоследствии изменены) и часть системных данных, хранящихся в полиморфных объектах и используемых для диспетчеризации виртуальных функций, локализации виртуальных базовых классов, а также классы обработки операторов `typeid` и `dynamic_cast`.

Проблема, связанная с рассматривавшейся выше двухэтапной процедурой присваивания, заключается в том, что скрыто заканчивается время существования одного объекта и начинается время жизни другого объекта в том же месте, таким образом, что компилятор может быть не способен это распознать. Следовательно, компилятор может предположить, что значение, полученное из предыдущего состояния объекта `Variant` все еще является действительным, в то время как фактически инициализация с размещающим `new` делает его недействительным. Без решения этой проблемы выполнение программы с использованием типа `Variant` с неизменяемыми членами-данными иногда может приводить к некорректным результатам в случае ее оптимизации для достижения высокой производительности. Такие ошибки, как правило, очень трудно отслеживать (части потому, что они встречаются редко, а отчасти потому, что в действительности они не видны в исходном коде).

Начиная с C++17, решением этой проблемы является доступ к адресу нового объекта через вызов `std::launder()`, который просто возвращает свой аргумент, но при этом заставляет компилятор распознать, что полученный адрес указывает на объект, который может отличаться от предположений компилятора об аргументе, передаваемом `std::launder()`. Однако учтите, что `std::launder()` фиксирует только возвращаемый адрес, но не аргумент, передаваемый `std::launder()`, потому что компиляторы “рассуждают” в терминах выражений, а не фактических адресов (так как последние не существуют до времени выполнения). Таким образом, после создания нового значения с помощью размещающего `new` мы должны гарантировать, что каждое следующее обращение будет использовать “очищенные”⁸ данные. Вот почему в `Variant` мы всегда “отмываем” указатель буфера. Есть возможность поступить немного лучше (например, добавляя дополнительный член-указатель, который указывает на буфер и получает “очищенный” адрес после каждого присваивания нового значения

⁸ Launder (англ.) — стирать, очищать. — Примеч. пер.

с размещающим `new`), но код при этом усложняется и становится трудным для сопровождения. Наш подход прост и корректен, пока мы обращаемся к буферу исключительно через члены `getBufferAs()`.

Ситуация с `std::laundrer()` не является полностью удовлетворительной: вопрос это очень тонкий, трудно воспринимаемый (например, мы не заметили его до самой отправки книги в печать) и трудно решаемый (`std::laundrer()` не очень прост в использовании). Поэтому несколько членов Комитета потребовали проделать дополнительную работу по поиску более удовлетворительного решения. Подробнее эта проблема описана в [44].

26.5. Посетители

Функции-члены `is()` и `get()` позволяют проверить, имеет ли действующее значение определенный тип, и получить доступ к значению с этим типом. Однако проверка всех возможных типов в пределах варианта быстро переходит в избыточную цепочку инструкций `if`. Например, следующий фрагмент выводит значение `Variant<int, double, string>` с именем `v`:

```
if (v.is<int>())
{
    std::cout << v.get<int>();
}
else if (v.is<double>())
{
    std::cout << v.get<double>();
}
else
{
    std::cout << v.get<string>();
}
```

Обобщение этого кода для вывода значения, хранящегося в произвольном объединении, требует рекурсивно инстанцируемого шаблона основной и вспомогательной функции. Например:

`variant/printrec.cpp`

```
#include "variant.hpp"
#include <iostream>

template<typename V, typename Head, typename... Tail>
void printImpl(V const& v)
{
    if (v.template is<Head>())
    {
        std::cout << v.template get<Head>();
    }
    else if constexpr(sizeof...(Tail) > 0)
    {
        printImpl<V, Tail...>(v);
    }
}
```

```

template<typename... Types>
void print(Variant<Types...> const& v)
{
    printImpl<Variant<Types...>, Types...>(v);
}

int main()
{
    Variant<int, short, float, double> v(1.5);
    print(v);
}

```

Это значительное количество кода для относительно простой операции. Чтобы его упростить, рассмотрим проблему расширения Variant операцией visit(). Затем клиент передает функциональный объект-посетитель, operator() которого будет вызываться с активным значением. Поскольку активное значение может быть любого из возможных типов Variant, этот operator(), вероятно, должен либо быть перегруженным, либо являться шаблоном функции. Например, универсальное лямбда-выражение обеспечивает шаблонный operator(), позволяющий нам лаконично представлять операцию вывода объединения v:

```

v.visit([](auto const& value)
{
    std::cout << value;
});

```

Это обобщенное лямбда-выражение примерно эквивалентно следующему функциональному объекту, полезному для компиляторов, если они еще не поддерживают обобщенные лямбда-выражения:

```

class VariantPrinter
{
public:
    template<typename T>
    void operator()(T const& value) const
    {
        std::cout << value;
    }
};

```

Ядро операции visit() похоже на рекурсивную операцию print: она обходит типы Variant, проверяя, имеет ли активное значение данный тип (с помощью `is<T>()`), а найдя подходящий тип, выполняет соответствующие действия:

`variant/variantvisitimpl.hpp`

```

template<typename R, typename V, typename Visitor,
         typename Head, typename... Tail>
R variantVisitImpl(V& variant, Visitor&& vis,
                    TypeList<Head, Tail...>)
{
    if (variant.template is<Head>())
    {

```

```

        return static_cast<R>(
            std::forward<Visitor>(vis) (
                std::forward<V>(variant).template get<Head>()));
    }
    else if constexpr(sizeof...(Tail) > 0)
    {
        return variantVisitImpl<R>(std::forward<V>(variant),
            std::forward<Visitor>(vis),
            TypeList<Tail...>());
    }
    else
    {
        throw EmptyVariant();
    }
}

```

`variantVisitImpl()` представляет собой шаблон свободной функции (не являющейся членом класса) с рядом параметров шаблона. Параметр шаблона `R` описывает тип результата операции посещения, к которому мы вернемся позднее. `V` является типом объединения, а `Visitor` представляет собой тип посетителя. `Head` и `Tail` используются для разделения типов `Variant` для эффективной рекурсии.

Первый `if` выполняет проверку (времени выполнения), имеет ли действующее значение данного `Variant` тип `Head`: если да, то значение извлекается из `Variant` с помощью `get<Head>()` и передается посетителю, завершая рекурсию. Второй `if` выполняет рекурсию, когда имеются иные элементы для рассмотрения. Если ни один из типов не подошел — значит, `Variant` не содержит значение⁹, и в этом случае реализация генерирует исключение `EmptyVariant`.

Не считая вычисления типа результата, предоставляемого `VisitResult` (будет обсуждаться в следующем разделе), реализация `visit()` проста:

`variant/variantvisit.hpp`

```

template<typename... Types>
template<typename R, typename Visitor>
VisitResult<R, Visitor, Types& ...>
Variant<Types...>::visit(Visitor&& vis) &
{
    using Result = VisitResult<R, Visitor, Types& ...>;
    return variantVisitImpl<Result>(*this, std::forward<Visitor>(vis),
        TypeList<Types...>());
}

template<typename... Types>
template<typename R, typename Visitor>
VisitResult<R, Visitor, Types const& ...>
Variant<Types...>::visit(Visitor&& vis) const &
{
    using Result = VisitResult<R, Visitor, Types const& ...>;
    return variantVisitImpl<Result>(*this, std::forward<Visitor>(vis),
        TypeList<Types...>());
}

```

⁹Этот случай подробно рассматривается в разделе 26.4.3.

```

template<typename... Types>
template<typename R, typename Visitor>
VisitResult < R, Visitor, Types&& ... >
Variant<Types...>::visit(Visitor&& vis)&&
{
    using Result = VisitResult < R, Visitor, Types && ... >;
    return variantVisitImpl<Result>(std::move(*this),
                                      std::forward<Visitor>(vis),
                                      Typelist<Types...>());
}

```

Реализации выполняют непосредственное делегирование функции `variantVisitImpl`, передавая ей само объединение, посетителя (с помощью прямой передачи) и полный перечень типов. Единственное различие между тремя приведенными реализациями заключается в том, как они передают объединение — как `Variant&`, `Variant const&` или `Variant &&`.

26.5.1. Возвращаемый тип `visit()`

Возвращаемый тип `visit()` остается загадкой. Данный посетитель может иметь различные перегрузки `operator()`, которые производят различные возвращаемые типы, шаблонные операторы `operator()`, возвращаемый тип которых зависит от типов параметров или их сочетания. Например, рассмотрим следующее обобщенное лямбда-выражение:

```
[ ](auto const& value)
{
    return value + 1;
}
```

Возвращаемый тип этого лямбда-выражения зависит от входного типа: для `int` оно дает `int`, но для `double` оно дает `double`. Если передать это обобщенное лямбда-выражение операции `visit()` объединения `Variant<int, double>`, то каков должен быть тип результата?

Единственного правильного ответа нет, поэтому наша операция `visit()` позволяет явно указывать тип результата. Например, может потребоваться захватить результат в другой `Variant<int, double>`. Можно явно указать тип результата `visit()` в качестве первого аргумента шаблона:

```
v.visit<Variant<int, double>>([ ](auto const& value)
{
    return value + 1;
});
```

Возможность явно указать возвращаемый тип важна, когда нет универсального решения. Однако требование явно указывать тип результата во всех случаях может давать излишне многословный код. Поэтому `visit()` предоставляет оба варианта, используя сочетание аргумента шаблона по умолчанию и простой макропрограммы. Вспомните объявление `visit()`:

```
template<typename R = ComputedResultType, typename Visitor>
VisitResult<R, Visitor, Types...> visit(Visitor && vis) &;
```

Параметр шаблона R, который мы явно указываем в приведенном выше примере, также имеет аргумент по умолчанию, так что он не всегда обязан быть указан явно. Этот аргумент по умолчанию представляет собой неполный тип ограничителя ComputedResultType:

```
class ComputedResultType;
```

Для вычисления своего возвращаемого типа visit() проходит по всем параметрам шаблона VisitResult — шаблона псевдонима, который обеспечивает доступ к новому свойству типа VisitResultT:

```
variant/variantvisitresult.hpp
```

```
// Явно указанный возвращаемый тип посетителя:
template<typename R, typename Visitor, typename... ElementTypes>
class VisitResultT
{
public:
    using Type = R;
};

template<typename R, typename Visitor, typename... ElementTypes>
using VisitResult =
    typename VisitResultT<R, Visitor, ElementTypes...>::Type;
```

Основное определение VisitResultT обрабатывает случаи, когда явно указан аргумент R, так что Type определен как R. Отдельная частичная специализация применяется, когда R получает аргумент по умолчанию, ComputedResultType:

```
template<typename Visitor, typename... ElementTypes>
class VisitResultT<ComputedResultType, Visitor, ElementTypes...>
{
    ...
}
```

Эта частичная специализация отвечает за вычисление соответствующего возвращаемого типа для общего случая и является предметом рассмотрения следующего раздела.

26.5.2. Общий возвращаемый тип

Как при вызове посетителя, который может производить различные возвращаемые типы для каждого из типов элементов объединения, можно объединить эти типы в один-единственный возвращаемый тип visit()? Есть некоторые очевидные случаи — если посетитель возвращает один и тот же тип для каждого типа элемента, то он должен быть возвращаемым типом visit().

C++ уже имеет понятие общего возвращаемого типа, которое было представлено в разделе 1.3.3: в тернарном выражении `b?x:y` тип выражения является *общим типом* для типов x и y. Например, если x имеет тип int, а y имеет

тип `double`, то общим типом является тип `double`, потому что `int` повышается до `double`. Можно представить понятие общего типа в виде свойства типа:

`variant/commontype.hpp`

```
using std::declval;
template<typename T, typename U>
class CommonTypeT
{
public:
    using Type = decltype(true ? declval<T>() : declval<U>());
};

template<typename T, typename U>
using CommonType = typename CommonTypeT<T, U>::Type;
```

Понятие общего типа расширяется для множества типов: общий тип — это тип, к которому могут быть расширены все типы множества. Для нашего посетителя мы хотим вычислить общий тип для возвращаемых типов, которые посетитель будет производить при вызове с каждым из типов объединения:

`variant/variantvisitresultcommon.hpp`

```
#include "accumulate.hpp"
#include "commontype.hpp"

// Возвращаемый тип, полученный при вызове
// посетителя со значением типа T:
template<typename Visitor, typename T>
using VisitElementResult = decltype(declval<Visitor>()(declval<T>()));

// Общий возвращаемый тип для посетителя,
// вызванного с каждым из данных типов элементов:
template<typename Visitor, typename... ElementTypes>
class VisitResultT<ComputedResultType, Visitor, ElementTypes...>
{
    using ResultTypes =
        Typelist<VisitElementResult<Visitor, ElementTypes>...>;
public:
    using Type =
        Accumulate<PopFront<ResultTypes>, CommonTypeT,
        Front<ResultTypes>>;
};
```

Вычисление `VisitResult` выполняется в два этапа. Сначала `VisitElementResult` вычисляет возвращаемый тип, создаваемый при вызове посетителя со значением типа `T`. Эта метафункция применяется к каждому из данных типов элементов, чтобы определить все возвращаемые типы, которые может производить посетитель, и захватывает результат в список типов `ResultTypes`.

Затем в вычислениях используется алгоритм `Accumulate`, описанный в разделе 24.2.6, для того, чтобы применить вычисление общего типа к списку возвращаемых типов. Его начальное значение (третий аргумент `Accumulate`)

представляет собой первый возвращаемый тип, посредством CommonTypeT объединяется с последовательными значениями из остальной части списка типа ResultTypes. Конечным результатом является общий тип, к которому могут быть приведены все возвращаемые типы посетителей, или ошибка, если возвращаемые типы несовместимы.

Начиная с C++11, стандартная библиотека предоставляет соответствующее свойство типов, std::common_type<>, которое, эффективно комбинируя CommonTypeT и Accumulate, использует этот подход для того, чтобы получить общий тип для произвольного количества переданных типов (см. раздел Г.5). С помощью std::common_type<> реализация VisitResultT упрощается:

variant/variantvisitresultstd.hpp

```
template<typename Visitor, typename... ElementTypes>
class VisitResultT<ComputedResultType, Visitor, ElementTypes...>
{
public:
    using Type = std::common_type_t<
        VisitElementResult<Visitor, ElementTypes>...>;
};
```

Следующая демонстрационная программа выводит тип, полученный путем передачи посетителю обобщенного лямбда-выражения, которое добавляет 1 к получаемому значению:

variant/visit.cpp

```
#include "variant.hpp"
#include <iostream>
#include <typeinfo>

int main()
{
    Variant<int, short, double, float> v(1.5);
    auto result = v.visit([](auto const & value)
    {
        return value + 1;
    });
    std::cout << typeid(result).name() << '\n';
}
```

Выходом этой программы будет имя type_info для double, потому что это тип, к которому могут быть преобразованы все возвращаемые типы.

26.6. Инициализация и присваивание Variant

Контролируемые объединения можно инициализировать и присваивать различными способами, включая конструирование по умолчанию, копирующее и перемещающее конструирования, а также копирующее и перемещающее присваивания. В этом разделе подробно описаны эти операции Variant.

Инициализация по умолчанию

Должны ли контролируемые объединения предоставлять конструкторы по умолчанию? Если нет, то такие объединения может быть излишне трудно использовать, потому что они всегда должны будут иметь начальное значение (даже когда программно оно не имеет смысла). Если же оно предоставляет конструктор по умолчанию, то какова должна быть его семантика?

Одна возможная семантика состоит в том, чтобы для инициализации по умолчанию иметь пустое объединение, представленное нулевым дискриминатором. Однако такие пустые объединения в общем случае полезными не являются (например, их нельзя посещать или искать значения для извлечения), а кроме того, такой режим инициализации по умолчанию будет способствовать тому, что уникальное состояние пустого объединения (описанное в разделе 26.4.3) превратится в достаточно распространенное.

В качестве альтернативы конструктор по умолчанию может строить значение *некоторого* типа. Для нашего объединения мы следуем семантике `std::variant`<> C++17 и конструируем по умолчанию значение первого типа в списке типов:

`variant/variantdefaultctor.hpp`

```
template<typename... Types>
Variant<Types...>::Variant()
{
    *this = Front<Typelist<Types...>>();
}
```

Этот подход прост и предсказуем, и в большинстве случаев позволяет избежать введения пустых объединений. Его поведение демонстрирует следующая программа:

`variant/variantdefaultctor.cpp`

```
#include "variant.hpp"
#include <iostream>

int main()
{
    Variant<int, double> v;

    if (v.is<int>())
    {
        std::cout << "Построенный по умолчанию v хранит int "
                  << v.get<int>() << '\n';
    }

    Variant<double, int> v2;

    if (v2.is<double>())
    {
        std::cout << "Построенный по умолчанию v2 хранит double "
                  << v2.get<double>() << '\n';
    }
}
```

Вывод данной программы имеет следующий вид:

```
Построенный по умолчанию v хранит int 0
Построенный по умолчанию v2 хранит double 0
```

Копирующая/перемещающая инициализация

Копирующая и перемещающая инициализация более интересна. Для копирования исходного объединения нужно определить, какой тип хранится в настоящее время, создать с помощью копирующего конструирования это значение в буфере и установить значение дискриминатора. К счастью, `visit()` обрабатывает декодирование активного значения исходного объединения, а оператор копирующего присваивания, унаследованный от `VariantChoice`, создает копию значения в буфере, что позволяет написать компактную реализацию¹⁰:

variant/variantcopyctor.hpp

```
template<typename... Types>
Variant<Types...>::Variant(Variant const& source)
{
    if (!source.empty())
    {
        source.visit([&] (auto const & value)
        {
            *this = value;
        });
    }
}
```

Перемещающий конструктор отличается только тем, что использует `std::move` при посещении исходного объединения и выполняет перемещающее присваивание из исходного значения:

variant/variantmovector.hpp

```
template<typename... Types>
Variant<Types...>::Variant(Variant&& source)
{
    if (!source.empty())
    {
        std::move(source).visit([&] (auto&& value)
        {
            *this = std::move(value);
        });
    }
}
```

¹⁰ Несмотря на синтаксис, использующий оператор присваивания (=) в лямбда-выражении, фактические реализации оператора присваивания в `VariantChoice` будут выполнять копирующее конструирование, поскольку изначально объединение не хранит значения.

Один из самых интересных аспектов реализации на основе идиомы посетителя заключается в том, что он работает и для шаблонных вариантов операций копирования и перемещения. Например, шаблонный копирующий конструктор может быть определен следующим образом:

variant/variantcopyctortmpl.hpp

```
template<typename... Types>
template<typename... SourceTypes>
Variant<Types...>::Variant(Variant<SourceTypes...> const& source)
{
    if (!source.empty())
    {
        source.visit([&](auto const & value)
        {
            *this = value;
        });
    }
}
```

Поскольку этот код посещает исходное объединение, присваивание `*this` будет выполняться для каждого из его типов. Разрешение перегрузки для этого присваивания будет находить наиболее подходящий целевой тип для каждого исходного типа, выполняя при необходимости неявные преобразования. В следующем примере показано построение и присваивание из различных типов объединения:

variant/variantpromote.cpp

```
#include "variant.hpp"
#include <iostream>
#include <string>

int main()
{
    Variant<short, float, char const*> v1((short)123);
    Variant<int, std::string, double> v2(v1);
    std::cout << "v2 содержит int " << v2.get<int>() << '\n';
    v1 = 3.14f;
    Variant<double, int, std::string> v3(std::move(v1));
    std::cout << "v3 содержит double " << v3.get<double>() << '\n';
    v1 = "hello";
    Variant<double, int, std::string> v4(std::move(v1));
    std::cout << "v4 содержит string " << v4.get<std::string>() <<
        '\n';
}
```

Построение или присваивание `v1` переменным `v2` или `v3` включает целочисленное расширение (`short` в `int`), расширение значений с плавающей точкой (`float` в `double`) и пользовательские преобразования (`char const*` в `std::string`). Выход этой программы имеет следующий вид:

```
v2 содержит int 123
v3 содержит double 3.14
v4 содержит string hello
```

Присваивание

Операторы присваивания Variant похожи на копирующий и перемещающий конструкторы, показанные выше. Здесь мы приводим только оператор копирующего присваивания:

variant/variantcopyassign.hpp

```
template<typename... Types>
Variant<Types...>& Variant<Types...>::operator= (Variant const&
    source)
{
    if (!source.empty())
    {
        source.visit([&] (auto const & value)
        {
            *this = value;
        });
    }
    else
    {
        destroy();
    }

    return *this;
}
```

Единственное интересное дополнение находится в ветви `else`: когда исходное объединение не содержит значения (на что указывает нулевой дискриминатор), мы уничтожаем значение, неявно устанавливая его дискриминатор равным 0.

26.7. Заключительные замечания

Андрей Александреску подробно рассмотрел контролируемые объединения в серии статей [4]. Наше рассмотрение объединения Variant основано на ряде подобных методов, таких как выровненные буферы для хранения значений и применение идиомы посетителя для извлечения значений. Некоторые различия обусловлены базовым языком: Александреску работал с C++98, так что, например, он не мог использовать вариативные шаблоны или унаследованные конструкторы. Он также уделил много времени вычислению выравнивания, которое в C++11 стало тривиальным с введением `alignas`. Наиболее интересная разница в дизайне заключается в обработке дискриминатора: в то время как мы решили использовать целочисленный дискриминатор для указания того, какой тип в настоящее время хранится в объединении, Андрей использовал подход со “статической vtable”, используя указатели на функции для создания, копирования, запроса и уничтожения типа базового элемента. Интересно, что этот подход имел большее влияние в качестве метода оптимизации для открытых объединений, таких как шаблон `FunctionPtr`, разработанный в разделе 22.2, и является распространенной оптимизацией для реализаций `std::function`, устраняющих

использование виртуальных функций. Тип Boost any [11] является другим примером типа контролируемого объединения, который был принят в стандартной библиотеке C++17 как `std::any`.

Позже библиотеки Boost [10] представили несколько типов контролируемых объединений [23], которые оказали влияние на тип, разрабатываемый в данной главе. Проектная документация Boost.Variant [23] включает в себя живое обсуждение вопросов безопасности исключений при присваивании (“гарантия непустоты объединения”) и различных решений, не полностью удовлетворяющих разработчиков.

В отличие от нашего шаблона Variant шаблон `std::variant` допускает несколько одинаковых аргументов шаблона (например, `std::variant<int, int>`). Включение этой функциональности в Variant потребовало бы значительных изменений в нашей конструкции, включая добавление метода для однозначного определения базовых классов VariantChoice и альтернативы раскрытиям вложенных пакетов, описанным в разделе 26.2.

Вариант операции `visit()`, описанный в этой главе, структурно идентичен схеме посетителя, описываемой Александреску в [5]. Посетитель Александреску позволяет упростить процесс проверки указателя на некоторый общий базовый класс, не указывает ли он на некоторый известный производный класс из набора (описываемого как список типов). Эта реализация для проверки указателей использует `dynamic_cast`, вызывая посетителя с указателем на производный класс при обнаружении совпадения.

Глава 27

Шаблоны выражений

В этой главе описывается метод шаблонного программирования, известный под названием *шаблоны выражений* (expression templates). Впервые этот метод возник при работе с классами числовых массивов, и именно в таком контексте он рассматривается в данной главе.

Класс числовых массивов поддерживает выполнение числовых операций с массивом как с целостным объектом. Например, к двум массивам можно применить операцию сложения; в результате ее выполнения получим массив, каждый элемент которого будет суммой соответствующих элементов исходных массивов. Аналогично массив можно умножить на скаляр. Это означает, что на скаляр умножается каждый элемент исходного массива. Возникает естественное желание реализовать для массивов операторные обозначения, так знакомые нам по работе со встроенными скалярными типами:

```
Array<double> x(1000), y(1000);  
***  
x = 1.2 * x + x * y;
```

Для серьезных высокопроизводительных вычислительных комплексов важно, чтобы выражения такого рода вычислялись настолько эффективно, насколько можно ожидать от платформы, на которой выполняется этот код. Добиться того, чтобы такие вычисления можно было выполнять с помощью компактной записи, приведенной в рассмотренном выше примере, — задача далеко не тривиальная, однако при ее решении на помощь приходят шаблоны выражений.

Шаблоны выражений напоминают шаблонное метапрограммирование. Частично это объясняется тем, что иногда они опираются на глубоко вложенное инстанцирование шаблонов, которое не отличается от рекурсивного инстанцирования, встречающегося в шаблонном метапрограммировании. Не исключено, что на сходство этих двух методов также повлияло то, что оба они первоначально разрабатывались для поддержки высокопроизводительных операций с массивами (см. пример применения шаблонов для развертывания циклов в разделе 23.1.3). Эти два метода, несомненно, дополняют друг друга. Например, метапрограммирование удобно применять для небольших массивов фиксированного размера, а шаблоны выражений весьма эффективны для выполнения операций со средними и большими массивами, размеры которых задаются во время работы программы.

27.1. Временные объекты и раздельные циклы

Чтобы обосновать применение шаблонов выражений, рассмотрим простейший, прямолинейный (возможно, даже наивный) подход к реализации шаблонов, позволяющих выполнять операции с числовыми массивами. Первичный

шаблон массива мог бы выглядеть, как показано ниже (имя SArray обозначает *простой массив* – simple array).

`exprtmp/sarray1.hpp`

```
#include <cstddef>
#include <cassert>
template<typename T>
class SArray
{
public:
    // Создание массива с начальным размером
    explicit SArray(std::size_t s)
        : storage(new T[s]), storage_size(s)
    {
        init();
    }
    // Копирующий конструктор
    SArray(SArray<T> const& orig)
        : storage(new T[orig.size()]), storage_size(orig.size())
    {
        copy(orig);
    }
    // Деструктор: освобождение памяти
    ~SArray()
    {
        delete[] storage;
    }
    // Оператор присваивания
    SArray<T>& operator= (SArray<T> const& orig)
    {
        if (&orig != this)
        {
            copy(orig);
        }

        return *this;
    }
    // Возврат размера
    std::size_t size() const
    {
        return storage_size;
    }
    // Оператор индекса для констант и переменных
    T const& operator[](std::size_t idx) const
    {
        return storage[idx];
    }
    T& operator[](std::size_t idx)
    {
        return storage[idx];
    }
protected:
    // Инициализация значений конструктором по умолчанию
    void init()
```

```

    {
        for (std::size_t idx = 0; idx < size(); ++idx)
        {
            storage[idx] = T();
        }
    }
    // Копирование значений другого массива
    void copy(SArray<T> const& orig)
    {
        assert(size() == orig.size());

        for (std::size_t idx = 0; idx < size(); ++idx)
        {
            storage[idx] = orig.storage[idx];
        }
    }
private:
    T* storage;           // Память для элементов
    std::size_t storage_size; // Количество элементов
};

```

Числовые операторы можно закодировать, как показано ниже:

exprtmp/sarrayops1.hpp

```

// Сложение двух SArrays
template<typename T>
SArray<T> operator+ (SArray<T> const& a, SArray<T> const& b)
{
    assert(a.size() == b.size());
    SArray<T> result(a.size());

    for (std::size_t k = 0; k < a.size(); ++k)
    {
        result[k] = a[k] + b[k];
    }

    return result;
}
// Умножение двух SArrays
template<typename T>
SArray<T> operator* (SArray<T> const& a, SArray<T> const& b)
{
    assert(a.size() == b.size());
    SArray<T> result(a.size());

    for (std::size_t k = 0; k < a.size(); ++k)
    {
        result[k] = a[k] * b[k];
    }

    return result;
}
// Перемножение скаляра и SArray
template<typename T>
SArray<T> operator* (T const& s, SArray<T> const& a)

```

```

{
    SArray<T> result(a.size());
    for (std::size_t k = 0; k < a.size(); ++k)
    {
        result[k] = s * a[k];
    }
    return result;
}
// Умножение SArray и скаляра
// Сложение скаляра и SArray
// Сложение SArray и скаляра
...

```

Можно было бы написать еще много версий этих и других операторов, однако для нашего примера достаточно того, что есть.

`exprtmp1/sarray1.cpp`

```

#include "sarray1.hpp"
#include "sarrayops1.hpp"

int main()
{
    SArray<double> x(1000), y(1000);
    ...
    x = 1.2 * x + x * y;
}

```

Однако оказывается, что приведенная выше реализация операторов крайне неэффективна. Тому есть две причины.

- При каждом применении оператора (за исключением оператора присваивания) создается как минимум один временный массив. Это означает, что в нашем примере создается по крайней мере три временных массива, в каждом из которых содержится по 1 000 элементов (это при условии, что компилятор выполняет все допустимые устраниния временных копий).
- При каждом применении оператора компилятору требуется выполнить дополнительный обход массивов-аргументов и результирующих массивов. В нашем примере в предположении генерации только трех временных объектов SArray это приводит к необходимости считывания приблизительно 6 000 и записи около 4 000 значений типа `double`.

Оценим количество операций, которые выполняются в циклах, обрабатывающих временные массивы.

```

tmp1 = 1.2*x;      // Цикл из 1000 операций
                  // плюс создание и удаление массива tmp1
tmp2 = x*y;        // Цикл из 1000 операций
                  // плюс создание и удаление массива tmp2

```

```

tmp3 = tmp1+tmp2; // Цикл из 1000 операций
                  // плюс создание и удаление массива tmp3
x = tmp3;          // 1000 операций считывания и
                  // 1000 операций записи

```

Если в компиляторе не применяется специальный высокопроизводительный распределитель памяти, то при работе с небольшими массивами основное время тратится на создание ненужных временных массивов. Для выполнения операций с очень большими массивами использование временных массивов абсолютно недопустимо, поскольку может оказаться, что их негде хранить. (Чтобы получить достоверные результаты при численном моделировании реальных систем, часто расходуется вся доступная память. Если же она будет применяться для хранения ненужных временных массивов, это приведет к снижению точности расчетов.)

В ранних реализациях численных библиотек, предназначенных для работы с массивами, эта проблема оставалась нерешенной, и пользователям рекомендовалось применять операторы вычисления с присваиванием (такие как `+=`, `*=` и т.д.). Их преимущество состоит в том, что в них массив является одновременно и аргументом, и целевым массивом, поэтому они не требуют временных массивов. Например, два объекта класса `SArray` можно было бы сложить так, как показано ниже.

`exprtmp1/sarrayops2.hpp`

```

// Сложение SArray с присваиванием
template<typename T>
SArray<T>& SArray<T>::operator+= (SArray<T> const& b)
{
    assert(size() == orig.size());

    for (std::size_t k = 0; k < size(); ++k)
    {
        (*this)[k] += b[k];
    }

    return *this;
}

// Умножение SArray с присваиванием
template<typename T>
SArray<T>& SArray<T>::operator*=( SArray<T> const& b)
{
    assert(size() == orig.size());

    for (std::size_t k = 0; k < size(); ++k)
    {
        (*this)[k] *= b[k];
    }

    return *this;
}

// Скалярное умножение SArray с присваиванием
template<typename T>
SArray<T>& SArray<T>::operator*=( T const& s )

```

```
{
    for (std::size_t k = 0; k < size(); ++k)
    {
        (*this)[k] *= s;
    }

    return *this;
}
```

С помощью подобных операторов наш вычислительный пример можно было бы переписать следующим образом:

`exprtmp1/sarray2.cpp`

```
#include "sarray2.hpp"
#include "sarrayops1.hpp"
#include "sarrayops2.hpp"

int main()
{
    SArray<double> x(1000), y(1000);
    /**
     * Вычисление x = 1.2*x + x*y
     */
    SArray<double> tmp(x);
    tmp *= y;
    x *= 1.2;
    x += tmp;
}
```

Понятно, что метод, при котором используются только операторы присваивания с вычислением, тоже не оправдывает наших ожиданий. Он обладает рядом недостатков:

- обозначения становятся громоздкими;
- полностью избавиться от ненужных временных массивов не удается;
- в теле цикла выполняется множество операций; в результате для его выполнения требуется выполнить около 6 000 считываний и 4 000 записей значений типа `double`.

На самом деле нам нужен некий *единий* “идеальный” цикл, в котором бы все выражение вычислялось для каждого индекса.

```
int main()
{
    SArray<double> x(1000), y(1000);
    /**
     */

    for (int idx = 0; idx < x.size(); ++idx)
    {
        x[idx] = 1.2 * x[idx] + x[idx] * y[idx];
    }
}
```

На этот раз удалось обойтись без временных массивов и ограничиться в каждой итерации цикла всего двумя операциями считывания (элементов массивов

$x[idx]$ и $y[idx]$) и одной операцией записи в память ($x[idx]$). В результате для выполнения всего цикла требуется выполнить около 2 000 считываний и 1 000 записей в память.

На современных высокопроизводительных компьютерных архитектурах пропускная способность памяти является ограничивающим фактором, снижающим скорость выполнения операций с массивами. Поэтому неудивительно, если на практике окажется, что производительность запрограммированного “вручную” цикла на один-два порядка выше, чем производительность продемонстрированного только что подхода, при котором применяется такая простая перегрузка оператора. Однако желательно добиться такой же производительности более элегантным путем, избегая непонятных обозначений или громоздкого кода, в котором легко допустить ошибку.

27.2. Программирование выражений в аргументах шаблонов

Ключом к решению нашей проблемы является попытка вычислить выражение не по частям, а сразу, откладывая вычисления до момента, пока не будет просмотрено все выражение (в нашем примере — до вызова оператора присваивания). Таким образом, перед вычислением нужно записать, какая операция к какому объекту будет применяться. Эти операции определяются во время компиляции, поэтому они могут быть запрограммированы как аргументы шаблонов.

Для нашего выражения

```
1.2*x + x*y;
```

это означает, что результат умножения $1.2*x$ — это не новый массив, а объект, в котором *каждое значение x умножается на 1.2*. Аналогично, в результате операции $x*y$ получается объект, в котором *каждый элемент массива x умножается на соответствующий элемент массива y*. И наконец, когда нам понадобятся значения результирующего массива, выполняются все упомянутые (и отложенные) вычисления.

Рассмотрим конкретную реализацию сформулированной выше программы действий. Наша реализация вычисляет выражение

```
1.2*x + x*y;
```

в объект следующего типа:

```
A_Add<A_Mult<A_Scalar<double>, Array<double>>,
A_Mult<Array<double>, Array<double>>>
```

В приведенных строках кода новый фундаментальный шаблон класса `Array` комбинируется с шаблонами классов `A_Scalar`, `A_Add` и `A_Mult`. Сопоставьте расположение имен шаблонов с синтаксическим деревом, соответствующим рассматриваемому выражению (рис. 27.1). Этот вложенный идентификатор шаблона представляет типы участвующих в вычислении объектов и операции над ними. Шаблон скалярного типа `A_Scalar` будет приведен ниже; по сути, это просто заместитель для фигурирующего в выражении скаляра.

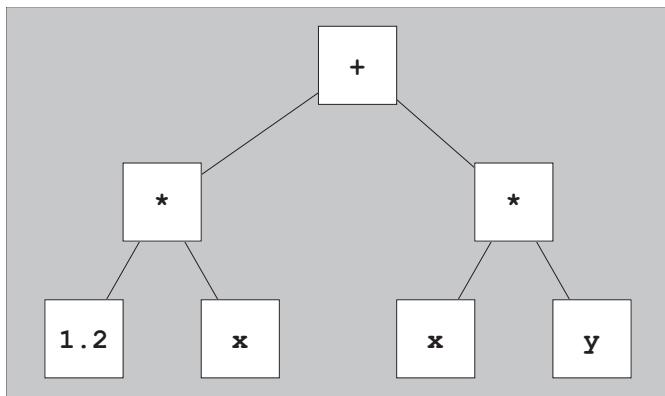


Рис. 27.1. Представление выражения $1.2 * x + x * y$ в виде дерева

27.2.1. Операнды шаблонов выражений

Чтобы придать завершенный вид приведенному выше представлению выражения, необходимо создать ссылки на каждый из объектов классов A_Add и A_Mult, а также записать значение в объект класса A_Scalar (или ссылку на это значение). Ниже приведен пример определения соответствующих операндов.

exprtmp1/exprops1.hpp

```
#include <cstddef>
#include <cassert>

// Включение вспомогательного шаблона класса свойств для
// выбора вида обращения к узлу шаблона выражения — по
// ссылке или по значению
#include "exprops1a.hpp"

// Класс для объектов, представляющих сложение двух операндов
template<typename T, typename OP1, typename OP2>
class A_Add
{
private:
    typename A_Traits<OP1>::ExprRef op1; // Первый операнд
    typename A_Traits<OP2>::ExprRef op2; // Второй операнд
public:
    // Конструктор, инициализирующий ссылки на операнды
    A_Add(OP1 const& a, OP2 const& b)
        : op1(a), op2(b)
    {
    }
    // Вычисление суммы по запросу
    T operator[](std::size_t idx) const
    {
        return op1[idx] + op2[idx];
    }
    // size — максимальный размер
    std::size_t size() const
{}
```

```

    {
        assert(op1.size() == 0 || op2.size() == 0
               || op1.size() == op2.size());
        return op1.size() != 0 ? op1.size() : op2.size();
    }
};

// Класс для объектов, представляющих умножение двух операндов
template<typename T, typename OP1, typename OP2>
class A_Mult
{
private:
    typename A_Traits<OP1>::ExprRef op1; // Первый операнд
    typename A_Traits<OP2>::ExprRef op2; // Второй операнд
public:
    // Конструктор, инициализирующий ссылки на операнды
    A_Mult(OP1 const & a, OP2 const & b)
        : op1(a), op2(b)
    {
    }
    // Вычисление произведения по запросу
    T operator[](std::size_t idx) const
    {
        return op1[idx] * op2[idx];
    }
    // size – максимальный размер
    std::size_t size() const
    {
        assert(op1.size() == 0 || op2.size() == 0
               || op1.size() == op2.size());
        return op1.size() != 0 ? op1.size() : op2.size();
    }
};

```

Как видите, здесь добавлены операция индексации и операция получения размера. Это позволит определять тип массива и значения его элементов, что понадобится в процессе выполнения операций, представленных “узлами” того поддерева, корнем которого является данный объект.

Если в операции принимают участие только массивы, то размер результирующего объекта равен размеру любого из двух операндов. Если же операция выполняется над массивом и скаляром, в результате получим объект, размер которого равен операнду-массиву. Чтобы отличить операнд-массив от скалярного операнда, размер последнего задан равным нулю. Поэтому шаблон `A_Scalar` определяется так, как показано ниже:

`exprtmp1/exprscalar.hpp`

```

// Класс для объектов, представляющих скаляры:
template<typename T>
class A_Scalar

```

```

{
    private:
        T const& s; // Значение скаляра
    public:
        // Конструктор, инициализирующий значение
        constexpr A_Scalar(T const & v)
            : s(v)
        {
        }
        // В индексных операциях скаляр представляет
        // собой значение каждого элемента
        constexpr T const& operator[](std::size_t) const
        {
            return s;
        }
        // Размер скаляра - нулевой
        constexpr std::size_t size() const
        {
            return 0;
        };
    };
}

```

(Мы объявили конструктор и функции-члены как `constexpr`, так что этот класс может быть использован во время компиляции. Однако для наших целей это не является строго необходимым.)

Заметим, что наши скаляры допускают использование операции индексации. Скаляр при этом можно рассматривать как массив с одинаковыми значениями всех элементов, равных значению скаляра.

Вероятно, вы обратили внимание на то, что в классах операторов для определения операндов-членов применяется вспомогательный класс `A_Traits`:

```
typename A_Traits<OP1>::ExprRef op1; // Первый операнд
typename A_Traits<OP2>::ExprRef op2; // Второй операнд
```

Это необходимо по следующей причине: в общем случае операнды можно объявить как ссылки, поскольку большинство временных узлов иерархической структуры вычисляемого выражения связаны с выражением верхнего уровня и потому остаются в памяти до завершения вычисления всего выражения. Единственным исключением являются узлы `A_Scalar`. Они фигурируют в операторных функциях и могут не оставаться в памяти до завершения вычисления всего выражения. Таким образом, чтобы избежать ссылок на уже не существующие скалярные объекты, скалярные операнды должны быть скопированы по значению.

Другими словами, нужны операнды-члены, обладающие такими свойствами:

- в общем случае являются ссылками на константы:

```
OP1 const& op1; // Обращение к первому операнду по ссылке
OP2 const& op2; // Обращение ко второму операнду по ссылке
```

- но для скаляров это обычные значения:

```
OP1 op1; // Обращение к первому операнду по значению
OP2 op2; // Обращение ко второму операнду по значению
```

Здесь отлично подходят классы свойств. В классе свойств тип в общем случае определяется как ссылка на константу, а в случае скаляра — как обычное значение:

`exprtmp1/exprops1a.hpp`

```
// Вспомогательный класс свойств, помогающий выбрать способ
// обращения к узловому объекту шаблона выражения
//   - в общем случае — по ссылке;
//   - для скаляров — по значению
template<typename T> class A_Scalar;

// Первичный шаблон
template<typename T>
class A_Traits
{
public:
    using ExprRef = T const&; // Тип — ссылка на константу
};

// Частичная специализация для скаляров
template<typename T>
class A_Traits<A_Scalar<T>>
{
public:
    using ExprRef = A_Scalar<T>; // Тип — обычное значение
};
```

Обратите внимание: поскольку объекты `A_Scalar` в выражении верхнего уровня обращаются к скалярам, эти скаляры могут использовать ссылочные типы, т.е. `A_Scalar<T>::s` представляет собой член-ссылку.

27.2.2. Тип `Array`

Теперь, когда мы умеем кодировать выражения с использованием простых шаблонов выражений, необходимо создать тип `Array`, который бы занимался реальным хранением данных и располагал информацией о шаблонах выражений. Конечно, с инженерной точки зрения было бы хорошо, если бы его интерфейс был как можно более похож на интерфейс обычного массива, и чтобы имелся интерфейс для представления выражений, которые определяют результирующие значения массива. Для этого объявим шаблон `Array` следующим образом:

```
template<typename T, typename Rep = SArray<T>>
class Array;
```

Тип `Rep` может быть типом `SArray`, если `Array` представляет собой обычный массив¹, либо быть вложенным идентификатором шаблона наподобие `A_Add` или `A_Mult`, который кодирует выражение. В любом случае мы имеем дело с инстанцированием класса `Array`, что значительно упрощает дальнейшие действия. Фактически, чтобы различить два этих случая, даже не нужны специализации

¹ Здесь было бы удобно повторно использовать разработанный ранее класс `SArray`, но в библиотеках промышленного назначения может быть предпочтительнее целевая реализация, в которой все особенности класса `SArray` не нужны.

определений шаблона `Array`, хотя при подстановке вместо `Rep` типов наподобие `A_Mult` некоторые из членов не могут быть инстанцированы.

Ниже приведено определение шаблона `Array`. Его функциональные возможности ограничены и мало отличаются от функциональных возможностей шаблона `SArray`. Однако если принцип работы кода понятен, то расширение возможностей шаблона не представит затруднений.

`exprtmpl/exprarray.hpp`

```
#include <cstddef>
#include <cassert>
#include "sarray1.hpp"

template<typename T, typename Rep = SArray<T>>
class Array
{
private:
    Rep expr_rep; // (Доступ к) данным массива
public:
    // Создание массива с начальным размером
    explicit Array(std::size_t s)
        : expr_rep(s)
    {
    }
    // Создание массива из возможного представления
    Array(Rep const& rb)
        : expr_rep(rb)
    {
    }
    // Оператор присваивания для того же типа
    Array& operator= (Array const& b)
    {
        assert(size() == b.size());

        for (std::size_t idx = 0; idx < b.size(); ++idx)
        {
            expr_rep[idx] = b[idx];
        }

        return *this;
    }
    // Оператор присваивания для массивов другого типа
    template<typename T2, typename Rep2>
    Array& operator= (Array<T2, Rep2> const& b)
    {
        assert(size() == b.size());

        for (std::size_t idx = 0; idx < b.size(); ++idx)
        {
            expr_rep[idx] = b[idx];
        }

        return *this;
    }
}
```

```

// size – размер представленных данных
std::size_t size() const
{
    return expr_rep.size();
}
// Оператор индекса для констант и переменных
decltype(auto) operator[](std::size_t idx) const
{
    assert(idx < size());
    return expr_rep[idx];
}
T& operator[](std::size_t idx)
{
    assert(idx < size());
    return expr_rep[idx];
}
// Возврат текущего представления массива
Rep const& rep() const
{
    return expr_rep;
}
Rep& rep()
{
    return expr_rep;
}
} ;

```

Как видно из представленного фрагмента кода, многие операции просто передаются в лежащий в основе объект класса Rep. Однако при копировании другого массива необходимо принимать во внимание, что этот другой массив на самом деле может быть создан на основе шаблона выражения. Таким образом, эта операция копирования параметризуется в терминах представления, лежащего в основе типа Rep.

Оператор индекса заслуживает небольшого отдельного обсуждения. Обратите внимание на то, что константная версия этого оператора использует выведенный тип возвращаемого значения вместо более традиционного типа T const&. Мы поступаем так, поскольку, если Rep представляет A_Mult или A_Add, его оператор индекса возвращает временное значение (т.е. *prvalue*), которое не может быть возвращено по ссылке (и decltype(auto) в этом случае выведет нессылочный тип). С другой стороны, если Rep представляет собой SArray<T>, то базовый оператор индекса создает константное *l-значение*, и в этом случае выводимый возвращаемый тип будет соответствующей константной ссылкой.

27.2.3. Операторы

Теперь у нас есть почти все, что необходимо для реализации эффективных числовых операторов при работе с шаблоном Array, за исключением самих операторов. Как уже отмечалось, эти операторы только собирают в единое выражение шаблонные объекты, сами результирующие массивы они не вычисляют.

Необходимо реализовать по три версии каждого обычного бинарного оператора для таких пар operandов: массив-массив, массив-скаляр и скаляр-массив. В частности, чтобы получить возможность вычислять выражение, приведенное в качестве примера в начале данной главы, понадобятся такие операторы:

exprtmpl/exprops2.hpp

```
// Сложение двух Array:
template<typename T, typename R1, typename R2>
Array<T, A_Add<T, R1, R2>>
operator+ (Array<T, R1> const& a, Array<T, R2> const& b)
{
    return Array<T, A_Add<T, R1, R2>>
        (A_Add<T, R1, R2>(a.rep(), b.rep()));
}

// Умножение двух Array:
template<typename T, typename R1, typename R2>
Array<T, A_Mult<T, R1, R2>>
operator* (Array<T, R1> const& a, Array<T, R2> const& b)
{
    return Array<T, A_Mult<T, R1, R2>>
        (A_Mult<T, R1, R2>(a.rep(), b.rep()));
}

// Умножение скаляра и Array:
template<typename T, typename R2>
Array<T, A_Mult<T, A_Scalar<T>, R2>>
operator* (T const& s, Array<T, R2> const& b)
{
    return Array<T, A_Mult<T, A_Scalar<T>, R2>>
        (A_Mult<T, A_Scalar<T>, R2>(A_Scalar<T>(s), b.rep()));
}

// Умножение Array и скаляра, сложение скаляра и Array,
// сложение Array и скаляра:
***
```

Как видно из рассмотренных примеров, объявление этих операторов является громоздким, однако нельзя сказать, чтобы в самих функциях выполнялось много действий. Например, в операторе суммирования двух массивов сначала создается объект класса `A_Add<>`, представляющий этот оператор и operandы

`A_Add<T,R1,R2>(a.rep(),b.rep())`

Затем этот объект помещается в объект класса `Array`. После этого результат можно использовать, как любой другой объект, с помощью которого представляются содержащиеся в массиве данные:

`return Array<T,A_Add<T,R1,R2>> (...);`

При вычислении скалярного произведения для создания объекта класса `A_Mult` применяется шаблон `A_Scalar`:

`A_Mult<T,A_Scalar<T>,R2>(A_Scalar<T>(s), b.rep())`

После этого результат снова помещается в объект класса `Array`:

```
return Array<T, A_Mult<T, A_Scalar<T>, R2>> (...);
```

Другие бинарные операторы, не являющиеся членами класса, настолько похожи друг на друга, что для большинства из них можно применить макросы с небольшим количеством исходного кода. С помощью других (меньших) макросов можно реализовать унарные операторы, не являющиеся членами.

27.2.4. Подведем итог

При первом знакомстве с понятием шаблонов выражений сложная взаимосвязь различных объявлений и определений может отпугивать. Поэтому проведем нисходящий обзор процессов, которые происходят при работе рассматриваемого примера. Анализируемый код выглядит следующим образом (его можно найти в файле `meta/exprmain.cpp`):

```
int main()
{
    Array<double> x(1000), y(1000);
    /**
     * x = 1.2 * x + x * y;
    */
}
```

Поскольку в объявлении объектов `x` и `y` аргумент `Rep` опущен, по умолчанию принимается, что это объекты класса `SArray<double>`. Таким образом, `x` и `y` — это массивы, в которых хранятся “реальные” значения, а не записи операций.

При анализе выражения

```
1.2*x + x*y
```

компилятор сначала применяет левую операцию `*`, операндами которой являются скаляр и массив. Поэтому в процессе разрешения перегрузки оператора умножения выбирается такая его форма:

```
template<typename T, typename R2>
Array<T, A_Mult<T, A_Scalar<T>, R2>>
operator* (T const& s, Array<T, R2> const& b)
{
    return Array<T, A_Mult<T, A_Scalar<T>, R2>>
        (A_Mult<T, A_Scalar<T>, R2>(A_Scalar<T>(s), b.rep()));
}
```

Типами operandов являются `double` и `Array<double, SArray<double>>`. Таким образом, тип результата представляет собой

```
Array<double, A_Mult<double, A_Scalar<double>, SArray<double>>>
```

Результирующее значение использует объект `A_Scalar<double>`, созданный из значения 1.2 типа `double`, и представление `SArray<double>` объекта `x`.

Затем компилятор переходит к обработке операции умножения `x*y`, операндами которой являются два массива. Для этого применяется соответствующая форма перегруженного оператора `operator*`:

```
template<typename T, typename R1, typename R2>
Array<T, A_Mult<T, R1, R2>>
operator* (Array<T, R1> const& a, Array<T, R2> const& b)
{
    return Array<T, A_Mult<T, R1, R2>>
        (A_Mult<T, R1, R2>(a.rep(), b.rep()));
}
```

Оба операнда имеют тип `Array<double, SArray<double>>`, поэтому тип результата следующий:

```
Array<double, A_Mult<double, SArray<double>, SArray<double>>>
```

На этот раз объект `A_Mult` использует два представления класса `SArray<double>`: одно для массива `x`, второе для массива `y`.

Наконец, выполняется операция `+`. Ее operandами, как и operandами предыдущей операции, являются два массива, типы которых получены в результате предыдущих операций. Таким образом, к паре массивов применяется оператор `+` такого вида:

```
template<typename T, typename R1, typename R2>
Array<T, A_Add<T, R1, R2>>
operator+ (Array<T, R1> const& a, Array<T, R2> const& b)
{
    return Array<T, A_Add<T, R1, R2>>
        (A_Add<T, R1, R2>(a.rep(), b.rep()));
}
```

Вместо параметра `T` подставляется `double`, в то время как вместо `R1` подставляется

```
A_Mult<double, A_Scalar<double>, SArray<double>>
```

`R2` заменяется на

```
A_Mult<double, SArray<double>, SArray<double>>
```

Таким образом, справа от знака присваивания находится выражение, имеющее такой тип:

```
Array<double,
      A_Add<double,
      A_Mult<double, A_Scalar<double>, SArray<double>>,
      A_Mult<double, SArray<double>, SArray<double>>>
```

Этот тип соответствует шаблону оператора присваивания из шаблона `Array`:

```
template<typename T, typename Rep = SArray<T>>
class Array
{
public:
    /**
     * Оператор присваивания для массивов отличающихся типов
     */
    template<typename T2, typename Rep2>
    Array& operator= (Array<T2, Rep2> const& b)
    {
        assert(size() == b.size());
```

```

        for (std::size_t idx = 0; idx < b.size(); ++idx)
        {
            expr_rep[idx] = b[idx];
        }
        return *this;
    }
    ...
}:

```

В результате выполнения оператора присваивания вычисляется каждый элемент целевого массива x . При этом оператор индекса применяется к представлению правой части, тип которой следующий:

```

A_Add<double,
    A_Mult<double, A_Scalar<double>, SArray<double>>,
    A_Mult<double, SArray<double>, SArray<double>> >

```

Детальный разбор результатов работы этого оператора показывает, что для данного индекса idx он вычисляет выражение

$$(1.2*x[idx]) + (x[idx]*y[idx])$$

которое представляет собой именно то, что нам нужно.

27.2.5. Присваивание шаблонам выражений

Для элемента массива с аргументом Rep , который в рассматриваемом примере входит в шаблоны выражений A_Mult и A_Add , невозможно инстанцировать операции записи (действительно, выражение $a+b=c$ не имеет смысла). Однако вполне допустимо написать другие шаблоны выражений, для которых присваивание результата возможно. Например, если массив состоит из целочисленных элементов, то он может играть роль индексов другого массива. Другими словами, выражение

$$x[y] = 2*x[y];$$

должно рассматриваться как

```

for (std::size_t idx = 0; idx < y.size(); ++idx)
{
    x[y[idx]] = 2 * x[y[idx]];
}

```

Чтобы можно было реализовать подобную операцию, в основании класса $Array$ должен находиться шаблон выражения, который ведет себя как l -значение (т.е. позволяет выполнить в него запись). Составные части такого шаблона выражения принципиально не отличаются, например, от шаблона A_Mult . Единственное важное отличие состоит в том, что для операторов индексации реализуется версия с модификатором $const$ и без него, и что эти операторы возвращают l -значения (ссылки):

exprtmp1/exprops3.hpp

<pre> template<typename T, typename A1, typename A2> class A_Subscript </pre>

```

{
public:
    // Конструктор инициализирует ссылки на операнды
    A_Subscript(A1 const& a, A2 const& b)
        : a1(a), a2(b)
    {
    }
    // Выполнение индексации при запросе значения
    decltype(auto) operator[](std::size_t idx) const
    {
        return a1[a2[idx]];
    }
    T& operator[](std::size_t idx)
    {
        return a1[a2[idx]];
    }
    // size – размер внутреннего массива
    std::size_t size() const
    {
        return a2.size();
    }
private:
    A1 const& a1; // Ссылка на первый operand
    A2 const& a2; // Ссылка на второй operand
};

```

Здесь `decltype(auto)` вновь помогает нам обработать индексацию массивов независимо от того, производит базовое представление `rvalue` или `lvalue`.

Для реализации предложенного ранее расширения оператора, возвращающего элемент массива по индексу, потребуется добавить в шаблон класса `Array` дополнительные операторы индекса. Один из этих операторов можно определить как показано ниже (по-видимому, понадобится также его версия с модификатором `const`):

```

exp/tmp/exprops4.hpp
template<typename T, typename R>
template<typename T2, typename R2>
Array<T, A_Subscript<T, R, R2>>
Array<T, R>::operator[](Array<T2, R2> const& b)
{
    return Array<T, A_Subscript<T, R, R2>>
        (A_Subscript<T, R, R2>(*this, b));
}

```

27.3. Производительность и ограничения шаблонов выражений

Чтобы убедить читателя в оправданности идеи шаблонов выражений, в этой главе доказывалось, что такие шаблоны позволяют значительно повысить производительность выполнения операций с массивами. Если проследить за обработкой шаблонов выражений, то можно обнаружить, что в них множество

встроенных функций вызывают друг друга, и что при этом в стеке размещается множество шаблонов объектов. Чтобы получился код, работающий так же хорошо, как написанный вручную, оптимизатор должен выполнять полное встраивание и исключение небольших объектов. Во времена первого издания книги редко встречались компиляторы C++, способные выполнять эти операции. Но с тех пор ситуация существенно изменилась, несомненно, частично и из-за того, что эта методика оказалась популярной.

Метод шаблонных выражений не решает всех проблем, возникающих при выполнении численных операций с массивами. Например, он не работает при умножении матрицы на вектор, записанном в виде

```
x = A*x;
```

где x — это вектор-столбец, а A — квадратная матрица $n \times n$. Без временного массива здесь не обойтись, поскольку каждый элемент результирующего массива зависит от каждого элемента исходного массива. К сожалению, в цикле, заданном в шаблоне выражения, первый элемент массива x тут же изменится, и в вычислении последующих значений этого массива будет участвовать уже не исходное, а измененное значение $x[0]$, что приведет к неправильному результату. С другой стороны, для вычисления несколько отличающегося выражения

```
x = A*y;
```

временный массив не нужен, если только переменные x и y не являются псевдонимами один для другого. Это означает, что во время выполнения такой операции необходимо знать о том, в каких взаимоотношениях находятся операнды. Для этого создается динамическая структура времени выполнения, представляющая дерево выражения вместо его кодирования в типе шаблона выражения. Впервые такой подход был применен Робертом Дэвисом (Robert Davies) в библиотеке NewMat [58] задолго до того, как были разработаны шаблоны выражений.

27.4. Заключительные замечания

Шаблоны выражений были изобретены независимо Тоддом Вельдхаузеном (Todd Veldhuizen), благодаря которому и появился этот термин, и Дэвидом Вандевурдом (David Vandevoorde). Это произошло, когда в языке программирования C++ еще не было шаблонов-членов (в то время казалось, что они никогда не будут добавлены в C++). Из-за этого возникали некоторые проблемы, связанные с реализацией оператора присваивания: он не мог быть параметризован для шаблонов выражений. Один из способов обойти эту проблему состоял в том, чтобы ввести в шаблон выражения оператор преобразования в класс `Copier`, параметризованный шаблоном выражения, но унаследованный от базового основного класса, параметризованного только типом элемента. Далее этот базовый класс предоставлял (виртуальный) интерфейс `copy_to`, на который мог ссылаться оператор присваивания.

Ниже этот механизм представлен схематически (имена шаблонов совпадают с использованными в данной главе).

```

template<typename T>
class CopierInterface
{
public:
    virtual void copy_to(Array<T, SArray<T>>&) const;
};

template<typename T, typename X>
class Copier : public CopierInterface<T>
{
public:
    Copier(X const& x) : expr(x)
    {
    }
    virtual void copy_to(Array<T, SArray<T>>&) const
    {
        // Реализация цикла присваивания
        ***
    }
private:
    X const& expr;
};

template<typename T, typename Rep = SArray<T>>
class Array
{
public:
    // Делегированный оператор присваивания
    Array<T, Rep>& operator=(CopierInterface<T> const& b)
    {
        b.copy_to(rep);
    }
    ***
};

template<typename T, typename A1, typename A2>
class A_mult
{
public:
    operator Copier<T, A_Mult<T, A1, A2>>();
    ***
};

```

При этом в программу добавляется дополнительный уровень сложности, а на обработку шаблонов выражений во время выполнения программы затрачиваются определенные дополнительные ресурсы. Однако даже с учетом этих замечаний производительность полученных программ была на то время впечатляющей.

В стандартной библиотеке C++ содержится шаблон класса `valarray`; предлагалось, что он будет полезен в приложениях, в которых применяются методы, использованные в данной главе для разработки шаблона `Array`. Предшественник `valarray` был разработан с учетом того, что на рынке программных продуктов появятся компиляторы, предназначенные для научных расчетов, которые смогут распознавать этот тип и преобразовывать использующую его программу в высокопроизводительный машинный код. Такие компиляторы должны были обладать

способностью эффективно работать с этим типом, однако они так и не появились (частично из-за того, что рынок программных продуктов в области научных расчетов занимает сравнительно небольшую долю, а частично из-за того, что после реализации класса `valarray` в виде шаблона проблема усложнилась). После изобретения метода шаблонов выражений один из его авторов, Дэвид Вандевурд, подал в Комитет по стандартизации C++ предложение преобразовать класс `valarray`, по сути, в шаблон `Array`, рассмотренный в данной главе (конечно, доработанный с тем, чтобы придать ему все функциональные возможности класса `valarray`). Это предложение было первым, в котором планировалось документировать концепцию параметра `Rep`. До этого фактические массивы, хранящие реальные данные, и псевдомассивы, предназначенные для реализации шаблонов выражений, были разными массивами. Например, если клиентский код вводил функцию `foo()`, аргументом которой являлся объект класса `Array` — например

```
double foo(Array<double> const&);
```

то ее вызов в виде `foo(1.2*x)` приводил к преобразованию шаблона выражения в реальный массив с фактическими данными, даже если для выполнения операций с этими аргументами временный массив не требуется. Если же в нашем распоряжении есть аргумент `Rep`, в который можно поместить шаблон выражения, то функцию `foo()` можно объявить так:

```
template<typename Rep>
double foo(Array<double, Rep> const&);
```

При этом не будет выполняться никаких преобразований, в которых нет необходимости.

Предложение о модификации класса `valarray` поступило в то время, когда процесс стандартизации C++ уже шел полным ходом. Чтобы его принять, пришлось бы переписать весь текст документации, касающийся класса `valarray`. В результате предложение было отклонено, а в документацию были внесены некоторые исправления, позволяющие описывать классы, основанные на шаблонах выражений. Однако практическое использование этих возможностей намного сложнее, чем рассмотренное в данной главе. На момент написания книги не было известно ни одной реализации этих возможностей, а стандартный класс `valarray`, вообще говоря, довольно неэффективен при выполнении тех операций, для которых он разработан.

Наконец, следует упомянуть, что многие новаторские подходы, рассмотренные в настоящей главе, а также те, что вошли в библиотеку STL², имеют одну общую особенность. Все они были первоначально реализованы на одном и том же компиляторе — версии 4 компилятора Borland C++. Он был, вероятно, первым компилятором, благодаря которому шаблонное программирование стало широко распространенным среди разработчиков на языке программирования C++.

² STL (Standard Template Library), или стандартная библиотека шаблонов, кардинально изменила представления о библиотеках C++; позже она вошла в стандартную библиотеку C++ [45].

Шаблоны выражений изначально применялись главным образом для операций над типами массивов. Однако через несколько лет были найдены новые их приложения. Среди наиболее новаторских была библиотека Boost.Lambda [48] Яакко Ярви (Jaakko Järvi) и Гэри Пауэлла (Gary Powell), которые представили полезные возможности лямбда-выражений за годы до того, как лямбда-выражения стали частью ядра языка³, и библиотека Boost.Proto Эрика Ниблера (Eric Niebler), которая представляет собой библиотеку для метапрограммирования шаблонов выражений с целью создания *встроенных предметно-ориентированных языков* в C++. Другие библиотеки Boost, такие как Boost.Fusion и Boost.Hana, также активно используют шаблоны выражений.

³Яакко также сыграл важную роль в развитии этой функциональной возможности языка.

Глава 28

Отладка шаблонов

Когда дело доходит до отладки шаблонов, возникают два класса проблем. Первый класс проблем — проблемы авторов шаблонов: как можно гарантировать, что написанные шаблоны будут функционировать для *любых* аргументов шаблонов, удовлетворяющих задокументированным автором условиям? Второй класс проблем по сути обратный: как пользователь шаблона может узнать, какие требования к параметрам шаблона он нарушил, когда шаблон ведет себя не так, как документировано?

Прежде чем приступить к серьезному обсуждению этих вопросов, полезно рассмотреть виды ограничений, которые могут быть наложены на параметры шаблона. В этой главе мы в основном имеем дело с ограничениями, которые при нарушении приводят к ошибкам компиляции, и называем эти ограничения *синтаксическими ограничениями*. Синтаксические ограничения могут включать необходимость наличия конструкторов определенного вида, однозначность некоторого вызова функции и так далее. Прочие виды ограничений мы называем *семантическими ограничениями*. Эти ограничения гораздо труднее проверить механически. В общем случае это даже может быть нецелесообразным. Например, мы можем потребовать, чтобы для типового параметра шаблона был определен оператор < (что является синтаксическим ограничением), но обычно мы также требуем, чтобы этот оператор фактически определял некоторое упорядочение в своей области определения (что является семантическим ограничением).

Для обозначения набора ограничений, которые многократно требуются в библиотеке шаблонов, часто используется термин *концепт*. Например, стандартная библиотека C++ опирается на такие концепты, как *оператор с произвольным доступом* или *имеющий конструктор по умолчанию*. С учетом данной терминологии можно сказать, что отладка кода шаблонов включает значительное количество работы по выявлению того, как в реализациях шаблонов и их использовании нарушаются концепты. Эта глава посвящена как проектированию, так и методам отладки, которые облегчают работу с шаблонами как для их авторов, так и для пользователей.

28.1. Поверхностное инстанцирование

Когда происходит ошибка в шаблоне, проблема часто выявляется после длинной цепочки инстанцирований, что приводит к длинным сообщениям об ошибках наподобие рассматривавшихся в разделе 9.4¹. Для иллюстрации этого рассмотрим следующий надуманный код:

¹ Если вы добрались до этого места книги, то наверняка сталкивались с такими сообщениями об ошибках, по сравнению с которыми этот упомянутый пример — просто надпись на заборе рядом с “Войной и миром”.

```

template<typename T>
void clear(T& p)
{
    *p = 0; // В предположении, что T – тип указателя
}

template<typename T>
void core(T& p)
{
    clear(p);
}

template<typename T>
void middle(typename T::Index p)
{
    core(p);
}

template<typename T>
void shell(T const& env)
{
    typename T::Index i;
    middle<T>(i);
}

```

Этот пример иллюстрирует типичную слоистую структуру разработки программного обеспечения: шаблоны функций высокого уровня, такие как `shell()`, основаны на таких компонентах, как `middle()`, которые сами используют фундаментальные средства наподобие `core()`. Когда инстанцируется `shell()`, должны инстанцироваться и все слои ниже. В данном примере проблема проявляется в наиболее глубоком слое: `core()` создается с типом `int` (получаемым из использования `Client::Index` в `middle()`), и выполняется попытка разыменования значения этого типа, что является ошибкой.

Эта ошибка обнаруживается только во время инстанцирования. Например:

```

class Client
{
public:
    using Index = int;
};

int main()
{
    Client mainClient;
    shell(mainClient);
}

```

Хорошая диагностика при обобщенном программировании включает трассировку всех слоев, которые привели к проблемам, но такое большое количество информации может оказаться излишне громоздким.

Отличное обсуждения основных идей, связанных с этой проблемой, можно найти в [66], где Бъярне Страуструп (Bjarne Stroustrup) определяет два класса подходов к как можно более раннему определению, удовлетворяют ли аргументы шаблона набору ограничений: через расширение языка или с помощью раннего

использования параметров. Первый вариант рассматривается в разделе 17.8 и в приложении Д, “Концепты”. Второй вариант состоит в выявлении любых ошибок в поверхностных инстанцированиях (*shallow instantiations*). Это достигается путем вставки неиспользуемого кода, не имеющего никакой иной цели, кроме как вызвать ошибку, если код создается с аргументами шаблона, которые не отвечают требованиям шаблонов на более глубоких уровнях.

В нашем предыдущем примере мы могли бы добавить в `shell()` код, который пытается разыменовать значения типа `T::Index`. Например:

```
template<typename T>
void ignore(T const&)
{
}

template<typename T>
void shell(T const& env)
{
    class ShallowChecks
    {
        void deref(typename T::Index ptr)
        {
            ignore(*ptr);
        }
    };
    typename T::Index i;
    middle(i);
}
```

Если `T` является типом, для которого `T::Index` не может быть разыменован, ошибка диагностируется в локальном классе `ShallowChecks`. Обратите внимание: поскольку этот локальный класс фактически не используется, добавленный код не влияет на время работы функции `shell()`. К сожалению, многие компиляторы будут предупреждать, что класс `ShallowChecks` не используется. Для подавления предупреждений могут использоваться такие трюки, как применение шаблона `ignore()`, но они привносят в код дополнительную сложность.

Проверка концепта

Очевидно, что разработка фиктивного кода в нашем примере может стать столь же сложной, как и разработка кода, реализующего фактическую функциональность шаблона. Для контроля этой сложности естественно попытаться собрать различные фрагменты фиктивного кода в своего рода библиотеки. Например, такая библиотека может содержать макросы, которые раскрываются в код, который вызывает ошибку, когда подстановка параметров шаблона нарушает концепт, лежащий в основе конкретного параметра. Наиболее популярная из таких библиотек, *Concept Check Library*, является частью дистрибутива Boost (см. [8]).

К сожалению, эта методика не особенно переносима (способ диагностики ошибки может значительно отличаться от одного компилятора к другому), а иногда маскировать проблемы, которые не могут быть перехвачены на более высоком уровне.

После того как *концепты* войдут в C++ (см. приложение Д, “Концепты”), у нас будут другие способы поддержки определения требований и ожидаемого поведения.

28.2. Статические утверждения

Макрос `assert()` часто используется в коде C++ для проверки выполнения некоторых определенных условий в некоторый момент выполнения программы. Если утверждение не выполняется, программа прерывается (с выводом соответствующей информации), так что программист может обнаружить и исправить проблему.

Ключевое слово C++ `static_assert`, введенное в C++11, служит той же цели, но вычисляется во время компиляции. Если условие (которое должно быть константным выражением) имеет значение `false`, компилятор выдаст сообщение об ошибке. Это сообщение об ошибке будет включать строку (которая является частью самого `static_assert`), указывающую программисту, что именно пошло не так. Например, следующее статическое утверждение гарантирует, что мы выполняем компиляцию на платформе с 64-битными указателями:

```
static_assert(sizeof(void*)*CHAR_BIT==64,"Не 64-разрядная платформа");
```

Статические утверждения могут использоваться для предоставления полезных сообщений об ошибках, когда аргумент шаблона не соответствует ограничениям шаблона. Например, с помощью методов, описанных в разделе 19.4, можно создать тип-свойство для выяснения, является ли данный тип разыменуемым:

debugging/hasderef.hpp

```
#include <utility>      // Для declval()
#include <type_traits> // Для true_type и false_type

template<typename T>
class HasDereference
{
private:
    template<typename U> struct Identity;
    template<typename U> static std::true_type
    test(Identity<decltype(*std::declval<U>())*>*);
    template<typename U> static std::false_type
    test(...);
public:
    static constexpr bool value    decltype(test<T>(nullptr))::value;
};
```

Теперь мы можем ввести в `shell()` статическое утверждение, которое обеспечивает лучшую диагностику, если шаблон `shell()` из предыдущего раздела инстанцируется с типом, который не является разыменовываемым:

```
template<typename T>
void shell(T const& env)
{
    static_assert(HasDereference<T>::value,
                 "Т не является разыменовываемым");
    typename T::Index i;
    middle(i);
}
```

При таком изменении компилятор выводит значительно более краткую диагностику, указывающую, что тип T не является разыменуемым.

Статические утверждения могут значительно облегчить жизнь пользователей при работе с шаблонной библиотекой, делая сообщения об ошибках короче и понятнее.

Обратите внимание на то, что их можно также применять к шаблонам классов и использовать все свойства типов, обсуждающиеся в приложении Г, “Стандартные утилиты для работы с типами”:

```
template<typename T>
class C
{
    static_assert(HasDereference<T>::value,
                  "T не является разыменовываемым");
    static_assert(std::is_default_constructible<T>::value,
                  "T не конструируем по умолчанию");

    ...
};
```

28.3. Архетипы

При написании шаблона сложно гарантировать, что его определение будет компилироваться для любых аргументов шаблона, удовлетворяющих указанным ограничениям для этого шаблона. Рассмотрим простой алгоритм `find()`, который ищет значения в массиве, а также его документированные ограничения:

```
// Т должен быть EqualityComparable, что означает:
// два объекта типа Т могут сравниваться с использованием
// оператора == и преобразованием результата в bool
template<typename T>
int find(T const* array, int n, T const& value);
```

Можно представить себе следующую простую реализацию этого шаблона функции:

```
template<typename T>
int find(T const* array, int n, T const& value)
{
    int i = 0;

    while (i != n && array[i] != value)
        ++i;

    return i;
}
```

У этого определения шаблона есть две проблемы, причем обе они проявляются как ошибки компиляции для определенных аргументов шаблона, которые *технически* отвечают требованиям шаблона, но ведут себя несколько иначе, чем ожидает автор шаблона. Мы будем использовать понятие *архетипов* (archetype) для проверки соответствия используемых в нашей реализации параметров шаблонов требованиям, указанным в шаблоне `find()`.

Архетипы представляют собой пользовательские классы, применяемые в качестве аргументов шаблонов для тестирования соблюдения ограничений, которые определение шаблона налагает на соответствующие параметры шаблона. Архетип специально разрабатывается для удовлетворения требованиям шаблона наименьшим возможным способом, без предоставления каких-либо посторонних операций. Если инстанцирование шаблона определения с архетипом в качестве его аргумента шаблона завершается успешно, то мы знаем, что определение шаблона не пытается использовать никакие операции, которые не указаны шаблоном явно.

Далее приводится пример архетипа, предназначенного для проверки соответствия требованиям концепта `EqualityComparable`, описанного в документации нашего алгоритма `find()`:

```
class EqualityComparableArchetype
{
};

class ConvertibleToBoolArchetype
{
public:
    operator bool() const;
};

ConvertibleToBoolArchetype
operator==(EqualityComparableArchetype const&,
            EqualityComparableArchetype const&);
```

Тип `EqualityComparableArchetype` не имеет функций-членов и членов-данных, и единственная операция, которую он обеспечивает — это перегруженный `operator==` для удовлетворения требованиям `find()`. Сам по себе `operator==` минималистичен, возвращая другой архетип, `ConvertibleToBoolArchetype`, единственной определенной операцией которого является пользовательское преобразование в тип `bool`.

Очевидно, что `EqualityComparableArchetype` соответствует указанным требованиям шаблона `find()`, так что мы можем проверить, будет ли реализация `find()` до конца выполнять свой контракт, пытаясь инстанцировать `find()` для `EqualityComparableArchetype`:

```
template int find(EqualityComparableArchetype const*, int,
                  EqualityComparableArchetype const&);
```

Инстанцирование `find<EqualityComparableArchetype>` оказывается неудачным, указывая, что мы встретились с первой проблемой: описание `EqualityComparable` описание наличия только оператора `==`, но реализация `find()` полагается на сравнение объектов типа `T` с помощью оператора `!=`. Наша реализация работала бы с большинством определяемых пользователем типов, которые реализуют пару операторов `==` и `!=`, но на самом деле работала бы неправильно. Архетипы предназначены для обнаружения подобных проблем на ранних этапах разработки шаблонных библиотек.

Изменяя реализацию алгоритма `find()` так, чтобы он использовал равенство вместо неравенства, мы решаем эту первую проблему, и шаблон `find()` теперь успешно компилируется с архетипом²:

```
template<typename T>
int find(T const* array, int n, T const& value)
{
    int i = 0;

    while (i != n && !(array[i] == value))
        ++i;

    return i;
}
```

Выявление второй проблемы в `find()` с использованием архетипов требует немного больше изобретательности. Обратите внимание на то, что новое определение `find()` применяет оператор `!` непосредственно к результату оператора `==`. В случае нашего архетипа код опирается на пользовательское преобразование в тип `bool` и встроенный оператор логического отрицания `!`. Более тщательная реализация `ConvertibleToBoolArchetype` убирает оператор `!` таким образом, чтобы он не мог использоваться ненадлежащим образом:

```
class ConvertibleToBoolArchetype
{
public:
    operator bool() const;
    bool operator!() =
        delete; // Логическое отрицание явно не требовалось
};
```

Мы могли бы расширить этот архетип, используя удаление функций³ для операторов `&&` и `||`, чтобы помочь найти проблемы в других определениях шаблонов. Обычно разработчик шаблона будет хотеть разработать архетип каждого концепта, определенного в библиотеке шаблонов, а затем использовать эти архетипы для тестирования каждого определения шаблона на соответствие заявленным требованиям.

28.4. Трассировщики

До сих пор мы обсуждали ошибки, возникающие при компиляции или компоновке программ, содержащих шаблоны. Однако наиболее сложная задача обеспечения корректной работы программы во время выполнения часто *следует за успешным построением программы*. Шаблоны могут сделать эту задачу более сложной, потому что поведение обобщенного кода, представленного шаблоном,

² Программа будет компилироваться, но не компоноваться, поскольку мы не определили `operator==`. Это обычная ситуация для архетипов, которые в общем случае предназначены только для проверок времени компиляции.

³ Удаленные функции участвуют в разрешении перегрузки как обычные функции. Однако если они выбираются разрешением перегрузки, то компилятор выдает сообщение об ошибке.

зависит от клиента этого шаблона в гораздо большей степени, чем для обычных классов и функций. *Трассировщик* (*tracer*) представляет собой программное обеспечение, которое может облегчить этот аспект отладки путем выявления проблем в определениях шаблонов на ранних этапах цикла разработки.

Трассировщик представляет собой пользовательский класс, который может использоваться в качестве аргумента проверяемого шаблона. Часто трассировщики являются также архетипами, написанными просто для удовлетворения требований шаблона. Однако более важным является то, что трассировщик должен генерировать *след* (*trace*) операций, которые вызываются для него. Это позволяет, например, экспериментально проверить как эффективность алгоритмов, так и последовательность операций.

Ниже приведен пример трассировщика, который может использоваться для проверки алгоритма сортировки⁴:

debugging/tracer.hpp

```
#include <iostream>
class SortTracer
{
private:
    // Сортируемое значение:
    int value;
    // Поколение трассировщика:
    int generation;
    // Количество вызовов конструкторов:
    inline static long n_created = 0;
    // Количество вызовов деструкторов:
    inline static long n_destroyed = 0;
    // Количество присваиваний:
    inline static long n_assigned = 0;
    // Количество сравнений:
    inline static long n_compared = 0;
    // Максимальное количество существующих объектов:
    inline static long n_max_live = 0;

    // Вычисление максимального количества существующих объектов:
    static void update_max_live()
    {
        if (n_created - n_destroyed > n_max_live)
        {
            n_max_live = n_created - n_destroyed;
        }
    }
public:
    static long creations()
    {
        return n_created;
    }
    static long destructions()
```

⁴ До C++17 мы должны инициализировать статические члены в одной единице трансляции вне объявления класса.

```
{      return n_destroyed;
}
static long assignments()
{
    return n_assigned;
}
static long comparisons()
{
    return n_compared;
}
static long max_live()
{
    return n_max_live;
}
public:
// Конструктор
SortTracer(int v = 0) : value(v), generation(1)
{
    ++n_created;
    update_max_live();
    std::cerr << "SortTracer #" << n_created
        << ", created generation " << generation
        << " (total: " << n_created - n_destroyed
        << ")\\n";
}
// Копирующий конструктор
SortTracer(SortTracer const& b)
    : value(b.value), generation(b.generation + 1)
{
    ++n_created;
    update_max_live();
    std::cerr << "SortTracer #" << n_created
        << ", copied as generation " << generation
        << " (total: " << n_created - n_destroyed
        << ")\\n";
}
// Деструктор
~SortTracer()
{
    ++n_destroyed;
    update_max_live();
    std::cerr << "SortTracer generation " << generation
        << " destroyed (total: "
        << n_created - n_destroyed << ")\\n";
}
// Присваивание
SortTracer& operator= (SortTracer const& b)
{
    ++n_assigned;
    std::cerr << "SortTracer assignment #" << n_assigned
        << " (generation " << generation
        << " = " << b.generation
        << ")\\n";
    value = b.value;
    return *this;
}
```

```

// Сравнение
friend bool operator < (SortTracer const& a,
                        SortTracer const& b)
{
    ++n_compared;
    std::cerr << "SortTracer comparison #" << n_compared
        << " (generation " << a.generation
        << " < " << b.generation
        << ")\n";
    return a.value < b.value;
}
int val() const
{
    return value;
}
};

```

В дополнение к сортируемому значению `value` трассировщик предоставляет несколько членов для отслеживания выполнения сортировки: для каждого объекта `generation` отслеживает количество операций копирования. Исходное значение имеет `generation==1`, у прямой копии оригинала `generation==2`, копия копии имеет `generation==3` и так далее. Другие статические члены отслеживают количество вызовов конструкторов, деструкторов, присваиваний, сравнений и максимальное количество одновременно существовавших объектов.

Этот конкретный трассировщик позволяет нам отслеживать схему создания и уничтожения объектов, а также выполнение присваиваний и сравнений данным шаблоном. Следующая тестовая программа иллюстрирует сказанное для алгоритма `std::sort()` стандартной библиотеки C++:

debugging/tracertest.cpp

```

#include <iostream>
#include <algorithm>
#include "tracer.hpp"

int main()
{
    // Образец входных данных:
    SortTracer input[] = { 7, 3, 5, 6, 4, 2, 0, 1, 9, 8 };

    // Вывод исходных значений:
    for (int i = 0; i < 10; ++i)
    {
        std::cerr << input[i].val() << ' ';
    }

    std::cerr << '\n';
    // Исходные условия:
    long created_at_start = SortTracer::creations();
    long max_live_at_start = SortTracer::max_live();
    long assigned_at_start = SortTracer::assignments();
    long compared_at_start = SortTracer::comparisons();

```

```

// Работа алгоритма:
std::cerr << "---[ Start std::sort() ]-----\n";
std::sort(&input[0], &input[9] + 1);
std::cerr << "---[ End std::sort() ]-----\n";

// Проверка результатов:
for (int i = 0; i < 10; ++i)
{
    std::cerr << input[i].val() << ' ';
}

std::cerr << "\n\n";
// Окончательный отчет:
std::cerr << "std::sort() для 10 SortTracer"
<< " выполнен с помощью:\n  "
<< SortTracer::creations() - created_at_start
<< " временных трассировщиков\n  "
<< "до "
<< SortTracer::max_live()
<< " трассировщиков одновременно ("
<< max_live_at_start << " вначале)\n  "
<< SortTracer::assignments() - assigned_at_start
<< " присваивания\n  "
<< SortTracer::comparisons() - compared_at_start
<< " сравнений\n\n";
}

```

Выполнение этой программы генерирует значительный вывод в консоль, но из заключительного отчета можно сделать многие выводы. Для одной из реализаций функции `std::sort()` мы получили следующее:

```

std::sort() для 10 SortTracer выполнен с помощью:
9 временных трассировщиков
до 11 трассировщиков одновременно (10 вначале)
33 присваивания
27 сравнений

```

Например, мы видим, что, хотя в процессе сортировки в нашей программе были созданы девять временных трассировщиков, в любой момент времени имеется не более двух дополнительных трассировщиков.

Таким образом, наш трассировщик выполняет две функции: доказывает, что стандартный алгоритм `sort()` требует не большую функциональность, чем может предоставить наш трассировщик (например, операторы `==` и `>` не нужны), и дает нам возможность оценить стоимость алгоритма. Однако он не свидетельствует о правильности шаблона сортировки.

28.5. Оракулы

Трассировщики относительно просты и эффективны, но они позволяют нам отслеживать выполнение шаблонов только для определенных входных данных и для определенного поведения связанной с ними функциональности. Например,

нас может интересовать, каким требованиям должен отвечать оператор сравнения для корректной сортировки, но в нашем примере мы проверили только оператор сравнения, который ведет себя так же, как оператор “меньше, чем” для целых чисел.

Расширения трассировщиков известны в определенных кругах как *оракулы* (*oracles*), или как *оракулы анализа времени выполнения*. Они являются трассировщиками, подключенными к *механизмам вывода* (inference engine) — программам, которые могут запоминать утверждения и их причины и делать на их основе определенные заключения.

Оракулы в ряде случаев позволяют нам проверить шаблоны алгоритмов динамически, без полного указания заменяемых аргументов шаблона (оракулы являются аргументами) или входных данных (механизм вывода может запросить некоторые исходные предположения для работы). Однако реально таким образом могут быть проанализированы только достаточно простые алгоритмы (из-за ограничений механизмов вывода), и такой анализ требует значительного объема работы. По этим причинам мы не вникали глубоко в эту тему, но заинтересовавшийся читатель может изучить публикации, упомянутые в заключительных замечаниях.

28.6. Заключительные замечания

Систематические попытки улучшения диагностики компиляторов C++ путем добавления фиктивного кода в высокоуровневые шаблоны можно найти у Джереми Сика (Jeremy Siek) в его библиотеке *Concept Check Library* [8]. Она является частью библиотеки Boost [10].

Роберт Кларер (Robert Klarer) и Джон Мэддок (John Maddock) предложили статическое утверждение — `static_assert`, призванное помочь программистам проверять условия во время компиляции. Это одна из первых функциональных возможностей, добавленных в стандарт C++11. До этого утверждения обычно выражались как библиотека или макрос с использованием методов, аналогичных описанным в разделе 28.1. Одной такой реализацией является библиотека Boost. StaticAssert.

Система *MELAS* предоставляет оракулы для определенных частей стандартной библиотеки C++, обеспечивая возможность верификации некоторых ее алгоритмов. Эта система обсуждается в [56]⁵.

⁵ Один из ее авторов, Дэвид Мицсер (David Musser), являлся также ключевой фигурой в разработке стандартной библиотеки C++. Среди прочего он спроектировал и реализовал первые ассоциативные контейнеры.

Приложение А

Правило одного определения

Правило одного определения (one-definition rule – ODR) — это краеугольный камень хорошо оформленного структурирования программ на C++. Наиболее общие следствия этого правила достаточно просты для того, чтобы их запомнить и использовать: следует определять невстраиваемые функции и объекты ровно один раз для всех файлов, а классы и встраиваемые функции и встраиваемые переменные — не более одного раза на единицу трансляции. При этом нужно следить за тем, чтобы все определения одного и того же объекта были идентичны.

Однако основные проблемы заключены в деталях, которые в сочетании с инстанцированием шаблона могут оказаться обескураживающими. Данное приложение поможет заинтересованному читателю получить всестороннее представление об ODR. Если отдельные темы подробно описаны в той или иной главе книги, об этом обязательно упоминается в тексте приложения.

A.1. Единицы трансляции

На практике программы на C++ пишутся путем заполнения файлов “кодом”. Однако границы, устанавливаемые файлом, не так важны в контексте ODR. Вместо этого важную роль играют так называемые единицы трансляции. Последняя, *единица трансляции* (translation unit) представляет собой результат препроцессорной обработки файла, переданного компилятору. Препроцессор удаляет части кода в соответствии с директивами условной компиляции (#if, #ifdef и связанными с ними), удаляет комментарии, (рекурсивно) вставляет файлы, определяемые директивой #include, и разворачивает макросы.

Следовательно, в контексте ODR файлы

```
// ===== header.hpp:
#ifndef DO_DEBUG
    #define debug(x) std::cout << x << '\n'
#else
    #define debug(x)
#endif

void debugInit();

И

// ===== myprog.cpp:
#include "header.hpp"

int main()
{
    debugInit();
    debug("main() ");
}
```

эквивалентны одному следующему файлу:

```
// ===== myprog.cpp:
void debugInit();
int main()
{
    debugInit();
}
```

Связи между единицами трансляции устанавливаются с помощью соответствующих объявлений с внешним связыванием в двух единицах трансляции (например, двух объявлений глобальной функции `debug_init()`).

Следует отметить, что концепция единицы трансляции несколько более абстрактна, чем просто концепция “файла, обработанного препроцессором”. Например, если дважды передать компилятору файл после обработки препроцессором для формирования единой программы, он выдаст две отдельные единицы трансляции (хотя, надо сказать, для этого нет причин).

A.2. Объявления и определения

В разговоре между собой программисты часто не делают различия между терминами *обявление* (*declaration*) и *определение* (*definition*). Тем не менее в контексте ODR важно точное значение этих слов¹.

Обявление — это конструкция C++, которая (обычно²) первый раз (или повторно) вводит в программу какое-либо имя. Обявление может одновременно быть и определением, в зависимости от того, какой объект и как вводится.

- **Пространства имен и псевдонимы пространств имен.** Объявления пространств имен и их псевдонимов также являются определениями, хотя термин “*определение*” в этом контексте необычен, поскольку список членов пространства имен позднее может быть расширен (в отличие, например, от классов и перечислимых типов).
- **Классы, шаблоны классов, функции, шаблоны функций, функции-члены и шаблоны функций-членов.** Обявление является определением тогда и только тогда, когда объявление включает связанное с именем тело, ограниченное фигурными скобками. Это правило включает объединения, операторы, операторы-члены, статические функции-члены, конструкторы и деструкторы, а также явные специализации шаблонных версий таких объектов (т.е. любого объекта типа класса и типа функции).
- **Перечисления.** Обявление является определением тогда и только тогда, когда оно включает список перечисленных значений, заключенный в фигурные скобки.

¹ Нам кажется, что при обмене идеями из С и C++ хорошей привычкой является крайне осторожное употребление этих терминов. Такой подход используется и в данной книге.

² Некоторые конструкции (такие как `static_assert`) не вводят никаких имен, но синтаксически рассматриваются как объявления.

- **Локальные переменные и нестатические члены-данные.** Эти объекты всегда рассматриваются как определения, хотя изредка всгречаются и редкие исключения. Обратите внимание на то, что объявление параметра функции в определении функции само по себе является определением, потому что обозначает локальную переменную, но параметр функции в объявлении функции, которое не является определением, также не является определением.
- **Глобальные переменные.** Если непосредственно перед объявлением не стоит ключевое слово `extern` или переменная инициализирована, такое объявление глобальной переменной является одновременно ее определением. В противном случае это не определение.
- **Статические данные-члены.** Объявление является определением тогда и только тогда, когда оно находится за пределами класса или шаблона класса, членом которого оно является, или если член определен как `inline` или `constexpr` в классе или шаблоне класса.
- **Явная и частичная специализации.** Объявление является определением, если объявление, следующее после `template<>` или `template<...>`, само по себе является определением, с тем исключением, что явная специализация статических членов-данных или шаблонов статических членов-данных является определением только тогда, когда включает инициализатор.

Прочие объявления определениями не являются. Сюда входят псевдонимы типов (получаемые с помощью `typedef` или `using`), объявления `using`, директивы `using`, объявления параметров шаблонов, директивы явного инстанцирования, объявления `static_assert` и т.д.

A.3. Детали правила одного определения

Как уже упоминалось во введении к данному приложению, практическое использование этого правила связано с массой тонкостей. Рассмотрим ограничения данного правила в соответствии с областью их действия.

A.3.1. Ограничения “одно на программу”

Перечисленные ниже объекты могут иметь лишь одно определение на программу.

- Невстраиваемые функции и невстраиваемые функции-члены (включая полные специализации шаблонов функций).
- Невстраиваемые переменные (по сути, переменные, объявленные в области видимости пространства имен или в глобальной области видимости, и без спецификатора `static`).
- Невстраиваемые статические данные-члены.

Например, приведенная ниже программа на C++, состоящая из двух единиц трансляции, неработоспособна.

```
// ===== Единица трансляции 1:
int counter;
// ===== Единица трансляции 2:
int counter; // Ошибка: повторное определение (нарушение ODR)
```

Это правило не применяется к объектам с *внутренним связыванием* (сущности, объявленные со спецификатором `static` в глобальной области видимости или области видимости пространства имен), поскольку даже когда два таких объекта имеют одно и то же имя, они считаются разными. Объекты, объявленные в безымянных пространствах имен, считаются различными, если они находятся в разных единицах трансляции; в C++11 и более поздних стандартах такие сущности также по умолчанию имеют внутреннее связывание, но до C++11 они имели по умолчанию внешнее связывание. Например, следующие две единицы трансляции можно объединить в корректную программу на C++:

```
// ===== Единица трансляции 1:
static int counter = 2; // Не связана с прочими единицами трансляции
namespace
{
    void unique()          // Не связана с прочими единицами трансляции
    {
    }
}

// ===== Единица трансляции 2:
static int counter = 0; // Не связана с прочими единицами трансляции
namespace
{
    void unique()          // Не связана с прочими единицами трансляции
    {
        ++counter;
    }
}
int main()
{
    unique();
}
```

Кроме того, в программе должен быть *только один* из упомянутых выше объектов, если упомянутые выше объекты *используются* в контексте, отличном от отброшенной ветви конструкции `if constexpr` (функциональная возможность, доступная только в C++17). У термина “используются” в данном контексте точное значение. Он означает, что к данному объекту где-то в программе есть обращение, которое приводит к необходимости генерации кода³. Это обращение может быть доступом к значению переменной, вызовом функции или получением адреса такого объекта. Это может быть явное обращение в исходном тексте, но может быть

³Различные оптимизации могут приводить к удалению этого кода, но сам язык не предполагает наличия такой оптимизации.

и неявное. Например, выражение с оператором `new` может создавать неявный вызов связанного с ним оператора `delete` для обработки ситуаций, когда конструктор генерирует исключение, которое требует очистки неиспользуемой (но выделенной) памяти. Другой пример — копирующие конструкторы, которые должны быть определены, даже если в конечном итоге они удаляются оптимизатором (кроме случаев, когда такое удаление требуется языком, что является распространенной практикой в C++17). Виртуальные функции также используются неявно (с помощью внутренних структур, которые обеспечивают их вызовы), если только это не чисто виртуальные функции. Существует несколько других видов неявного использования, однако для краткости изложения здесь они не рассматриваются.

Существуют некоторые обращения, которые *не являются* использованием в указанном смысле. Это применение объекта в качестве *невычисляемого операнда* (например, operand оператора `sizeof` или `decltype`). Operand оператора `typeid` (см. раздел 9.1.1) является *невычисляемым* только в некоторых случаях. Конкретно, если ссылка появляется как часть оператора `typeid`, это не является использованием в указанном ранее смысле, если только аргумент оператора `typeid` не используется с полиморфным объектом (объектом с — возможно, унаследованными — виртуальными функциями). Рассмотрим, например, следующую программу, состоящую из одного файла:

```
#include <typeinfo>

class Decider
{
#if defined(DYNAMIC)
    virtual ~Decider()
    {
    }
#endif
};

extern Decider d;

int main()
{
    char const* name = typeid(d).name();
    return (int)sizeof(d);
}
```

Эта программа корректна тогда и только тогда, когда не определен символ препроцессора `DYNAMIC`. На самом деле переменная `d` не определена, однако обращение к `d` в операторе `sizeof(d)` не является использованием, а обращение в операторе `typeid(d)` может быть использованием, только если `d` — объект полиморфного типа (поскольку в общем случае не всегда удается определить результат применения полиморфного оператора `typeid` до момента выполнения).

В соответствии со стандартом C++ ограничения, описанные в данном разделе, не требуют диагностических сообщений от компилятора C++. На практике же компоновщик почти всегда сообщает о них как о двойных или пропущенных определениях.

A.3.2. Ограничения “одно на единицу трансляции”

Ни одна сущность не может быть определена в единице трансляции более одного раза. Так, приведенный ниже код на C++ работать не будет.

```
inline void f() {}  
inline void f() {} // Ошибка: двойное определение
```

Это одна из причин для того, чтобы окружить код в заголовочных файлах так называемыми *предохранителями*:

```
// ===== guarddemo.hpp:  
#ifndef GUARDDEMO_HPP  
#define GUARDDEMO_HPP  
  
...  
#endif // GUARDDEMO_HPP
```

Такие предохранители обеспечивают игнорирование содержимого файла при его повторном включении директивой `#include`, что позволяет избежать двойного определения любого класса, встроенной функции или шаблона, содержащихся в нем.

Правило ODR также указывает, что в определенных обстоятельствах некоторые объекты *должны* быть определены. Это требуется, например, в случае классовых типов, встроенных функций и неэкспортированных шаблонов. Рассмотрим вкратце конкретные правила.

Классовый тип X (включая `struct` и `union`) *должен* быть определен в единице трансляции *до того*, как в этой единице трансляции будет выполнена какая-либо из описанных ниже операций.

- Создание объекта типа X (например, путем объявления переменной или с помощью оператора `new`). Создание может быть непрямым, например когда создается объект, который содержит в себе объект типа X.
- Объявление члена-данного класса X.
- Применение оператора `sizeof` или `typeid` к объекту типа X.
- Явный или неявный доступ к членам типа X.
- Преобразование выражения в тип X или из типа X с помощью процедуры преобразования любого вида; преобразование выражения в указатель или ссылку на тип X либо из указателя или ссылки на тип X (за исключением `void*`) с помощью операции неявного приведения, а также с помощью операторов `static_cast` или `dynamic_cast`.
- Присваивание значения объекту типа X.
- Определение или вызов функции с аргументом либо возвращаемым значением типа X. Вместе с тем простое объявление такой функции не требует определения типа.

Правила для типов применяются также к типам, генерируемым из шаблонов классов. Это означает, что в тех ситуациях, когда должен быть определен такой тип X, должны быть определены и соответствующие шаблоны. Эти ситуации

создают так называемые *точки инстанцирования* (points of instantiation – POI, см. раздел 14.3.2).

Встраиваемые функции должны быть определены в каждой единице трансляции, в которой они используются (в которой они вызываются или в которой определяются их адреса). Однако в отличие от типов классов их определение может идти после точки использования.

```
inline int notSoFast();

int main()
{
    notSoFast();
}

inline int notSoFast()
{
}
```

Хотя это правильный код на C++, некоторые компиляторы не встраивают вызов функции с телом, которого еще не видно; поэтому желаемого эффекта можно не достичь.

Как и в случае шаблонов классов, использование функции, сгенерированной из параметризованного объявления функции (шаблона функции или функции-члена либо функции-члена шаблона класса), создает точку инстанцирования. Однако в отличие от шаблонов классов соответствующее определение может находиться после этой точки.

Аспекты ODR, о которых идет речь в этом приложении, как правило, легко проверить с помощью компиляторов C++. Поэтому стандарт C++ требует, чтобы компиляторы выполняли определенную диагностику при нарушении этих правил. Исключением является отсутствие определения для параметризованной функции. Такие ситуации обычно не диагностируются.

A.3.3. Ограничения эквивалентности единиц перекрестной трансляции

Способность определять некоторые виды объектов в более чем одной единице трансляции создает потенциальную возможность возникновения ошибки нового вида: многократные определения, которые не согласуются друг с другом. К сожалению, такие ошибки трудно обнаружить с помощью традиционной технологии компиляции, в которой единицы трансляции обрабатываются по одной за раз. Таким образом, стандарт C++ *не требует* обнаружения или диагностирования различий во многократных определениях (но, конечно, *позволяет* делать это). Если это ограничение на единицы перекрестной трансляции нарушается, стандарт C++ квалифицирует это как фактор, ведущий к неопределенному поведению, что означает возможность любых предсказуемых и непредсказуемых событий. Обычно такие ошибки, не поддающиеся диагностике, могут вызывать аварийные ситуации или приводить к неправильным результатам, но в принципе

они могут повлечь за собой и другие, более прямые виды ущерба (например, повреждение файлов)⁴.

Ограничения на единицы перекрестной трансляции указывают, что, когда объект определен в двух различных местах кода, эти два места должны состоять из одинаковой последовательности токенов (ключевых слов, операторов и т.п., оставшихся после обработки препроцессором). Кроме того, в одинаковом контексте эти токены должны иметь одинаковое значение (например, может потребоваться, чтобы идентификаторы ссылались на одну и ту же переменную).

Рассмотрим следующий пример:

```
// ===== Единица трансляции 1:
static int counter = 0;
inline void increaseCounter()
{
    ++counter;
}
int main()
{
}

// ===== Единица трансляции 2:
static int counter = 0;
inline void increaseCounter()
{
    ++counter;
}
```

Этот пример выдает сообщение об ошибке, поскольку, хотя последовательность токенов для встроенной функции `increaseCounter()` выглядит идентично в обеих единицах трансляции, они содержат токен `counter`, который указывает на два разных объекта. Действительно, поскольку две переменные `counter` имеют внутреннее связывание (спецификатор `static`), они никак не связаны между собой, несмотря на одинаковое имя. Отметим, что это ошибка, несмотря на то, что в данном случае не используется ни одна из встроенных функций.

Размещение в заголовочных файлах (подключаемых к программе с помощью директивы `#included`) определений объектов, которые могут быть определены в нескольких единицах трансляции, обеспечивает идентичность последовательности лексем почти во всех ситуациях⁵. При таком подходе ситуации, в которых два идентичных токена ссылаются на различные объекты, становятся довольно редкими. Но, если такая ситуация все же случается, возникающие при этом ошибки часто загадочны и их сложно отследить.

Ограничения на перекрестные единицы трансляции касаются не только объектов, которые могут быть определены в нескольких местах, но и аргументов по

⁴ Забавно, что версия 1 компилятора gcc действительно делала это, запуская в подобных ситуациях игру *Rogue*.

⁵ Иногда директивы условной компиляции по-разному вычисляются в разных единицах трансляции. Такие директивы следует использовать осторожно. Возможны также и другие отличия, но они встречаются реже.

умолчанию в объявлении. Иными словами, приведенная ниже программа обладает непредсказуемым поведением.

```
// ===== Единица трансляции 1:
void unused(int = 3);
int main()
{
}

// ===== Единица трансляции 2:
void unused(int = 4);
```

Следует отметить, что с эквивалентностью потоков токенов иногда могут быть связаны неявные тонкие эффекты. Следующий пример (слегка измененный) взят из стандарта C++:

```
// ===== Единица трансляции 1:
class X
{
public:
    X(int, int);
    X(int, int, int);
};
X::X(int, int = 0)
{
}
class D
{
    X x = 0;
};
D d1;      // D() вызывает X(int, int)

// ===== Единица трансляции 2:
class X
{
public:
    X(int, int);
    X(int, int, int);
};
X::X(int, int = 0, int = 0)
{
}
class D : public X
{
    X x = 0;
};
D d2;      // D() вызывает X(int, int, int)
```

В этом примере проблема связана с тем, что неявные конструкторы по умолчанию класса D, генерируемые в двух единицах трансляции, отличаются. В одной из них вызывается конструктор X, принимающий один аргумент, в другой вызывается конструктор X, принимающий два аргумента. Несомненно, данный пример — это еще один стимул ограничить аргумент по умолчанию единственным местом в программе (по возможности это место должно быть в заголовочном

файле). К счастью, размещение аргумента по умолчанию в определениях за пределами класса — довольно редкое явление.

Существует также исключение из правила, согласно которому идентичные токены должны ссылаться на идентичные сущности. Если идентичные токенызываются на не связанные между собой константы, которые имеют одно и то же значение, а адрес результирующего выражения не используется (даже неявно путем связывания ссылки на переменную, создающую константу), то такие токены считаются эквивалентными. Это исключение позволяет использовать следующие программные структуры:

```
// ===== header.hpp:  
#ifndef HEADER_HPP  
#define HEADER_HPP  
int const length = 10;  
class MiniBuffer  
{  
    char buf[length];  
    ...  
};  
#endif // HEADER_HPP
```

В принципе, когда этот заголовочный файл включается в две разные единицы трансляции, создаются две отдельные константные переменные под названием `length`, поскольку в данном контексте `const` подразумевает `static`. Однако такие константные переменные часто предназначены для определения константных значений времени компиляции, а не конкретных адресов памяти во время выполнения программы. Следовательно, если нас ничто не заставляет иметь конкретное место в памяти (например, обращение к переменной по адресу), то двум константам вполне достаточно иметь одно и то же значение.

И наконец, замечание о шаблонах. Имена в шаблонах связываются в две фазы. Так называемые *независимые имена* связываются в момент, когда шаблон определяется. Для них правила эквивалентности обрабатываются таким же образом, как и для других нешаблонных определений. Для имен, которые связываются в точке инстанцирования, правила эквивалентности должны быть применимы в этой точке и связывания должны быть эквивалентны.

Приложение Б

Категории значений

Выражения являются краеугольным камнем языка C++, предоставляя основной механизм, позволяющий выразить необходимые вычисления. Каждое выражение имеет тип, который описывает статический тип значения, производимого при вычислении. Так, выражение 7 имеет тип `int`, как и выражение `5+2` или выражение `x`, если `x` является переменной типа `int`. Каждое выражение имеет также *категорию значения* (*value category*), которая описывает то, как было сформировано значение, и влияет на поведение выражения.

Б.1. Традиционные l- и r-значения

Исторически изначально имелись только две категории значений: l-значения и r-значения. L-значениями являются выражения, ссылающиеся на фактические значения, хранящиеся в памяти или в регистре компьютера, такие как выражение `x`, где `x` — имя переменной. Эти выражения могут быть изменяемыми, позволяя обновлять сохраненное значение. Например, если `x` является переменной типа `int`, то следующее присваивание заменит значение `x` на 7:

```
x = 7;
```

Термин *l-значение* (*lvalue*) является производным от роли, которую это выражение могло бы играть в рамках присваивания: буква *l* означает “левый” потому что (исторически в языке программирования С) слева от знака присваивания могут находиться только l-значения. И наоборот, *r-значения* (*rvalues*, где *r* означает “справа”) может находиться только в правой части выражения присваивания.

Однако после стандартизации языка С в 1989 году все изменилось: хотя по-прежнему значение `int const` хранится в памяти, оно не может находиться в левой части присваивания:

```
int const x; // x - неизменяемое значение
x = 7;        // Ошибка: слева должно быть изменяемое значение
```

Изменения в C++ пошли еще дальше: класс r-значений может находиться слева в присваиваниях. Такие присваивания на самом деле являются вызовами функций операторов присваивания соответствующих классов, а не “простыми” присваиваниями скалярных типов, так что они управляются (отдельными) правилами вызовов функций-членов.

Учитывая все эти изменения, термин *l-значение* теперь иногда трактуется как *локализуемое значение*. Выражения, которые ссылаются на переменную, являются не единственными выражениями l-значений. Еще один класс выражений, которые являются l-значениями, включает операцию разыменования указателя (например, `*p`), и ссылается на значения, хранящиеся по адресу, на который указывает указатель, а также выражения, которые ссылаются на члены объектов

классов (например, `p->data`). Даже вызовы функций, возвращающих значения “традиционных” ссылочных на l-значения типов, объявленных с использованием &, являются l-значениями. Например (см. подробности в разделе Б.4):

```
std::vector<int> v;
v.front() // l-значение, так как возвращаемый тип
           // является ссылкой на l-значение
```

Возможно, это покажется удивительным, но строковые литералы также являются (неизменяемыми) l-значениями.

R-значения являются чисто математическими значениями (например, 7 или символ 'a'), которые не обязательно имеют связанное с ними место хранения; они существуют только для вычислений, но после того, как они были использованы, на них нельзя ссылаться. В частности, любые литеральные значения, за исключением строковых литералов (например, 7, 'a', `true`, `nullptr`), являются r-значениями, как и результаты многих встроенных арифметических вычислений (например, `x+5` для `x` целочисленного типа) и вызовов функций, возвращающих результат по значению. То есть все временные значения являются r-значениями. (Хотя это не относится к именованным ссылкам, которые ссылаются на них.)

Б.1.1. Преобразования l-значений в r-значения

В силу своей эфемерной природы r-значения ограничены присутствием только с правой стороны (“простых”) присваиваний: присваивание `7=8` не имеет смысла, потому что математическое значение 7 не может быть переопределено. С другой стороны, l-значения подобного ограничения не имеют: безусловно можно вычислить присваивание `x=y`, когда и `x`, и `y` являются переменными совместимого типа, даже если оба выражения `x` и `y` являются l-значениями.

Присваивание `x=y` работает благодаря тому, что выражение справа, `y`, подвергается неявному преобразованию, которое называется *преобразованием l-значения в r-значение* (*lvalue-to-rvalue conversion*). Как предполагает название, это преобразование принимает l-значение и выполняет чтение из памяти или регистра, связанных с l-значением, r-значения того же типа. Таким образом, это преобразование выполняет два действия: во-первых, оно гарантирует, что l-значения могут использоваться везде, где ожидается r-значение (например, в правой части присваивания или в математическом выражении наподобие `x+y`). Во-вторых, оно определяет, где в программе компилятор может включать (до оптимизации) инструкцию “загрузка” для чтения значения из памяти.

Б.2. Категории значений, начиная с C++11

Когда в C++11 для поддержки семантики перемещения были введены ссылки на r-значение, традиционного деления выражений на l-значения и r-значения стало недостаточно, чтобы описать все поведение C++11. Поэтому комитет по стандартизации C++ изменил систему категорий значений, основав ее на трех основных и двух составных категориях (см. рис. Б.1). Основными категориями

являются: *l-значение*, *pr-значение* (pure rvalue – чистое r-значение) и *x-значение*. Составными категориями являются *gl-значение* (generalized lvalue – обобщенное l-значение, которое является объединением l-значения и x-значения) и *r-значение* (объединение x-значения и pr-значения).

Обратите внимание на то, что все выражения по-прежнему являются l-значениями или r-значениями, но теперь категория r-значения разделяется.

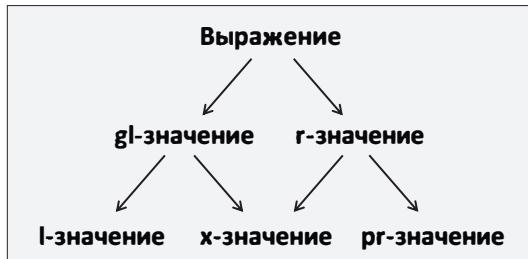


Рис. Б.1. Категории значений, начиная с C++11

Эта категоризация C++11 остается в силе, но в C++17 характеристики категорий были пересмотрены следующим образом.

- **gl-значение** представляет собой выражение, вычисление которого определяет идентичность объекта, битового поля или функции (сущности, имеющей место хранения).
- **pr-значение** представляет собой выражение, вычисление которого инициализирует объект или битовое поле, или вычисляет значение операнда оператора.
- **x-значение** представляет собой gl-значение, обозначающее объект или битовое поле, ресурсы которого могут быть повторно использованы (обычно из-за того, что вскоре истечет его время жизни – так что “*x*” в *xvalue* означает “expiring value” – завершающееся значение).
- **l-значение** представляет собой gl-значение, не являющееся x-значением.
- **r-значение** представляет собой выражение, являющееся либо pr-значением, либо x-значением.

Обратите внимание на то, что в C++17 (и в некоторой степени в C++11 и C++14) разделение gl-значений и pr-значений, возможно, более фундаментальное, чем традиционное отличие l-значений от r-значений.

Хотя это описание характеристик введено в C++17, оно применимо и в C++11 и C++14 (более ранние описания эквивалентны по смыслу, но с ними трудно работать).

За исключением битовых полей, gl-значения производят сущности с адресами. Этот адрес может быть адресом подобъектов большего охватывающего объекта. В случае подобъекта базового класса тип gl-значения (выражения) называется *статическим типом*, а тип наиболее производного объекта, частью которого

является базовый класс, называется *динамическим типом gl-значения*. Если gl-значение не производит подобъект базового класса, статические и динамические типы являются идентичными (типов выражения).

Примеры *l-значений* таковы.

- Выражения, обозначающие переменные или функции.
- Применения встроенного унарного оператора * (разыменование указателя).
- Выражение, являющееся простым строковым литералом.
- Вызов функции с возвращаемым типом, являющимся ссылкой на l-значение.

Примеры *pr-значений* таковы.

- Выражения, которые состоят из литерала, не являющегося строковым литералом или литералом, определяемым пользователем¹.
- Применения встроенного унарного оператора & (взятие адреса выражения).
- Применение встроенных арифметических операторов.
- Вызов функции с возвращаемым типом, который *не является* ссылочным типом.
- Лямбда-выражения.

Примеры *x-значений* таковы.

- Вызов функции с возвращаемым типом, который представляет собой ссылку на г-значение на тип объекта (например, std::move()).
- Приведение к г-значению ссылки на тип объекта.

Обратите внимание на то, что ссылки на г-значение на типы функций производят l-значения, а не x-значения.

Стоит подчеркнуть, что gl-значения, pr-значения, x-значения и так далее являются *выражениями*, а *не значениями*² или сущностями. Например, переменная не является l-значением, несмотря на то, что выражение, описывающее переменную, является таковым:

```
int x = 3; // Здесь x - переменная, но не l-значение. З является
           // pr-значением, инициализирующим переменную x.
int y = x; // Здесь x - l-значение. Вычисление этого выражения
           // l-значения не дает значения 3, но обозначает объект,
           // содержащий значение 3. Такое l-значение затем
           // преобразуется в pr-значение, которое инициализирует y.
```

¹ Пользовательские литералы могут давать l-значения или г-значения, в зависимости от возвращаемого типа соответствующего литерального оператора. К сожалению, это означает, что используемые термины являются неподходящими.

² К сожалению, это означает, что используемые термины являются неподходящими.

Б.2.1. Временная материализация

Ранее мы упоминали, что l-значения часто проходят преобразования³ l-значения в r-значение, потому что rг-значения представляют собой разновидность выражений, которые инициализируют объекты (или предоставляют операнды для большинства встроенных операторов).

В C++17 существует дуальное преобразование к данному, известное как *временная материализация* (но его можно было бы также назвать преобразованием rг-значения в x-значение): в любой момент, когда rг-значение корректно появляется там, где ожидается gl-значение (которое включает x-значение), создается временный объект, который инициализируется rг-значением (вспомните, что rг-значения являются главным образом инициализирующими значениями), и rг-значение заменяется временным x-значением. Например:

```
int f(int const&);  
int r = f(3);
```

Поскольку `f()` в этом примере имеет ссылочный параметр, ожидается аргумент, являющийся gl-значением. Однако выражение `3` является rг-значением. Поэтому вступает в дело правило временной материализации, и выражение `3` “преобразуется” в x-значение, обозначающее временный объект, инициализированный значением `3`.

В целом временное значение материализуется с инициализацией rг-значением в следующих ситуациях.

- rг-значение связано со ссылкой (например, как в вызове `f(3)` выше).
- Доступ к члену rг-значения классового типа.
- Индексация массива rг-значений.
- Массив rг-значений преобразуется в указатель на его первый элемент (*низведение массива*).
- rг-значение находится в списке инициализации в фигурных скобках, который для некоторого типа `X` инициализирует объект типа `std::initializer_list<X>`.
- Применение операторов `sizeof` или `typeid` к rг-значению.
- rг-значение представляет собой выражение верхнего уровня в инструкции вида “`expr;`” или выражение, приводимое к `void`.

Таким образом, в C++17 объект, инициализированный rг-значением, всегда определяется контекстом, и в результате временные значения создаются только тогда, когда они действительно необходимы. До C++17 rг-значения (в частности, классового типа) всегда подразумевали временные объекты. Копии этих временных объектов позже могли (неизбежно) быть устранины, но компилятор по-прежнему должен был обеспечивать большинство семантических ограничений

³ В мире категорий значений C++11 более точной будет фраза *преобразование gl-значения в pr-значение*, но традиционный термин остается более распространенным.

операции копирования (например, мог требоваться вызываемый копирующий конструктор). В следующем примере показано следствие пересмотра правил в C++17:

```
class N
{
public:
    N();
    N(N const&) = delete; // Этот класс некопируемый ...
    N(N&&) = delete;     // ... и не перемещаемый
};

N make_N()
{
    return N{};           // До C++17 всегда создавался концептуальный
                        // временный объект. В C++17 временный объект
                        // в этой точке не создается.
auto n = make_N(); // До C++17 – ошибка, так как pr-значение требует
                    // концептуального копирования. Начиная с C++17,
                    // все в порядке, так как n инициализируется
                    // непосредственно pr-значением.
```

До C++17 pr-значение N{} создавало временный объект типа N, но компиляторам разрешалось устранять копирования и перемещения этого временного объекта (что всегда делалось на практике). В данном случае это означает, что временный результат вызова make_N() может быть сконструирован непосредственно в хранилище n; при этом не требуется никаких операций копирования или перемещения. К сожалению, компиляторы, соответствующие стандартам до C++17, по-прежнему должны проверять, что в этом примере *могут* быть выполнены операции копирования или перемещения; это не представляется возможным, потому что копирующий и перемещающий конструкторы N удалены. Таким образом, компиляторы с поддержкой стандарта C++11 и C++14 должны выдавать для этого примера сообщение об ошибке.

В случае C++17 pr-значение N само по себе не создает временный объект. Вместо этого оно инициализирует объект, определяемый контекстом. В нашем примере этим объектом является объект, обозначаемый как n. Никакие операции копирования или перемещения не рассматриваются (и это не оптимизация, а гарантия языка), так что приведенный код является корректным кодом C++17.

В заключение приведем пример, который показывает различные ситуации с категориями значений:

```
class X
{
};

X v;
X const c;

void f(X const&); // Принимает выражение любой категории значений
void f(X&&);   // Принимает только pr- и x-значения, но
                  // демонстрирует для них лучшее соответствие,
                  // чем предыдущее объявление
f(v);            // Передача модифицируемого l-значения в первую f()
```

```
f(c); // Передача неизменяемого l-значения в первую f()
f(X()); // Передача pr-значения (начиная с C++17
         // материализуется как x-значение) во вторую f()
f(std::move(v)); // Передача x-значения во вторую f()
```

Б.3. Проверка категорий значений с помощью decltype

С помощью ключевого слова `decltype` (введенного в C++11) можно проверить категорию значения любого выражения C++. Для любого выражения `x decltype((x))` (обратите внимание на двойные скобки!) дает:

- `type`, если `x` представляет собой `rg`-значение;
- `type&`, если `x` представляет собой `l`-значение;
- `type&&`, если `x` представляет собой `x`-значение.

Двойные скобки в `decltype((x))` необходимы во избежание получения объявленного типа именованной сущности в случае, когда выражение `x` представляет собой имя сущности (в других случаях скобки не производят никакого действия). Например, если выражение `x` представляет собой просто имя переменной `v`, то конструкция без скобок `decltype(v)` вернет тип переменной `v`, а не тип, отражающий категорию значения выражения `x`, ссылающегося на эту переменную.

Таким образом, используя свойства типов для выражения `e`, мы можем проверить его категорию следующим образом:

```
if constexpr(std::is_lvalue_reference<decltype((e))>::value)
{
    std::cout << "Выражение является l-значением\n";
}
else if constexpr(std::is_rvalue_reference<decltype((e))>::value)
{
    std::cout << "Выражение является x-значением\n";
}
else
{
    std::cout << "Выражение является pr-значением\n";
}
```

Подробнее этот вопрос рассматривается в разделе 15.10.2.

Б.4. Ссылочные типы

Ссылочные типы в C++ – такие как `int&` – взаимодействуют с категориями значений двумя важными способами. Во-первых, ссылка может ограничить категорию значения выражения, с которым она связана. Например, неконстантная ссылка на `l`-значение типа `int&` может быть инициализирована только выражением, которое является `l`-значением типа `int`. Аналогично ссылка на `g`-значение типа `int&&` может быть инициализирована только выражением, которое представляет собой `g`-значение типа `int`.

Во-вторых, значение категории взаимодействует со ссылками в возвращаемых типах функций, где использование ссылочного типа в качестве возвращаемого типа влияет на категорию значения вызова этой функции. В частности:

- вызов функции, чей возвращаемый тип представляет собой ссылку на l-значение, дает l-значение;
- вызов функции, чей возвращаемый тип представляет собой ссылку на g-значение на тип объекта, дает x-значение (ссылки на g-значение на типы функций всегда дают l-значения);
- вызов функции, которая возвращает нессылочный тип, дает pr-значение.

Проиллюстрируем взаимодействия между ссылочными типами и категориями значений в следующем примере. Пусть имеются следующие объявления:

```
int& lvalue();
int&& xvalue();
int prvalue();
```

`decltype` может определять как категорию значения, так и тип данного выражения. Как показано в разделе 15.10.2, для описания ситуаций, когда выражение является l- или x-значением, используются ссылочные типы:

```
// true, так как результат - l-значение:
std::is_same_v<decltype(lvalue()), int&>
// true, так как результат - x-значение:
std::is_same_v<decltype(xvalue()), int&&>
// true, так как результат - pr-значение:
std::is_same_v<decltype(prvalue()), int>
```

Таким образом, возможны следующие вызовы:

```
// OK: ссылка на l-значение связана с l-значением:
int& lref1 = lvalue();
// Ошибка: ссылка на l-значение не может быть связана с pr-значением:
int& lref3 = prvalue();
// Ошибка: ссылка на l-значение не может быть связана с x-значением:
int& lref2 = xvalue();
// Ошибка: ссылка на r-значение не может быть связана с l-значением:
int&& rref1 = lvalue();
// OK: ссылка на r-значение может быть связана с pr-значением:
int&& rref2 = prvalue();
// OK: ссылка на r-значение может быть связана с x-значением:
int&& rref3 = xvalue();
```

Приложение В

Разрешение перегрузки

Разрешение перегрузки — это процесс выбора функции, к которой обращается данное выражение вызова. Рассмотрим простой пример:

```
void display_num(int);      // #1
void display_num(double);   // #2
int main()
{
    display_num(399);       // #1 соответствует лучше, чем #2
    display_num(3.99);      // #2 соответствует лучше, чем #1
}
```

Принято говорить, что в этом примере имя функции `display_num()` *перегружено*. Когда это имя используется в вызове, компилятор C++ должен выбирать вызываемую функцию среди различных кандидатов с использованием дополнительной информации, главным образом о типах аргументов вызова. В нашем примере интуиция подсказывает целесообразность вызова `int`-версии, когда функция вызывается с целочисленным аргументом, и `double`-версии, когда используется аргумент с плавающей точкой. Формальный процесс попытки моделирования этого интуитивного выбора — это и есть процесс разрешения перегрузки.

Общие идеи, лежащие в основе принципов, регулирующих разрешение перегрузки, достаточно просты. Однако подробности их реализации значительно усложнились в процессе стандартизации C++. Эта сложность — в основном результат желания обеспечить поддержку реально существующих примеров, которые интуитивно кажутся человеку “наилучшим выбором”. Однако при формализации такого интуитивного выбора возникают различные тонкости.

В этом приложении достаточно подробно рассматриваются правила разрешения перегрузки. Однако этот процесс настолько сложен, что мы не можем претендовать на полноту охвата данной темы.

B.1. Когда используется разрешение перегрузки

Разрешение перегрузки — это только одна часть процесса обработки вызова функции. В действительности разрешение перегрузки не является частью любого вызова функции. Во-первых, для вызовов с помощью указателей и вызовов с помощью указателей на функции-члены не требуется разрешения перегрузки, поскольку вызываемая функция полностью определена (во время работы программы) с помощью указателей. Во-вторых, макросы функционального вида нельзя перегружать, поэтому они не попадают под действие разрешения перегрузки.

На очень высоком уровне вызов именованной функции может быть обработан следующим образом.

- Поиск имени создает начальное множество перегрузки.
- При необходимости это множество может быть уточнено различными способами (например, путем вывода аргументов шаблонов и подстановкой, что может привести к отбрасыванию некоторых кандидатов — шаблонов функций).
- Любая функция-кандидат, которая совсем не соответствует вызову (даже после учета неявных преобразований и аргументов по умолчанию), удаляется из множества перегрузки. В результате формируется множество так называемых жизнеспособных функций-кандидатов (viable function candidates).
- Для поиска *наилучшего* кандидата выполняется разрешение перегрузки. Если таковой есть, выбираем его; в противном случае исход вызова неоднозначен.
- Выбранный кандидат проверяется. Например, если это удаленная функция (определенная с использованием `=delete`) или недоступный закрытый член, выводится соответствующая диагностика).

Каждый из этих шагов имеет свои тонкости, однако можно утверждать, что разрешение перегрузки — самый сложный из них. К счастью, несколько простых принципов позволяют прояснить большинство ситуаций. Далее эти принципы будут подробно рассмотрены.

B.2. Упрощенное разрешение перегрузки

Разрешение перегрузки располагает жизнеспособные функции-кандидаты по рангу путем сравнения того, насколько каждый аргумент вызова совпадает с соответствующим параметром кандидата. Один из кандидатов считается лучше другого, если любой из его параметров соответствует вызову не хуже, чем соответствующий параметр другого кандидата. Данный подход можно проиллюстрировать приведенным ниже примером:

```
void combine(int, double);
void combine(long, int);

int main()
{
    combine(1, 2); // Неоднозначность!
}
```

В этом примере вызов `combine()` неоднозначен, поскольку первый кандидат *лучше* соответствует первому аргументу (литерал 1 типа `int`), тогда как второй кандидат *лучше* соответствует второму аргументу. Можно утверждать, что `int` в некотором смысле ближе к `long`, чем к `double` (что поддерживает выбор в пользу второго кандидата). Однако C++ не пытается определить меру близости для нескольких аргументов вызова.

Для этого принципа необходимо указать, насколько хорошо данный аргумент совпадает с соответствующим параметром жизнеспособного кандидата. В первом приближении возможные соответствия можно расположить по рангу (от лучшего к худшему) так, как описано ниже.

1. Точное соответствие. Тип параметра совпадает с типом выражения, или его тип является ссылкой на тип выражения (возможно, с добавлением спецификаторов `const` и/или `volatile`).
2. Соответствие, достигаемое минимумом подгонок. Включает, например, низведение переменной массива к указателю на его первый элемент или добавление спецификатора `const` для обеспечения соответствия аргумента типа `int**` параметру типа `int const* const*`.
3. Соответствие, достигаемое расширением (promotion) типа. Расширение – это вид неявного преобразования, которое включает преобразование меньших целочисленных типов (таких как `bool`, `char`, `short`, и иногда перечислимых типов) в тип `int`, `unsigned int`, `long` или `unsigned long`, а также преобразование типа `float` в тип `double`.
4. Соответствие, достигаемое только за счет стандартного преобразования. Оно включает любой вид стандартного преобразования (например, типа `int` в тип `float`) или преобразование производного класса в один из его открытых однозначных базовых классов, но исключает неявный вызов оператора преобразования или конструктора преобразования.
5. Соответствие за счет преобразования, определяемого пользователем. Позволяет выполнять любой тип неявного преобразования.
6. Соответствие за счет многоточия (...) в объявлении функции. Параметр, подставляемый на место многоточия, может совпадать почти с любым типом. Однако имеется одно исключение: типы классов с нетривиальными копирующими конструкторами могут рассматриваться и как допустимые, и как недопустимые (реализация компилятора может их разрешать или запрещать).

Приведенный ниже запутанный пример иллюстрирует некоторые из этих соответствий.

```

int f1(int);      // #1
int f1(double);  // #2
f1(4);           // Вызов #1 : точное соответствие (#2 требует
                  // применения стандартного преобразования)
int f2(int);     // #3
int f2(char);    // #4
f2(true);        // Вызов #3 : соответствие с расширением (#4 требует
                  // более сильного стандартного преобразования)
class X
{
  public:
    X(int);
};
int f3(X);       // #5

```

```
int f3(...);      // #6
f3(7);          // Вызов #5 : соответствие с помощью пользовательского
                // преобразования (#6 требует соответствия многоточию)
```

Отметим, что разрешение перегрузки осуществляется *после* вывода аргумента шаблона, и этот вывод не учитывает все эти виды преобразований. Проиллюстрируем это на примере:

```
template<typename T>
class MyString
{
public:
    MyString(T const*);      // Преобразующий конструктор
    ...
};

template<typename T>
MyString<T> truncate(MyString<T> const&, int);
int main()
{
    MyString<char> str1, str2;
    str1 = truncate<char>("Hello World", 5); // OK
    str2 = truncate("Hello World", 5);        // Ошибка
}
```

Неявное преобразование, предусмотренное в преобразующем конструкторе, не учитывается при выводе аргумента шаблона. Присваивание `str2` не обнаруживает жизнеспособной функции `truncate()`, так что разрешение перегрузки не выполняется вообще.

В контексте вывода аргумента шаблона вспомним также, что ссылка на `г`-значение на параметр шаблона может быть выведена либо как тип ссылки на `л`-значение (после свертки ссылок), если соответствующий аргумент представляет собой `л`-значение, либо как тип ссылки на `г`-значение, если этот аргумент представляет собой `г`-значение (см. раздел 15.6). Например:

```
template<typename T> void strange(T&&, T&&);
template<typename T> void bizarre(T&&, double&&);

int main()
{
    strange(1.2, 3.4); // OK: Т выводится как double
    double val = 1.2;
    strange(val, val); // OK: Т выводится как double&
    strange(val, 3.4); // Ошибка: конфликт вывода
    bizarre(val, val); // Ошибка: л-значение не соответствует double&&
}
```

Изложенные принципы — это только первое приближение, но они охватывают очень много случаев. Однако имеется ряд распространенных ситуаций, которые нельзя адекватно объяснить этими правилами. Далее кратко обсуждаются самые важные уточнения этих правил.

B.2.1. Неявный аргумент для функций-членов

Вызовы нестатических функций-членов имеют скрытый параметр, доступ к которому возможен в определении функции-члена как к `*this`. Для функции-члена класса `MyClass` скрытый параметр обычно имеет тип `MyClass&` (для функций-членов неконстантного типа) или `MyClass const&` (для функций-членов константного типа)¹. Несколько удивляет то, что `this` имеет тип указателя. Лучше было бы сделать его эквивалентным параметру, который сейчас определен как `*this`. Однако `this` был частью первых версий языка C++ еще до того, как в нем появился ссылочный тип данных. К моменту введения этого типа данных было уже написано слишком много кода, который зависел от параметра `this`, являющегося указателем.

Скрытый параметр `*this` принимает участие в разрешении перегрузки точно так же, как и явные параметры. Почти всегда это вполне естественно, но иногда — неожиданно. В следующем примере приведен строковый класс, который работает не так, как ожидалось (такой код приходится видеть и в реальной жизни):

```
#include <cstddef>
class BadString
{
public:
    BadString(char const*);
    ...
    // Доступ к символу через индексацию:
    char& operator[](std::size_t); // #1
    char const& operator[](std::size_t) const;
    // Неявное преобразование в строку в стиле C:
    operator char*(); // #2
    operator char const*();
    ...
};

int main()
{
    BadString str("correkt");
    str[5] = 'c'; // Возможная неоднозначность перегрузки!
}
```

Вначале кажется, что в выражении `str[5]` нет ничего неоднозначного. Оператор индексации в строке #1 выглядит совершенно корректно. Однако это не *совсем* так, поскольку аргумент 5 имеет тип `int`, а оператор ожидает целое число без знака (`size_t` и `std::size_t` обычно имеют тип `unsigned int` или `unsigned long`, но никогда не `int`). При этом простое стандартное преобразование целочисленного типа легко делает #1 жизнеспособным. Вместе с тем есть и другой жизнеспособный кандидат — встроенный оператор индексации. Действительно, если применить оператор неявного преобразования к типу `str` (который является неявным аргументом функции-члена), получится указатель.

¹ Это может быть также тип `MyClass volatile&` или `MyClass const volatile&`, если функция-член описана как `volatile`, но это чрезвычайно редкий случай.

И теперь можно применять встроенный оператор индексации. Этот встроенный оператор принимает аргумент типа `ptrdiff_t`, который на многих платформах эквивалентен `int`, и потому полностью соответствует аргументу 5. Поэтому, даже если встроенный оператор индексации плохо соответствует неявному аргументу (с помощью пользовательского преобразования типов), это все же лучшее соответствие, чем оператор, определенный в #1 для действительной индексации! Это источник потенциальной неоднозначности². Чтобы решить эту проблему для конкретной платформы, необходимо объявить оператор `[]` с параметром `ptrdiff_t` либо заменить неявное преобразование типа в `char*` явным (что обычно рекомендуется делать в любом случае, независимо от прочих моментов).

Множество жизнеспособных кандидатов может содержать как статические, так и нестатические члены. При сравнении статического члена с нестатическим качество соответствия неявных аргументов игнорируется (только нестатический член имеет неявный параметр `*this`).

По умолчанию нестатическая функция-член имеет неявный параметр `*this`, который имеет тип ссылки на l-значение, но в C++11 введен синтаксис, позволяющий сделать его ссылкой на r-значение. Например:

```
struct S
{
    void f1();    // *this неявно является ссылкой на l-значение
    void f2()&&; // *this явно является ссылкой на r-значение
    void f3()&;  // *this явно является ссылкой на l-значение
};
```

Как видно из этого примера, можно не только сделать неявный параметр ссылкой на r-значение (с помощью суффикса `&&`), но и подтвердить случай ссылки на l-значение (с помощью суффикса `&`). Интересно, что указание суффикса `&` не эквивалентно его отсутствию: старый частный случай позволяет r-значению связываться с ссылкой на l-значение для неконстантного типа, когда эта ссылка является традиционным неявным параметром `*this`, но этот (несколько опасный) частный случай не применим, если ссылка на l-значение была запрошена явно. Таким образом, при указанном выше определении S:

```
int main()
{
    S().f1(); // OK: старое правило позволяет r-значению S()
              // соответствовать типу неявной ссылки на
              // l-значение S& параметра *this
    S().f2(); // OK: r-значение S() соответствует типу ссылки
              // на r-значение параметра *this
    S().f3(); // Ошибка: r-значение S() не может соответствовать
              // явному типу ссылки на l-значение параметра *this
}
```

² Отметим, что неоднозначность существует только на платформах, где `size_t` — синоним для `unsigned int`. На платформах, где это синоним для `unsigned long`, тип `ptrdiff_t` — синоним типа `long` и неоднозначности нет, поскольку встроенный оператор индексации также требует преобразования индексного выражения.

B.2.2. Улучшение точного соответствия

Для аргумента типа X имеется четыре типа параметров, которые дают точное соответствие: X, X&, X const& и X&& (X const&& также дает точное соответствие, но редко применяется). Однако чаще всего происходит перегрузка функции по двум видам ссылок. До C++11 это означало наличие перегрузок наподобие показанной:

```
void report(int&);           // #1
void report(int const&);     // #2
int main()
{
    for (int k = 0; k < 10; ++k)
    {
        report(k);          // Вызов #1
    }

    report(42);            // Вызов #2
}
```

Здесь версия без const предпочтительна для l-значений, в то время как для r-значений годится только версия с const.

С появлением ссылок на r-значения в C++11 требуется различать еще один распространенный случай двух точных соответствий, проиллюстрированных в следующем примере:

```
struct Value
{
    ...
};

void pass(Value const&); // #1
void pass(Value&&);   // #2
void g(X&& x)
{
    pass(x);              // Вызов #1, т.к. x является l-значением
    pass(X());             // Вызов #2, т.к. X() является r-значением
                           // (фактически - pr-значением)
    pass(std::move(x));   // Вызов #2, т.к. std::move(x) является
                           // r-значением (фактически - x-значением)
}
```

В этот раз версия, принимающая ссылку на r-значение, рассматривается как имеющая лучшее соответствие для r-значений, но не может соответствовать l-значениям.

Обратите внимание на то, что это относится также к неявному аргументу вызова функции-члена:

```
class Wonder
{
public:
    void tick();           // #1
    void tick() const;     // #2
    void tack() const;     // #3
};
```

```
void run(Wonder& device)
{
    device.tick();      // Вызов #1
    device.tack();      // Вызов #3, т.к. нет неконстантной
}                      // версии Wonder::tack()
```

Наконец, следующее изменение нашего предыдущего примера показывает, что два точных соответствия также могут создать неоднозначность, если выполняется перегрузка со ссылкой и без нее:

```
void report(int);          // #1
void report(int&);        // #2
void report(int const&);  // #3
int main()
{
    for (int k = 0; k < 10; ++k)
    {
        report(k); // Неоднозначность: #1 и #2
    }              // подходят одинаково хорошо

    report(42);    // Неоднозначность: #1 и #3
}                      // подходят одинаково хорошо
```

B.3. Детали перегрузки

Предыдущий раздел охватывает большую часть ситуаций перегрузки, которые встречаются в повседневном программировании на C++. К сожалению, есть еще очень много правил и исключений из этих правил – больше, чем было бы разумно включать в книгу, которая не посвящена перегрузке функций в C++. Тем не менее обсудим некоторые из них, поскольку они используются несколько чаще, чем другие правила, и с той степенью подробности, которая здесь имеет смысл.

B.3.1. Предпочтение нешаблонных функций или более специализированных шаблонов

Если все прочие аспекты разрешения перегрузки равны, нешаблонная функция предпочтительнее экземпляра шаблонной функции (не имеет значения, сгенерирован экземпляр последней из определения обобщенного шаблона или же получен в результате явной специализации). Например:

```
template<typename T> int f(T); // #1
void f(int);                  // #2
int main()
{
    return f(7);               // Ошибка: выбрана функция #2,
}                                // не возвращающая значения
```

Этот пример также ясно показывает, что разрешение перегрузки обычно не включает возвращаемый тип выбранной функции.

Однако при незначительных отличиях других аспектов разрешения перегрузки (например, различные квалификаторы `const` или ссылки) сначала применяются общие правила разрешения перегрузки. Это может случайно вызывать удивительное

поведение, когда функции-члены определены как принимающие те же аргументы, что и копирующие или перемещающие конструкторы (см. раздел 16.2.4).

Если делается выбор из двух шаблонов, то предпочтение отдается *более специализированному* (при условии, что один из них более специализирован, чем другой). Подробное объяснение этого подхода дано в разделе 16.2.2. Один частный случай такого предпочтения — когда два шаблона отличаются только тем, что один добавляет завершающий пакет параметров: шаблон без пакета параметров считается более специализированным, а потому является более предпочтительным, если он соответствует вызову. Пример этой ситуации можно найти в разделе 4.1.2.

B.3.2. Последовательности преобразований

Неявное преобразование в общем случае может быть последовательностью элементарных преобразований. Рассмотрим приведенный ниже код:

```
class Base
{
public:
    operator short() const;
};

class Derived : public Base
{
};

void count(int);
void process(Derived const& object)
{
    count(object); // Соответствие при использовании
                    // пользовательского преобразования
```

Вызов `count (object)` работает, поскольку `object` может быть неявно преобразован в `int`. Однако это преобразование требует, чтобы были выполнены перечисленные ниже действия.

1. Преобразование `object` из `Derived const` в `Base const` (это преобразование gl-значения; оно сохраняет идентичность объекта).
2. Пользовательское преобразование полученного объекта из типа `Base const` в тип `short`.
3. Расширение `short` до `int`.

Это наиболее общий вид последовательности преобразований: стандартное преобразование (в данном случае из производного типа в базовый), затем пользовательское преобразование, после чего другое стандартное преобразование. Хотя в последовательности может быть не более одного пользовательского преобразования, возможна также последовательность, в которую входят только стандартные преобразования.

Важный принцип разрешения перегрузки состоит в том, что если есть последовательность, являющаяся подпоследовательностью другой последовательности

преобразований, то предпочтительнее использовать подпоследовательность. Так, если бы в рассмотренном примере имелась дополнительная функция-кандидат

```
void count(short)
```

то она была бы предпочтительнее из-за отсутствия необходимости третьего пункта (расширения типа) в последовательности преобразований.

B.3.3. Преобразования указателей

Указатели и указатели на члены класса могут подвергаться различным специальным стандартным преобразованиям, включая следующие:

- преобразования в тип `bool`;
- преобразования из типа произвольного указателя в тип `void*`;
- преобразования указателей на производный тип в указатель на базовый тип;
- преобразования указателей на члены базового класса в указатели на члены производного класса.

Хотя все они могут обеспечивать “соответствие за счет только стандартных преобразований”, ранг у них разный.

Прежде всего, преобразование в тип `bool` (как из обычного указателя, так и из указателя на член класса) считается менее предпочтительным, чем любой другой стандартный тип преобразования. Например:

```
void check(void*);      // #1
void check(bool);       // #2
void rearrange(Matrix* m)
{
    check(m);           // Вызов #1
    ...
}
```

В категории преобразований обычных указателей преобразование в тип `void*` считается менее предпочтительным, чем преобразование из указателя на производный класс в указатель на базовый класс. Кроме того, если есть указатели на различные классы, связанные наследованием, то предпочтительнее преобразование в указатель на наиболее производный (т.е. ближайший базовый) класс. Приведем небольшой пример:

```
class Interface
{
    ...
};

class CommonProcesses : public Interface
{
    ...
};

class Machine : public CommonProcesses
{
    ...
};
```

```

char* serialize(Interface*);           // #1
char* serialize(CommonProcesses*);     // #2
void dump(Machine* machine)
{
    char* buffer = serialize(machine); // Вызывает #2
    ...
}

```

Преобразование из `Machine*` в `CommonProcesses*` предпочтительнее, чем преобразование в `Interface*`, что вполне понятно интуитивно.

Схожее правило применяется к указателям на члены класса: из двух преобразований типов указателей на члены классов, связанных наследованием, предпочтительнее преобразование к “ближайшему базовому классу” в диаграмме наследования.

B.3.4. Списки инициализаторов

Аргументы списка инициализаторов (инициализаторы, передаваемые в фигурных скобках) могут быть преобразованы в несколько различных видов параметров: `initializer_list`, классовые типы с конструктором `initializer_list`, классовые типы, для которых элементы списка инициализаторов могут рассматриваться как (отдельные) параметры конструктора, или агрегатные классовые типы, члены которых могут быть инициализированы элементами списка инициализаторов. Следующая программа иллюстрирует эти случаи:

overload/initlist.cpp

```

#include <initializer_list>
#include <string>
#include <vector>
#include <complex>
#include <iostream>

void f(std::initializer_list<int>)
{
    std::cout << "#1\n";
}

void f(std::initializer_list<std::string>)
{
    std::cout << "#2\n";
}

void g(std::vector<int> const& vec)
{
    std::cout << "#3\n";
}

void h(std::complex<double> const& cmplx)
{
    std::cout << "#4\n";
}

```

```

struct Point
{
    int x, y;
};

void i(Point const& pt)
{
    std::cout << "#5\n";
}

int main()
{
    f({1, 2, 3});                                // Выводит #1
    f({"hello", "initializer", "list"});          // Выводит #2
    g({1, 1, 2, 3, 5});                          // Выводит #3
    h({1.5, 2.5});                               // Выводит #4
    i({1, 2});                                   // Выводит #5
}

```

В первых двух вызовах `f()` аргументы списка инициализаторов преобразуются в значения `std::initializer_list`, что включает преобразование каждого из элементов списка в тип элементов списка `std::initializer_list`. В первом вызове все элементы уже имеют тип `int`, поэтому дополнительное преобразование не требуется. Во втором вызове каждый строковый литерал в списке инициализаторов преобразуется в `std::string` путем вызова конструктора `string(char const*)`. Третий вызов (для `g()`) выполняет пользовательское преобразование с помощью конструктора `std::vector(std::initializer_list<int>)`. Следующий за ним вызов вызывает конструктор `std::complex(double, double)`, как если бы мы написали `std::complex<double>(1.5, 2.5)`. Последний вызов выполняет агрегатную инициализацию, инициализируя члены экземпляра класса `Point` элементами из списка инициализаторов без вызова конструктора `Point`³.

Есть несколько интересных случаев перегрузки для списков инициализаторов. При преобразовании списка инициализаторов в `initializer_list`, как в первых двух вызовах в показанном выше примере, полное преобразование получает тот же ранг, что и *наихудшее* преобразование любого заданного элемента в списке инициализатора к типу элемента `initializer_list` (т.е. к `T` в `initializer_list<T>`). Это может привести к некоторым сюрпризам, как показано в следующем примере:

overload/initlistov1.cpp

```
#include <initializer_list>
#include <iostream>
```

³ Агрегатная инициализация доступна только для агрегатных типов C++, которыми являются либо массивы, либо простые C-образные классы, не имеющие ни пользовательских конструкторов, ни закрытых или защищенных нестатических членов-данных, ни базовых классов, ни виртуальных функций. До C++14 они также должны были не иметь инициализаторов элементов по умолчанию. Начиная с C++17, допускается наличие открытых базовых классов.

```

void ovl(std::initializer_list<char>)           // #1
{
    std::cout << "#1\n";
}

void ovl(std::initializer_list<int>)             // #2
{
    std::cout << "#2\n";
}

int main()
{
    ovl({'h', 'e', 'l', 'l', 'o', '\0'}); // Выводит #1
    ovl({'h', 'e', 'l', 'l', 'o', 0});   // Выводит #2
}

```

В первом вызове `ovl()` каждый элемент списка инициализаторов представляет собой `char`. Для первой функции `ovl()` эти элементы не требуют никакого преобразования. Для второй функции `ovl()` требуется расширение до `int`. Поскольку точное соответствие лучше расширения, первый вызов `ovl()` приводит к вызову #1.

Во втором вызове `ovl()` первые пять элементов имеют тип `char`, в то время как последний имеет тип `int`. Для первой функции `ovl()` точно подходят элементы `char`, но `int` требует стандартного преобразования, так что полное преобразование получает тот же ранг, что и стандартное преобразование. Для второй функции `ovl()` элементы `char` требуют расширения до `int`, в то время как элемент `int` в конце списка соответствует точно. Полное преобразование для второй функции `ovl()` ранжируется как расширение, что делает ее лучшим кандидатом, чем первая `ovl()`, несмотря на то, что лучшим оказалось преобразование только одного элемента.

При инициализации объекта классового типа списком инициализаторов, как в вызовах `g()` и `h()` в нашем исходном примере, разрешение перегрузки протекает в два этапа.

- На первом этапе рассматриваются только *конструкторы со списками инициализаторов*, т.е. конструкторы, у которых единственный параметр (к тому же не имеющий значения по умолчанию) имеет тип `std::initializer_list<T>` для некоторого типа `T` (после удаления ссылки верхнего уровня и квалификаторов `const/volatile`).
- Если такой жизнеспособный конструктор не был найден, то на втором этапе рассматриваются все прочие конструкторы.

Существует одно исключение из этого правила: если список инициализаторов пуст, а класс имеет конструктор по умолчанию, первый этап пропускается, так что будет вызван конструктор по умолчанию.

Результат применения этого правила состоит в том, что *любой* конструктор со списком инициализаторов имеет лучшее соответствие, чем любой конструктор без списка инициализаторов, как показано в следующем примере:

overload/initlistctor.cpp

```
#include <initializer_list>
#include <string>
#include <iostream>

template<typename T>
struct Array
{
    Array(std::initializer_list<T>)
    {
        std::cout << "#1\n";
    }
    Array(unsigned n, T const&)
    {
        std::cout << "#2\n";
    }
};

void arr1(Array<int>)
{
}

void arr2(Array<std::string>)
{
}

int main()
{
    arr1({1, 2, 3, 4, 5}); // Выводит #1
    arr1({1, 2}); // Выводит #1
    arr1({10u, 5}); // Выводит #1
    arr2({"hello", "initializer", "list"}); // Выводит #1
    arr2({10, "hello"}); // Выводит #2
}
```

Обратите внимание: второй конструктор, который принимает `unsigned` и `T const&`, при инициализации объекта `Array<int>` списком инициализаторов не вызывается, потому что его конструктор со списком инициализаторов всегда имеет лучшее соответствие, чем его конструкторы без списка инициализаторов. Однако в случае `Array<string>` во втором вызове `arr2()` будет вызван конструктор без списка инициализаторов, так как конструктор со списком инициализаторов не является жизнеспособным.

B.3.5. Функторы и функции-суррогаты

Ранее упоминалось, что после поиска имени функции для формирования начального множества перегрузки это множество обрабатывается различными способами. Интересная ситуация возникает, когда выражение вызова ссылается на объект классового типа, а не на функцию. В этом случае возможны два потенциальных дополнения ко множеству перегрузки.

Первое дополнение очевидно: ко множеству можно добавить любой оператор-член () (оператор вызова функции). Объекты с такими операторами обычно называются *функциями-суррогатами* (см. раздел 11.1).

Менее очевидное дополнение возникает в ситуации, когда объект классового типа содержит оператор неявного преобразования в указатель на тип функции (или в ссылку на тип функции)⁴. В такой ситуации к набору перегрузки добавляется фиктивная функция (так называемая функция-суррогат). Эта функция-суррогат рассматривается как имеющая неявный параметр, тип которого определяется функцией преобразования, в дополнение к параметрам, типы которых соответствуют таковым в целевой функции. Приведенный ниже пример значительно проясняет ситуацию:

```
using FuncType = void (double, int);

class IndirectFunctor
{
public:
    ...
    void operator()(double, double) const;
    operator FuncType* () const;
};

void activate(IndirectFunctor const& funcObj)
{
    funcObj(3, 5); // Ошибка: неоднозначность
}
```

Вызов funcObj(3, 5) рассматривается как вызов с тремя аргументами: funcObj, 3 и 5. Жизнеспособные функции-кандидаты включают член operator() (который рассматривается как имеющий параметры типа IndirectFunctor const&, double и double) и функцию-суррогат с параметрами типа FuncType*, double и int. Суррогат обладает худшим соответствием для неявного параметра (поскольку требуется пользовательское преобразование), однако лучшим соответствием для последнего параметра. Следовательно, невозможно отдать предпочтение какому-либо из этих кандидатов, поэтому вызов будет неоднозначным.

Функции-суррогаты относятся к одной из самых “темных” областей C++ и редко используются на практике (к счастью).

В.3.6. Другие контексты перегрузки

Мы уже обсудили перегрузку в контексте определения того, к какой функции должно идти обращение в выражении вызова. Однако существует несколько других контекстов, в которых также необходимо сделать подобный выбор.

Первый контекст возникает, когда требуется адрес функции. Рассмотрим следующий пример:

⁴ Оператор преобразования должен также быть применим, в том смысле, что, например, оператор без спецификатора const не может использоваться с объектами, имеющими спецификатор const.

```
int numElems(Matrix const&); // #1
int numElems(Vector const&); // #2
...
int (*funcPtr)(Vector const&) = numElems; // Выбирается #2
```

Здесь имя numElems обращается ко множеству перегрузки, однако необходим адрес только одной функции. Затем разрешение перегрузки проверяет соответствие нужного типа функции (тип funcPtr в этом примере) одному из доступных кандидатов.

Другой контекст, который требует разрешения перегрузки, — это *инициализация*. К сожалению, в этой теме есть масса тонкостей, которые невозможно охватить в данном приложении. Тем не менее приведем простой пример, который хотя бы проиллюстрирует этот дополнительный аспект разрешения перегрузки.

```
#include <string>

class BigNum
{
public:
    BigNum(long n);           // #1
    BigNum(double n);         // #2
    BigNum(std::string const&); // #3
    ...
    operator double();        // #4
    operator long();          // #5
    ...
};

void initDemo()
{
    BigNum bn1(100103);       // Выбирается #1
    BigNum bn2("7057103224.095764"); // Выбирается #3
    int in = bn1;              // Выбирается #5
}
```

В этом примере разрешение перегрузки необходимо для выбора соответствующего конструктора или оператора преобразования. В частности, инициализация bn1 вызывает первый конструктор, инициализация bn2 — третий, а для in вызывается operator long(). В подавляющем большинстве случаев правила перегрузки дают интуитивно понятный результат. Тем не менее подробности этих правил довольно сложны, а некоторые приложения основаны на менее понятных принципах этой области языка C++.

Приложение Г

Стандартные утилиты

для работы с типами

Стандартная библиотека C++ в основном состоит из шаблонов, многие из которых опираются на различные методы, представленные и всесторонне рассмотренные в этой книге. По этой причине некоторые методы были “стандартизованы” в том смысле, что стандартная библиотека определяет несколько шаблонов для реализации библиотек с обобщенным кодом. Эти утилиты для работы с типами (свойства типов и другие вспомогательные шаблоны) перечислены и разъясняются здесь, в этом приложении.

Обратите внимание на то, что некоторые свойства типов требуют поддержки компилятора, в то время как другие могут быть реализованы в библиотеке только с помощью стандартных возможностей языка (некоторые из этих вопросов рассматриваются в главе 19, “Реализация свойств типов”).

Г.1. Использование свойств типов

В общем случае при использовании свойств типов необходимо включение заголовочного файла `<type_traits>`:

```
#include <type_traits>
```

Далее их применение зависит от того, дает свойство тип или значение.

- Если свойство дает **тип**, обратиться к нему можно следующим образом:

```
typename std::trait<...>::type  
std::trait_t<...> // Начиная с C++14
```
- Если свойство дает **значение**, обратиться к нему можно следующим образом:

```
std::trait<...>::value  
std::trait<...>() // Неявное преобразование в его тип  
std::trait_v<...> // Начиная с C++17
```

Например:

```
utils/traits1.cpp
```

```
#include <type_traits>  
#include <iostream>  
  
int main()  
{  
    int i = 42;  
    std::add_const<int>::type c = i; // c - int const  
    std::add_const_t<int> c14 = i; // Начиная с C++14  
    static_assert(std::is_const<decltype(c)>::value,  
                 "c должна быть const");
```

```

    std::cout << std::boolalpha;
    std::cout << std::is_same<decltype(c), int const>::value // true
                << '\n';
    std::cout << std::is_same_v<decltype(c), int const>
                << '\n';                                         // Начиная с C++17

    // Неявное преобразование в bool:
    if (std::is_same<decltype(c), int const> {})
    {
        std::cout << "Однаковы \n";
    }
}
}

```

В разделе 2.8 описан способ определения `_t`-версии свойств, а в разделе 5.6 – определения `_v`-версии свойств.

Г.1.1. `std::integral_constant` и `std::bool_constant`

Все стандартные типы, дающие **значение**, являются производными от экземпляра вспомогательного шаблона класса `std::integral_constant`:

```

namespace std
{
    template<typename T, T val>
    struct integral_constant
    {
        static constexpr T value = val; // Значение свойства
        using value_type = T;           // Тип значения
        using type = integral_constant<T, val>;
        constexpr operator value_type() const noexcept
        {
            return value;
        }
        // Начиная с C++14:
        constexpr value_type operator()() const noexcept
        {
            return value;
        }
    };
}

```

Из приведенного шаблона видно следующее.

- Можно использовать член `value_type` для запроса типа результата. Поскольку многие свойства, дающие значения, являются *предикатами*, `value_type` часто является простым `bool`.
- Объекты с типами свойств имеют неявное преобразование типа в тип значения, производимый свойством типа.
- В C++14 (и далее) объекты типа свойств являются также функциональными объектами (функционаторами), для которых “вызов функций” дает их значения.
- Член `type` просто дает базовый экземпляр `integral_constant`.

Если свойства дают булевые значения, то они также используют¹

```
namespace std
{
    template<bool B>
    using bool_constant = integral_constant<bool,B>; // Начиная с C++17
    using true_type = bool_constant<true>;
    using false_type = bool_constant<false>;
}
```

так что эти логические свойства наследуют `std::true_type`, если определенное свойство справедливо, и `std::false_type`, если нет. Это также означает, что соответствующие члены `value` равны `true` или `false`. Наличие различных типов для результирующих значений `true` и `false` позволяет нам прибегать к диспетчеризации дескрипторов, основанной на результатах свойств типа (см. разделы 19.3.3 и 20.2).

Например:

`utils/traits2.cpp`

```
#include <type_traits>
#include <iostream>
int main()
{
    using namespace std;
    cout << boolealpha;

    using MyType = int;
    cout << is_const<MyType>::value << '\n';           // Выводит false

    using VT = is_const<MyType>::value_type;             // bool
    // integral_constant<bool, false>;
    using T = is_const<MyType>::type;
    cout << is_same<VT, bool>::value << '\n';           // Выводит true
    cout << is_same<T, integral_constant<bool, false>>::value
        << '\n';                                         // Выводит true
    cout << is_same<T, bool_constant<false>>::value // Выводит true
        << '\n';                                         // (не работает до C++17)

    auto ic = is_const<MyType>(); // Объект типа свойства

    cout << is_same<decltype(ic),
                  is_const<int>::value << '\n'; // true
    cout << ic() << '\n';                // Вызов функции (выводит false)

    static constexpr auto mytypeIsConst = is_const<MyType> {};
    if constexpr(mytypeIsConst) // Проверка времени компиляции
```

¹ До C++17 стандарт не включал шаблон-псевдоним `bool_constant<>`. Однако `std::true_type` и `std::false_type` существовали еще в C++11 и C++14, и были определены через `integral_constant<bool, true>` и `integral_constant<bool, false>` соответственно.

```

{
    ...
}

static_assert(!std::is_const<MyType> {},           // начиная с C++17 => false
              "MyType не должен быть константой");
}

```

Наличие различных типов для небулевых специализаций `integral_constant` полезно также в ряде контекстов метапрограммирования. Обсуждение аналогичного типа `CTValue` в разделе 24.3 и его применение для доступа к элементу кортежа приведено в разделе 25.6.

Г.1.2. Что вы должны знать при использовании свойств

Говоря о применениях свойств, следует упомянуть о следующем.

- Свойства типов применяются непосредственно к типам, но `decltype` позволяет нам также проверять свойства выражений, переменных и функций. Напомним, однако, что `decltype` дает тип переменной или функции, только если сущность именована и не находится в лишних скобках; для любого прочего выражения эта конструкция дает тип, который отражает также категорию типа выражения. Например:

```

void foo(std::string&& s)
{
    // Проверка типа s:
    // false:
    std::is_lvalue_reference<decltype(s)>::value
    // true, согласно объявлению:
    std::is_rvalue_reference<decltype(s)>::value

    // Проверка категории значения s как выражения:
    // true, s используется как l-значение:
    std::is_lvalue_reference<decltype((s))>::value
    // false:
    std::is_rvalue_reference<decltype((s))>::value
}

```

Подробности представлены в разделе 15.10.2.

- Некоторые свойства для начинающего программиста могут демонстрировать неинтуитивное поведение (см. примеры в разделе 11.2).
- Некоторые свойства предъявляют определенные требования или предусловия. Нарушение этих предусловий ведет к неопределенному поведению². Некоторые примеры приведены в разделе 11.2.1.

² Комитет по стандартизации C++ рассмотрел предложение в стандарт C++17 о том, чтобы нарушения предусловий свойств типов всегда вели к ошибке времени компиляции. Однако, поскольку некоторые свойства типов в настоящее время предъявляют требования, которые сильнее, чем это строго необходимо (как, например, безусловное требование полных типов), это предложение было отложено.

- Многие свойства требуют полных типов (см. раздел 10.3.1). Чтобы иметь возможность использовать их для неполных типов, мы иногда можем ввести шаблоны для того, чтобы отложить их вычисление (см. подробности в разделе 11.5).
- Иногда логические операторы `&`, `|`, и `!` не могут использоваться для определения нового свойства типа, основанного на других свойствах типов. Кроме того, работа со свойствами типов, которые могут давать сбои, может стать проблемой или по крайней мере привести к определенным недостаткам. По этой причине предоставляются специальные свойства, которые позволяют нам логически объединять булевые свойства типов. Подробнее об этом рассказывается в разделе Г.6.
- Хотя стандартные шаблоны псевдонимов (заканчивающиеся `_t` или `_v`) часто удобны, они также имеют недостатки, делающие их непригодными для использования в некоторых контекстах метaproграммирования. Подробнее об этом говорится в разделе 19.7.3.

Г.2. Основные и составные категории типов

Начнем со стандартных свойств, которые проверяют основные и составные категории типов (см. рис. Г.1)³. В общем случае каждый тип принадлежит ровно к одной основной категории типа (белые элементы на рисунке). Составные категории типов объединяют основные категории типов в концепции более высокого уровня.

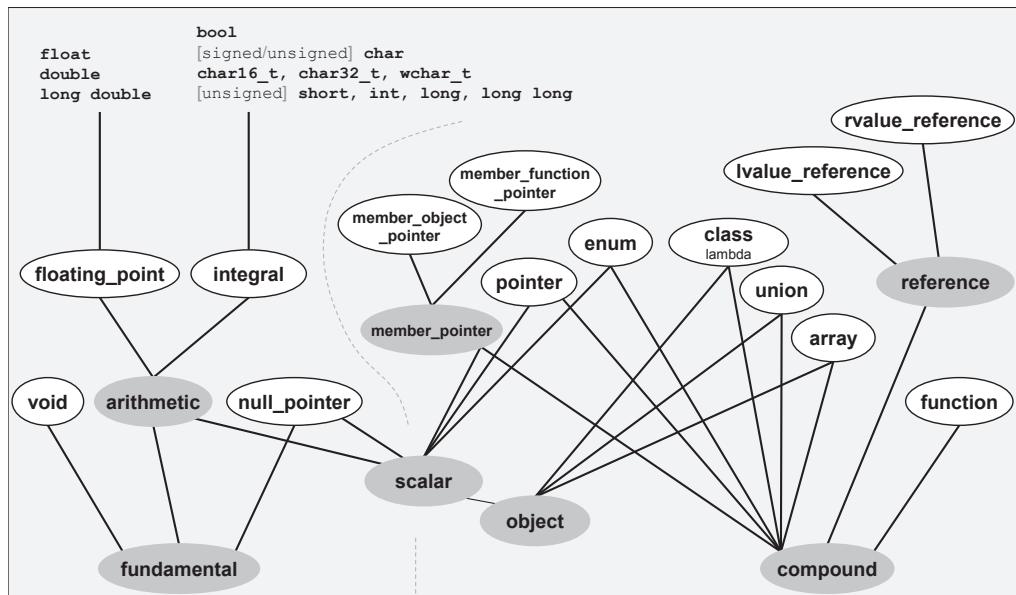


Рис. Г.1. Основные и составные категории типов

³Эта иерархия типов разработана Говардом Хиннантом (Howard Hinnant) в <http://howardhinnant.github.io/TypeHierarchy.pdf>.

Г.2.1. Проверка основных категорий типов

Этот раздел описывает утилиты, которые проверяют основную категорию данного типа. Для любого заданного типа ровно одна основная категория имеет статический член `value`, который имеет значение `true`⁴. Результат не зависит от того, квалифицирован тип с помощью `const` или `volatile` (*cv*-квалифицированный).

Обратите внимание на то, что для типов `std::size_t` и `std::ptrdiff_t` свойство `is_integral<T>` дает значение `true`. Для типа `std::max_align_t` то, какая из основных категорий типа выдаст `true`, зависит от реализации (т.е. это может быть целочисленный тип, тип с плавающей точкой или классовый тип). Язык определяет, что тип лямбда-выражения — это классовый тип (см. раздел 15.10.6), поэтому применение `is_class` к этому типу дает значение `true`.

Таблица Г.1. Свойства для проверки основных категорий типов

Свойство	Эффект
<code>is_void<T></code>	Тип <code>void</code>
<code>is_integral<T></code>	Целочисленный тип (включая <code>bool</code> , <code>char</code> , <code>char16_t</code> , <code>char32_t</code> , <code>wchar_t</code>)
<code>is_floating_point<T></code>	Тип с плавающей точкой (<code>float</code> , <code>double</code> , <code>long double</code>)
<code>is_array<T></code>	Обычный массив (не тип <code>std::array</code> !)
<code>is_pointer<T></code>	Тип указателя (включая указатели на функции, но не указатели на нестатические члены)
<code>is_null_pointer<T></code>	Тип <code>nullptr</code> (начиная с C++14)
<code>is_member_object_pointer<T></code>	Указатель на нестатический член-данное
<code>is_member_function_pointer<T></code>	Указатель на нестатическую функцию-член
<code>is_lvalue_reference<T></code>	Ссылка на l-значение
<code>is_rvalue_reference<T></code>	Ссылка на r-значение
<code>is_enum<T></code>	Тип перечисления
<code>is_class<T></code>	Тип класса/структурьи или лямбда-выражения, но не объединения
<code>is_union<T></code>	Тип объединения
<code>is_function<T></code>	Тип функции

`std::is_void<T>::value`

- Дает `true`, если тип `T` представляет собой (*cv*-квалифицированный) `void`.
- Например:

```
is_void_v<void>           // Дает true
is_void_v<void const>     // Дает true
```

⁴ До C++14 единственным исключением был тип `nullptr` — `std::nullptr_t`, для которого все утилиты основных категорий типов давали `false`, так как свойство `is_null_pointer<T>` не было частью стандарта C++11.

```
is_void_v<int>           // Дает false
void f();                  // Дает false (f имеет тип функции)
is_void_v<decltype(f())> // Дает true (возвращаемый тип f())
```

std::is_integral<T>::value

- Даёт true, если тип T представляет собой один из следующих (возможно, cv-квалифицированных) типов:
 - bool;
 - символьный тип (char, signed char, unsigned char, char16_t, char32_t или wchar_t);
 - целочисленный тип (знаковые или беззнаковые варианты short, int, long или long long; включает std::size_t и std::ptrdiff_t).

std::is_floating_point<T>::value

- Даёт true, если тип T представляет собой (cv-квалифицированный) float, double или long double.

std::is_array<T>::value

- Даёт true, если тип T представляет собой (cv-квалифицированный) тип массива.
- Помните, что *параметр*, объявленный как массив (с длиной или без таковой) по правилам языка в действительности имеет тип указателя.
- Обратите внимание на то, что класс std::array<> является типом класса, а не массива.
- Например:

```
is_array_v<int[]>           // Дает true
is_array_v<int[5]>           // Дает true
is_array_v<int*>             // Дает false
void foo(int a[], int b[5], int* c)
{
    is_array_v<decltype(a)> // Дает false (a имеет тип int*)
    is_array_v<decltype(b)> // Дает false (b имеет тип int*)
    is_array_v<decltype(c)> // Дает false (c имеет тип int*)
}
```

- Детали реализации рассмотрены в разделе 19.8.2.

std::is_pointer<T>::value

- Даёт true, если тип T представляет собой (cv-квалифицированный) указатель.

Сюда входят:

- указатели на статические/глобальные функции (и статические функции-члены);
- параметры, объявленные как массивы (с длиной или без таковой) или типы функций.

Сюда не входят:

- типы указателей на члены (например, тип `&X::m`, где `X` – классовый тип, а `m` – нестатическая функция-член или нестатический член-данное);
- тип `nullptr` – `std::nullptr_t`.

- Например:

```
is_pointer_v<int>           // Дает false
is_pointer_v<int*>          // Дает true
is_pointer_v<int* const>      // Дает true
is_pointer_v<int*&>          // Дает false
is_pointer_v<decltype(nullptr)> // Дает false
int* foo(int a[5], void(f)())
{
    is_pointer_v<decltype(a)>    // Дает true (тип a – int*)
    is_pointer_v<decltype(f)>    // Дает true (тип f – void(*)())
    is_pointer_v<decltype(foo)>  // Дает false
    is_pointer_v<decltype(&foo)> // Дает true
    // Дает true (возвращаемый тип – int*):
    is_pointer_v<decltype(foo(a,f))>
}
```

- Детали реализации приведены в разделе 19.8.2.

`std::is_null_pointer<T>::value`

- Даёт `true`, если тип `T` представляет собой (св-квалифицированный) `std::nullptr_t`, являющийся типом `nullptr`.

- Например:

```
is_null_pointer_v<decltype(nullptr)> // Дает true
void* p = nullptr;
// Даёт false (p не имеет тип std::nullptr_t):
is_null_pointer_v<decltype(p)>
```

- В стандарте, начиная с C++14.

`std::is_member_object_pointer<T>::value`

`std::is_member_function_pointer<T>::value`

- Даёт `true`, если тип `T` представляет собой (св-квалифицированный) тип указателя на член (например, `int X::*` или `int (X::*())()` для некоторого классового типа `X`).

`std::is_lvalue_reference<T>::value`

`std::is_rvalue_reference<T>::value`

- Даёт `true`, если тип `T` представляет собой (св-квалифицированный) тип ссылки на l-значение или ссылки на r-значение соответственно. Например:

```
is_lvalue_reference_v<int>    // Дает false
is_lvalue_reference_v<int&>   // Дает true
is_lvalue_reference_v<int&&>  // Дает false
is_lvalue_reference_v<void>    // Дает false
is_rvalue_reference_v<int>     // Дает false
is_rvalue_reference_v<int&>   // Дает false
is_rvalue_reference_v<int&&>  // Дает true
is_rvalue_reference_v<void>    // Дает false
```

- Детали реализации рассмотрены в разделе 19.8.2.

`std::is_enum<T>::value`

- Даёт `true`, если тип `T` представляет собой (св-квалифицированный) тип перечисления. Применим как к перечислениям с областью видимости, так и к перечислениям без области видимости.
- Детали реализации приведены в разделе 19.8.5.

`std::is_class<T>::value`

- Даёт `true`, если тип `T` представляет собой (св-квалифицированный) классовый тип, объявленный с использованием ключевого слова `class` или `struct`, включая тип, сгенерированный инстанцированием шаблона класса. Обратите внимание — язык гарантирует, что тип лямбда-выражения представляет собой классовый тип (см. раздел 15.10.6).
- Даёт `false` для объединений, типов перечислений с областью видимости (не смотря на объявление `enum class`), `std::nullptr_t` и любых прочих типов.
- Например:

```
is_class_v<int>           // Даёт false
is_class_v<std::string>    // Даёт true
is_class_v<std::string const> // Даёт true
is_class_v<std::string&>    // Даёт false
auto l1 = [] {};
// Даёт true (лямбда-выражение является объектом класса):
is_class_v<decltype(l1)>
```

- Детали реализации рассмотрены в разделе 19.8.4.

`std::is_union<T>::value`

- Даёт `true`, если тип `T` представляет собой (св-квалифицированный) тип объединения `union`, включая объединение, генерируемое из шаблона класса, который является шаблоном объединения.

`std::is_function<T>::value`

- Даёт `true`, если тип `T` представляет собой (св-квалифицированный) тип функции. Даёт `false` для типа указателя на функцию, типа лямбда-выражения или любого другого типа.
- Помните, что *параметр*, объявленный как тип функции, по правилам языка в действительности имеет тип указателя.
- Например:

```
void foo(void(f)())
{
    // Даёт false (f имеет тип void(*)()):
    is_function_v<decltype(f)>
    is_function_v<decltype(foo)>    // Даёт true
    is_function_v<decltype(&foo)>    // Даёт false
    // Даёт false (для возвращаемого типа):
    is_function_v<decltype(foo(f))>
}
```

- Детали реализации приведены в разделе 19.8.3.

Г.2.2. Проверка составных категорий типов

Рассматриваемые далее утилиты позволяют определить, принадлежит ли тип к более широкой категории типов, которая является объединением некоторых основных категорий типов. Составные категории типов не образуют строгого деления: тип может принадлежать к нескольким составным категориям одновременно (например, тип указателя является одновременно и скалярным, и составным). И вновь cv-квалификация (`const` и `volatile`) при классификации типа не учитывается.

Таблица Г.2. Свойства для проверки составных категорий типов

Свойство	Эффект
<code>is_reference<T></code>	Ссылка на l-значение или ссылка на r-значение
<code>is_member_pointer<T></code>	Указатель на нестатический член
<code>is_arithmetic<T></code>	Целочисленный тип (включая <code>bool</code> и символьные типы) или тип с плавающей точкой
<code>is_fundamental<T></code>	<code>void</code> , целочисленный тип (включая <code>bool</code> и символьные типы), тип с плавающей точкой или <code>std::nullptr_t</code>
<code>is_scalar<T></code>	Целочисленный тип (включая <code>bool</code> и символьные типы), тип с плавающей точкой, перечисление, указатель, указатель на член или <code>std::nullptr_t</code>
<code>is_object<T></code>	Любой тип, за исключением <code>void</code> , функции или ссылки
<code>is_compound<T></code>	Противоположность <code>is_fundamental<T></code> : массив, перечисление, объединение, класс, функция, ссылка, указатель или указатель на член

`std::is_reference<T>::value`

- Дает `true`, если тип `T` представляет собой ссылочный тип.
- То же, что и `is_lvalue_reference_v<T> || is_rvalue_reference_v<T>`.
- Детали реализации приведены в разделе 19.8.2.

`std::is_member_pointer<T>::value`

- Дает `true`, если тип `T` представляет собой любой тип указателя на член.
- То же, что и `!(is_member_object_pointer_v<T> || is_member_function_pointer_v<T>)`.

`std::is_arithmetic<T>::value`

- Дает `true`, если тип `T` представляет собой арифметический тип (`bool`, символьный тип, целочисленный тип или тип с плавающей точкой).
- То же, что и `is_integral_v<T> || is_floating_point_v<T>`.

`std::is_fundamental<T>::value`

- Дает `true`, если тип `T` представляет собой фундаментальный тип (арифметический тип, `void` или `std::nullptr_t`).
- То же, что и `is_arithmetic_v<T> || is_void_v<T> || is_null_pointer_v<T>`.
- То же, что и `!is_compound_v<T>`.
- Подробности реализации рассмотрены в разделе 19.8.1 при разработке шаблона `IsFundamental`.

`std::is_scalar<T>::value`

- Дает `true`, если тип `T` представляет собой “скалярный” тип.
- То же, что и `is_arithmetic_v<T> || is_enum_v<T> || is_pointer_v<T> || is_member_pointer_v<T> || is_null_pointer_v<T>`.

`std::is_object<T>::value`

- Дает `true`, если тип `T` описывает тип объекта.
- То же, что и `is_scalar_v<T> || is_array_v<T> || is_class_v<T> || is_union_v<T>`.
- То же, что и `!(is_function_v<T> || is_reference_v<T> || is_void_v<T>)`.

`std::is_compound<T>::value`

- Дает `true`, если тип `T` представляет собой тип, составленный из других типов.
- То же, что и `!is_fundamental_v<T>`.
- То же, что и `is_enum_v<T> || is_array_v<T> || is_class_v<T> || is_union_v<T> || is_reference_v<T> || is_pointer_v<T> || is_member_pointer_v<T> || is_function_v<T>`.

Г.3. Характеристики и операции над типами

Следующая группа свойств проверяет другие характеристики отдельных типов, а также наличие определенных операций (например, обмен значений), которые могут выполняться с ними.

Г.3.1. Прочие характеристики типов

Таблица Г.3. Свойства, проверяющие простые характеристики типов

Свойство	Эффект
<code>is_signed<T></code>	Знаковый арифметический тип
<code>is_unsigned<T></code>	Беззнаковый арифметический тип
<code>is_const<T></code>	<code>const</code> -квалифицированный тип
<code>is_volatile<T></code>	<code>volatile</code> -квалифицированный тип

Окончание табл. Г.3

Свойство	Эффект
<code>is_aggregate<T></code>	Агрегат (начиная с C++17)
<code>is_trivial<T></code>	Скаляр, тривиальный класс или массивы этих типов
<code>is_trivially_copyable<T></code>	Скаляр, тривиально копируемый класс или массивы этих типов
<code>is_standard_layout<T></code>	Скаляр, класс со стандартной схемой размещения или массивы этих типов
<code>is_pod<T></code>	Простой старый тип данных (в котором можно копировать объекты с помощью <code>memcpy()</code>)
<code>is_literal_type<T></code>	Скаляр, ссылка, класс или массивы этих типов (не рекомендовано, начиная с C++17)
<code>is_empty<T></code>	Класс без членов, виртуальных функций-членов или виртуальных базовых классов
<code>is_polymorphic<T></code>	Класс с (производной) виртуальной функцией-членом
<code>is_abstract<T></code>	Абстрактный класс (как минимум одна чисто виртуальная функция)
<code>is_final<T></code>	Финальный класс (класс, который нельзя наследовать; начиная с C++14)
<code>has_virtual_destructor<T></code>	Класс с виртуальным деструктором
<code>has_unique_object_representations<T></code>	Любые два объекта с одним и тем же значением имеют одинаковое представление в памяти (начиная с C++17)
<code>alignment_of<T></code>	Эквивалентно <code>alignof(T)</code>
<code>rank<T></code>	Количество размерностей типа массива (или 0)
<code>extent<T, I=0></code>	Длина по размерности I (или 0)
<code>underlying_type<T></code>	Базовый тип для типа перечисления
<code>is_invocable<T, Args...></code>	Может использоваться как вызываемый объект для Args... (начиная с C++17)
<code>is_nothrow_invocable<T, Args...></code>	Может использоваться как вызываемый объект для Args... с гарантией отсутствия исключений (начиная с C++17)
<code>is_invocable_r<RT, T, Args...></code>	Может использоваться как вызываемый объект для Args..., с возвращаемым типом RT (начиная с C++17)
<code>is_nothrow_invocable_r<RT, T, Args...></code>	Может использоваться как вызываемый объект для Args..., с возвращаемым типом RT с гарантией отсутствия исключений (начиная с C++17)
<code>invoke_result<T, Args...></code>	Возвращаемый тип при использовании в качестве вызываемого объекта для Args... (начиная с C++17)
<code>result_of<F, ArgTypes></code>	Возвращаемый тип при вызове F с типами аргументов ArgTypes (не рекомендовано, начиная с C++17)

std::is_signed<T>::value

- Даёт `true`, если `T` представляет собой знаковый арифметический тип (т.е. арифметический тип, который включает представление отрицательных значений; сюда входят такие типы, как (`signed`) `int`, `float`).
- Для типа `bool` даёт `false`.
- Для типа `char` возвращаемое значение — `true` или `false` — зависит от реализации.
- Для всех неарифметических типов (включая типы перечислений) `is_signed` даёт `false`.

std::is_unsigned<T>::value

- Даёт `true`, если `T` представляет собой беззнаковый арифметический тип (т.е. арифметический тип, который не включает представление отрицательных значений; сюда входят такие типы, как `unsigned int` и `bool`).
- Для типа `char` возвращаемое значение — `true` или `false` — зависит от реализации.
- Для всех неарифметических типов (включая типы перечислений) `is_unsigned` даёт `false`.

std::is_const<T>::value

- Даёт `true`, если тип квалифицирован как `const`.
- Обратите внимание на то, что константный указатель является `const`-квалифицированным типом, в то время как неконстантный указатель (или ссылка на константный тип) не является `const`-квалифицированным типом. Например:

```
is_const<int* const>::value // true
is_const<int const*>::value // false
is_const<int const&>::value // false
```

- Язык определяет массивы как `const`-квалифицированные, если тип элемента является `const`-квалифицированным⁵. Например:

```
is_const<int[3]>::value      // false
is_const<int const[3]>::value // true
is_const<int[]>::value       // false
is_const<int const[]>::value // true
```

std::is_volatile<T>::value

- Даёт `true`, если тип `volatile`-квалифицирован.
- Обратите внимание на то, что `volatile`-указатель имеет `volatile`-квалифицированный тип, в то время как указатель или ссылка на `volatile` тип, но сами не являющиеся `volatile`, не являются `volatile`-квалифицированными. Например:

⁵ Это было разъяснено в резолюции по базовому вопросу 1059 уже после опубликования C++11.

```
is_volatile<int* volatile>::value // true
is_volatile<int volatile*>::value // false
is_volatile<int volatile&>::value // false
```

- Язык определяет массив как volatile-квалифицированный, если тип элементов является volatile-квалифицированным⁶. Например:

```
is_volatile<int[3]>::value // false
is_volatile<int volatile[3]>::value // true
is_volatile<int[]>::value // false
is_volatile<int volatile[]>::value // true
```

std::is_aggregate<T>::value

- Дает `true`, если `T` является *агрегатным* типом (массив либо класс/структура/объединение, у которого нет пользовательского, явного или унаследованного конструктора, нет закрытых или защищенных нестатических членов-данных, нет виртуальных функций, а также нет виртуальных, закрытых или защищенных базовых классов)⁷.
- Помогает выяснить, требуется ли список инициализации. Например:

```
template<typename Coll, typename... T>
void insert(Coll& coll, T& ... val)
{
    if constexpr(!std::is_aggregate_v<typename Coll::value_type>)
    {
        // Для агрегатов не годится:
        coll.emplace_back(std::forward<T>(val)...);
    }
    else
    {
        coll.emplace_back(typename
            Coll::value_type{std::forward<T>(val)...});
    }
}
```

- Требует, чтобы данный тип либо был полным (см. раздел 10.3.1) либо (св-квалифицированным) `void`.
- Доступно, начиная с C++17.

std::is_trivial<T>::value

- Дает `true`, если тип является “тривиальным” типом:
 - скалярный тип (целочисленный, с плавающей точкой, перечисление, указатель; см. выше свойство `is_scalar()`);
 - тип тривиального класса (класс, не имеющий виртуальных функций, виртуальных базовых классов, (косвенных) пользовательских кон-

⁶Это было разъяснено в резолюции по базовому вопросу 1059 уже после опубликования C++11.

⁷ Обратите внимание на то, что члены базовых классов и/или члены-данные агрегатов не обязаны быть агрегатами. До C++14 типы классов агрегатов не могли иметь инициализаторов членов по умолчанию. До C++17 агрегаты не могли иметь открытых базовых классов.

структур по умолчанию, копирующих/перемещающих конструкторов, копирующих/перемещающих операторов присваивания или деструкторов, инициализаторов для нестатических членов-данных, членов, объявленных как `volatile`, а также нетривиальных членов);

массив таких типов;

- а также cv-квалифицированные версии этих типов.

- Дает `true`, если `is_trivially_copyable_v<T>` дает `true`, и существует тривиальный конструктор по умолчанию.
- Требует, чтобы данный тип либо был полным (см. раздел 10.3.1), либо (cv-квалифицированным) `void`.

`std::is_trivially_copyable<T>::value`

- Дает `true`, если тип является “тривиально копируемым” типом:
 - скалярный тип (целочисленный, с плавающей точкой, перечисление, указатель; см. выше свойство `is_scalar()`);
 - тип тривиального класса (класс, не имеющий виртуальных функций, виртуальных базовых классов, (косвенных) пользовательских конструкторов по умолчанию, копирующих/перемещающих конструкторов, копирующих/перемещающих операторов присваивания или деструкторов, инициализаторов для нестатических членов-данных, членов, объявленных как `volatile`, а также нетривиальных членов);
 - массив таких типов;
 - а также cv-квалифицированные версии этих типов.
- Дает то же значение, что и `is_trivial_v<T>`, за исключением того, что может дать `true` для типа класса без тривиального конструктора по умолчанию.
- В отличие от `is_standard_layout<>`, `volatile`-члены не допускаются, ссылки разрешены, члены могут иметь различный доступ, а также члены могут быть распределены среди различных (базовых) классов.
- Требует, чтобы данный тип либо был полным (см. раздел 10.3.1), либо (cv-квалифицированным) `void`.

`std::is_standard_layout<T>::value`

- Дает `true`, если тип имеет стандартную схему размещения, что, например, упрощает обмен значений данного типа с другими языками:
 - скалярный тип (целочисленный, с плавающей точкой, перечисление, указатель; см. выше свойство `is_scalar()`);
 - тип со *стандартной схемой размещения* (класс, не имеющий виртуальных функций, виртуальных базовых классов, нестатических членов-ссылок, все нестатические члены находятся в одном и том же (базовом) классе, определенном с тем же доступом, все члены также являются типами со стандартной схемой размещения);
 - массив таких типов;
 - а также cv-квалифицированные версии этих типов.

- В отличие от `is_trivial<>`, `volatile`-члены разрешены, ссылки не разрешены, члены не могут иметь различный доступ, а также члены не могут быть распределены среди различных (базовых) классов.
- Требует, чтобы данный тип (для массивов – базовый тип) либо был полным (см. раздел 10.3.1), либо (св-квалифицированным) `void`.

`std::is_pod<T>::value`

- Дает `true`, если `T` представляет собой простой старый тип данных (*plain old datatype* – POD).
- Объекты таких типов могут копироваться путем копирования их памяти (например с помощью `memcpy()`).
- То же, что и `is_trivial_t<T> && is_standard_layout_v<T>`.
- Дает `false` для:
 - классов, не имеющих тривиального конструктора по умолчанию, копирующего/перемещающего конструктора, копирующего/перемещающего присваивания или деструктора;
 - классов, имеющих виртуальные члены или виртуальные базовые классы;
 - классов, имеющих `volatile`-члены или члены-ссылки;
 - классов, которые имеют члены в различных (базовых) классах или с различным доступом;
 - типов лямбда-выражений (именуемых *типами замыканий*);
 - функций;
 - `void`;
 - типов, составленных из перечисленных выше типов.
- Требует, чтобы данный тип либо был полным (см. раздел 10.3.1), либо (св-квалифицированным) `void`.

`std::is_literal_type<T>::value`

- Дает `true`, если данный тип является корректным возвращаемым типом для `constexpr`-функции (что в особенности исключает любой тип, требующий нетривиального уничтожения).
- Дает `true`, если `T` является *литеральным типом*:
 - скалярный тип (целочисленный, с плавающей точкой, перечисление, указатель; см. выше свойство `is_scalar()`);
 - ссылка;
 - тип класса с по меньшей мере одним `constexpr`-конструктором, который не является копирующим/перемещающим конструктором, в каждом базовом классе, не имеющем пользовательского или виртуального деструктора в любом (базовом) классе или члене, и в котором каждая инициализация нестатического члена-данного является константным выражением;
 - массив таких типов.

- Требует, чтобы данный тип либо был полным (см. раздел 10.3.1), либо (свалифицированным) `void`.
- Обратите внимание на то, что, начиная со стандарта C++17, это свойство не рекомендуется к употреблению, поскольку “оно слишком слабое, чтобы содержательно использовать его в обобщенном коде. Что действительно необходимо — это возможность знать, что конкретное конструирование будет приводить к константной инициализации”.

`std::is_empty<T>::value`

- Дает `true`, если тип `T` является типом класса (но не объединения), объекты которого не содержат данных.
- Дает `true`, если `T` определен как `class` или `struct`, у которого
 - нет нестатических членов-данных, отличных от битовых полей нулевой длины;
 - нет виртуальных функций-членов;
 - нет виртуальных базовых классов;
 - нет непустых базовых классов.
- Требует, чтобы данный тип либо был полным (см. раздел 10.3.1), если это `class/struct` (неполный `union` разрешен).

`std::is_polymorphic<T>::value`

- Дает `true`, если `T` представляет собой *полиморфный* тип класса (класс, объявляющий или наследующий виртуальную функцию).
- Требует, чтобы данный тип либо был полным (см. раздел 10.3.1), либо не был ни `class`, ни `struct`.

`std::is_abstract<T>::value`

- Даёт `true`, если `T` представляет собой *абстрактный* тип класса (класс, для которого не может быть создан ни один объект, так как он имеет по крайней мере одну чисто виртуальную функцию-член).
- Требует, чтобы данный тип либо был полным (см. раздел 10.3.1), если это `class/struct` (неполный `union` разрешен).

`std::is_final<T>::value`

- Дает `true`, если `T` — *окончательный* (`final`) тип класса (класс или объединение, который не может быть использован в качестве базового, так как объявлен как `final`).
- Для всех типов, не являющихся классами/объединениями, такими как `int`, возвращает `false` (т.е. это не то же, как нечто наподобие *может быть наследован*).
- Требует, чтобы данный тип либо был полным (см. раздел 10.3.1), либо не был `class/struct` или `union`.
- Доступно, начиная со стандарта C++14.

std::has_virtual_destructor<T>::value

- Дает `true`, если тип `T` имеет виртуальный деструктор.
- Требует, чтобы данный тип либо был полным (см. раздел 10.3.1), если это `class/struct` (неполный `union` разрешен).

std::has_unique_object_representations<T>::value

- Дает `true`, если любые два объекта типа `T` имеют одинаковое представление объектов в памяти. Следовательно, два идентичных значения всегда представлены с использованием одинаковых последовательностей значений байтов.
- Объекты с таким свойством могут производить надежное хеш-значение путем хеширования связанной с ними последовательности байтов (отсутствует риск, что некоторые биты, не участвующие в значении объекта, могут отличаться от одного объекта к другому).
- Требует, чтобы данный тип был тривиально копируемым (см. раздел Г.3.1) и либо полным (см. раздел 10.3.1), либо (св-квалифицированным) `void` или массивом с неизвестными границами.
- Доступно, начиная со стандарта C++17.

std::alignment_of<T>::value

- Дает выравнивание объекта типа `T` как значение типа `std::size_t` (для массивов — типа элемента; для ссылок — типа, на который выполняется ссылка).
- То же, что и `alignof(T)`.
- Это свойство было введено в C++11 до конструкции `alignof(...)`. Однако оно все еще востребовано, поскольку свойство может быть передано как тип класса, что может быть полезным для некоторых метапрограмм.
- Требует, чтобы `alignof(T)` было корректным выражением.
- Используйте `aligned_union<>` для получения общего выравнивания нескольких типов (см. раздел Г.5).

std::rank<T>::value

- Дает количество размерностей массива типа `T` в виде значения типа `std::size_t`.
- Дает 0 для всех прочих типов.
- Указатели не имеют связанных с ними размерностей. Неопределенная граница в типе массива указывает размерность. (Как обычно, параметр функции, объявленный как тип массива, в действительности не имеет типа массива, так же как и тип `std::array` не является типом массива — см. раздел Г.2.1.) Например:

```
int a2[5][7];
rank_v<decltype(a2)>; // Дает 2
rank_v<int*>; // Дает 0 (no array)
extern int p1[];
```

```
rank_v<decltype(p1)>; // Дает 1
```

```
std::extent<T>::value
```

```
std::extent<T, IDX>::value
```

- Дает размер первой или IDX-й размерности массива типа T в виде значения типа std::size_t.
- Дает 0, если T не является массивом, размерность не существует, или размер данной размерности неизвестен.
- Подробности реализации описаны в разделе 19.8.2.

```
int a2[5][7];
extent_v<decltype(a2)>; // Дает 5
extent_v<decltype(a2), 0>; // Дает 5
extent_v<decltype(a2), 1>; // Дает 7
extent_v<decltype(a2), 2>; // Дает 0
extent_v<int*>; // Дает 0
extern int p1[];
extent_v<decltype(p1)>; // Дает 0
```

```
std::underlying_type<T>::type
```

- Дает базовый тип для типа перечисления T.
- Требует, чтобы данный тип был полным (см. раздел 10.3.1) типом перечисления. Для всех прочих типов имеет неопределенное поведение.

```
std::is_invocable<T, Args...>::value
```

```
std::is_nothrow_invocable<T, Args...>::value
```

- Дает true, если T можно использовать для вызова с аргументами Args... (с гарантией отсутствия исключений).
- Таким образом, мы можем использовать эти свойства, чтобы проверить, можем ли мы использовать вызов или std::invoke() для данного *вызываемого* типа T для аргументов Args.... (Подробная информация о *вызываемых* объектах и std::invoke() приведена в разделе 11.1.)
- Требует, чтобы все указанные типы были полными (см. раздел 10.3.1) или (cv-квалифицированным) void, или массивом с неизвестными границами.
- Например:

```
struct C
{
    bool operator()(int) const
    {
        return true;
    }
};
std::is_invocable<C>::value // false
std::is_invocable<C, int>::value // true
std::is_invocable<int*>::value // false
std::is_invocable<int(*)()>::value // true
```

- Доступно, начиная со стандарта C++17⁸.

⁸ В процессе стандартизации C++17 is_invocable был переименован из is_callable.

```
std::is_invocable_r<RET_T, T, Args...>::value
std::is_nothrow_invocable_r<RET_T, T, Args...>::value
```

- Дает true, если T можно использовать для вызова с аргументами Args... (с гарантией отсутствия исключений) и возвращаемым значением, преобразуемым к типу RET_T.
- Таким образом, эти свойства позволяют проверить, можем ли мы использовать вызов или std::invoke() для данного *вызываемого* типа T для аргументов Args... и применить возвращаемое значение как RET_T. (Подробная информация о *вызываемых* объектах и std::invoke() приведена в разделе 11.1.)
- Требует, чтобы все указанные типы были полными (см. раздел 10.3.1) или (св-квалифицированным) void, или массивом с неизвестными границами.
- Например:

```
struct C
{
    bool operator()(int) const
    {
        return true;
    }
};

std::is_invocable_r<bool, C, int>::value           // true
std::is_invocable_r<int, C, long>::value           // true
std::is_invocable_r<void, C, int>::value           // true
std::is_invocable_r<char*, C, int>::value          // false
std::is_invocable_r<long, int(*)(int)>::value      // false
std::is_invocable_r<long, int(*)(int), int>::value // true
std::is_invocable_r<long, int(*)(int), double>::value // true
```

- Доступно, начиная со стандарта C++17.

```
std::invoke_result<T,Args...>::value
std::result_of<T,Args...>::value
```

- Дает возвращаемый тип *вызываемого* типа T, вызванного с аргументами Args....
- Обратите внимание на небольшие отличия в синтаксисе:
 - В invoke_result<> вы должны передать в качестве параметров как вызываемый тип, так и типы аргументов.
 - В result_of<> вы должны передать “объявление функции” с использованием соответствующих типов.
- Если вызов невозможен, член type не определен, так что его использование является ошибкой (это позволяет использовать принцип SFINAE для отбрасывания шаблона функции с помощью использования свойства в ее объявлении; см. раздел 8.4).
- Таким образом, мы можем использовать эти свойства для получения возвращаемого типа при использовании вызова или std::invoke() для дан-

ногого вызываемого типа T для аргументов Args.... (Подробная информация о вызываемых объектах и std::invoke() приведена в разделе 11.1.)

- Требует, чтобы все указанные типы были полными (см. раздел 10.3.1) или (cv-квалифицированным) void, или массивом с неизвестными границами.
- Свойство invoke_result<> доступно, начиная с C++17, и заменяет свойство result_of<>, которое не рекомендовано к употреблению, начиная с C++17, поскольку invoke_result<> предоставляет определенные улучшения, такие как более простой синтаксис и разрешение T быть абстрактным типом.
- Например:

```
std::string foo(int);
using R0 = typename
    std::result_of<decltype(&foo)(int)>::type; // C++11
using R1 = std::result_of_t<decltype(&foo)(int)>; // C++14
using R2 = std::invoke_result_t<decltype(foo), int>; // C++17
struct ABC
{
    virtual ~ABC() = 0;
    void operator()(int) const
    {
    }
};
using T1 = typename
    std::result_of<ABC(int)>::type; // Ошибка: ABC - абстрактный
using T2 = typename
    std::invoke_result<ABC,int>::type; // OK, начиная с C++17
```

Полный пример можно найти в разделе 11.1.3.

Таблица Г.4. Свойства для проверки определенных операций

Свойства	Эффект
is_constructible<T,Args...>	Можно инициализировать тип T типами Args
is_trivially_constructible<T,Args...>	Можно тривиально инициализировать тип T типами Args
is_nothrow_constructible<T,Args...>	Можно инициализировать тип T типами Args, и эта операция не генерирует исключений
is_default_constructible<T>	Можно инициализировать тип T без аргументов
is_trivially_default_constructible<T>	Можно тривиально инициализировать тип T без аргументов
is_nothrow_default_constructible<T>	Можно инициализировать тип T без аргументов, и эта операция не генерирует исключений
is_copy_constructible<T>	Тип T можно копировать
is_trivially_copy_constructible<T>	Тип T можно тривиально копировать

Окончание табл. Г.4

Свойства	Эффект
<code>is_nothrow_copy_constructible<T></code>	Тип T можно копировать, и эта операция не генерирует исключений
<code>is_move_constructible<T></code>	Тип T можно перемещать
<code>is_trivially_move_constructible<T></code>	Тип T можно тривиально перемещать
<code>is_nothrow_move_constructible<T></code>	Тип T можно перемещать, и эта операция не генерирует исключений
<code>is_assignable<T, T2></code>	Можно присваивать тип T2 типу T
<code>is_trivially_assignable<T, T2></code>	Можно тривиально присваивать тип T2 типу T
<code>is_nothrow_assignable<T, T2></code>	Можно присваивать тип T2 типу T, и эта операция не генерирует исключений
<code>is_copy_assignable<T></code>	Можно выполнять копирующее присваивание T
<code>is_trivially_copy_assignable<T></code>	Можно выполнять тривиальное копирующее присваивание T
<code>is_nothrow_copy_assignable<T></code>	Можно выполнять копирующее присваивание T, и эта операция не генерирует исключений
<code>is_move_assignable<T></code>	Можно выполнять перемещающее присваивание T
<code>is_trivially_move_assignable<T></code>	Можно выполнять тривиальное перемещающее присваивание T
<code>is_nothrow_move_assignable<T></code>	Можно выполнять перемещающее присваивание T, и эта операция не генерирует исключений
<code>is_destructible<T></code>	Можно уничтожать T
<code>is_trivially_destructible<T></code>	Можно тривиально уничтожать T
<code>is_nothrow_destructible<T></code>	Можно уничтожать T, и эта операция не генерирует исключений
<code>is_swappable<T></code>	Для этого типа можно вызывать <code>swap()</code> (начиная с C++17)
<code>is_nothrow_swappable<T></code>	Для этого типа можно вызывать <code>swap()</code> , и эта операция не генерирует исключений (начиная с C++17)
<code>is_swappable_with<T, T2></code>	Для этих двух типов с указанной категорией значений можно вызывать <code>swap()</code> (начиная с C++17)
<code>is_nothrow_swappable_with<T, T2></code>	Для этих двух типов с указанной категорией значений можно вызывать <code>swap()</code> , и эта операция не генерирует исключений (начиная с C++17)

В таблице Г.4 перечислены свойства типов, которые позволяют нам проверять некоторые определенные операции. Формы с `is_trivially_...` дополнительно проверяют, все ли (суб)операции, вызываемые для объекта, членов или базовых классов, являются тривиальными (не определяемыми пользователем и не виртуальными). Формы с `is_nothrow_...` дополнительно проверяют, гарантируется ли отсутствие генерации исключений данной операцией. Обратите внимание на то, что все проверки `is_..._constructible` неявно выполняют соответствующие проверки `is_..._destructible`. Например:

`utils/isconstructible.cpp`

```
#include <iostream>

class C
{
public:
    C()          // Конструктор по умолчанию не имеет noexcept
    {
    }
    virtual ~C() = default; // Делает тип С нетривиальным
};

int main()
{
    using namespace std;
    cout << is_default_constructible_v<C> << '\n';           // true
    cout << is_trivially_default_constructible_v<C> << '\n'; // false
    cout << is_nothrow_default_constructible_v<C> << '\n';   // false
    cout << is_copy_constructible_v<C> << '\n';           // true
    cout << is_trivially_copy_constructible_v<C> << '\n'; // true
    cout << is_nothrow_copy_constructible_v<C> << '\n'; // true
    cout << is_destructible_v<C> << '\n';           // true
    cout << is_trivially_destructible_v<C> << '\n'; // false
    cout << is_nothrow_destructible_v<C> << '\n'; // true
}
```

Из-за определения деструктора как виртуального все операции больше не являются тривиальными. А поскольку мы определяем конструктор по умолчанию без `noexcept`, он может генерировать исключения. Все другие операции по умолчанию гарантируют отсутствие исключений.

`std::is_constructible<T,Args...>::value`
`std::is_trivially_constructible<T,Args...>::value`
`std::is_nothrow_constructible<T,Args...>::value`

- Даёт `true`, если объект типа `T` может быть инициализирован аргументами типов, задаваемых `Args...` (без использования нетривиальной операции или с гарантией отсутствия генерации исключений). Таким образом, следующий код должен быть корректен⁹:

`T t(std::declval<Args>()...);`

⁹ См. информацию о `std::declval` в разделе 11.2.3.

- Значение `true` подразумевает, что объект может быть соответствующим образом уничтожен (т.е. `is_destructible_v<T>`, `is_trivially_destructible_v<T>` или соответственно `is_nothrow_destructible_v<T>` дадут `true`).
- Требует, чтобы все данные типы были либо полными (см. раздел 10.3.1), (свалифицированным) `void`, либо массивами с неизвестными границами.
- Например:

```

is_constructible_v<int>                                // true
is_constructible_v<int, int>                            // true
is_constructible_v<long, int>                           // true
is_constructible_v<int, void*>                          // false
is_constructible_v<void*, int>                          // false
is_constructible_v<char const*, std::string>           // false
is_constructible_v<std::string, char const*>           // true
is_constructible_v<std::string, char const*, int, int> // true

```

- Обратите внимание на то, что `is_convertible` имеет иной порядок исходного и целевого типов.

```

std::is_default_constructible<T>::value
std::is_trivially_default_constructible<T>::value
std::is_nothrow_default_constructible<T>::value

```

- Дает `true`, если объект типа `T` может быть инициализирован без аргументов инициализации (без использования нетривиальной операции или с гарантией отсутствия генерации исключений).
- То же, что и `is_constructible_v<T>`, `is_trivially_constructible_v<T>` или `is_nothrow_constructible_v<T>` соответственно.
- Значение `true` подразумевает, что объект может быть соответствующим образом уничтожен (т.е. `is_destructible_v<T>`, `is_trivially_destructible_v<T>` или соответственно `is_nothrow_destructible_v<T>` дает `true`).
- Требует, чтобы данный тип был либо полным (см. раздел 10.3.1), (свалифицированным) `void` или массивом с неизвестными границами.

```

std::is_copy_constructible<T>::value
std::is_trivially_copy_constructible<T>::value
std::is_nothrow_copy_constructible<T>::value

```

- Дает `true`, если объект типа `T` может быть создан путем копирования другого значения типа `T` (без использования нетривиальной операции или с гарантией отсутствия генерации исключений).
- Дает `false`, если `T` не является типом, на который можно ссылаться (referenceable type) ((свалифицированный) `void` или тип функции, свалифицированный с помощью `const`, `volatile`, `&`, `&&`).
- Указывает, является ли тип `T` типом, на который можно ссылаться, так же как `is_constructible<T, Tconst&>::value`, `is_trivially_constructible`

`<T, T const&>::value` или `is_nothrow_constructible<T, T const&>::value` соответственно.

- Чтобы выяснить, может ли объект типа `T` быть сконструирован копированием `г-значения` типа `T`, используйте `is_constructible<T, T&&>` (и т.д.).
- Значение `true` подразумевает, что объект может быть соответствующим образом уничтожен (т.е. `is_destructible_v<T>`, `is_trivially_destructible_v<T>` или соответственно `is_nothrow_destructible_v<T>` дает `true`).
- Требует, чтобы данный тип был полным (см. раздел 10.3.1), (св-квалифицированным) `void` либо массивом с неизвестной границей.
- Например:

```
is_copy_constructible_v<int>                                // true
is_copy_constructible_v<void>                               // false
is_copy_constructible_v<std::unique_ptr<int>>             // false
is_copy_constructible_v<std::string>                         // true
is_copy_constructible_v<std::string&>                       // true
is_copy_constructible_v < std::string&& >                  // false

// В отличие от:
is_constructible_v<std::string, std::string>               // true
is_constructible_v<std::string&, std::string&>              // true
is_constructible_v < std::string&&, std::string&& > // true

std::is_move_constructible<T>::value
std::is_trivially_move_constructible<T>::value
std::is_nothrow_move_constructible<T>::value
```

- Дает `true`, если объект типа `T` может быть создан из `г-значения` типа `T` (без использования нетривиальной операции или с гарантией отсутствия генерации исключений).
- Дает `false`, если `T` не является типом, на который можно ссылаться (`referenceable type`) ((св-квалифицированный) `void` или тип функции, квалифицированный с помощью `const`, `volatile`, `&`, `&&`).
- Указывает, является ли тип `T` типом, на который можно ссылаться, также как и `is_constructible<T, T&&>::value`, `is_trivially_constructible<T, &&>::value` или `is_nothrow_constructible<T, &&>::value` соответственно.
- Значение `true` подразумевает, что объект может быть соответствующим образом уничтожен (т.е. `is_destructible_v<T>`, `is_trivially_destructible_v<T>` или соответственно `is_nothrow_destructible_v<T>` дает `true`).
- Заметьте, что нет никакого способа проверить, не генерируется ли исключение перемещающим конструктором без возможности вызова его непосредственно для объекта типа `T`. Для конструктора недостаточно быть открытим и не удаленным; он также требует, чтобы соответствующий тип не

являлся абстрактным классом (ссылки или указатели на абстрактные классы работают корректно).

- Подробности реализации представлены в разделе 19.7.2.
- Например:

```
is_move_constructible_v<int> // true
is_move_constructible_v<void> // false
is_move_constructible_v<std::unique_ptr<int>> // true
is_move_constructible_v<std::string> // true
is_move_constructible_v<std::string&> // true
is_move_constructible_v<std::string&&> // true

// В отличие от:
is_constructible_v<std::string, std::string> // true
is_constructible_v<std::string&, std::string&> // true
is_constructible_v<std::string&&, std::string&&> // true

std::is_assignable<TO, FROM>::value
std::is_trivially_assignable<TO, FROM>::value
std::is_nothrow_assignable<TO, FROM>::value
```

- Дает `true`, если объект типа `FROM` может быть присвоен объекту типа `TO` (без использования нетривиальной операции или с гарантией отсутствия генерации исключений).
- Требует, чтобы данный тип был полным (см. раздел 10.3.1), (свалифицированным) `void`, или же массивом с неизвестной границей.
- Обратите внимание на то, что `is_assignable_v<>` для первого типа, не являющегося ссылкой или классом, всегда дает `false`, поскольку такие типы производят `r`-значения. Таким образом, инструкция `42=77`; является некорректной. Однако для типов классов возможно присваивание `r`-значениям при наличии соответствующего оператора присваивания (благодаря старому правилу о том, что неконстантные функции-члены могут вызываться для `r`-значений типов классов)¹⁰.
- Обратите внимание: `is_convertible` имеет иной порядок исходного и целевого типов.
- Например:

```
is_assignable_v<int, int> // false
is_assignable_v<int&, int> // true
is_assignable_v < int&&, int> // false
is_assignable_v<int&, int&> // true
is_assignable_v < int&&, int&&> // false
is_assignable_v<int&, long&> // true
is_assignable_v<int&, void*> // false
is_assignable_v<void*, int> // false
is_assignable_v<void*, int&> // false
is_assignable_v<std::string, std::string> // true
is_assignable_v<std::string&, std::string&> // true
is_assignable_v < std::string&&, std::string&&> // true
```

¹⁰ Выражаем благодарность Даниэлю Крюглеру (Daniel Krügler) за указание на этот факт.

```
std::is_copy_assignable<T>::value
std::is_trivially_copy_assignable<T>::value
std::is_nothrow_copy_assignable<T>::value
```

- Даёт `true`, если значение типа `T` может быть присвоено (с копированием) объекту типа `T` (без использования нетривиальной операции или с гарантией отсутствия генерации исключений).
- Даёт `false`, если `T` не является типом, на который можно ссылаться (`referenceable type`) ((`cv`-квалифицированный) `void` или тип функции, квалифицированный с помощью `const`, `volatile`, `&`, `&&`).
- Указывает, является ли тип `T` типом, на который можно ссылаться, также как и `is_assignable<T&, T const&>::value`, `is_trivially_assignable<T&, T const&>::value` или соответственно `is_nothrow_assignable<T&, T const&>::value`.
- Чтобы выяснить, можно ли г-значение типа `T` присвоить копированием другому г-значению типа `T`, используйте `is_assignable<T&&, T&&>` и т.д.
- Обратите внимание на то, что копирующее присваивание неприменимо для типов `void`, встроенных типов массивов и классов с удаленным оператором копирующего присваивания.
- Требует, чтобы данный тип был полным (см. раздел 10.3.1), (`cv`-квалифицированным) `void` или массивом с неизвестной границей.
- Например:

```
is_copy_assignable_v<int> // true
is_copy_assignable_v<int&> // true
is_copy_assignable_v<int&&> // true
is_copy_assignable_v<void> // false
is_copy_assignable_v<void*> // true
is_copy_assignable_v<char[]> // false
is_copy_assignable_v<std::string> // true
is_copy_assignable_v<std::unique_ptr<int>> // false
```

```
std::is_move_assignable<T>::value
std::is_trivially_move_assignable<T>::value
std::is_nothrow_move_assignable<T>::value
```

- Даёт `true`, если г-значение типа `T` может быть присвоено перемещением объекту типа `T` (без использования нетривиальной операции или с гарантией отсутствия генерации исключений).
- Даёт `false`, если `T` не является типом, на который можно ссылаться (`referenceable type`) ((`cv`-квалифицированный) `void` или тип функции, квалифицированный с помощью `const`, `volatile`, `&`, `&&`).
- Указывает, является ли тип `T` типом, на который можно ссылаться, также как и `is_assignable<T&, T const&>::value`, `is_trivially_assignable<T&, T const&>::value` или соответственно `is_nothrow_assignable<T&, T const&>::value`.

- Обратите внимание на то, что перемещающее присваивание неприменимо для типов `void`, встроенных типов массивов и классов с удаленным оператором перемещающего присваивания.
- Требует, чтобы данный тип был полным (см. раздел 10.3.1), (свалифицированным) `void` или массивом с неизвестной границей.
- Например:

```
is_moveAssignable_v<int> // true
is_moveAssignable_v<int&> // true
is_moveAssignable_v<int&&> // true
is_moveAssignable_v<void> // false
is_moveAssignable_v<void*> // true
is_moveAssignable_v<char[]> // false
is_moveAssignable_v<std::string> // true
is_moveAssignable_v<std::unique_ptr<int>> // true
```

`std::is_destructible<T>::value`
`std::is_trivially_destructible<T>::value`
`std::is_nothrow_destructible<T>::value`

- Дает `true`, если объект типа `T` может быть уничтожен (без использования нетривиальной операции или с гарантией отсутствия генерации исключений).
- Всегда дает `true` для ссылок.
- Всегда дает `false` для `void`, типов массивов с неизвестными границами и типов функций.
- `is_trivially_destructible` дает `true`, если никакой из деструкторов `T`, любого базового класса или любого нестатического члена-данного не является пользовательским или виртуальным.
- Требует, чтобы данный тип был полным (см. раздел 10.3.1), (свалифицированным) `void` или массивом с неизвестной границей.
- Например:

```
isDestructible_v<void> // false
isDestructible_v<int> // true
isDestructible_v<std::string> // true
isDestructible_v<std::pair<int, std::string>> // true
isTriviallyDestructible_v<void> // false
isTriviallyDestructible_v<int> // true
isTriviallyDestructible_v<std::string> // false
isTriviallyDestructible_v<std::pair<int, int>> // true
isTriviallyDestructible_v<std::pair<int, std::string>> // false
```

`std::is_swappable_with<T1, T2>::value`
`std::is_nothrow_swappable_with<T1, T2>::value`

- Дает `true`, если выражение типа `T1` может быть обменено (с использованием `swap()`) с выражением типа `T2` с тем исключением, что только ссылочные типы определяют категорию значения выражения (с гарантией отсутствия генерации исключений).

- Требует, чтобы данный тип был полным (см. раздел 10.3.1), (cv-квалифицированным) void или массивом с неизвестной границей.
- Обратите внимание на то, что `is_swappable_with_v<T>` для типов, не являющихся ссылками или классами, всегда дает `false`, поскольку такие типы производят `rg`-значения. Таким образом, `swap(42, 77)` является некорректной инструкцией.
- Например:

```

is_swappable_with_v<int, int>           // false
is_swappable_with_v<int&, int>           // false
is_swappable_with_v<int&&, int>          // false
is_swappable_with_v<int&, int&>          // true
is_swappable_with_v < int&&, int&& >      // false
is_swappable_with_v<int&, long&>          // false
is_swappable_with_v<int&, void*>          // false
is_swappable_with_v<void*, int>           // false
is_swappable_with_v<void*, int&>          // false
is_swappable_with_v<std::string, std::string> // false
is_swappable_with_v<std::string&, std::string&> // true
is_swappable_with_v < std::string&&, std::string&& > // false

```

- Доступно, начиная со стандарта C++17.

```

std::is_swappable<T>::value
std::is_nothrow_swappable<T>::value

```

- Дает `true`, если `l`-значение типа `T` может быть обменено (с гарантией отсутствия генерации исключений).
- Указывает, является ли тип `T` типом, на который можно ссылаться, так же как `is_swappable_with<T&, T&>::value` или `is_nothrow_swappable_with<T&, T&>::value` соответственно.
- Дает `false`, если `T` не является типом, на который можно ссылаться (`referenceable type`) ((cv-квалифицированный) `void` или тип функции, квалифицированный с помощью `const`, `volatile`, `&`, `&&`).
- Чтобы выяснить, можно ли `g`-значение типа `T` обменивать с другим `g`-значением типа `T`, используйте `is_swappable_with<T&&, T&&>`.
- Требует, чтобы данный тип был полным (см. раздел 10.3.1), (cv-квалифицированным) `void` или массивом с неизвестной границей.
- Например:

```

is_swappable_v<int>           // true
is_swappable_v<int&>           // true
is_swappable_v < int&& >        // true
is_swappable_v<std::string&&> // true
is_swappable_v<void>           // false
is_swappable_v<void*>          // true
is_swappable_v<char[]>          // false
is_swappable_v<std::unique_ptr<int>> // true

```

- Доступно, начиная со стандарта C++17.

Г.3.3. Взаимоотношения между типами

В табл. Г.5 перечислены свойства типов, которые позволяют проверять определенные взаимоотношения между типами. Сюда включается проверка, какие конструкторы и операторы присваивания предоставлены для типов классов.

Таблица Г.5. Свойства проверки взаимоотношений типов

Свойство	Эффект
<code>is_same<T1, T2></code>	T1 и T2 являются одним и тем же типом (включая квалифиликаторы <code>const/volatile</code>)
<code>is_base_of<T, D></code>	Тип T является базовым классом для типа D
<code>is_convertible<T, T2></code>	Тип T конвертируем в тип T2

`std::is_same<T1, T2>::value`

- Дает `true`, если T1 и T2 именуют один и тот же тип, включая cv-квалификаторы (`const` и `volatile`).
- Дает `true`, если один тип является псевдонимом другого типа.
- Дает `true`, если два объекта были инициализированы объектами одного и того же типа.
- Дает `false` для типов (замыканий), связанных с двумя различными лямбда-выражениями, даже если они определяют одно и то же поведение.
- Например:

```
auto a = nullptr;
auto b = nullptr;
is_same_v<decltype(a), decltype(b)> // Дает true
using A = int;
is_same_v<A, int> // Дает true
auto x = [](int) {};
auto y = x;
auto z = [](int) {};
is_same_v<decltype(x), decltype(y)> // Дает true
is_same_v<decltype(x), decltype(z)> // Дает false
```

- Подробности реализации представлены в разделе 19.3.3.

`std::is_base_of<B, D>::value`

- Дает `true`, если B является базовым классом для D или если B — тот же класс, что и D.
- Не имеет значения, является ли тип cv-квалифицированным, использовано ли закрытое или защищенное наследование, имеет ли D несколько базовых классов типа B, или D имеет B в качестве базового класса через множественные пути наследования (через виртуальное наследование).
- Дает `false`, если хотя бы один из типов представляет собой `union`.

- Требует, чтобы тип D либо был полным (см. раздел 10.3.1), имел тот же тип, что и B (игнорируя любые квалификаторы `const/volatile`), либо не был ни `struct`, ни `class`.
- Например:

```
class B
{
};
class D1 : B
{
};
class D2 : B
{
};
class DD : private D1, private D2
{
};
is_base_of_v<B, D1>      // Дает true
is_base_of_v<B, DD>       // Дает true
is_base_of_v<B const, DD> // Дает true
is_base_of_v<B, DD const> // Дает true
is_base_of_v<B, B const>  // Дает true
is_base_of_v<B&, DD&>    // Дает false (не тип класса)
is_base_of_v<B[3], DD[3]> // Дает false (не тип класса)
is_base_of_v<int, int>    // Дает false (не тип класса)
```

`std::is_convertible<FROM, TO>::value`

- Дает `true`, если выражение типа FROM может быть преобразовано в тип TO. Таким образом, должен быть корректен следующий код¹¹:

```
TO test()
{
    return std::declval<FROM>();
}
```

- Ссылка на тип FROM используется только для определения категории значения преобразуемого выражения; типом исходного выражения является базовый тип.
- Обратите внимание на то, что из `is_constructible` *не* всегда следует `is_convertible`. Например:

```
class C
{
public:
    explicit C(C const&); // Неявного копирующего
    ...                  // конструктора нет
};
is_constructible_v<C, C> // true
is_convertible_v<C, C>  // false
```

- Требует, чтобы данный тип был полным (см. раздел 10.3.1), (св-квалифицированным) `void` или массивом с неизвестной границей.

¹¹ См. информацию о `std::declval` в разделе 11.2.3.

- Обратите внимание на то, что `is_constructible` и `is_assignable` имеют различный порядок исходного и целевого типов.
- Подробности реализации представлены в разделе 19.5.

Г.4. Построение типов

Свойства, перечисленные в табл. Г.6, позволяют создавать типы из других типов.

Таблица Г.6. Свойства для построения типов

Свойство	Эффект
<code>remove_const<T></code>	Соответствующий тип без <code>const</code>
<code>remove_volatile<T></code>	Соответствующий тип без <code>volatile</code>
<code>remove_cv<T></code>	Соответствующий тип без <code>const</code> и <code>volatile</code>
<code>add_const<T></code>	Соответствующий <code>const</code> -тип
<code>add_volatile<T></code>	Соответствующий <code>volatile</code> -тип
<code>add_cv<T></code>	Соответствующий <code>const volatile</code> -тип
<code>make_signed<T></code>	Соответствующий знаковый нессылочный тип
<code>make_unsigned<T></code>	Соответствующий беззнаковый нессылочный тип
<code>remove_reference<T></code>	Соответствующий нессылочный тип
<code>add_lvalue_reference<T></code>	Соответствующий тип ссылки на <code>l</code> -значение (<code>g</code> -значения становятся <code>l</code> -значениями)
<code>add_rvalue_reference<T></code>	Соответствующий тип ссылки на <code>g</code> -значение (<code>l</code> -значения остаются <code>l</code> -значениями)
<code>remove_pointer<T></code>	Тип, на который указывает указатель (для не указателей тип остается неизменным)
<code>add_pointer<T></code>	Тип указателя на соответствующий нессылочный тип
<code>remove_extent<T></code>	Типы элементов массива (для немассивов тип остается неизменным)
<code>remove_all_extents<T></code>	Типы элементов многомерных массивов (для немассивов тип остается неизменным)
<code>decay<T></code>	Преобразование в соответствующий тип “передачи по значению”

```
std::remove_const<T>::type
std::remove_volatile<T>::type
std::remove_cv<T>::type
```

- Дает тип `T` без `const` или/и `volatile` на верхнем уровне.

- Обратите внимание на то, что const-указатель представляет собой const-квалифицированный тип, в то время как не-const указатель или ссылка на константный тип не являются const-квалифицированными. Например:

```
remove_cv_t<int>           // int
remove_const_t<int const>    // int
remove_cv_t<int const volatile> // int
remove_const_t<int const&>   // int const& (только
                             // ссылается на int const)
```

Очевидно, что порядок применения свойств имеет значение¹²:

```
remove_const_t<remove_reference_t<int const&>> // int
remove_reference_t<remove_const_t<int const&>> // int const
```

Вместо этого можно использовать свойство `std::decay<>`, которое, однако, преобразует типы массивов и функций в соответствующие типы указателей:

```
decay_t<int const&> // int
```

- Подробности реализации приведены в разделе 19.3.2.

```
std::add_const<T>::type
std::add_volatile<T>::type
std::add_cv<T>::type
```

- Дает тип T с квалификатором `const` или/и `volatile`, добавленным на верхнем уровне.
- Применение одного из этих свойств к ссылочному типу или типу функции не выполняет никаких действий. Например:

```
add_cv_t<int>           // int const volatile
add_cv_t<int const>     // int const volatile
add_cv_t<int const volatile> // int const volatile
add_const_t<int>         // int const
add_const_t<int const>   // int const
add_const_t<int&>       // int&
```

```
std::make_signed<T>::type
std::make_unsigned<T>::type
```

- Дает соответствующий знаковый/беззнаковый тип T.
- Требует, чтобы тип T был типом перечисления или (cv-квалифицированным) целочисленным типом, отличным от `bool`. Все прочие типы ведут к неопределенному поведению (см. в разделе 19.7.1 обсуждение вопроса о том, как избежать этого неопределенного поведения). Например:

```
make_unsigned_t<char>      // Дает unsigned char
make_unsigned_t<int>        // Дает unsigned int
make_unsigned_t<int const&> // Неопределенное поведение
```

```
std::remove_reference<T>::type
```

- Дает тип, на который указывает ссылочный тип T (или сам тип T, если он не является ссылочным типом).

¹² Стандарт, следующий за C++17, вероятно, по этой причине введет свойство `remove_refcv`.

- Например:

```
remove_reference_t<int>           // Дает int
remove_reference_t<int const>       // Дает int const
remove_reference_t<int const&>     // Дает int const
remove_reference_t < int&& >      // Дает int
```

- Обратите внимание на то, что сам по себе ссылочный тип не является константным. По этой причине порядок применения свойств построения типов имеет значение¹³:

```
remove_const_t<remove_reference_t<int const&>> // int
remove_reference_t<remove_const_t<int const&>> // int const
```

Вместо этого можно использовать свойство `std::decay<>`, которое, однако, преобразует типы массивов и функций в соответствующие типы указателей:

```
decay_t<int const&> // Дает int
```

- Подробности реализации приведены в разделе 19.3.2.

```
std::add_lvalue_reference<T>::type
std::add_rvalue_reference<T>::type
```

- Дает ссылку на l-значение или ссылку на r-значение типа T, если тип T позволяет создавать ссылку на него.
- Дает T, если тип T позволяет создавать на него ссылку (является либо (свалифицированным) `void`, либо типом функции, квалифицированным с помощью `const`, `volatile`, `&` или `&&`).
- Обратите внимание на то, что, если T уже является ссылочным типом, эти свойства используют правила свертки ссылок (см. раздел 15.6.1): результат является ссылкой на r-значение только тогда, когда используется `add_rvalue_reference`, а T представляет собой ссылку на r-значение.

- Например:

```
add_lvalue_reference_t<int>           // int&
add_rvalue_reference_t<int>           // int&&
add_rvalue_reference_t<int const>     // int const&&
add_lvalue_reference_t<int const&>   // int const&
add_rvalue_reference_t<int const&>   // int const&
                                         // (согласно правилам свертки ссылок)
add_rvalue_reference_t<
    remove_reference_t<int const&>> // int&&
add_lvalue_reference_t<void>          // void
add_rvalue_reference_t<void>          // void
```

- Подробности реализации описаны в разделе 19.3.2.

```
std::remove_pointer<T>::type
```

- Дает тип, на который указывает тип указателя T (или сам тип T, если он не является типом указателя).

¹³ Стандарт, следующий за C++17, вероятно, по этой причине введет свойство `remove_refcv`.

- Например:

```
remove_pointer_t<int> // int
remove_pointer_t<int const*> // int const
remove_pointer_t<int const* const* const> // int const* const
```

`std::add_pointer<T>::type`

- Дает тип указателя на Т, или, в случае ссылочного типа Т, тип указателя на базовый тип Т.
- Дает Т, если такой тип не существует (применяется к скв-квалифицированным типам функций).
- Например:

```
add_pointer_t<void> // void*
add_pointer_t<int const* const> // int const* const*
add_pointer_t<int&> // int*
add_pointer_t<int[3]> // int(*)[3]
add_pointer_t<void(&)(int)> // void(*)(int)
add_pointer_t<void(int)> // void(*)(int)
add_pointer_t<void(int) const> // void(int) const
// (без изменений)
```

`std::remove_extent<T>::type`

`std::remove_all_extents<T>::type`

- Для данного типа массива `remove_extent` производит тип элемента (который сам по себе может быть типом массива), а `remove_all_extents` удаляет все “слои массивов” для того, чтобы получить базовый тип элементов (который, таким образом, больше не может быть массивом). Если Т не является типом массива, свойство дает сам тип Т.
- Указатели не имеют какого-либо связанного с ними измерения. Не указанная граница в типе массива не определяет измерения. (Как обычно, параметр функции, объявленный с типом массива, не имеет фактического типа массива, как и `std::array` не является типом массива.)
- Например:

```
remove_extent_t<int> // int
remove_extent_t<int[10]> // int
remove_extent_t<int[5][10]> // int[10]
remove_extent_t<int[][10]> // int[10]
remove_extent_t<int*> // int*
remove_all_extents_t<int> // int
remove_all_extents_t<int[10]> // int
remove_all_extents_t<int[5][10]> // int
remove_all_extents_t<int[][10]> // int
remove_all_extents_t<int(*)[5]> // int(*)[5]
```

- Подробности реализации представлены в разделе 23.1.2.

`std::decay<T>::type`

- Дает низведененный тип Т.
- Подробнее – для типа Т выполняются следующие преобразования:

- сначала применяется `remove_reference`;
- если результат представляет собой тип массива, создается указатель на тип элемента (см. раздел 7.1);
- в противном случае, если результат представляет собой тип функции, создается тип, полученный путем применения `add_pointer` к этому типу функции (см. раздел 11.1.1);
- в противном случае результат получается без каких бы то ни было квалификаторов `const/volatile` верхнего уровня.
- Свойство `decay<>` моделирует передачу аргументов в функцию по значению или преобразования типов при инициализации объектов типа `auto`.
- Свойство `decay<>` в особенности полезно для обработки параметров шаблонов, вместо которых могут подставляться ссылочные типы, но использоваться для определения возвращаемого типа или типа параметров другой функции. Применение `std::decay<>()` рассмотрено в разделах 1.3.2 и 7.6 (в последнем также излагается история реализации `std::make_pair<>()`).
- Например:


```
decay_t<int const&>      // Дает int
decay_t<int const[4]>      // Дает int const*
void foo();
decay_t<decltype(foo)>    // Дает void(*)()
```
- Подробности реализации приведены в разделе 19.3.2.

Г.5. Прочие свойства

В табл. Г.7 перечислены все остальные свойства типов. Они запрашивают специальные характеристики или обеспечивают более сложные преобразования типов.

Таблица Г.7. Прочие свойства типов

Свойство	Эффект
<code>enable_if<B, T=void></code>	Дает тип T, только если значение B типа <code>bool</code> равно <code>true</code>
<code>conditional<B, T, F></code>	Дает тип T, если значение B типа <code>bool</code> равно <code>true</code> , и тип F в противном случае
<code>common_type<T1, ...></code>	Общий тип для всех переданных типов
<code>aligned_storage<Len></code>	Тип Len байтов с выравниванием по умолчанию
<code>aligned_storage<Len, Align></code>	Тип Len байтов с выравниванием согласно делителю Align типа <code>size_t</code>
<code>aligned_union<Len, Types...></code>	Тип Len байтов с выравниванием для объединения типов Types...

```
std::enable_if<cond>::type
std::enable_if<cond, T>::type
```

- Дает `void` или `T` в качестве члена `type`, если значение `cond` равно `true`. В противном случае член `type` не определен.
- Поскольку член `type` при `cond`, равном `false`, не определен, это свойство может использоваться (и обычно используется) для отключения (в частности, с использованием принципа SFINAE) шаблона функции на основании заданного условия.
- В разделе 6.3 подробно обсуждается это свойство и приведен соответствующий пример; в разделе Г.6 рассмотрен другой пример с использованием пакетов параметров.
- Подробности реализации `std::enable_if` описаны в разделе 20.3.

```
std::conditional<cond, T, F>::type
```

- Дает `T`, если значение `cond` равно `true`, и `F` в противном случае.
- Это стандартная версия свойства `IfThenElseT`, введенного в разделе 19.7.1.
- Обратите внимание: в отличие от обычных инструкций `if` C++ аргументы шаблона вычисляются для обоих ветвей до осуществления выбора, так что ни одна ветвь не может содержать некорректный код, иначе программа становится некорректной. Как следствие, возможно, придется добавить уровень косвенности для того, чтобы избежать вычисления выражений в неиспользуемой ветви. Этот способ продемонстрирован в разделе 19.7.1 для свойства `IfThenElseT`, имеющего такое же поведение.
- Пример приведен в разделе 11.5.
- Детали реализации `std::conditional` рассматриваются в разделе 19.7.1.

```
std::common_type<T...>::type
```

- Дает “общий тип” для данных типов T_1, T_2, \dots, T_n .
- Вычисление общего оказывается несколько более сложным, чем мы хотели бы видеть в приложении. Грубо говоря, общий тип двух типов `U` и `V` является типом, который возвращает условный оператор `?:`, когда его второй и третий operandы имеют типы `U` и `V` (с ссылочными типами, используемыми только для выяснения категорий значений двух operandов). Если соответствующее выражение недопустимо, общего типа не существует. К полученному результату применяется `decay_t`. Вычисления по умолчанию могут быть перекрыты пользовательской специализацией `std::common_type<U, V>` (в стандартной библиотеке C++ имеется частичная специализация для длительностей и точек времени).
- Если типы не заданы или общего типа не существует, то член `type` не определен, так что его использование является ошибкой (которая может использоваться с принципом SFINAE для отбрасывания шаблона функции).
- Если задан единственный тип, то результатом является применение к нему свойства `decay_t`.

- Для более чем двух типов `common_type` рекурсивно заменяет первые два типа T_1 и T_2 их общим типом. Если в любой момент процесс оказывается неудачным, общего типа не существует.
- При обработке общего типа переданные типы низводятся, так что свойство всегда дает низведенный тип (см. раздел Г.4).
- Обсуждение этого свойства и пример его применения приведены в разделе 1.3.3.
- Первичный шаблон этого свойства обычно реализуется с помощью кода наподобие следующего (здесь использованы только два параметра):

```
template<typename T1, typename T2>
struct common_type<T1, T2>
{
    using type = std::decay_t<decltype(true ?
        std::declval<T1>() :
        std::declval<T2>())>;
};
```

`std::aligned_union<MIN_SZ, T...>::type`

- Дает *простой старый тип данных* (POD), используемый как неинициализированное хранилище, имеющее размер как минимум `MIN_SZ` и походящее для хранения любого из заданных типов T_1, T_2, \dots, T_n .
- Кроме того, это свойство дает статический член `alignment_value`, значение которого представляет собой наиболее строгое выравнивание для всех данных типов, которое для результирующего типа `type` эквивалентно следующему:
 - `std::alignment_of_v<type>::value` (см. раздел Г.3.1);
 - `alignof(type)`.
- Требует, чтобы был предоставлен по крайней мере один тип.
- Например:

```
using POD_T = std::aligned_union_t<0, char,
    std::pair<std::string, std::string>>;
std::cout << sizeof(POD_T) << '\n';
std::cout << std::aligned_union<0, char,
    std::pair<std::string, std::string>
    >::alignment_value;
<< '\n';
```

Обратите внимание на то, что применение `aligned_union` вместо `aligned_union_t` дает вместо типа значение выравнивания.

`std::aligned_storage<MAX_TYPE_SZ>::type`

`std::aligned_storage<MAX_TYPE_SZ, DEF_ALIGN>::type`

- Дает *простой старый тип данных* (POD), используемый как неинициализированное хранилище для хранения всех возможных типов с размером до `MAX_TYPE_SZ`, с учетом выравнивания по умолчанию или выравнивания, переданного как `DEF_ALIGN`.

- Требует, чтобы значение `MAX_TYPE_SZ` было больше нуля, а на платформе имелся хотя бы один тип со значением выравнивания `DEF_ALIGN`.
- Например:

```
using POD_T = std::aligned_storage_t<5>;
```

Г.6. Комбинирование свойств типов

В большинстве контекстов несколько свойств типов, являющихся предикатами, могут быть объединены с помощью логических операторов. Однако в некоторых контекстах шаблонного метапрограммирования этого недостаточно:

- если вы работаете со свойством, которое *может* сбить (например, из-за неполноты типов);
- если вы хотите комбинировать определения свойств типов.

Для этой цели предназначены свойства типов `std::conjunction<>`, `std::disjunction<>` и `std::negation<>`.

Одним из примеров является применение этих вспомогательных свойств для короткого вычисления логических выражений (прекратить вычисление после первого значения `false` для операции `&&`, или первого значения `true` для операции `||` соответственно)¹⁴. Например, если используются неполные типы

```
struct X
{
    X(int); // Преобразование из int
};

struct Y; // Неполный тип
```

тот приведенный ниже код может не компилироваться, поскольку `is_constructible` приведет к неопределенному поведению для неполных типов (хотя некоторые компиляторы могут принять этот код):

```
// Неопределенное поведение:
static_assert(std::is_constructible<X, int> {}
              || std::is_constructible<Y, int> {},
              "нельзя инициализировать X или Y значением int");
```

Однако следующий код гарантированно компилируется, так как вычисление `is_constructible<X, int>` всегда дает `true`:

```
// OK:
static_assert(std::disjunction<std::is_constructible<X, int>,
              std::is_constructible<Y, int> {}),
              "нельзя инициализировать X или Y значением int");
```

Еще одним применением является простой способ определить новое свойство типа, логически объединяя существующие свойства. Например, можно легко определить свойство, проверяющее, является ли тип “не указателем” (т.е. не является ни указателем, ни указателем на член, ни нулевым указателем):

¹⁴ Выражаем благодарность Говарду Хиннанту (Howard Hinnant) за указание на этот факт.

```
template<typename T>
struct isNoPtrT :
    std::negation<
        std::disjunction<std::is_null_pointer<T>,
                        std::is_member_pointer<T>,
                        std::is_pointer<T>
        >
    > {};
```

Здесь мы не можем использовать логические операторы, потому что мы объединяем соответствующие классы свойств. Имея такое определение свойства, мы можем написать следующий код:

```
std::cout << isNoPtrT<void*>::value << '\n';      // false
std::cout << isNoPtrT<std::string>::value << '\n';   // true
auto np = nullptr;
std::cout << isNoPtrT<decltype(np)>::value << '\n'; // false
```

А при наличии соответствующего шаблона переменной:

```
template<typename T>
constexpr bool isNoPtr = isNoPtrT<T>::value;
```

можно записать:

```
std::cout << isNoPtr<void*> << '\n'; // false
std::cout << isNoPtr<int> << '\n'; // true
```

В качестве последнего примера следующий шаблон функции доступен, только если все его аргументы шаблона не являются ни классами, ни объединениями:

```
template<typename... Ts>
std::enable_if_t<std::conjunction_v<
    std::negation<std::is_class<Ts>>...,
    std::negation<std::is_union<Ts>>...
>>
print(Ts...)
{
    ...
}
```

Обратите внимание на троеточие, размеченное после каждого отрицания; таким образом, отрицание применяется к каждому элементу пакета параметров.

Таблица Г.8. Свойства типов для комбинирования других свойств типов

Свойство	Эффект
<code>conjunction<B...></code>	Логическое И для булевых свойств B... (начиная с C++17)
<code>disjunction<B...></code>	Логическое ИЛИ для булевых свойств B... (начиная с C++17)
<code>negation</code>	Логическое НЕ для булева свойства B (начиная с C++17)

```
std::conjunction<B...>::value
std::disjunction<B...>::value
```

- Проверяет, все ли (или одно из них) переданные булевые свойства B... дают `true`.
- Применяет логические операторы `&&` или `||` соответственно к переданным свойствам.
- Оба свойства используют краткие вычисления (вычисление прекращается после первого `false` (или, соответственно, `true`)).
- Выше представлен соответствующий пример.
- Доступно, начиная со стандарта C++17.

```
std::negation<B>::value
```

- Проверяет, дает ли переданное булево свойство B значение `false`.
- Применяет логический оператор `!` к переданному свойству.
- Выше приведен соответствующий пример.
- Доступно, начиная со стандарта C++17.

Г.7. Прочие утилиты

Стандартная библиотека C++ предоставляет несколько других утилит, полезных при написании переносимого обобщенного кода.

Таблица Г.9. Прочие утилиты для метапрограммирования

Свойство	Эффект
<code>declval<T>()</code>	Дает “объект” (ссылку на г-значение) типа без фактического создания объекта
<code>addressof(r)</code>	Дает адрес объекта или функции

```
std::declval<T>()
```

- Определено в заголовочном файле `<utility>`.
- Даёт “объект” или функцию любого типа без вызова каких бы то ни было конструкторов или инициализаций.
- Если T представляет собой `void`, то возвращаемый тип – `void`.
- Это свойство может быть использовано для работы с объектами или функциями любого типа в невычисляемых выражениях.
- Свойство определено как

```
template<typename T>
add_rvalue_reference_t<T> declval() noexcept;
```

так что

- если T является простым типом или ссылкой на г-значение, свойство даёт `T&&`;

- если T является ссылкой на l-значение, свойство дает $T\&;$;
 - если T представляет собой `void`, свойство дает `void`.
- Подробности приведены в разделах 19.3.4 и 11.2.3, а в описании `common_type<>` в разделе Г.5 – пример применения.

`std::addressof(r)`

- Определено в заголовочном файле `<memory>`.
- Дает адрес объекта или функции r , даже если `operator&` для этого типа перегружен.
- Подробности представлены в разделе 11.2.2.

Приложение Д

Концепты

На протяжении многих лет разработчики языка C++ изучали способы ограничения параметров шаблонов. Например, в нашем шаблоне функции `max()` мы хотели бы с самого начала объявить, что он не должен вызываться для типов, которые не сопоставимы с использованием оператора “меньше”. Для других шаблонов может потребоваться, чтобы они инстанцировались с типами, которые представляют собой корректные типы итераторов (для некоторого официального определения этого термина) или для корректных “арифметических” типов (понятие, которое может быть более широким, чем просто множество встроенных арифметических типов).

Концепт представляет собой именованный набор ограничений на один или несколько параметров шаблона. При разработке C++11 для него была создана богатая система концептов, но интеграция этой возможности в спецификации языка в конечном итоге потребовала слишком много ресурсов Комитета, так что концепты в конечном итоге были исключены из C++11. Некоторое время спустя был предложен иной их дизайн, и похоже, что именно он в конечном итоге войдет в язык в той или иной форме. Непосредственно перед тем, как оригинал этой книги отправился в печать, Комитет по стандартизации проголосовал за интеграцию нового дизайна в проект C++20. В этом приложении мы опишем основные элементы этого нового дизайна.

Мы уже говорили о концептах и показали некоторые их применения в основном тексте этой книги.

- В разделе 6.5 показано, как использовать требования и концепты для того, чтобы разрешить конструктор только в том случае, когда параметр шаблона преобразуем в строку (чтобы избежать случайного использования конструктора в качестве копирующего конструктора).
- В разделе 18.4 показано, как использовать концепты для указания и обеспечения выполнения ограничений на типы, используемые для представления геометрических объектов.

Д.1. Использование концептов

Сначала рассмотрим использование концептов в коде клиента (т.е. коде, который определяет шаблоны без обязательного определения концептов, применимых к параметрам шаблона).

Работа с требованиями

Вот как выглядит наш привычный шаблон `max()` с двумя параметрами при использовании ограничения:

```
template<typename T> requires LessThanComparable<T>
T max(T a, T b)
{
    return b < a ? a : b;
}
```

Единственным добавлением к нему является *конструкция требования* (*requires clause*)

```
requires LessThanComparable<T>
```

которая полагает, что ранее был объявлен — скорее всего, путем включения заголовочного файла — *концепт LessThanComparable*.

Такой концепт представляет собой булев предикат (т.е. выражение, которое производит значение типа `bool`), вычисляющий константное выражение. Это важно, потому что ограничения проверяются во время компиляции и поэтому не создают никаких накладных расходов с точки зрения генерированного кода: этот шаблон с ограничением будет производить столь же быстрый код, как и для рассматривавшейся ранее версии без ограничений.

При попытках использовать этот шаблон он не будет инстанцироваться до тех пор, пока не будет вычислена конструкция требования и не выяснится, что она дает значение `true`. Если она дает значение `false`, может быть выведено сообщение об ошибке, объясняющее, какая часть требования оказалась не выполненной (или может быть выбран перегруженный шаблон, не приводящий к нарушениям требований).

Конструкции требований *не обязаны* выражаться в терминах концептов (хотя это хорошая практика, имеющая тенденцию к производству лучшей диагностики): может использоваться любое булево константное выражение. Например, как показано в разделе 6.5, приведенный ниже пример кода гарантирует, что шаблонный конструктор нельзя использовать в качестве копирующего конструктора:

```
class Person
{
private:
    std::string name;
public:
    template<typename STR>
    requires std::is_convertible_v<STR, std::string>
    explicit Person(STR&& n)
        : name(std::forward<STR>(n))
    {
        std::cout << "Шаблонный конструктор для '" << name << "'\n";
    }
    ...
};
```

Здесь именованный концепт не используется, что может оказаться вполне приемлемым решением, поскольку тут применено соответствующее булево выражение (в данном случае с использованием свойства типа)

```
std::is_convertible_v<STR, std::string>
```

позволяющее решить проблему с помощью шаблонного конструктора в качестве копирующего конструктора. Подробности организации концептов и ограничений по-прежнему активно исследуются сообществом C++ и, скорее всего, будут развиваться с течением времени, но представляется, что достигнуто общее согласие о том, что концепты должны отражать смысл кода, а не то, компилируется ли он.

Работа с множественными требованиями

В приведенном выше примере есть только одно требование, но не редкость случаи, когда имеется несколько требований. Например, можно представить концепт `Sequence`, который описывает последовательность значений элементов (соответствуя тому же понятию в стандарте) и шаблон `find()`, который для данной последовательности и значения возвращает итератор, указывающий на первое вхождение значения в последовательность (если таковые имеются). Этот шаблон может быть определен следующим образом:

```
template<typename Seq>
requires Sequence<Seq>&&
EqualityComparable<typename Seq::value_type>
typename Seq::iterator find(Seq const& seq,
                           typename Seq::value_type const& val)
{
    return std::find(seq.begin(), seq.end(), val);
}
```

Здесь любой вызов данного шаблона будет сначала проверять поочередно каждое требование, и, только если все требования будут давать значение `true`, шаблон может быть выбран для вызова и инстанцирования (конечно, при условии, что разрешение перегрузки не отбросит шаблон по другим причинам, таким как наличие другого, лучше подходящего шаблона).

Можно также выразить “альтернативные” требования с помощью `||`. Это редко необходимо, и обращаться к этому способу следует с осторожностью, так как чрезмерное использование оператора `||` в конструкциях требований может потенциально потреблять ресурсы компилятора (и заметно замедлять компиляцию). Однако в некоторых ситуациях это может быть очень удобно. Например:

```
template<typename T>
requires Integral<T> ||
FloatingPoint<T>
T power(T b, T p);
```

Одно требование может включать несколько параметров шаблона, и один концепт может выразить предикат над несколькими параметрами шаблона. Например:

```
template<typename T, typename U>
requires SomeConcept<T, U>
auto f(T x, U y) -> decltype(x + y)
```

Таким образом, концепт может устанавливать связь между параметрами типов.

Сокращенная запись единственного требования

Чтобы уменьшить накладные расходы конструкций требований, доступно синтаксическое сокращение, когда ограничение включает только один параметр. Пожалуй, его наиболее легко проиллюстрировать в объявлении нашего шаблона `max()` с ограничениями:

```
template<LessThanComparable T>
T max(T a, T b)
{
    return b < a ? a : b;
}
```

Функционально это определение эквивалентно предыдущему определению `max()`. Однако при повторном объявлении шаблона с ограничениями должна использоваться та же форма, что и в исходном объявлении (в этом смысле объявления разного вида эквивалентны только функционально).

Мы можем использовать такое сокращение для одного из двух требований в шаблоне `find()`:

```
template<Sequence Seq>
requires EqualityComparable<typename Seq::value_type>
typename Seq::iterator find(Seq const& seq,
                           typename Seq::value_type const& val)
{
    return std::find(seq.begin(), seq.end(), val);
}
```

Это определение эквивалентно приведенному выше определению шаблона `find()` для типов последовательностей.

Д.2. Определение концептов

Концепты очень похожи на шаблоны `constexpr`-переменных типа `bool`, но тип явно не указывается:

```
template<typename T> concept LessThanComparable = ... ;
```

Многоточие здесь заменяется выражением, которое использует различные свойства типов, чтобы установить, является ли тип `T` действительно сравнимым с помощью оператора `<`. Предложение о концептах в стандарт предоставляет средство для упрощения этой задачи: *выражение требования* (которое отличается от *конструкции требования*, описанной выше). Вот как может выглядеть полное определение концепта:

```
template<typename T>
concept LessThanComparable = requires(T x, T y)
{
    { x < y } -> bool;
};
```

Обратите внимание на то, что выражение требования может включать необязательный список параметров: эти параметры никогда не заменяются аргументами,

а вместо этого рассматриваются как “фиктивные переменные” для выражений требований. В нашем случае есть только одно такое требование, выражаемое как

```
{ x < y } -> bool;
```

Этот синтаксис означает, что 1) выражение `x < y` должно быть корректным в смысле SFINAЕ, и 2) результат этого выражения должен быть преобразуем в `bool`. Перед токеном `->` может быть добавлено ключевое слово `noexcept` для указания того факта, что выражение в скобках должно гарантированно не генерировать исключения (т.е. `noexcept(...)`, примененное к этому выражению, должно быть `true`). Часть неявного преобразования (т.е. `-> type`) может быть полностью опущена, если такое ограничение не требуется, и, если должна проверяться только корректность выражения, могут быть опущены фигурные скобки, так что все сводится к одному выражению. Например:

```
template<typename T>
concept Swappable = requires(T x, T y)
{
    swap(x, y);
};
```

Выражения требований могут также указывать требования к связанным типам. Рассмотрим концепт `Sequence`, о котором мы говорили ранее: помимо требования корректности таких выражений, как `seq.begin()`, требуется также наличие соответствующих типов итераторов последовательности, что можно выразить следующим образом:

```
template<typename Seq>
concept Sequence = requires(Seq seq)
{
    typename Seq::iterator;
    { seq.begin() } -> Seq::iterator;
    ...
};
```

Здесь выражение `typename type;` выражает требование существования типа (и называется *требованием типа*). В этом примере тип, который должен существовать, является членом параметра шаблона концепта, но это не обязательно. Мы могли бы, например, потребовать, чтобы существовал тип `IteratorFor<Seq>`, и это можно было бы достичь с помощью выражения

```
...
typename IteratorFor<Seq>;
...
```

Определение концепта `Sequence` выше показывает, как можно объединять требования, просто перечисляя их одно за другим. Существует третий класс требований, которые состоят в простом вызове другого концепта. Например, предположим, что у нас есть концепт для понятия *итератора*. Мы хотели бы, чтобы наш концепт `Sequence` не только требовал, чтобы `Seq::iterator` был типом, но и чтобы этот тип удовлетворял ограничениям концепта `Iterator`. Это выражается следующим образом:

```
template<typename Seq>
concept Sequence = requires(Seq seq)
{
    typename Seq::iterator;
    requires Iterator<typename Seq::iterator>;
    { seq.begin() } -> Seq::iterator;
    ...
};
```

Т.е. мы просто добавляем конструкцию требования в выражение требования (и получаем *вложенное требование*).

Д.3. Перегрузка ограничений

Предположим на минуту, что мы определили концепты `IntegerLike<T>` и `StringLike<T>` и решили написать шаблоны для вывода значения типов каждого концепта. Мы могли бы сделать это следующим образом:

```
template<IntegerLike T> void print(T val); // #1
template<StringLike T> void print(T val); // #2
```

Если бы не различие в ограничениях, эти два объявления объявляли бы один и тот же шаблон. Однако ограничения являются частью сигнатуры шаблона, что позволяет шаблонам быть различимыми во время разрешения перегрузки. В частности, если оба шаблона оказываются жизнеспособными кандидатами, но ограничения оказываются удовлетворены только для шаблона #1, то перегрузка выберет именно этот шаблон. Например, предполагая, что `int` удовлетворяет требованию `IntegerLike`, а `std::string` удовлетворяет требованию `StringLike`, но не наоборот:

```
int main()
{
    printf(1);      // Выбран шаблон #1
    printf("1"s);   // Выбран шаблон #2
}
```

Можно представить строковый тип, поддерживающий вычисления с целыми числами. Например, если `"6"_NS` и `"7"_NS` представляют собой два литерала для этого типа, то умножение этих литералов будет давать значение `"42"_NS`. Такой тип может удовлетворять как требованиям `IntegerLike`, так и `StringLike`, и, таким образом, вызов `print("42"_NS)` будет неоднозначным.

Д.3.1. Поглощение ограничений

Наше обсуждение перегруженных шаблонов функций, отличающихся ограничениями, включает ограничения, которые в общем случае могут быть взаимоисключающими. Например, в нашем примере с `IntegerLike` и `StringLike` мы могли бы представить типы, удовлетворяющие обеим концептам, но мы ожидаем, что такое может быть достаточно редко, так что наши перегруженные шаблоны `print` остаются полезными.

Существуют, однако, множества концептов, которые не являются взаимоисключающими, в которых один концепт “включает” другой. Классическими примерами этого являются категории итераторов стандартной библиотеки: входного итератора, одностороннего итератора, двунаправленного итератора, итератора с произвольным доступом (и в C++17 — непрерывного итератора¹ (*contiguous iterator*)). Предположим, у нас есть определение для `ForwardIterator`:

```
template<typename T>
concept ForwardIterator = ...;
```

Тогда “уточненный” концепт `BidirectionalIterator` может быть определен следующим образом:

```
template<typename T>
concept BidirectionalIterator =
    ForwardIterator<T> &&
    requires(T it)
{
    { --it } -> T&;
};
```

Следовательно, к возможностям, предоставляемым односторонними итераторами, мы добавляем возможность применения префиксного оператора `operator--`.

Рассмотрим теперь алгоритм `std::advance()` (который мы назовем `advanceIter()`), перегруженный для односторонних и двунаправленных итераторов с использованием шаблонов с ограничениями:

```
template<ForwardIterator T, typename D>
void advanceIter(T& it, D n)
{
    assert(n >= 0);

    for (; n != 0; --n)
    {
        ++it;
    }
}

template<BidirectionalIterator T, typename D>
void advanceIter(T& it, D n)
{
    if (n > 0)
    {
        for (; n != 0; --n)
        {
            ++it;
        }
    }
}
```

¹ Непрерывные итераторы представляют собой уточнение итераторов произвольного доступа в C++17. Для них не был добавлен специальный дескриптор `std::contiguous_iterator_tag`, потому что тогда существующие алгоритмы, которые основаны на использовании дескриптора `std::random_access_iterator_tag`, перестали бы корректно работать.

```

    else if (n < 0)
    {
        for (; n != 0; ++n)
        {
            --it;
        }
    }
}

```

При вызове `advanceIter()` с простым односторонним итератором (который не обладает свойствами двунаправленного итератора) будут удовлетворяться только ограничения первого шаблона, так что разрешение перегрузки очень простое: выбирается первый шаблон. Однако двунаправленный итератор будет удовлетворять ограничениям обоих шаблонов. В таких случаях, когда разрешение перегрузки не в состоянии предпочесть одного кандидата другому, оно предпочитает кандидата, ограничения которого *поглощают* (*subsume*) ограничения другого кандидата, в то время как обратное неверно. Точное определение поглощения выходит за рамки этого ознакомительного приложения, но достаточно знать, что если ограничение `C2<Ts...>` определяется как требующее ограничения `C1<Ts...>` и дополнительных ограничений (т.е. использует `&&`), то ограничение `C2` включает `C1`². Очевидно, что в нашем примере `BidirectionalIterator<T>` включает `ForwardIterator<T>`, а потому второй шаблон `advanceIter()` предпочтительнее при вызове с двунаправленным итератором.

Д.3.2. Ограничения и диспетчеризация дескрипторов

Напомним, что в разделе 20.2 мы решили вопрос перегрузки `advanceIter()` с помощью *диспетчеризации дескрипторов*. Этот метод может быть интегрирован в шаблоны с ограничениями довольно элегантным способом. Например, входные и односторонние итераторы не различимы по их синтаксическим интерфейсам. Так что вместо этого мы можем перейти к дескрипторам, определяя один в терминах другого:

```

template<typename T>
concept ForwardIterator =
    InputIterator<T> &&
    requires
    {
        typename std::iterator_traits<T>::iterator_category;
        is_convertible_v<std::iterator_traits<T>::iterator_category,
        std::forward_iterator_tag>;
    };

```

Теперь `ForwardIterator<T>` поглощает `InputIterator<T>`, и мы можем перегружать шаблоны с ограничениями для обеих категорий итераторов.

² Предлагаемая для стандарта спецификация несколько более мощная, чем изложено здесь. Она разбивает ограничения на множества “атомарных компонентов” (включая части выражений требований) и анализирует эти множества, чтобы выяснить, является ли одно множество строгим подмножеством другого.

Д.4. Советы

Хотя над концептами C++ работа продолжалась на протяжении многих лет, и в той или иной форме более десяти лет доступны различные экспериментальные реализации, широкий опыт работы с ними только начинает появляться. Мы надеемся, что будущее издание этой книги будет в состоянии предоставить гораздо более практическое руководство по вопросу разработки библиотек на базе шаблонов с ограничениями. Пока что мы предлагаем вашему вниманию три наблюдения.

Д.4.1. Проверка концептов

Концепты являются логическими предикатами, которые представляют собой корректные константные выражения. Поэтому для заданного концепта C и некоторых типов T₁, T₂, ..., моделирующих концепт, мы можем использовать статические проверки:

```
static_assert(C<T1, T2, ...>, "Модель не работает");
```

Поэтому при разработке концептов рекомендуется также разрабатывать простые типы для проверки концептов этим способом. Сюда входят типы, которые расширяют рамки концептов, отвечая на следующие вопросы.

- Должны ли интерфейсы и/или алгоритмы копировать или перемещать объекты моделируемых типов?
- Какие соглашения приемлемы? Какие необходимы?
- Каков базовый набор операций предполагается в наличии у шаблона? Например, можно ли использовать операцию *= или только * =?

Здесь также может пригодиться понятие *архетипов* (см. раздел 28.3).

Д.4.2. Гранулированность концептов

Когда концепты станут частью языка C++, вполне естественно, возникнет желание создавать “библиотеки концептов”, так же, как мы создаем библиотеки классов и библиотеки шаблонов. Как и в случае с другими библиотеками, естественным будет желание разделения концептов на слои различными способами. Мы кратко обсудили пример категорий итераторов, и не такой уж большой скачок следует сделать, чтобы перейти к “категориям диапазонов”, затем, возможно, к “концептам последовательностей” на их основе, и так далее.

С другой стороны, соблазнительно построить все эти концепты поверх некоторых элементарных синтаксических концептов. Например, можно представить:

```
template<typename T, typename U>
concept Addable =
    requires(T x, U y)
{
    x + y;
}
```

Однако такой подход не рекомендуется, потому что этот концепт не имеет четкого семантического оттенка, и ей будут удовлетворять самые разные типы. Например, она будет удовлетворена, когда T и U представляют собой `std::string`, или когда один тип является указателем, а другой — целочисленным типом, и, конечно же, когда они являются арифметическими типами. Тем не менее во всех этих трех случаях понятие `Addable` означает нечто принципиально иное (соответственно конкатенацию, перемещение итератора и варианты арифметического сложения). Введение такого концепта поэтому является рецептом для библиотек с нечеткими интерфейсами и, вероятно, приводящим к странным неоднозначностям.

Похоже, что концепты лучше всего использовать для моделирования реальных семантических понятий, которые возникают в предметной области задачи. Дисциплинированно поступая таким образом, мы улучшим общий дизайн наших библиотек, так как он будет добавлять последовательность и ясность в интерфейсы, представляемые клиентам. Например, когда стандартная библиотека шаблонов (STL) была включена в стандартную библиотеку C++, это было большим шагом вперед. Она, хотя и не основана на концептах, была спроектирована с учетом этого понятия (например, итераторы и иерархия итераторов).

Д.4.3. Бинарная совместимость

Опытным программистам на C++ известно, что при компиляции некоторых сущностей (в частности, функций и функций-членов) до машинного кода связанные с ними имена включают имя объявленного типа и области видимости. Такое имя, обычно именуемое *декорированным* (*mangled*) именем сущности, используется компоновщиком объектного кода для разрешения ссылок на сущности (например, из других объектных файлов). Так, декорированное имя функции, определенной как

```
namespace X
{
    void f() {}
}
```

в Itanium C++ ABI [43] имеет вид `_ZN1X1fEv` (буквы X и f указывают имя пространства имен и имя функции соответственно).

Декорированные имена в рамках программы не должны приводить к коллизиям. Таким образом, если две функции потенциально могут сосуществовать в программе, они должны иметь собственные декорированные имена. Это, в свою очередь, означает, что в имени функции должны быть закодированы ограничения (поскольку специализации шаблонов, идентичные во всех отношениях, за исключением их ограничений и тел функций, могут находиться в различных единицах трансляции). Рассмотрим две следующие единицы трансляции:

```
#include <iostream>
template<typename T>
concept HasPlus = requires(T x, T y)
{
    x + y;
};
```

```
template<typename T> int f(T p) requires HasPlus<T>
{
    std::cout << "TU1\n";
}
void g();
int main()
{
    f(1);
    g();
}
```

И

```
#include <iostream>
template<typename T>
concept HasMult = requires(T x, T y)
{
    x * y;
};
template<typename T> int f(T p) requires HasMult<T>
{
    std::cout << "TU2\n";
}
template int f(int);
void g()
{
    f(2);
}
```

Эта программа должна выводить

TU1
TU2

т.е. имена двух определений `f()` должны быть декорированы по-разному³.

³ Экспериментальная реализация концептов в GCC 7.1 считается в этом отношении несовершенной.

Библиография

В этом приложении перечислены все информационные ресурсы, упомянутые в данной книге. Немалая часть информации о современных достижениях в области программирования доступна только в электронном виде, так что не удивляйтесь обилию различных веб-узлов, перечисленных наряду с традиционными книгами и статьями. Мы не претендуем на полноту и завершенность представленного здесь списка, но все перечисленное в нем тем или иным образом имеет отношение к тематике нашей книги.

Веб-узлы гораздо более изменчивы, чем книги и статьи, а потому указанные здесь ссылки в будущем могут оказаться недействительными. Поэтому мы приняли решение поддерживать обновляемый список ссылок на узле по адресу (и надеемся, что он будет более стабилен):

<http://www.tplbook.com>

Прежде чем перейти к книгам, статьям и веб-узлам, перечислим интерактивные ресурсы информации, а именно форумы.

Форумы

В первом издании этой книги мы ссылались на группы новостей Usenet (большая коллекция онлайн-форумов до создания Веба) как источник дискуссий о языке программирования C++. С тех пор эти группы почти исчезли, но возникли многие другие онлайн-сообщества, посвященные программированию, в том числе и C++. Мы перечислим здесь некоторые наиболее популярные.

- “Вики” (коллективно редактируемый сайт) *Cppreference* со справочной информацией по C и C++ (на разных языках).

<http://www.cppreference.com>

На русском языке:

<http://ru.cppreference.com>

- *Stackoverflow* – широкое сообщество разработчиков, охватывающее разные темы и языки программирования, в том числе и шаблоны C++.

<https://stackoverflow.com/questions/tagged/c%2b%2b>

<https://stackoverflow.com/questions/tagged/c%2b%2b%20templates>

Имеется сообщество *StackOverflow* на русском языке:

<https://ru.stackoverflow.com/questions/tagged/c%2b%2b>

- Форум *Quora* подобен StackOverflow, но ограничивается техническими обсуждениями.

<https://www.quora.com/topic/C%2B%2B-programming-language>

- *Standard C++ Foundation* – некоммерческая организация, находящаяся в ведении некоторых видных членов Комитета по стандартизации C++

(хотя две группы в ней являются отдельными) для поддержки сообщества программистов на C++. Она помогает финансировать некоторые аспекты заседаний Комитета по стандартизации, а также CppCon — крупную ежегодную конференцию по C++. Сайт содержит каталог онлайн-форумов, которые охватывают различные темы C++.

<https://isocpp.org/forums>

<https://cppcon.org>

- *Association of C and C++ Users* (ACCU) — организация, находящаяся в Великобритании, для “тех, кто заинтересован в развитии и совершенствовании навыков программирования”. Она проводит ежегодные конференции по программированию с особенно сильной направленностью в сторону C++.
<https://www.accu.org>

Книги и веб-сайты

1. David Abrahams and Aleksey Gurtovoy, *C++ Template Metaprogramming – Concepts, Tools, and Techniques from Boost and Beyond*: Addison-Wesley, Boston, MA, 2005.
2. *Adobe Poly Library*, https://stlab.adobe.com/group_poly_related.html.
3. Andrei Alexandrescu, *Modern C++ Design – Generic Programming and Design Patterns Applied*: Addison-Wesley, Boston, MA, 2001.
 Русский перевод: Андрей Александреску, Современное проектирование на C++. М.: ООО “И.Д. Вильямс”, 2002.
4. Andrei Alexandrescu, *Discriminated Unions (parts I, II, III)*: C/C++ Users Journal, April/June/August, 2002.
5. Andrei Alexandrescu, *Generic Programming: Typelists and Applications*: Dr. Dobb's Journal, February, 2002.
6. Matthew H. Austern, *Generic Programming and the STL – Using and Extending the C++ Standard Template Library*: Addison-Wesley, Boston, MA, 1999.
7. John J. Barton and Lee R. Nackman, *Scientific and Engineering C++ – An Introduction with Advanced Techniques and Examples*: Addison-Wesley, Boston, MA, 1994.
8. Jeremy Siek, *The Boost Concept Check Library*: http://www.boost.org/libs/concept_check/concept_check.htm.
9. Todd Veldhuizen, *Blitz++: Object-Oriented Scientific Computing*: <http://blitz.sourceforge.net/>.
10. *The Boost Repository for Free, Peer-Reviewed C++ Libraries*: <http://www.boost.org>.
11. Kevlin Henney, *Boost Any Library*: <http://www.boost.org/libs/any>.
12. Douglas Gregor, *The Boost Fusion Library*: <http://boost.org/libs/function>.
13. Joel de Guzman, Dan Marsden, and Tobias Schwinger, *The Boost Fusion Library*: <http://boost.org/libs/fusion>.
14. Louis Dionne, *The Boost Hana Library for Metaprogramming*: <http://boostorg.github.io/hana>.

15. David Abrahams, Jeremy Siek, Thomas Witt, *Boost Iterator*: <http://www.boost.org/libs/iterator>.
16. Aleksey Gurtovoy and David Abrahams, *Boost MPL*: <http://www.boost.org/libs/mpl>.
17. David Abrahams, *Boost Operators*: <http://www.boost.org/libs/utility/operators.htm>.
18. Jaakko Järvi, *The Boost Tuple Library*: <http://boost.org/libs/tuple>.
19. Steven Watanabe, *Boost.TypeErasures Library*: \a href="http://www.boost.org/doc/libs/1_66_0/libs/type_erasure/index.html">http://www.boost.org/doc/libs/1_66_0/libs/type_erasure/index.html.
20. Fernando Luis Cacciola Carballal, *Boost Optional Library*: <http://www.boost.org/libs/optional>.
21. *Smart Pointer Library*: http://www.boost.org/libs/smart_ptr.
22. *Type Traits Library*: http://www.boost.org/libs/type_traits.
23. Eric Friedman and Itay Maman, *Boost Variant Library*: <http://www.boost.org/libs/variant>.
24. Walter E. Brown, *Introduction to the SI Library of Unit-Based Computation*: <http://lss.fnal.gov/archive/1998/conf/Conf-98-328.pdf>.
25. ISO, *Information Technology—Programming Languages—C++*: ISO/IEC, Document Number 14882-1998, 1998.
26. ISO, *Information Technology—Programming Languages—C++*: ISO/IEC, Document Number 14882-2003, 2003.
27. ISO, *Information Technology—Programming Languages—C++*: ISO/IEC, Document Number 14882-2011, 2011.
28. ISO, *Information Technology—Programming Languages—C++*: ISO/IEC, Document Number 14882-2014, 2014.
29. ISO, *Information Technology—Programming Languages—C++*: ISO/IEC, Document Number 14882-2017, 2017.
30. Fernando Luis Cacciola Carballal and Andrzej Krzemiński, *A Proposal to Add a Utility Class to Represent Optional Objects*: <http://wg21.link/n3527>.
31. Tom Cargill, *Exception Handling: A False Sense of Security*: C++ Report, November-December 1994.
32. James O. Coplien, *Curiously Recurring Template Patterns*: C++ Report, February 1995.
33. *C++ Standard Core Issue 1395*: <http://wg21.link/cwg1395>.
34. Krzysztof Czarnecki and Ulrich W. Eisenecker, *Generative Programming – Methods, Tools, and Applications*: Addison-Wesley, Boston, MA, 2000.
35. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*: Addison-Wesley, Boston, MA, 1995.
36. Gabrial Dos Reis and Mat Marcus, *Proposal to Add Template Aliases to C++*: <http://wg21.link/n1449>.
37. Edison Design Group, *Compiler Front Ends for the OEM Market*: <http://www.edg.com>.

38. Ulrich W. Eisenecker, Frank Blinn, and Krzysztof Czarnecki, *Mixin-Based Programming in C++*: Dr. Dobbs Journal, January, 2001.
39. Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual (ARM)*: Addison-Wesley, Boston, MA, 1990.
40. Douglas Gregor, Jaakko Järvi, and Gary Powell, *Variadic Templates*: <http://wg21.link/n2080>.
41. Kevlin Henney, *Valued Conversions*: C++ Report 12(7), July-August 2000.
42. Jaakko Järvi, Jeremiah Willcock, Howard Hinnant, and Andrew Lumsdaine, *Function Overloading Based on Arbitrary Properties of Types*: C/C++ Users Journal 12 (6), June, 2003.
43. *Itanium C++ ABI*: <http://itanium-cxx-abi.github.io/cxx-abi/>.
44. Nicolai Josuttis, *On launder()*: <https://wg21.link/p0532r0>.
45. Nicolai M. Josuttis, *The C++ Standard Library – A Tutorial and Reference (2nd edition)*: Addison-Wesley, Boston, MA, 2012.
Русский перевод: Николай М. Джосаттис. *Стандартная библиотека C++: справочное руководство. 2-е издание*. — М.: ООО “И.Д. Вильямс”, 2015.
46. Bjorn Karlsson, *The Safe Bool Idiom*: C++ Source, July, 2004.
47. Andrew Koenig and Barbara E. Moo, *Accelerated C++ – Practical Programming by Example*: Addison-Wesley, Boston, MA, 2000.
Русский перевод: Эндрю Кёниг, Барбара Му. *Эффективное программирование на C++*. — М.: ООО “И.Д. Вильямс”, 2002.
48. Jaakko Järvi and Gary Powell, *LL, The Lambda Library*: <http://www.boost.org/libs/lambda>.
49. *C++ Library Issue 181*: <http://wg21.link/lwg181>.
50. Stanley B. Lippman, *Inside the C++ Object Model*: Addison-Wesley, Boston, MA, 1996.
51. Scott Meyers, *Counting Objects In C++*: C/C++ Users Journal, April 1998.
52. Scott Meyers, *Effective C++ – 50 Specific Ways to Improve Your Programs and Design (2nd edition)*: Addison-Wesley, Boston, MA, 1998.
53. Scott Meyers, *More Effective C++ – 35 New Ways to Improve Your Programs and Designs*: Addison-Wesley, Boston, MA, 1996.
54. David A. Moon, *Object-oriented programming with Flavors*: Conference proceedings on Object-oriented programming systems, languages and applications, 1986.
55. Andrew Lumsdaine and Jeremy Siek, *MTL, The Matrix Template Library*: <http://www.osl.iu.edu/research/mtl>.
56. D. R. Musser and C. Wang, *Dynamic Verification of C++ Generic Algorithms*: IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997.
57. Nathan C. Myers, *Traits: A New and Useful Template Technique*: <http://www.cantrip.org/traits.html>.
58. Robert Davies, *NewMat10, A Matrix Library in C++*: http://www.robertnz.net/nm_intro.htm.

59. Leslie Brown (Ed.), *The New Shorter Oxford English Dictionary (4th edition)*: Oxford University Press, Oxford, 1993.
60. *POOMA: A High-Performance C++ Toolkit for Parallel Scientific Computation*: <http://www.nongnu.org/freepooma/>.
61. Yannis Smaragdakis and Don S. Batory, *Mixin-Based Programming in C++*: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering, October, 2000.
62. John Spicer, *Solving the SFINAE Problem for Expressions*: <http://wg21.link/n2634>
63. Alexander Stepanov and Meng Lee, *The Standard Template Library – HP Laboratories Technical Report 95-11(R.1)*: November 14, 1995.
64. Alexander Stepanov, *Notes on Programming*: <http://stepanovpapers.com/notes.pdf>.
65. Bjarne Stroustrup, *The C++ Programming Language* (Special Edition): Addison-Wesley, Boston, MA, 2000.
66. Bjarne Stroustrup, *The Design and Evolution of C++*: Addison-Wesley, Boston, MA, 1994.
67. Bjarne Stroustrup, *Bjarne Stroustrup's C++ Glossary*: <http://www.stroustrup.com/glossary.html>
68. Herb Sutter, *Exceptional C++ – 47 Engineering Puzzles, Programming Problems, and Solutions*: Addison-Wesley, Boston, MA, 2000.
Русский перевод: Герб Саттер. *Решение сложных задач на C++*. — М.: Издательский дом “Вильямс”, 2002.
69. Herb Sutter, *More Exceptional C++ – 40 New Engineering Puzzles, Programming Problems, and Solutions*: Addison-Wesley, Boston, MA, 2001.
Русский перевод: Герб Саттер. *Новые сложные задачи на C++*. — М.: Издательский дом “Вильямс”, 2005.
70. Erwin Unruh, *Original Metaprogram for Prime Number Computation*: <http://www.erwin-unruh.de/primorig.html>.
71. David Vandevoorde and Nicolai M. Josuttis, *C++ Templates: The Complete Guide*: Addison-Wesley, Boston, MA, 2003.
Русский перевод: Дэвид Вандевурд, Николаи М. Джосаттис. *Шаблоны C++: справочник разработчика*. — М.: Издательский дом “Вильямс”, 2003.
72. David Vandevoorde, *C++ Solutions*: Addison-Wesley, Boston, MA, 1998.
73. Todd Veldhuizen, *Using C++ Template Metaprograms*: C++ Report, May 1995.

Глоссарий

Данный глоссарий представляет собой набор самых важных технических терминов, встречающихся в этой книге. Наиболее полный на текущий момент толковый словарь используемых в C++ терминов представлен в [67].

Abstract class	Абстрактный класс	Класс, для которого невозможно создание конкретных объектов (экземпляров). Абстрактные классы могут использоваться для объединения общих свойств различных классов в одном типе либо для определения полиморфного интерфейса. Поскольку абстрактные классы используются в качестве базовых, зачастую употребляется аббревиатура <i>ABC</i> , означающая <i>abstract base class</i> (абстрактный базовый класс)
ADL	Поиск, зависящий от аргументов	Аббревиатура от <i>argument dependent lookup</i> (поиск, зависящий от аргументов). ADL представляет собой процесс поиска имени функции (или оператора) в пространствах имен и классах, тем или иным способом связанных с аргументами вызова функции, в котором упоминается данная функция или оператор. По историческим причинам иногда называется <i>расширенным поиском Кёнига</i> (или просто <i> поиском Кёнига</i> ; данное название применимо в основном к поиску имен операторов)
Alias template	Шаблон-псевдоним	Конструкция, представляющая семейство шаблонов типов. Определяет шаблон, из которого могут быть сгенерированы фактические шаблоны типов путем подстановки параметров шаблонов. Шаблон псевдонима может быть членом класса
Angle bracket hack	Коррекция угловых скобок	Нестандартная способность компилятора воспринимать два последовательных символа > как две закрывающие угловые скобки. Например, таким образом выражение <code>vector<list<int>></code> трактуется как <code>vector<list<int> ></code> . Считается <i>хаком</i> , поскольку не соответствует формальной спецификации (в частности, <i>грамматике</i>) языка C++
Angle brackets	Угловые скобки	Символы < и >, используемые для выделения списка аргументов или параметров шаблонов
ANSI	ANSI	<i>American National Standard Institute</i> (Американский национальный институт стандартов). Некоммерческая организация, координирующая усилия по выработке различных стандартов. См. <i>INCITS</i>
Argument	Аргумент	Значение (в широком смысле), которое замещает <i>параметр</i> . Например, в вызове функции <code>abs (-3)</code> аргументом является значение -3. В некоторых сообществах программистов <i>аргумент</i> называется <i>фактическим параметром</i> , в то время как <i>параметр</i> называется <i>формальным параметром</i> . См. также <i>Template argument</i> (<i>Аргумент шаблона</i>)

Argument-dependent lookup	Поиск, зависящий от аргументов	См. <i>ADL</i> (<i>Поиск, зависящий от аргументов</i>)
Class	Класс	Описание категории объектов. Класс определяет множество характеристик любого объекта. Сюда входят его данные (<i>атрибуты, члены-данные</i>) и операции (<i>методы, функции-члены</i>). В C++ классы — это структуры с членами, которые могут представлять собой функции и являются субъектами ограничения доступа. Классы объявляются с использованием ключевого слова <code>class</code> или <code>struct</code>
Class template	Шаблон класса	Конструкция, представляющая семейство классов. Определяет схему, по которой могут быть сгенерированы действительные классы путем подстановки вместо параметров шаблона конкретных аргументов. Шаблоны классов иногда называют <i>параметризованными классами</i>
Class type	Тип класса, классовый тип	Тип, объявленный в C++ с использованием ключевых слов <code>class</code> , <code>struct</code> или <code>union</code>
Collection class	Класс коллекции	Класс, используемый для управления группой объектов. В C++ классы коллекций называются также <i>контейнерами</i>
Compiler	Компилятор	Программа или компонент библиотеки, который транслирует исходный текст в единице трансляции в объектный код (машинный код с символьными аннотациями, позволяющими компоновщику разрешать ссылки между единицами трансляции)
Complete type	Полный тип	Любой тип, не являющийся <i>неполным</i> : определенный класс, массив полных элементов, размер которого известен, тип перечисления с указанным базовым типом, и любой фундаментальный тип, за исключением <code>void</code> (с необязательными <code>const</code> и/или <code>volatile</code>)
Concept	Концепт	Именованное множество ограничений, которое может быть применено к одному или нескольким параметрам шаблона. См. приложение Д, “Концепты”
Constant-expression	Константное выражение	Выражение, значение которого вычисляется компилятором во время компиляции. Иногда называется также <i>истинной константой</i> , чтобы избежать путаницы с <i>константным выражением</i> . Последнее включает выражения, которые являются константами, но не могут быть вычислены в процессе компиляции
const member function	Константная функция-член	Функция-член, которая может быть вызвана для константного или временного объектов, поскольку обычно она не модифицирует объект <code>*this</code>
Container	Контейнер	См. <i>Collection class</i> (<i>Класс коллекции</i>)
Conversion operator	Оператор преобразования типа	Специальная функция-член, определяющая, каким образом объект может быть неявно (или явно) преобразован в объект другого типа. Объявляется в виде <code>operator type()</code>

CPP file	Файл .cpp	Файл, в котором располагаются <i>определения</i> переменных и невстраиваемых функций. Большая часть выполнимого (в отличие от декларативного) кода обычно находится в файлах, расширение которых, как правило, — .cpp. По историческим причинам расширения могут быть и иными, например, .cc или .cxx. См. также <i>Header file</i> (Заголовочный файл) и <i>Translation unit</i> (Единица трансляции)
Curiously recurring template pattern CRTP	Модель необычного рекуррентного шаблона	Аббревиатура от <i>curiously recurring template pattern</i> (модель необычного рекуррентного шаблона). Обозначает код, в котором класс X порождается из базового класса, для которого X выступает в роли аргумента шаблона
Decay	Низведение	Неявное преобразование массива или функции в указатель. Например, строковый литерал "Hello" имеет тип <code>char const[6]</code> , но во многих контекстах C++ он неявно преобразуется в указатель <code>char const*</code> (который указывает на первый символ строки)
Declaration	Объявление	Конструкция C++, которая вводит (возможно, повторно) имя в область видимости C++. См. <i>Definition (Определение)</i>
Deduction	Вывод	Процесс, который неявно определяет аргументы шаблона из контекста, в котором используется шаблон. Полностью термин звучит как <i>template argument deduction</i> (вывод аргумента шаблона)
Definition	Определение	Объявление, которое раскрывает детали объявляемого объекта или в случае использования переменных приводит к выделению памяти для объявляемого объекта. Для определений типов классов и функций — объявление, включающее тело, заключенное в фигурные скобки. Для объявлений внешних переменных — либо объявление без ключевого слова <code>extern</code> , либо объявление с инициализатором
Dependent base class	Зависимый базовый класс	Базовый класс, который зависит от параметра шаблона. Особое внимание должно уделяться доступу к членам зависимого базового класса. См. <i>Two-phase lookup</i> (Двухфазный поиск)
Dependent name	Зависимое имя	Имя, смысл которого зависит от параметра шаблона. Например, <code>A<T>::x</code> является зависимым именем, если <code>A</code> или <code>T</code> представляют собой параметр шаблона. Имя функции в вызове функции также является зависимым, если любой из аргументов вызова имеет тип, зависящий от параметра шаблона. Например, <code>f</code> в вызове <code>f((T*) 0)</code> является зависимым именем, если <code>T</code> — параметр шаблона. Однако само имя параметра шаблона не рассматривается как зависимое. См. <i>Two-phase lookup</i> (Двухфазный поиск)

Digraph	Диграф	Комбинация из двух последовательных символов, эквивалентная некоторому отдельному символу в программе C++. Назначение диграфов — обеспечить возможность ввода кода на C++ на клавиатурах, на которых отсутствуют некоторые символы. Хотя диграфы используются относительно редко, в тексте может встретиться диграф < : , который образуется, если после открывающей угловой скобки без пробела следует оператор разрешения области видимости :: . C++11 вводит лексический хак для отключения интерпретации диграфов в таких ситуациях
Empty base class optimization EBCO	Оптимизация пустого базового класса	Аббревиатура от <i>empty base class optimization</i> (оптимизация пустого базового класса). Оптимизация, выполняемая большинством современных компиляторов, когда подобъект “пустого” базового класса не занимает место в памяти
Explicit instantiation directive	Директива явного инстанцирования	Конструкция C++, единственная цель которой — создание <i>точки инстанцирования</i>
Explicit specialization	Явная специализация	Конструкция, которая объявляет или дает альтернативное определение шаблона с подстановкой конкретных параметров. Исходный (обобщенный) шаблон называется <i>первичным шаблоном</i> . Если альтернативное определение продолжает зависеть от одного или нескольких параметров шаблона, такое определение называется <i>частичной специализацией</i> , в противном случае это <i>полная специализация</i>
Expression template	Шаблон выражения	Шаблон класса, используемый для представления части выражения. Шаблон сам по себе представляет операцию определенного вида. Параметры шаблона обозначают типы операндов, к которым применима операция
Friend name injection	Внедрение имени друга	Процесс, делающий видимым имя функции, когда единственное ее объявление — в качестве друга
Full specialization	Полная специализация	См. <i>Explicit specialization</i> (Явная специализация)
Function object	Функциональный объект	Объект, который может быть вызван с использованием <i>синтаксиса вызова функции</i> . В C++ это указатели на функции, классы с перегруженным оператором <code>operator()</code> (см. <i>Functor</i> (Функтор)) и классы с функцией преобразования, дающие указатель или ссылку на функцию
Function template	Шаблон функции	Конструкция, которая представляет семейство функций. Шаблон функции определяет модель, из которой генерируются действительные функции путем подстановки вместо параметров шаблонов конкретных аргументов. Заметим, что шаблон функции является шаблоном, но не функцией. Шаблоны функций иногда называют “ <i>параметризованными</i> ” функциями

Functor	Функтор	Объект типа класса с перегруженным оператором <code>operator()</code> , который может быть вызван с использованием синтаксиса вызова функции. Включает тип замыкания лямбда-выражений
glvalue	gl-значение	Категория выражений, которая дает местоположение хранимого значения (обобщенное локализуемое значение). gl-значение может быть l-значением или x-значением. См. раздел Б.2
Header file	Заголовочный файл	Файл, предназначенный быть частью единицы трансляции с помощью директивы <code>#include</code> . Такие файлы часто содержат объявления переменных и функций, которые используются более чем в одной единице трансляции, а также определения типов, встраиваемых функций, шаблонов, констант и макросов. Такие файлы обычно имеют расширения <code>.hpp</code> , <code>.h</code> , <code>.H</code> , <code>.hh</code> или <code>.hxx</code> и иногда называются включаемыми файлами. См. также <i>CPP file</i> (Файл <code>.cpp</code>) и <i>Translation unit</i> (Единица трансляции)
INCITS	INCITS	Сокращение для <i>InterNational Committee for Information Technology Standards</i> , организация по разработке стандартов США, ранее известная как X3. Подкомитет J16 является ведущей силой стандартизации C++. Тесно сотрудничает с международной организацией по стандартизации (ISO)
Include file	Включаемый файл	См. <i>Header file</i> (Заголовочный файл)
Incomplete type	Неполный тип	Класс, который объявлен, но не определен, массив с элементами неполного типа или с неизвестным размером, тип перечисления без указанного базового типа, или <code>void</code> (с необязательными <code>const</code> и/или <code>volatile</code>)
Indirect call	Косвенный вызов	Вызов функции, в котором вызываемая функция не известна до момента реального вызова в процессе работы программы
Initializer	Инициализатор	Конструкция, которая указывает, каким образом должен быть инициализирован именованный объект. Например, в <code>std::complex<float> z1 = 1.0, z2(0.0, 1.0);</code> инициализаторами являются <code>= 1.0</code> и <code>(0.0, 1.0)</code>
Initializer list	Список инициализаторов	Список выражений, разделенных запятыми, помещенный в фигурные скобки и используемый для инициализации объектов и ссылок. Обычно используется для инициализации переменных, но может также, например, инициализировать члены и базовые классы в определениях конструкторов. Инициализация может выполняться непосредственно или через промежуточный объект типа <code>std::initializer_list</code>

Injected class name	Внедренное имя класса	Имя класса, как оно видимо в пределах собственной области видимости. Для шаблонов классов имя шаблона рассматривается в пределах области видимости шаблона как имя класса, если за этим именем не следует список аргументов
Instance	Экземпляр	Термин <i>instance</i> (экземпляр) имеет в C++ два значения. Значение, взятое из объектно-ориентированной терминологии, означает <i>экземпляр класса</i> , т.е. объект, являющийся реализацией класса. Например, в C++ <code>std::cout</code> является экземпляром класса <code>std::ostream</code> . Другое значение (используемое практически повсеместно в книге) этого термина — <i>экземпляр шаблона</i> , т.е. класс, функция или функция-член, получающиеся в результате подстановки вместо всех параметров шаблонов конкретных значений. В этом смысле <i>экземпляр</i> называется также <i>специализацией</i> , хотя последний термин часто ошибочно употребляется вместо <i>явной специализации</i>
Instantiation	Инстанцирование	Процесс создания конкретной сущности (класса, функции, переменной или псевдонима) из шаблона путем подстановки вместо параметров шаблона конкретных значений. Если подстановка выполняется только в объявление, но не в определение, иногда используется термин <i>частичного инстанцирования шаблона</i> . Альтернативный смысл — создание <i>экземпляра</i> (объекта) класса — в данной книге не используется. См. <i>Instance</i> (Экземпляр)
ISO	ISO	Международная организация стандартизации (International Organization for Standardization). Рабочая группа ISO с именем WG21 представляет собой движущую силу по стандартизации и развитию C++
Iterator	Итератор	Объект, который обеспечивает обход последовательности элементов. Зачастую эти элементы принадлежат коллекции. См. <i>Collection class</i> (Класс коллекции)
Linkable entity	Связываемый объект	Невстраиваемая функция или функция-член, глобальная переменная или статический член-данное, включая сгенерированные из шаблонов, и видимые компоновщику
Linker	Компоновщик	Программа или служба операционной системы, объединяющая вместе скомпилированные единицы трансляции и разрешающая ссылки на связываемые объекты между различными единицами трансляции
lvalue	l-значение	Категория выражений, которая дает местоположение хранимого значения, которое не является перемещаемым (т.е. gl-значение, но не x-значение). Типичными примерами являются выражения, описывающие именованные объекты (переменные или члены) и строковые литералы. См. раздел Б.1

Member class template	Шаблон класса-члена	Конструкция, которая представляет семейство членов классов. Это шаблон класса, объявленный внутри другого класса или шаблона класса. Он обладает собственным множеством параметров шаблона (в отличие от члена класса шаблона класса)
Member function template	Шаблон функции-члена	Конструкция, которая представляет семейство функций-членов. Она имеет собственное множество параметров шаблона (в отличие от функции-члена шаблона класса). Эта конструкция очень похожа на шаблон функции, но при подстановке всех ее параметров шаблонов в результате получается функция-член, а не обычная функция. Шаблоны функций-членов не могут быть виртуальными
Member template	Шаблон члена	<i>Member class template</i> (<i>Шаблон класса-члена</i>), <i>Member function template</i> (<i>Шаблон функции-члена</i>) или <i>Static data member template</i> (<i>Шаблон статического члена-данного</i>)
Modern C++	Современный C++	C++ стандарта C++11 или более позднего
Nondependent name	Независимое имя	Имя, которое не зависит от параметра шаблона. См. <i>Dependent name</i> (<i>Зависимое имя</i>) и <i>Two-phase lookup</i> (<i>Двухфазный поиск</i>)
ODR	Правило одного определения	Аббревиатура от <i>one-definition rule</i> (<i>правило одного определения</i>). Это правило накладывает определенные ограничения на <i>определения</i> , находящиеся в программе на C++. См. раздел 10.4 и Приложение А, “Правило одного определения”
Overload resolution	Разрешение перегрузки	Процесс, который выбирает, какая конкретно функция будет вызвана при наличии нескольких кандидатов (обычно имеющих одно и то же имя). См. приложение В, “Разрешение перегрузки”
Parameter	Параметр	Заместитель, вместо которого в некой точке будет подставлено фактическое “значение” (<i>аргумент</i>). В случае параметров макросов и параметров шаблонов данная подстановка выполняется во время компиляции. В случае параметров вызова функции эта замена осуществляется в процессе выполнения программы. В некоторых сообществах программистов <i>параметр</i> называется <i>формальным параметром</i> , в то время как <i>аргумент</i> называется <i>фактическим параметром</i> . См. также <i>Argument</i> (<i>Аргумент</i>)
Parametrized class	Параметризованный класс	Шаблон класса или класс, вложенный в шаблон класса. Оба они называются <i>параметризованными</i> , поскольку не могут однозначно соответствовать некоторому единственному классу до тех пор, пока не будут определены аргументы шаблона

Parametrized function	Параметризованная функция	Шаблон функции или функции-члена либо функция-член в шаблоне класса. Все они называются <i>параметризованными</i> , поскольку не могут однозначно соответствовать некоторой единственной функции (или функции-члену) до тех пор, пока не будут определены аргументы шаблона
Partial specialization	Частичная специализация	Конструкция, которая объявляет или дает альтернативное определение для некоторых <i>подстановок</i> шаблона. Исходный (обобщенный) шаблон называется <i>первичным шаблоном</i> . Альтернативное определение продолжает зависеть от параметров шаблона. В настоящее время эта конструкция применяется только для шаблонов классов. См. также <i>Explicit specialization</i> (<i>Явная специализация</i>)
POD	Обычные данные	Аббревиатура от <i>plain old data (type)</i> (обычный старый тип данных). Типы POD представляют собой типы, которые могут быть определены без использования возможностей C++ (таких, как виртуальные функции-члены, модификаторы доступа и т.п.). Например, обычная структура языка C является POD
POI	Точка инстанцирования	Аббревиатура от <i>point of instantiation</i> (точка инстанцирования). POI — это место в исходном коде, в котором шаблон (или член шаблона) концептуально развертывается путем подстановки аргументов шаблона вместо параметров шаблона. На практике такое развертывание не обязательно выполняться в каждой точке инстанцирования. См. также <i>Explicit instantiation directive</i> (<i>Директива явного инстанцирования</i>)
Policy class	Класс стратегии	Класс или шаблон класса, члены которого описывают настраиваемое поведение обобщенного компонента. Стратегии, как правило, передаются в качестве аргументов шаблонов. Например, шаблон сортировки может иметь стратегию упорядочения. Классы стратегий называют также <i>шаблонами стратегий</i> или просто <i>стратегиями</i> . См. также <i>Traits template</i> (<i>Шаблон свойств</i>)
Polymorphism	Полиморфизм	Способность операции (идентифицируемой ее именем) быть примененной к объектам различных типов. В C++ традиционная объектно-ориентированная концепция полиморфизма (именуемая также <i>полиморфизмом времени выполнения</i> или <i>динамическим полиморфизмом</i>) достигается посредством виртуальных функций, переопределенных в производных классах. Кроме того, шаблоны C++ обеспечивают так называемый <i>статический полиморфизм</i>

Precompiled header	Предварительно скомпилированный заголовочный файл	Исходный код в обработанном виде, быстро загружаемом компилятором. Исходный код предварительно скомпилированного заголовочного файла должен быть первой частью <i>единицы трансляции</i> (другими словами, он не может начинаться где-то в середине единицы трансляции). Зачастую предварительно скомпилированный заголовочный файл соответствует нескольким заголовочным файлам. Использование предварительно скомпилированных заголовочных файлов может существенно сократить время, необходимое для построения больших приложений, разработанных на C++
Primary template	Первичный шаблон	Шаблон, не являющийся <i>частичной специализацией</i>
prvalue	рг-значение	Категория выражений, выполняющих инициализацию. Можно рассматривать их как предназначенные для чисто математических значений, таких как 1 или true, и временных переменных (в частности, значений, возвращаемых из функций по значению). Любое г-значение до C++11 является рг-значение в C++11.
Qualified name	Полное (квалифицированное) имя	Имя, содержащее квалификатор области видимости ::
Reference counting	Подсчет ссылок	Стратегия управления ресурсами, которая отслеживает количество объектов, ссылающихся на некоторый ресурс. Когда эта величина снижается до 0, ресурс может быть освобожден
rvalue	г-значение	Категория выражений, не являющихся l-значениями. Г-значение может быть рг-значением (таким как временные объекты) или х-значением (например, l-значение после применения std::move()). То, что называлось г-значениями до C++11, в C++11 называется рг-значение. См. раздел Б.2
SFINAE	SFINAE	Аббревиатура для <i>substitution failure is not an error</i> — ошибка подстановки ошибкой не является. Механизм, позволяющий молча отбрасывать шаблоны вместо вывода сообщений об ошибках компиляции при некорректной попытке подстановки аргументов шаблона. При этом другие шаблоны в множестве перегрузки получают шанс быть выбранными, если подстановка в них оказывается успешной
Source file	Исходный файл	<i>Header file</i> (Заголовочный файл) или <i>CPP file</i> (Файл .cpp)
Specialization	Специализация	Результат подстановки вместо параметров шаблона фактических значений. Специализация может быть создана путем <i>инстанцирования</i> или <i>явной специализации</i> . Данный термин иногда ошибочно путают с <i>явной специализацией</i> . См. также <i>Instance</i> (Экземпляр)

Static data member template	Шаблон статического члена-данного	Шаблон переменной, являющейся членом класса или шаблона класса
Substitution	Подстановка	Процесс замены параметров шаблона в шаблонных сущностях фактическими типами, значениями или шаблонами. Степень замещения зависит от контекста. При разрешении перегрузки, например, выполняется только минимальное количество замещений, чтобы установить тип функции-кандидата (если замена ведет к недопустимым конструкциям, применяются правила SFINAE). См. также <i>Instantiation</i> (<i>Инстанцирование</i>)
Template	Шаблон	Конструкция, которая представляет семейство типов, функций, функций-членов или переменных. Она определяет модель, по которой путем подстановки вместо параметров шаблона конкретных аргументов могут быть сгенерированы действительные типы, функции, функции-члены или переменные. В этой книге данный термин не включает классы, функции, статические данные-члены и псевдонимы типов, которые параметризованы постольку, поскольку являются членами шаблона класса. См. <i>Class template</i> (<i>Шаблон класса</i>), <i>Parametrized class</i> (<i>Параметризованный класс</i>), <i>Function template</i> (<i>Шаблон функции</i>) и <i>Parametrized function</i> (<i>Параметризованная функция</i>)
Template argument	Аргумент шаблона	“Значение”, подставляемое вместо <i>параметра шаблона</i> . Это значение обычно представляет собой тип, хотя корректными аргументами могут быть также некоторые константные значения и шаблоны
Template argument deduction	Вывод аргумента шаблона	См. <i>Deduction</i> (<i>Вывод</i>)
Template-id	Идентификатор шаблона	Комбинация имени шаблона, за которым следуют его <i>аргументы в угловых скобках</i> (например, <code>std::list<int></code>)
Template parameter	Параметр шаблона	Обобщенный заместитель в шаблоне. Наиболее общий вид параметров шаблонов — <i>параметры типа</i> , которые представляют различные типы. <i>Параметры, не являющиеся типами</i> , представляют собой константные значения некоторого типа, а <i>шаблонные параметры шаблонов</i> являются шаблонами типов
Template entity	Шаблонная сущность	Шаблоны или сущности, определенные или созданные в шаблоне. Последние включают такие вещи, как обычные функции-члены шаблонов класса или типы замыканий лямбда-выражений, находящихся в шаблоне

Traits template	Шаблон свойств	Шаблон, члены которого описывают характеристики (свойства) аргументов шаблона. Обычно цель шаблонов свойств — помочь избежать слишком большого количества параметров шаблона. См. также <i>Policy class</i> (<i>Класс стратегии</i>)
Translation unit	Единица трансляции	Исходный .C-файл со всеми заголовочными файлами и заголовками стандартной библиотеки, включенными с помощью директив #include, исключая текст, который устранен из компиляции с помощью директив препроцессора типа #if. Для простоты можно считать единицу трансляции результатом обработки .C-файла препроцессором. См. <i>CPP file</i> (<i>Файл .cpp</i>) и <i>Header file</i> (<i>Заголовочный файл</i>)
True constant	Истинная константа	Выражение, значение которого может быть вычислено компилятором во время компиляции. См. <i>Constant-expression</i> (<i>Константное выражение</i>)
Tuple	Кортеж	Обобщение концепции структуры языка C, в котором обращение к членам может осуществляться по их номерам
Two-phase lookup	Двухфазный поиск	Механизм поиска имен, используемый для поиска имен в шаблонах. Две фазы представляют собой, во-первых, фазу, когда компилятор впервые встречается с определением шаблона, и, во-вторых, инстанцирование шаблона с конкретными аргументами шаблона. Поиск <i>независимых имен</i> выполняется только во время первой фазы, но при этом не рассматриваются <i>независимые базовые классы</i> . <i>Зависимые имена</i> с квалификатором области видимости (:) ищутся только во второй фазе. Поиск зависимых имен без квалификатора области видимости может проводиться в обеих фазах, но поиск, зависящий от аргументов, выполняется только во второй фазе
Type alias	Псевдоним типа	Альтернативное имя типа, вводимое конструкцией <code>typedef</code> , объявлением псевдонима или инстанцированием шаблона-псевдонима
Universal reference	Универсальная ссылка	Одно из двух названий для ссылки на <i>r</i> -значение вида <code>T&</code> , где <code>T</code> — выводимый параметр шаблона. Для нее применяются особые правила, отличные от правил для обычной ссылки на <i>r</i> -значение (см. раздел 6.1). Термин был придуман Скоттом Мейерсом (Scott Meyers) как универсальный термин для <i>ссылки на l-значение</i> и <i>ссылки на r-значение</i> . Из-за слишком большой универсальности стандарт C++17 ввел вместо этого термин “ <i>передаваемая ссылка</i> ”
User-defined conversion	Пользовательское преобразование типов	Преобразование типов, определенное программистом. Может иметь вид <i>конструктора</i> (который может быть вызван с одним аргументом, или <i>оператора преобразования типа</i>). Если конструктор не объявлен с ключевым словом <code>explicit</code> , преобразование типов может выполняться неявно

Value category	Категория значения	Классификация выражений. Традиционные категории значений (l-значения и r-значения) были унаследованы от С. В С++11 введены альтернативные категории: gl-значения (обобщенные l-значения), вычисление которых идентифицирует хранимые объекты, и rg-значения (чистые r-значения), вычисление которых инициализирует объекты. Есть дополнительное подразделение gl-значений на l-значения (локализуемые) и x-значения (завершающиеся). Кроме того, в С++11 r-значения служат в качестве общей категории для x-значений и rg-значений (до С++11 r-значение являлось тем же, чем в С++11 является rg-значение). Подробности см. в приложении Б, "Категории значений"
Variable template	Шаблон переменной	Конструкция, представляющая семейство переменных или статических членов-данных. Определяет схему, по которой путем подстановки конкретных сущностей вместо параметров шаблона генерируются фактические переменные или статические члены-данные
Whitespace	Пробельный символ	В С++ это символ, который служит разделителем лексем (идентификаторов, литералов, символов и т.п.) в исходном коде. Помимо традиционных символов пробела, начала новой строки и табуляции, сюда входят и комментарии. Другие символы (например, символ подачи страницы) также иногда являются корректными пробельными символами
xvalue	x-значение	Категория выражений, которая дает местоположение сохраненного объекта, который можно рассматривать как более не нужный. Типичным примером является l-значение после применения <code>std::move()</code> . См. раздел Б.2

Предметный указатель

Символы

.template, 114

A

add_const, 793

add_cv, 793

add_lvalue_reference, 794

add_pointer, 795

addressof, 207; 802

add_rvalue_reference, 794

add_volatile, 793

ADL, 264; 821

aligned_storage, 798

aligned_union, 798

alignment_of, 772; 778

auto, 46; 85; 144; 155; 346; 349

auto&&, 347

B

bool_constant, 763

C

C++03, 32; 330

C++11, 32; 62; 259; 274; 348; 402; 471

C++14, 32; 114; 203; 232; 349; 354

C++17, 32; 92; 117; 141; 143; 200; 223;

228; 231; 356; 357; 360; 368; 402; 471;
534

C++98, 32; 259

char_traits, 522

class, 38; 227

common_type, 47; 686; 797

conditional, 213; 503; 797

conjunction, 801

constexpr, 55; 161

D

decay, 46; 468; 795

decay_t, 48

decltype, 169; 352; 743

decltype(auto), 203; 354; 356

declval, 207; 473; 801

disjunction, 801

E

enable_if, 132; 168; 529; 548; 797

применение, 133; 134

enable_if_t, 133

extent, 772; 779

extern, 192

F

false_type, 471

forward, 333

friend, 254

function, 583

H

has_unique_object_representations,

772; 778

has_virtual_destructor, 772; 778

I

if

constexpr, 170; 317; 534

времени компиляции, 170; 316; 534

initializer_list, 327; 755

inline, 55; 176

integer_sequence, 648

integral_constant, 762

invoke, 200; 202

invoke_result, 772; 780

is_abstract, 772; 777

is_aggregate, 772; 774

is_arithmetic, 770

is_array, 514; 766; 767

is_assignable, 786

is_base_of, 790
 is_class, 517; 766; 769
 is_compound, 770; 771
 is_const, 771; 773
 is_constructible, 136; 783
 is_convertible, 134; 491; 791
 is_copy_assignable, 787
 is_copy_constructible, 784
 is_default_constructible, 784
 is_destructible, 788
 is_empty, 772; 777
 is_enum, 766; 769
 is_final, 772; 777
 is_floating_point, 766; 767
 is_function, 517; 766; 769
 is_fundamental, 770
 is_integral, 766; 767
 is_invocable, 772; 779
 is_invocable_r, 772; 780
 is_literal_type, 776
 is_lvalue_reference, 513; 766; 768
 is_member_function_pointer, 514;
 766; 768
 is_member_object_pointer, 514; 766; 768
 is_member_pointer, 514; 770
 is_move_assignable, 787
 is_move_constructible, 213; 506; 785
 is_nothrow_assignable, 786
 is_nothrow_constructible, 783
 is_nothrow_copy_assignable, 787
 is_nothrow_copy_constructible, 784
 is_nothrow_default_constructible, 784
 is_nothrow_destructible, 788
 is_nothrow_invocable, 772; 779
 is_nothrow_invocable_r, 772; 780
 is_nothrow_move_assignable, 787
 is_nothrow_move_constructible, 785
 is_nothrow_swappable, 789
 is_nothrow_swappable_with, 788
 is_null_pointer, 766; 768
 is_object, 770; 771
 is_pod, 772; 776
 is_pointer, 512; 766; 767
 is_polymorphic, 772; 777

is_reference, 770
 is_rvalue_reference, 513; 766; 768
 is_same, 790
 is_scalar, 770; 771
 is_signed, 771; 773
 is_standard_layout, 772; 775
 is_swappable, 789
 is_swappable_with, 788
 is_trivial, 772; 774
 is_trivially_assignable, 786
 is_trivially_constructible, 783
 is_trivially_copyable, 772; 775
 is_trivially_copy_assignable, 787
 is_trivially_copy_constructible, 520; 784
 is_trivially_default_constructible, 784
 is_trivially_destructible, 788
 is_trivially_move_assignable, 787
 is_trivially_move_constructible, 520; 785
 is_union, 517; 766; 769
 is_unsigned, 771; 773
 is_void, 766
 is_volatile, 771; 773
 iterator_traits, 523

L

launder, 679

M

make_pair, 157
 make_signed, 793
 make_unsigned, 793

N

negation, 801
 noexcept, 343; 503

R

rank, 772; 778
 reference_wrapper, 149
 remove_all_extents, 795
 remove_const, 792
 remove_cv, 792
 remove_extent, 795
 remove_pointer, 794

`remove_reference`, 154; 793
`remove_volatile`, 792

S

SFINAE, 133; 165; 336; 474
 лямбда-выражение, 480
 перегрузка функций, 474
 частичные специализации, 479
`sizeof`, 283; 459
`sizeof...`, 91; 244
`static_assert`, 63; 718

T

`template`, 279
`template<>`, 190
`true_type`, 471
`tuple`, 598
`typedef`, 72
`typeid`, 174
`typename`, 38; 101; 227; 410

U

UCN, 262
`underlying_type`, 772; 779
`using`, 72; 99; 279

V

`variant`, 665
`void_t`, 480

A

Алгоритм, 438
 специализация, 526
Аргумент
 вызова
 по умолчанию, 222; 233
 пакет, 244
 раскрытие, 244; 632
 вложенное, 250
 шаблона, 234
 именованный, 414
 нетиповой, 237
 по умолчанию, 47; 70
 типовой, 237

функции
 явный, 343

Архетип, 719

B

Безопасность типов, 434
 Безымянное объединение, 296
 Библиотека, 310
 обобщенная, 214
 стандартная, 318; 547
 шаблонов, 438

B

Висячая ссылка, 154
 Встраиваемая переменная, 450
 Встраивание, 145
 Вывод аргументов, 44; 74; 321
`auto`, 357
`decltype(auto)`, 354
 вариативные правила, 98
 возвращаемый тип, 349
 выводимый контекст, 324
 лямбда-выражения, 364
 ограничения, 339
 особые ситуации, 325
 пакет параметров, 327
 передаваемая ссылка, 331
 правила, 76
 вариативные, 98
 спецификации исключений, 342
 список инициализации, 326
 структурированное связывание, 360
 шаблон класса, 368
 шаблон псевдонима, 367
 Выводимый контекст, 324
Выражение
 вариативное, 96
 зависящее от значения, 283
 зависящее от инстанцирования, 283
 зависящее от типа, 282
 константное, 195; 240; 608
 параметризации, 219
 свертки, 92; 247; 252
 эквивалентное, 242
 функционально, 243

Д

Двухфазный поиск, 41; 300
 Дискриминатор, 666
 Диспетчеризация дескрипторов, 471; 527
 Дружба, 254

Е

Единица трансляции, 193; 304; 727; 831

З

Завершающий возвращаемый тип, 45; 169; 335; 349
 Заместитель типа, 85
 Замыкание, 200; 365

И

Идеальная передача. См. Прямая передача
 Идентификатор, 262
 квалифицированный, 262
 неквалифицированный, 262
 шаблона, 262
 Имя, 263
 внедренное класса, 374
 друга
 внесение, 268; 560
 инъекция, 268
 зависимое, 261; 263
 квалифицированное, 261; 263
 независимое, 263
 неквалифицированное, 263
 поиск, 263
 двухфазный, 300
 зависящий от аргументов, 264
 обычный, 264
 полное, 261
 Инициализация
 значением, 103; 444
 нулем, 102

Инстанцирование, 40; 175; 190; 293
 if constexpr, 317
 автоматическое, 293
 жадное, 308

итеративное, 311
 модель, 300
 неполное, 190
 неявное, 293
 отложенное, 295
 поверхностное, 717
 по запросу, 309
 полное, 296
 по требованию, 293
 рекурсивное, 603
 ручное, 313
 текущее, 270
 частичное, 296
 явное, 312
 директива, 312
 объявление, 315

Инъекция имени друга, 268
 Исключение, 503
 Использование, 730
 Итератор, 438; 458
 адаптер, 568
 дескриптор, 547

К

Категория значения, 335; 737
 Класс
 базовый
 абстрактный, 427
 вариативный, 100
 свойств, 462
 связанный, 266
 стратегии, 453; 456
 шаблонный, 189
 Константное if, 317
 Контейнер, 36; 438; 822
 Контекст
 выводимый, 324
 непосредственный, 337
 Концепт, 63; 138; 419; 435; 440; 536; 717; 803
 Кортеж, 598; 637
 операции, 641
 распаковка, 653

Л

Лямбда-выражение, 200
обобщенное, 114; 364
SFINAE, 481

М

Макросы свойств, 480

Массив, 361

копирование, 362

Метапрограммирование, 159; 593

гетерогенное, 599

гибридное, 596

значений, 593

рефлексивное, 420

типов, 595

Метод Бартона–Нэкмана, 559

Миксин, 247; 570; 571

странный, 572

Модель

включения, 173; 175; 305

разделения, 187; 305

Модуль, 176; 188; 423

О

Обобщенная связка, 440

Обобщенное

проектирование, 441

Обратный вызов, 197

Объединение

контролируемое, 665

Объект

времени компиляции, 195

вызываемый, 198

замыкания, 365

функциональный, 198; 365

Объявление, 190; 191; 728

класса, 728

параметризованное, 219

перечисления, 728

повторное, 399

предварительное, 294

пространства имен, 728

псевдонима, 72

явного инстанцирования, 305; 315

Определение, 190; 191; 728

явного инстанцирования, 306; 313

Оптимизация

пустого базового класса, 551; 654

Оракул, 726

Отложенные вычисления, 212

П

Пакет параметров, 89; 248

функции, 248

Параметр, 827

вызыва, 43; 108

типа, 37

фактический, 194

формальный, 194

шаблона

шаблонный, 118

Передача

аргументов

общие рекомендации, 155

по значению, 54; 142

по ссылке, 54; 144

прямая, 125

Переменная

встраиваемая, 450

Перемещение, 125

Подстановка, 190; 194

Полиморфизм, 427; 581; 828

динамический, 427; 433

инвазивный, 433

интрузивный, 433

неограниченный, 433

ограниченный, 433

связанный, 433

статический, 427; 430; 433

Правила вывода, 76; 219; 369

explicit, 375

вариативные, 98

внедренное имя класса, 374

неявные, 371

передаваемая ссылка, 374

Правило

двухфазного поиска, 287

доминирования, 577
одного определения, 187; 193; 727
пропуска копирования, 402
свертки ссылок, 330
упорядочения перегруженных шаблонов, 386
Предикат, 762
Принцип максимального поглощения, 273
Программирование обобщенное, 437
объектно-ориентированное, 437
с виртуальными функциями, 438
с шаблонами, 438
Пропуск копирования, 402
Прямая передача, 95; 125; 333
временных значений, 208
Псевдоним объявление, 72
типа, 71

P

Разбухание кода, 404
Разрешение перегрузки, 745
Раскрытие пакета место, 246
схема, 245
Рефлексия, 420; 523; 603

C

Свертка, 252
ссылок, 330
Свойства итераторов, 458
типов, 205; 455
преобразующие, 462
фиксированные, 452; 455
Свойства типов предикаты, 468
результата, 471
Связанное пространство имен, 266
Связываемый объект, 193
Связывание внутреннее, 730
структурированное, 360

Семантика перемещения, 125
Сигнатура, 383
Специализация, 190; 293 алгоритма, 526
встраиваемая, 399
неизвестная, 270
полная, 394
объявление, 396; 399
шаблона класса, 394
переменной, 400
функции, 398 частичная, 67; 163; 191; 227; 394
SFINAE, 405
шаблона класса, 403
переменной, 407
функции, 414
шаблона, 66; 190 явная, 191; 394
Список индексов, 648
инициализации, 326
типов, 515
Ссылка висячая, 154
константная, 144
неконстантная, 146
передаваемая, 127; 331; 374; 831
свертка, 330
универсальная, 127; 831
Стирание типа, 583; 588
Странно рекурсивный шаблон проектирования, 557; 572; 668
Стратегия, 453; 455
класс, 453
Структурированное связывание, 360
Суффикс _t, 73
_v, 117

T

Тип агрегатный, 774
безопасность, 434

вызываемый, 198
 выражения, 209
 зависимый, 270; 276
 задаваемый правилом, 369
 заместитель, 203; 354
 замыкания, 365
 класса, 189
 литеральный, 412
 классификация, 508
 литеральный, 776
 лямбда-выражения, 517
 неполный, 192
 низведение, 143; 322; 466
 низведенный, 46
 объявленный, 352
 полный, 192
 пользовательское
 преобразование, 238
 псевдоним, 71; 228
 свойства. См. Свойства типов
 список, 613
 ссылочный, 743
 стирание, 583; 588
 фундаментальный, 509
 функционального объекта, 198

Точка
 инстанцирования, 300; 301; 312;
 733; 828

Точка инстанцирования, 824

Трассировщик, 722

Требование, 536

У

Универсальная ссылка, 127

Ф

Файл
 CPP, 173; 187
 заголовочный, 173; 175; 187; 423
 предкомпилированный, 177

Фасад, 563

Функтор, 198; 759; 825

Функция
 constexpr, 161
 встраиваемая, 309
 дружественная, 64
 значения, 459
 сигнатура, 383
 суррогат, 759
 типа, 459

III

Шаблоид, 223

Шаблон, 36; 830
 аргумент, 194; 234
 нетиповой, 237
 пакет, 244
 раскрытие, 244; 632
 по умолчанию, 47; 70; 233
 типовoy, 237

вариативный, 89; 232; 243

выражения, 693

двуэтапная трансляция, 40

дружественный, 258

идентификатор, 194; 235

инстанцирование. См.
 Инстанцирование
 класса, 57; 189
 вариативный, 97
 дружественный класс, 254
 инстанцирование, 61
 использование, 61
 частичное, 62
 наследование, 285
 объявление, 58
 реализация функций-членов, 59
 специализация, 66
 частичная, 67

Класса, 822

метапрограммирование. См.
 Метапрограммирование

необычный рекуррентный, 434

объединения, 222

оператора литерала, 329

отладка, 715

- параметр, 37; 194; 227
нетиповый, 79
ограничения, 83
пакет, 89; 227; 231
шаблонный, 118; 229
первичный, 226
переменной, 115; 190
и свойства типов, 507
псевдонима, 72; 367; 473
для типов-членов, 73
свойств, 446; 455
связывание, 224
синтаксический анализ, 271
- специализация, 37; 190; 235
вариативный, 391
возврат значения, 153
дружественной, 64
перегрузка, 49; 382
передача аргументов, 54
члена, 190
члена класса, 108
специализация, 112
экземпляр, 40
экспортирование, 187
Шаблонная сущность, 223; 258