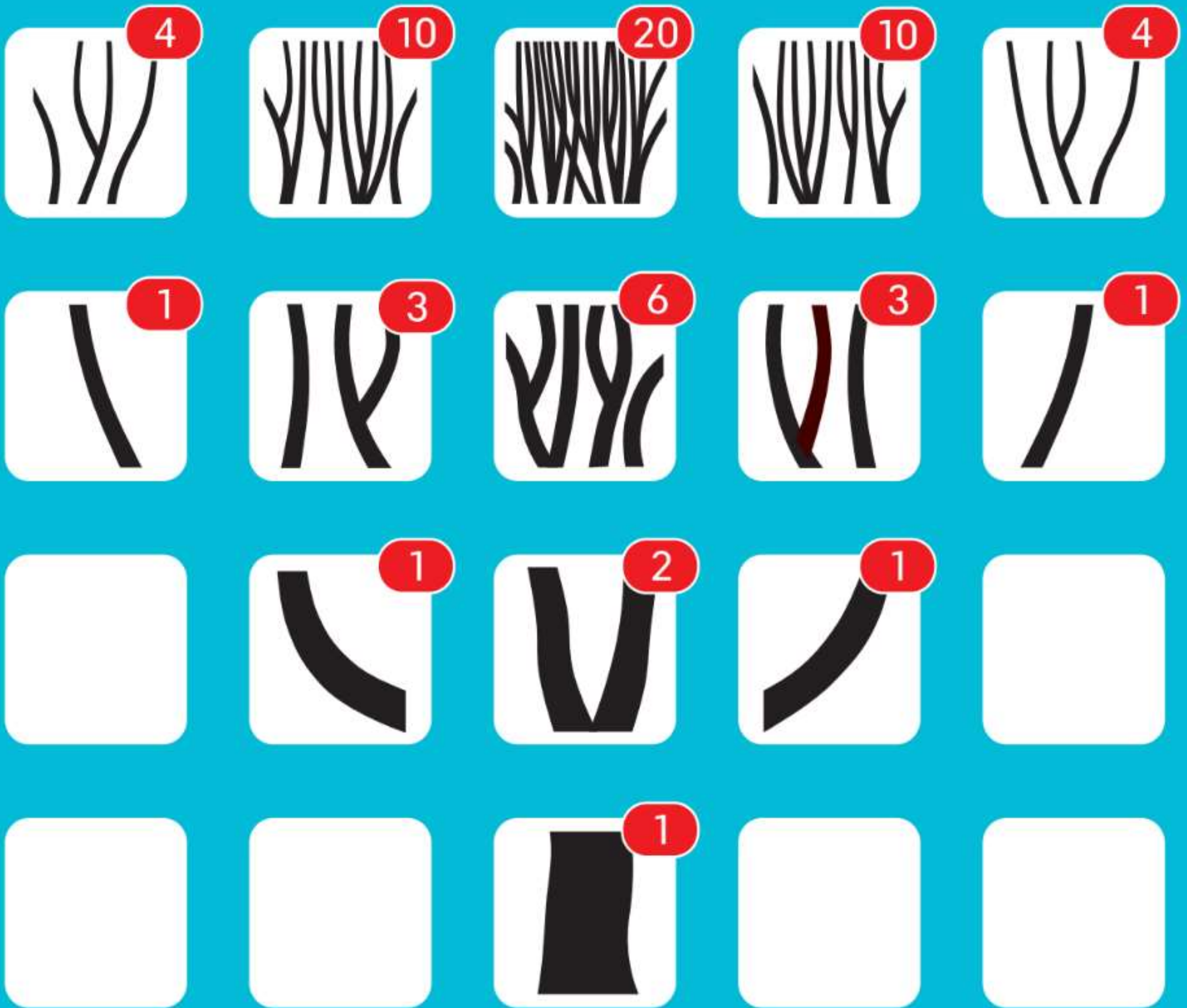


MARCO CANTÙ

OBJECT PASCAL HANDBOOK

DELPHI 10.4 SYDNEY EDITION

-

Руководство по Object Pascal для Delphi 10.4 Sydney

Полное руководство по языку программирования Object Pascal для разработчиков Delphi.

Оригинальное издание: Piacenza (Italy), Июль 2015

Delphi 10.4 Edition: Piacenza (Italy), Ноябрь 2020

Автор: Марко Канту

Издатель: Марко Канту

Редактор первого издания: Питер В. А. Вуд

Дизайнер обложки: Фабрицио Скьяви (www.fsd.it)

Авторское право 1995-2020 годов Марко Канту, Пьяченца, Италия. Мировые права защищены.

В данной публикации автор создал примеры кода специально для свободного использования его читателями. Исходный код этой книги является защищенным Freeware и распространяется через проект GitHub, указанный в книге и на сайте книги. Авторские права не позволяют вам переиздавать код в печатном или электронном виде без разрешения. Читателям даётся ограниченное разрешение использовать этот код в своих приложениях, при условии, что сам код не распространяется, не продаётся и не используется в коммерческих целях в качестве самостоятельного продукта.

Кроме этого особого исключения, касающегося исходного кода, никакая часть данной публикации не может храниться в поисковой системе, передаваться или воспроизводиться каким-либо способом в оригинале или на переведенном языке, включая, но не ограничиваясь, ксерокопирование, фотографию, магнитную или иную запись, без предварительного согласия и письменного разрешения издателя.

Delphi является торговой маркой Embarcadero Technologies. Другие товарные знаки являются собственностью соответствующих владельцев, на которые имеются ссылки в тексте. Несмотря на то, что автор и издатель приложили все усилия при подготовки этой книги, они не делают никаких заявлений и не дают никаких гарантий в отношении полноты или точности содержания здесь и не принимают на себя никакой ответственности любого рода за, включая, но не ограничиваясь, производительность, коммерческую пригодность, пригодность для любых конкретных целей, или любые убытки или ущерб любого рода, вызванные или предположительно вызванные прямо или косвенно из этой книги.

Руководство по Object Pascal для Delphi 10.4 Сидней

ISBN-10: будет назначен

ISBN-13: будет назначен

Электронное издание этой книги было лицензировано Embarcadero Technologies Inc. Она также продается непосредственно автором. *Не распространяйте PDF-версию этой книги без разрешения.* Печатное издание публикуется через Kindle Direct Publishing и продается в нескольких торговых точках.

-

Дополнительную информацию можно получить на сайте ***http://www.marcocantu.com/objectpascal***.

begin

Моей семье, Раффаэлле, Бенни и Джакопо, со всей моей любовью и огромной благодарностью за все, что вы делаете, чтобы превзойти моиожидания.

Мощность и простота, выразительность и читабельность, отлично подходят как для обучения, так и для профессионального развития — вот некоторые из черт сегодняшнего Object Pascal, языка с длинной историей, живым настоящим и блестящим будущим.

Object Pascal – это многогранный язык. Он сочетает в себе мощь объектно-ориентированного программирования, расширенную поддержку программирования с generic и динамические конструкции, такие как атрибуты, но не исключает поддержки более традиционного стиля процедурного программирования. Инструмент для всех, с компиляторами и инструментами разработки, охватывающими мобильную эру. Язык, готовый к будущему, но с прочными корнями в прошлом.

Для чего нужен язык Object Pascal? От написания настольных приложений до клиент-серверных приложений, от массивных модулей веб-серверов до многоуровневого программного обеспечения, от офисной автоматизации до приложений для новейших телефонов и планшетов, от систем промышленной автоматизации до виртуальных телефонных сетей в Интернете... это не то, для чего можно было бы использовать язык, а то, для чего он используется в *настоящее время*, в реальном мире.

Ядро языка Object Pascal в том виде, в котором мы используем сегодня, берет свое начало от его определения в 1995 году, потрясающем году для языков программирования, учитывая, что именно в этот год были изобретены Java и JavaScript. Хотя корни языка восходят к его предку Паскалю, его эволюция не остановилась в 1995 году, а улучшения в ядре языка продолжаются и по сей день, с компиляторами для настольных и мобильных устройств, которые созданы Embarcadero Technologies и входят в состав Delphi и RAD Studio.

Книга о современном языке

Учитывая меняющуюся роль языка, его расширение на протяжении многих лет, а также тот факт, что он сейчас привлекает новых разработчиков, я посчитал важным написать книгу, которая предлагает полный охват языка Object Pascal в том виде, в каком он есть сегодня. Цель состоит в том, чтобы предложить руководство по языку для новых разработчиков, для разработчиков, владеющих другими схожими языками, а также для ветеранов различных диалектов Паскаля, которые хотят узнать больше о последних изменениях в языке.

Новичкам, конечно, нужны некоторые из основ, но, учитывая изменения, которые появились повсеместно, даже старожилы найдут что-то новое в первых главах.

Не считая маленького Приложения, кратко освещающего историю языка Object Pascal, эта книга была написана для того, чтобы представить язык, каким он является сегодня. Большая часть основных особенностей языка не претерпела существенных изменений со времен ранних версий Delphi, первой реализации современного Object Pascal в 1995 году.

Я буду напоминать на протяжении всей книги, что язык был далеко не застойным в течение всех этих лет, он развивался довольно быстрыми темпами. В других книгах, которые я писал в прошлом, я придерживался более *хронологического* подхода, охватывая сначала классический Паскаль, а затем его расширения, более или менее, как они появились с течением времени. В этой книге, однако, идея заключается в том, чтобы использовать более *логичный* подход, продвигаясь вперед по темам и освещая то, как язык работает сегодня, и как его лучше всего использовать, а не то, как он развивался с течением времени.

В качестве примера, нативные типы данных, восходящие к оригинальному языку Pascal, имеют возможности использования методов (благодаря встроенным хелперам типов), введенные недавно. Поэтому во второй главе я расскажу, как использовать эту возможность, хотя это произойдет не раньше, чем вы сами разберетесь, как сделать такие пользовательские расширения типов.

Другими словами, эта книга охватывает язык Object Pascal в том виде, в каком он существует сегодня, обучая ему с нуля, лишь с очень ограниченной исторической перспективой. Даже если вы использовали этот язык в прошлом, вы, возможно, захотите просмотреть весь текст с начала в поисках новых возможностей, а не сосредотачиваться только на заключительных главах.

Учись на собственном опыте

Идея книги состоит в том, чтобы объяснить основные понятия и сразу же представить короткие демо-примеры, которые читателям предлагается попытаться выполнить,

поэкспериментировать и расширить, чтобы лучше понять понятия и усвоить их. Книга не является справочным пособием, объясняющим, что должен делать язык в теории, и перечисляющим все возможные краеугольные случаи. Если быть точным, основное внимание уделяется обучению языку, предлагая практическое пошаговое руководство. Примеры, как правило, очень просты, потому что цель состоит в том, чтобы они были сфокусированы на одной характеристике за раз.

Весь исходный код доступен в онлайн-репозитории кода на GitHub. Вы можете скачать его в виде одного файла, клонировать репозиторий или просто просмотреть его онлайн и скачать только код конкретных проектов. Если вы получаете из репозитория, вы легко обновите свой код в случае, если я опубликую какие-либо изменения или дополнительные примеры. Местоположение на GitHub:

`https://github.com/MarcoDelphiBooks/ObjectPascalHandbook104.`

Для компиляции и тестирования демонстрационного кода вам понадобится свежая версия Delphi (по крайней мере 10.4, чтобы запустить все, но большая часть демо-примеров будет работать и на релизах 10.x).

Если у вас нет лицензии для Delphi, то доступна пробная версия, которую вы можете использовать, позволяющая 30-дневное бесплатное использование компилятора и IDE. Существует также бесплатная версия Delphi Community Edition (в настоящее время обновленная до версии 10.3), которая свободна для использования любым человеком, у которого нет или ограничена возможность зарабатывать на разработке программ.

Благодарности

Как и любая другая книга, эти появились благодаря многим людям, слишком многим, чтобы перечислять по одному. Человек, который разделил большую часть работы над первым изданием этой книги, был мой редактор Питер Вуд, который продолжал придерживаться моего постоянно меняющегося графика и смог значительно сгладить мой технический английский язык, помогая сделать эту книгу (как и мои предыдущие руководства) такой, какая она есть.

После первого издания один из читателей, Андреас Тот, прислал мне обширный отзыв о предыдущем издании, и его предложения помогли улучшить текст. Это новое издание также было рассмотрено несколькими другими экспертами Delphi (большинство из них среди MVP Embarcadero), включая xxx.

Учитывая мою нынешнюю должность менеджера по продукции в Embarcadero Technologies, я многим обязан всем своим коллегам и членам команды исследователей и разработчиков, так как за время моей работы в компании мое понимание продукта и его технологии еще больше расширилось благодаря тому, что я получил огромное количество информации в бесчисленных беседах, встречах и электронной переписке. Учитывая то, как трудно убедиться, что все были упомянуты, я не буду пытаться, а только упомяну трех человек, которые сыграли огромную роль и внесли непосредственный вклад в первое издание этой книги: Дэвид Интерсимоне (David I) из отдела по связям с разработчиками, Джон Томас (JT), глава отдела управления продуктами для средств разработки, и главный архитектор RAD Studio Аллен Бауэр.

В последнее время я много работал с двумя другими менеджерами по продукции RAD Studio, Сариной Дюпон и Дэвидом Миллингтоном, с нашим нынешним *евангелистом* Джимом Маккитом, а также с группой нынешних выдающихся архитекторов продуктов.

Другие люди за пределами Embarcadero также являлись важными контактами и иногда вносили прямой вклад, от многих итальянских экспертов Delphi до бесчисленных клиентов, торговых и технических партнеров Embarcadero, членов сообщества Delphi, MVP и даже разработчиков, использующих другие языки и инструменты, с которыми я так часто встречаюсь.

Если и есть в этой группе человек, с которым я проводил много времени, прежде чем присоединиться к Embarcadero, то это Кэри Дженсен, с которым я организовал несколько раундов Delphi Developer Days в Европе и США.

И, наконец, большое спасибо моей семье за то, что вы вынесли мой график путешествий, ночи встреч, плюс несколько дополнительных книг, написанных по выходным. Еще раз спасибо Лелле, Бенни и Джакопо.

О себе, об авторе

Большую часть последних 25 лет я посвятил написанию, преподаванию и консультированию по вопросам разработки программного обеспечения на языке Object Pascal. Я написал серию бестселлеров Mastering Delphi, а позже самостоятельно опубликовал несколько Справочников по инструменту разработки (о различных версиях от Delphi 2007 до Delphi XE).

Я выступал на большом количестве конференций по программированию на большинстве континентов и преподавал

тысячам разработчиков на конференциях, мероприятиях для разработчиков Delphi, классах, проводимых компаниями, онлайн-вебинарах и конференциях CodeRage.

Проработав много лет в качестве независимого консультанта и тренера, в 2013 году моя карьера резко изменилась: Я занял должность менеджера по продуктам Delphi, а теперь – RAD Studio в Embarcadero Technologies, компании, которая занимается созданием и продажей этих инструментов разработки.

Чтобы не раздражать вас больше, я только добавлю, что в настоящее время я живу в Италии, езжу в Калифорнию (теперь чуть меньше), у меня есть прекрасная жена и двое замечательных детей, и мне нравится возвращаться к программированию так часто, как я могу.

Надеюсь, вам понравится читать книгу, как мне понравилось писать это новое издание (моя 23-я работа в печати). Для получения дополнительной информации используйте любой из следующих веб-сайтов и каналов социальных сетей:

```
http://www.marcocantu.com/objectpascalhandbook
http://blog.marcocantu.com
http://twitter.com/marcocantu
```

Оглавление

Оглавление

Книга о современном языке 5
Учись на собственном опыте 6
Благодарности 8
О себе, об авторе 9
Оглавление 11
Часть I: основы 21
Содержимое части I 22
01: кодирование в Паскале 23
Начнем с кода 24
Первое консольное приложение 24
Первое визуальное приложение 26
Синтаксис и стиль кодирования 30
Комментарии 31
Комментарии и XML Doc 33
Символические идентификаторы 35
Нечувствительность к регистру и использование прописных букв 36
Пустое пространство, пробелы 38
Ключевые слова языка 43
Структура программы 53
Имена модулей и программ 55
Имена модулей с точками 55
Подробнее о структуре модулей 57
Раздел Uses 59
Использование модулей, подобно пространствам имён 61
Файл программы 62
Директивы компилятора 64
Условные определения 65
Версии компилятора 66
Включение файлов 68
02: Переменные и типы данных 70
Переменные и присвоения 71
Литералы 73

Оператор присваивания ... 74
Присвоение и преобразование ... 75
Инициализация глобальных переменных ... 76
Инициализация локальных переменных ... 77
Инициализация inline-переменных ... 78
Вывод типа для inline-переменных ... 79
Константы ... 80
Inline-константы ... 81
Константы Resource String ... 82
Жизненный цикл и видимость переменных ... 83
Типы данных ... 85
Порядковый и цифровой типы ... 86
Псевдонимы целых типов ... 87
Целочисленный тип, 64-разрядный, NativeInt и LargeInt. ... 88
Стандартные операции порядкового типа ... 91
Операции Out-of-Range (вне диапазона) ... 93
Boolean ... 94
Символы ... 95
Операции с типом данных Char ... 96
Char как порядковый тип данных ... 97
Типы с плавающей точкой ... 98
Чем отличаются значения с плавающей точкой? ... 101
Простые пользовательские типы данных ... 103
Выражения и операторы ... 112
Использование операторов ... 113
Операторы и Приоритет ... 115
Дата и время ... 119
Приведение и преобразование типов ... 123
03: языковые утверждения ... 129
Простые и составные утверждения ... 130
Условное выражение IF ... 132
Оператор Case ... 134
Цикл For ... 136
Циклы While и Repeat ... 143
Примеры циклов ... 144
Прерывание выполнения цикла с break и continue ... 146
04: Процедуры и функции ... 150
Процедуры и функции ... 150
Предварительные декларации ... 154
Рекурсивная функция ... 156
Что такое Метод? ... 158
Параметры и возвращаемые значения ... 158
Выход с результатом ... 160
Ссылочные параметры ... 162

Постоянные параметры 164
Перегрузка функций 165
Перегрузка и неоднозначные вызовы 168
Параметры по умолчанию 170
Встраивание 172
Расширенные возможности функций 177
Конвенции по вызову в Object Pascal 177
Процедурные типы 179
Декларация внешних функций 182
05: Массивы и записи 186
Типы массивов данных 187
Статические массивы 187
Размеры и границы массива 189
Многомерные статические массивы 190
Динамические массивы 192
Открытый массив параметров 198
Типы данных record 203
Использование массивов записей 205
Вариантные записи 207
Выравнивание полей 208
Особенности оператора With 210
Методы записей 212
Self: Магия записей 215
Инициализация записей 217
Записи и конструкторы 218
Операторы получают новую основу 219
Операторы и управляемые записи 225
Тип данных Variant 231
Варианты не имеют типа 232
Варианты в подробностях 234
Варианты - медленные 235
А что насчет указателей? 237
Типы файлов, кто-нибудь? 242
06: Все о строках 244
Юникод: Алфавит для всего мира 245
Символы из прошлого: от ASCII до ISO кодировки 245
Unicode Code Points и Graphemes 247
От кодовых точек к байтам (UTF) 248
Отметка Byte Order Mark 251
Рассматривая Юникод 252
Вернемся к типу Char 256
Операции в Юникод из модуля Character 257
Символьные литералы Юникода 260
Как насчет 1-байтовых Char? 262
Тип данных String 262

Передача строк как праметры ... 267
Использование [] и режимов подсчета строковых символов ... 268
Конкатенация строк ... 271
Операции хелпера строк ... 273
Больше RTL функций для строк ... 278
Строки форматирования ... 279
Внутренняя структура строк ... 283
Глядя на строки в памяти ... 285
Строки и кодирование ... 287
Другие типы для строк ... 291
Тип UCS4String ... 292
Старые типы строк ... 292
Часть II: ООП в Object Pascal ... 294
Резюме части II ... 295
07: Объекты ... 297
Представляем классы и объекты ... 297
Определение класса ... 299
Классы в других ООП-языках ... 300
Методы и классы ... 302
Создание объекта ... 302
Объектная модель ... 304
Утилизация объектов ... 305
Что такое "nil"? ... 307
Отличие Record от Class в памяти ... 307
Private, Protected и Public ... 309
Пример Private данных ... 311
Инкапсуляция и формы ... 313
Ключевое слово Self ... 316
Динамическое создание компонентов ... 318
Конструкторы ... 320
Управление локальными данными класса с помощью конструкторов и деструкторов ... 323
Перегруженные методы и конструкторы ... 325
Полный класс TDate ... 327
Вложенные типы и вложенные константы ... 331
08: Наследование ... 335
Наследование от существующих типов ... 336
Общий базовый класс ... 339
Защищенные поля Protected и инкапсуляция ... 340
Использование приема "Protected Hack" ... 341
От наследования к полиморфизму ... 344
Наследование и совместимость типов ... 345
Позднее связывание и полиморфизм ... 347
Методы переопределения, повторного определения и повторного объявления ... 350

Наследование и конструкторы 354
Виртуальный и динамический методы 355
Абстрактные методы и классы 357
Абстрактные методы 358
Запечатанные классы и окончательные методы 360
Безопасные операторы приведения типа 362
Наследование визуальной формы 365
Наследование от базовой формы 367
09: Обработка Исключений 373
Блоки Try-Except 375
Иерархия исключений 377
Поднятие (вызов) исключения 380
Исключения и Стек 381
Блок Finally 383
Восстановление курсора с блоком Finally. 385
Восстановить курсор с помощью управляемой записи. 386
Исключения в реальном мире 387
Обработка глобальных исключений 388
Исключения и конструкторы 390
Расширенные возможности исключений 393
Вложенные исключения и Механизм Внутренних Исключений 394
Перехват исключения 398
10: Свойства и события 401
Определение свойств 402
Свойства сравнительно с другими языками программирования 404
Code Completion и свойства 406
Добавление свойств в формы 407
Добавление свойств в класс TDate 410
Использование массива свойств 413
Установка значения свойств по ссылке 414
Спецификатор доступа published 416
Свойства при проектировании 418
Published и формы 419
Автоматическая RTTI 420
Программирование, ориентированное на события 422
Указатели на методы 424
Концепция делегирования 426
События являются свойствами 429
Добавление события в класс TDate 431
Создание компонента TDate 434
Реализация поддержки перечислений в классе 437
15 советов по RAD и ООП 441
Совет 1: Форма - это класс 441
Совет 2: Название Компоненты 442
Совет 3: Имя События 442

Совет 4: Используйте методы формы....443
Совет 5: Добавить конструкторы формы....443
Совет 6: Избегайте глобальных переменных....444
Совет 7: Никогда не используйте Form1 в методах TForm1.....444
Совет 8: Редко используйте Form1 в других формах.....445
Совет 9: Удалить Глобальную переменную Form1....445
Совет 10: Добавить свойства формы....446
Совет 11: Показать свойства компонентов....447
Совет 12: Используйте массив свойств, когда это необходимо....447
Совет 13: Запуск операций в свойствах....448
Совет 14: Спрячьте компоненты....448
Совет 15: Используйте мастер создания формы ООП....450
Советы: Заключение....450
11: Интерфейсы 452
Использование интерфейсов....453
Объявление интерфейса....454
Реализация интерфейса....456
Интерфейсы и подсчет ссылок....458
Ошибки при смешивании способов применения ссылок....460
Слабые и небезопасные ссылки на интерфейс.....462
Расширенные технологии интерфейсов....465
Свойства у интерфейса.....466
Делегирование интерфейса....467
Несколько интерфейсов и псевдонимы методов....469
Полиморфизм интерфейсов....471
Получение объектов из ссылок на интерфейс....473
Реализация паттерна адаптера с интерфейсами....476
12: Работа с классами.... 481
Методы и данные классов....482
Данные классов....483
Виртуальные методы класса и скрытый параметр self....484
Статические методы классов....484
Свойства класса....487
Класс со счетчиком инстанций....488
Конструкторы (и деструкторы) классов....489
Конструкторы классов в RTL....491
Реализация паттерна Singleton....492
Ссылки на классы....493
Ссылки на класс в RTL....496
Создание компонентов с использованием ссылок на классы....496
Помощники классов и записей....499
Помощники класса....501
Помощники классов и наследование....504
Добавление перечисления элементов управления с помощью помощника класса....506

Помощники записей для внутренних типов ... 509
Помощники для псевдонимов типов ... 511
13: Объекты и память ... 514
Глобальные данные, стек и куча ... 515
Глобальная память ... 516
Стек ... 517
Куча ... 518
Объектная модель ... 519
Объекты как параметры ... 520
Советы по управлению памятью ... 522
Уничтожение объектов, которые вы создаете ... 523
Уничтожение объектов только один раз ... 525
Управление памятью и интерфейсы ... 528
Подробнее о Слабых (Weak) Ссылках ... 528
Атрибут Unsafe ... 533
Отслеживание и проверка памяти ... 534
Статус памяти ... 535
FastMM4 ... 535
Отслеживание утечек и другие глобальные настройки ... 536
Перерасход буфера в Full FastMM4 ... 539
Управление памятью на платформах, отличных от Windows ... 542
Отслеживание выделения памяти класса ... 543
Написание надежных приложений ... 543
Конструкторы, деструкторы и исключения ... 544
Вложенные блоки Finally ... 546
Динамическая проверка типа ... 547
Является ли этот указатель ссылкой на объект? ... 549
Часть III: Расширенные функции ... 553
Главы части III ... 554
14: Дженерики ... 555
Generic пары ключей-значений ... 556
Встроенные переменные и выведение типа у дженериков ... 560
Правила типов для дженериков ... 561
Дженерики в Object Pascal ... 562
Правила совместимости Generic типов ... 563
Generic методы для стандартных классов ... 565
Инициализация типа Generic ... 567
Функции типа Generic ... 570
Конструкторы классов для Generic классов ... 573
Ограничения Generic ... 575
Ограничения класса ... 576
Конкретные ограничения классов ... 578
Ограничения, основанные на интерфейсе ... 579
Ссылки на интерфейс vs. Generic ограничения по интерфейсу ... 582
Ограничение конструктор по умолчанию ... 583

Сводка ограничений и их комбинация 585
Готовые Generic контейнеры 587
Использование TList<T> 588
Сортировка TList<T> 590
Сортировка анонимным методом 592
Контейнеры для объектов 594
Использование Generic словаря 595
Словари или списки строк? 599
Generic интерфейсы 601
Предопределенные Generic интерфейсы 604
Умные указатели в Object Pascal 605
Использование записей для умных указателей 606
Реализация умных указателей с управляемой generic записью 607
Реализация умного указателя с Generic записью и интерфейсом 610
Добавление неявного преобразования 612
Сравнение решений на основе интеллектуальных указателей 613
Ковариантные возвратные типы с дженериками 614
Животных, собак и кошек. 614
Метод с Generic результатом 616
Возвращение производного объекта другого класса 617
15: Анонимные методы 619
Синтаксис и семантика анонимных методов 620
Переменная Анонимного метода 621
Параметр анонимного метода 622
Использование локальных переменных 623
Продление срока службы локальных переменных 624
Анонимные методы: за кулисами 626
(Потенциально) пропущенная скобка 627
Реализация анонимных методов 629
Готовые к использованию ссылочные типы 629
Анонимные методы в реальном мире 631
Анонимные обработчики событий 632
Анонимные методы работы со временем 634
Синхронизация потоков 636
AJAX в Object Pascal 638
16: Отражение и атрибуты 644
Расширенная RTTI 646
Первый пример 646
Генерируемая компилятором информация 647
Слабые и сильные связи типов 650
Модуль RTTI 651
Классы RTTI в модуле Rtti 654
Жизненный цикл объектов RTTI и запись TRttiContext 655
Отображение информации о классе 657

RTTI для пакетов ... 659
Структура TValue ... 660
Чтение свойств с TValue ... 664
Вызов методов ... 664
Использование атрибутов ... 665
Что такое "Атрибут"? ... 666
Классы атрибутов и декларации атрибутов ... 667
Атрибуты просмотра ... 670
Перехват виртуальных методов ... 673
Сценарии использования RTTI ... 677
Атрибуты для ID и описания ... 678
Потоковая передача XML ... 684
Другие библиотеки на базе RTTI ... 692
17: TObject и модуль System ... 694
Класс TObject. ... 695
Конструкторы и деструкторы ... 696
Знания об объекте ... 696
Дополнительные методы класса TObject ... 698
Виртуальные методы TObject ... 701
Краткое описание класса ... 705
Юникод и имена классов ... 706
Unit System ... 707
Избранные системные типы ... 708
Интерфейсы в System Unit ... 710
Избранные системные подпрограммы ... 711
Предопределенные атрибуты RTTI ... 712
18: Другие базовые классы RTL ... 714
Unit Classes ... 715
Классы в модуле Classes ... 716
Класс TPersistent ... 718
Класс TComponent ... 719
Современный доступ к файлам ... 723
Модуль управления вводом/выводом ... 724
Введение в потоки ... 726
Использование Reader и Writer ... 729
Построение строк и списков строк ... 732
Класс TStringBuilder ... 732
Использование списков строк ... 734
Библиотека RTL на самом деле большая. ... 735
В заключение ... 739
end. ... 740
Резюме раздела приложений ... 740
A: Эволюция Object Pascal ... 741
Паскаль Вирта ... 742
Турбо-Паскаль ... 743

Ранние дни Object Pascal в Delphi 744
Object Pascal из CodeGear в Embarcadero 745
Переход к мобильности 747
Эпоха Delphi 10.x 749
B: Глоссарий 750
A 750
B 751
C 752
D 753
E 754
F 755
G 755
H 756
I 757
M 758
O 758
P 759
R 760
S 761
U 762
V 762
W 763
C: Index 764

Часть I: основы

Object Pascal - чрезвычайно мощный язык, основанный на таких базовых принципах, как хорошая структура программы и расширяемые типы данных. Эти основы частично происходят от традиционного языка Паскаль, но даже возможности основного языка с первых дней существования получили множество расширений.

В этой первой части книги вы узнаете о синтаксисе языка, стиле кодирования, структуре программ, использовании переменных и типов данных, фундаментальных языковых операторах (таких как условия и циклы), использовании процедур и функций, а также основных конструкциях, таких как массивы, записи и строки.

Это основы более продвинутых функций, от классов до типов generic, которые мы рассмотрим во второй и третьей частях книги. Изучение языка похоже на строительство дома, и нужно начинать с твердой основы и хорошего фундамента, иначе все остальное сверху и сверху будет сияющим... но шатким.

Содержимое части I

Глава 1: Кодирование в Паскале

Глава 2: Переменные и типы данных

Глава 3: Языковые выражения

Глава 4: Процедуры и функции

Глава 5: Массивы и записи

Глава 6: Все о строках

01: кодирование в Паскале

Эта глава начинается с некоторых строительных блоков приложения Object Pascal, в которых рассматриваются стандартные способы написания кода и связанных с ним комментариев, вводятся ключевые слова и описывается структура программы. Я начну писать некоторые простые приложения, пытаясь объяснить, что они делают, и таким образом представлю некоторые другие ключевые понятия, рассмотренные более подробно в следующих главах.

Начнем с кода

В этой главе рассказывается об основах языка, но мне понадобится несколько глав, которые помогут вам разобраться в деталях законченного рабочего приложения. А пока давайте посмотрим на две первые программы (разные по своей структуре), не вдаваясь в подробности. Здесь я просто хочу показать вам структуру программ, которые я буду использовать для создания примеров, объясняющих специфические конструкции языка, прежде чем я смогу охватить все различные элементы. Учитывая, что я хочу, чтобы вы могли как можно скорее начать применять на практике информацию, приведенную в книге, разбор демо-примеров с самого начала был бы хорошей идеей.

Object Pascal был разработан для работы рука об руку с интегрированной средой разработки. Именно благодаря этой мощной комбинации Object Pascal может соответствовать скорости разработки удобных для программиста языков и в то же время соответствовать скорости работы дружественных к машине языков.

IDE позволяет создавать пользовательские интерфейсы, помогает писать код, запускать программы и многое, многое другое. Я буду использовать IDE на протяжении всей книги по мере того, как буду знакомить вас с языком Object Pascal.

Первое консольное приложение

В качестве отправной точки, я покажу вам код простого консольного приложения *Hello, World*, показывающего некоторые структурные элементы программы Object Pascal. Консольное приложение – это программа без графического

интерфейса пользователя, отображающая текст и принимающая ввод с клавиатуры, и обычно выполняемая из консоли операционной системы или командной строки. Консольные приложения, как правило, имеют мало смысла на мобильных платформах и редко используются на ПК в наши дни.

Я пока не буду объяснять, что означают различные элементы кода, приведенные ниже, так как именно для этого и предназначаются первые несколько глав книги. Вот код из проекта приложения `HelloConsole`:

```
program HelloConsole;

{$APPTYPE CONSOLE}

var
  StrMessage: string;

begin
  StrMessage := 'Hello, World';
  writeln (StrMessage);
  // wait until the Enter key is pressed
  readln;
end.
```

примечание Как поясняется во введении, полный исходный код всех демо-примеров книги доступен в онлайн-репозитории на GitHub. Подробнее о том, как получить эти примеры, см. во введении к книге. В тексте я ссылаюсь на название проекта (в данном случае `HelloConsole`), которое также является названием папки, содержащей различные файлы примеров. Папки проекта сгруппированы по главам, так что вы найдете эту первую демонстрацию под `01/HelloConsole`

Имя программы можно увидеть в первой строке после конкретного объявления, директивы компилятора (префиксом по символу `$` и заключенным в фигурные скобки), объявления переменной (строка с заданным именем) и трех строк кода плюс комментарий внутри основного `начального` блока. Эти три строки кода копируют значение в строку, вызывают системную функцию для записи этой строки в консоль и вызывают другую системную функцию для чтения строки пользовательского ввода (или, в данном случае, для ожидания нажатия

пользователем клавиши Enter). Как мы увидим, вы можете определить свои собственные функции, но Object Pascal поставляется с сотнями предопределенных.

Опять же, мы скоро узнаем обо всех этих элементах, так как этот начальный раздел служит только для того, чтобы дать вам представление о том, как выглядит маленькая, но полная программа Pascal. Конечно, вы можете открыть и запустить это приложение, которое будет выдавать следующие результаты (реальная версия Windows показана на Рисунке 1.1).

```
Hello, World
```

Рисунок 1.1: Вывод Пример HelloConsole работающий на Windows

Первое визуальное приложение

Современное приложение, правда, редко похоже на эту старомодную консольную программу и, как правило, состоит из визуальных элементов (так называемых элементов управления), отображаемых в окошках (так называемых формах). В большинстве случаев в этой книге я буду строить визуальные примеры (даже если в большинстве случаев они будут сводиться к отображению простого текста), используя библиотеку FireMonkey (которая также известна как FMX).

примечание В Дельфи визуальное управление имеет две разновидности: VCL (библиотека визуальных компонентов для Windows) и FireMonkey (кроссплатформенная библиотека для разных устройств, для всех поддерживаемых платформ, настольных и мобильных). В любом случае, адаптировать примеры к Windows-специфичной VCL-библиотеке должно быть достаточно просто.

Чтобы понять точную структуру визуального приложения, вам придется прочитать значительную часть этой книги, так как форма является объектом данного класса и имеет методы, обработчики событий, свойства... все функции, и это займет некоторое время. Но чтобы иметь возможность создавать такие приложения, вам не нужно быть экспертом, так как все, что вам нужно сделать, это использовать команду меню для создания нового настольного или мобильного приложения. В начальной части книги я буду основывать примеры на платформе FireMonkey и просто использовать контекст форм и операции нажатия кнопок. Для начала можно создать форму любого типа (настольную или мобильную, я бы выбрал New Multi-device Application, так как оно также будет работать под Windows), и поместить на него кнопку управления, с многострочным текстовым элементом управления (Memo) под ней, чтобы отобразить вывод.

На рисунке 1.2 показано, как ваша форма приложения будет выглядеть для мобильного приложения в Delphi IDE, после выбора предварительного просмотра в стиле Android (см. комбо-бокс над поверхностью формы) и добавления одного элемента управления - кнопки.

Рисунок 1.2: Простое мобильное приложение с одной кнопкой из примера HelloVisual

Все, что требуется для создания подобного приложения — это добавить кнопку на пустую форму. Теперь, чтобы добавить нужный код, который является единственной вещью, которая нас интересует на данный момент, дважды щелкните по кнопке, вам будет представлен следующий скелет кода (или что-то очень похожее):

```
procedure TForm1.Button1Click (Sender: TObject)
begin

end;
```

Даже если вы не знаете, что такое метод класса (что такое Button1Click), вы можете напечатать что-нибудь в этом фрагменте кода (что означает в пределах ключевых слов `begin` и `end`), и этот код будет выполнен при нажатии кнопки.

Наша первая "визуальная" программа имеет код, совпадающий с кодом первого консольного приложения, только в другом контексте и вызывая другую библиотечную функцию, а именно глобальную функцию `ShowMessage`, используемую для отображения некоторой строки в окне сообщения. Именно такой код вы можете найти в проекте приложения `HelloVisual` и достаточно легко попробовать перестроить его с нуля:

```
procedure TForm1.Button1Click (Sender: TObject)
var
  StrMessage: string;
begin
  StrMessage := 'Hello, World';
  ShowMessage (StrMessage);
end;
```

end;

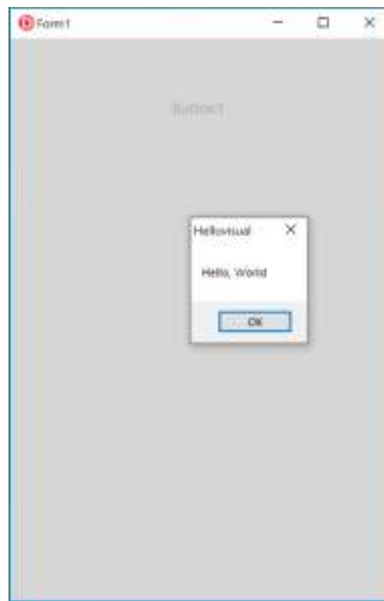
Обратите внимание, как нужно разместить объявление переменной `strMessage` перед оператором begin и реальный код после него. Опять же, не волнуйтесь, если это неясно, все будет объяснено в свое время в подробностях.

Примечание Исходный код этой демонстрации находится в папке под контейнером 01 для главы. В этом случае, однако, есть не только имя файла проекта, как у примера, но

и вторичный файл с добавленным после имени проекта словом "Form". Это стандарт, которому я буду следовать в книге. Структура проекта рассматривается в конце этой главы

На Рисунке 1.3 вы можете увидеть результаты этой простой программы, работающей под Windows с включенным режимом FMX MobilePreview (вы можете запустить эту пример и на Android, и на iOS, и на MacOS, но это требует некоторых дополнительных настроек в IDE)

Рисунок 1.3: **:** Простое мобильное приложение с одной кнопкой, используемое примером HelloVisual



примечание FireMonkey Mobile Preview делает приложение Windows немного похожим на мобильное приложение. Я включил этот режим в большинстве примеров из этой книги. Это делается путем добавления MobilePreview в набор Uses в исходный код проекта

Теперь, когда у нас есть возможность написать и протестировать демо-программу, давайте вернемся к началу, охватив все детали первых нескольких строительных блоков приложения, как я обещал в начале этой главы. Первое, что вам нужно знать, это как *читать* программу, как написаны различные элементы, и какова структура приложения, которое мы только что собрали (в котором есть и PAS-файл, и DPR-файл).

Синтаксис и стиль кодирования

Прежде чем перейти к теме написания реальных выражений языка Object Pascal, важно выделить некоторые элементы стиля кодирования Object Pascal. Вопрос, который я рассматриваю, вот в чем: кроме правил синтаксиса (которые мы до сих пор не изучили), как следует писать код? На этот вопрос нет единственно верного ответа, так как личный вкус может диктовать различные стили. Тем не менее, есть некоторые принципы, которые нужно знать относительно комментариев, прописных букв, пробелов и того, что много лет назад называлось *красиво напечатано* (красиво для нас, людей, а не для компьютера) - термин, который сейчас считается устаревшим.

В общем, целью любого стиля кодирования является ясность. Стиль и решения по форматированию, которые вы принимаете, представляют собой форму сокращения, указывающего на назначение данного фрагмента кода. Важным инструментом для ясности является последовательность - какой бы стиль вы ни выбрали, обязательно следуйте ему на протяжении всего проекта и во всех проектах.

Совет IDE (Integrated Development Environment) имеет поддержку автоматического форматирования кода (на уровне модуля или проекта): Вы можете скомандовать

редактору переформатировать ваш код с помощью клавиш Ctrl+D, следуя набору правил, которые вы можете изменить, настроив около 40 различных элементов форматирования (находящихся среди Options IDE), и даже поделиться этими настройками с другими разработчиками в вашей команде, чтобы сделать форматирование согласованным. Однако, автоматическое форматирование не поддерживает некоторые из последних возможностей языка

Комментарии

Несмотря на то, что код часто бывает понятен сам по себе, уместно добавить значительное количество комментариев в исходный код программы, чтобы в дальнейшем объяснить другим (и себе, когда вы будете просматривать свой код в отдаленном будущем), почему код был написан определенным образом и каковы были установки.

В традиционном Pascal комментарии были заключены либо в фигурные скобки, либо в скобки, за которыми следовала звезда. Современные версии языка дополнительно допускают однострочные комментарии в стиле C++, с двойной косой чертой, которые действуют до конца строки и не требуют наличия символа, обозначающего конец комментария:

```
// это комментарий до конца строки

{ это многострочный комментарий }

(* это еще один
многострочный комментарий *)
```

Первая форма короче и чаще используется. Вторая форма часто предпочиталась в Европе, потому что на многих европейских клавиатурах отсутствует символ фигурной скобки (или его использование затруднено). Третья форма комментария заимствована из языка C/C++, который также

использует синтаксис `/* комментария */` для многострочных комментариев, наряду с C#, Objective-C, Java и JavaScript.

Комментарии до конца строки очень полезны для коротких комментариев и для комментирования одной строки кода. На сегодняшний день они являются наиболее распространенной формой комментариев на языке Object Pascal.

примечание В редакторе IDE можно прокомментировать или разделить текущую строку (или группу выделенных строк) прямым нажатием клавиши. Это Ctrl+/ на американской клавиатуре и другая комбинация (с физической / клавишей) на других клавиатурах: действующая клавиша перечислена во всплывающем меню редактора.

Для маркировки вложенных комментариев может быть полезным наличие трех различных форм комментариев. Если вы хотите прокомментировать несколько строк исходного кода, чтобы отключить их, и эти строки содержат некоторые реальные комментарии, вы не можете использовать один и тот же идентификатор комментария:

```
{
  code...
  {nested comment, creating problems}
  code...
}
```

Приведенный выше код приводит к ошибке компилятора, так как первая закрывающая скобка указывает на конец всей закомментированной секции. Со вторым вложенным комментарием можно написать следующий код, который является корректным:

```
{
  code...
  // this comment is OK
  code...
}
```

Альтернативой является комментирование группы строк, как объяснялось выше, поскольку это добавит второй // комментарий к комментируемой строке, вы можете легко

удалить, разместив тот же самый блок (сохранив исходный комментарий).

примечание Если за открытой скобкой или скобкой со звездочкой следует знак доллара ($), то это уже не комментарий, а директива компилятора, как мы видели в первом примере в строке {$APPTYPE CONSOLE}. Директивы компилятора инструктируют компилятор делать что-то особенное, и кратко объясняются в конце этой главы. На самом деле, директивы компилятора все еще являются комментариями. Например, {$X+ *Это комментарий} является* легальным. Это и действительная директива, и комментарий, хотя большинство *здравомыслящих* программистов, скорее всего, будут склонны разделять директивы и комментарии

Комментарии и XML Doc

Существует специальная версия комментариев, общая и для других языков программирования, к которой компилятор относится особым образом. Эти специальные комментарии генерируют дополнительную документацию, доступную непосредственно в IDE Help Insight и в XML-файлах, генерируемых компилятором.

примечание В Delphi IDE Help Insight автоматически отображает информацию о символе (включая его тип и место, где он был определен). С помощью комментариев XML Doc вы можете дополнить эту информацию конкретными деталями, написанными в самом исходном коде

XML Doc включается с помощью /// комментариев или {! комментариев. В этих комментариях вы можете использовать общий текст или (лучше) специфические XML-теги для указания информации о закомментированном символе, его параметрах и возвращаемом значении и многое другое. Это очень простой случай текста в свободной форме:

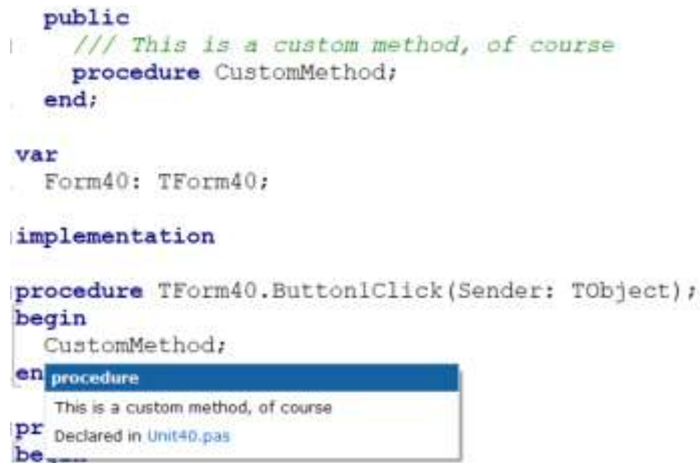
```
public
  /// This is a custom method, of course
  procedure CustomMethod;
```

Информация добавляется в XML-выход, генерируемый компилятором, если вы включили генерацию XML Doc, следующим образом:

```
<procedure name="CustomMethod" visibility="public">
  <devnotes>
    This is a custom method, of course
  </devnotes>
</procedure>
```

Та же информация отображается в IDE при наведении курсора на символ, как показано на рисунке 1.4.

Рисунок 1.4: Help Insight в Delphi IDE показывает XML Doc записанную в комментариях



Если вы предоставите в комментарии раздел с кратким описанием, следуя рекомендуемым соглашениям, это также отобразится в окне Help Insight:

```
public
/// <summary>This is a custom method, of course</summary>
  procedure CustomMethod;
```

Преимущество заключается в том, что существует множество других XML-тегов, которые можно использовать для определения параметров, возвращаемых значений и более подробной информации. Доступные теги перечислены по адресу:

http://docwiki.embarcadero.com/RADStudio/en/XML_Documentation_Comments

Символические идентификаторы

Программа состоит из множества различных символов, которые можно использовать для наименования различных элементов (типов данных, переменных, функций, объектов, классов и т.д.). Несмотря на то, что вы можете использовать практически любой идентификатор, существует несколько правил, которым вы должны следовать:

- Identifiers cannot include spaces (as spaces do separate identifiers from other language elements)

- Идентификаторы не могут включать пробелы (поскольку пробелы отделяют идентификаторы от других элементов языка).
- Идентификаторы могут использовать буквы и цифры, включая буквы всего алфавита Unicode; таким образом, вы можете называть символы на вашем родном языке, если хотите (что не очень рекомендуется, так как часть инструментов IDE могут не предлагать такую же поддержку).
- Из традиционных ASCII-символов идентификаторы могут использовать только символ подчеркивания (_); все остальные ASCII-символы кроме букв и цифр не допускаются. К недопустимым символам в идентификаторах относятся символы (+, -, *, /, =), все обычные и фигурные скобки ({}[]), знаки препинания, специальные символы (в том числе @, #, $, %, ^, &, \, |). Однако вы можете использовать символы Юникода, например ☼♪ или ∞.
- Идентификаторы должны начинаться с буквы или подчеркивания, начинаться с цифры не разрешается (другими

словами, можно использовать цифры, но не как первый символ). Здесь под числами мы подразумеваем ASCII-цифры, от 0 до 9, в то время как другие Unicode-представления чисел разрешены.

Ниже приведены примеры классических идентификаторов, перечисленных в приложении `IdentifiersTest`:

```
MyValue
Value1
My_Value
_Value
Val123
_123
```

Это примеры легальных Unicode-идентификаторов (где последний немного экстремален):

```
Cantù (Latin accented letter)
结 (Cash Balance in Simplified Chinese)
画像 (picture in Japanese)
☼ (Sun Unicode symbol)
```

Вот несколько примеров *недопустимых* идентификаторов

```
123
1Value
My Value
My-Value
My%Value
```

совет Если вы хотите проверить правильность идентификатора во время выполнения (что-то редко требуется, если только вы не пишете инструмент в помощь другим разработчикам), то в библиотеке времени выполнения есть функция, которую вы можете использовать, называемая IsValidIdent.

Нечувствительность к регистру и использование прописных букв

В отличие от многих других языков, в том числе всех, основанных на синтаксисе C (таких как C++, Java, C# и JavaScript), компилятор Object Pascal игнорирует регистр или заглавные буквы идентификаторов. Поэтому идентификаторы

`Myname`, `MyName`, `myname`, `myName` и `MYNAME` абсолютно одинаковы. По моему личному мнению, нечувствительность к регистру, определенно, является положительным моментом, так как синтаксические ошибки и другие скрытые ошибки могут быть вызваны неправильным использованием заглавных букв в регистрозависимых языках.

Если учесть тот факт, что для идентификаторов можно использовать Юникод, то все немного усложняется, так как заглавная версия буквы рассматривается как один и тот же элемент, в то время как акцентированная версия той же буквы рассматривается как отдельный символ. Другими словами:

```
cantu: Integer;
Cantu: Integer; // error: duplicate identifier
cantù: Integer; // correct: different identifier
```

примечание Есть только одно исключение из правила нечувствительности к регистру языка: процедура Register пакета компонентов должна начинаться с верхнего регистра R, из-за проблемы совместимости с C++. Конечно, при обращении к идентификаторам, экспортируемым другими языками (например, родной функцией операционной системы), может потребоваться использование правильной заглавной буквы.

Однако есть пара нюансов. Во-первых, вы должны знать, что эти идентификаторы на самом деле одни и те же, поэтому вы должны избегать использования их в качестве различных элементов. Во-вторых, следует стараться быть последовательным в использовании заглавных букв, чтобы улучшить читабельность кода.

Последовательное использование регистра не принуждается компилятором, но это хорошая привычка. Общий подход заключается в том, чтобы сделать заглавными только первую букву каждого идентификатора. Когда идентификатор состоит из нескольких последовательных слов (в идентификатор нельзя вставить пробел), каждая первая буква слова должна быть заглавной:

```
MyLongIdentifier
MyVeryLongAndAlmostStupidIdentifier
```

Это часто называют "Паскаль-стилем" (PascalCase), чтобы противопоставить его так называемому "Верблюжьему стилю" (CamelCase) Java и других языков, основанному на синтаксисе C, в котором внутренние слова заглавными буквами начинаются с нижнего регистра, как в случае с

```
myLongIdentifier
```

На самом деле все чаще встречается код Object Pascal, в котором локальные переменные используют Camel Case (нижний регистр в начале), в то время как элементы класса, параметры и другие более глобальные элементы используют PascalCase. Во всяком случае, в исходных текстах книг я попытался последовательно использовать Pascal-стиль для всех символов.

Пустое пространство, пробелы

Другие элементы, которые компилятор полностью игнорирует — это пробелы, новые строки и табуляции, которые вы добавляете в исходный код. Все эти элементы коллективно известны как *пробелы*. Пробел используется только для улучшения читабельности кода; это никак не влияет на компиляцию.

В отличие от традиционного BASIC, Object Pascal позволяет писать оператор на нескольких строках кода, разбивая длинную инструкцию на две или более строк. Недостатком допустимости операторов более чем на одну строку является то, что вы должны помнить о добавлении точки с запятой для обозначения конца оператора, или, точнее, для отделения одного оператора от другого. Единственным ограничением при разбиении программных операторов на разные строки

является то, что строковый литерал не может охватывать несколько строк.

Необычно, но все следующие строки представляют собой одно и то же утверждение:

```
A := B + 10;

A :=
  B
  +
  10;

A
:=
// this is a mid-statement comment
B + 10;
```

Опять же, нет жестких требований использования пробелов и многострочных операторов, только некоторые правила:

- В редакторе есть вертикальная строка, которую можно разместить после 80 или около того символов. Если вы используете эту строку и стараетесь не превышать этот предел, ваш исходный код будет выглядеть лучше, и вам не придется прокручивать его по горизонтали, чтобы прочитать на компьютере с меньшим экраном. Изначальная идея 80 символов заключалась в том, чтобы сделать код красивее при печати, что не очень распространено в наши дни (но все же полезно).
- Когда функция или процедура имеет несколько сложных параметров, обычной практикой является размещение параметров на разных строках, привычка, которая в основном исходит от языка Си.
- Вы можете оставить полностью белую (пустую) строку перед комментарием или разделить длинный кусок кода на более мелкие части. Даже эта простая идея может улучшить читабельность кода.
- Используйте пробелы для разделения параметров вызова функции, и, возможно, даже пробел перед начальной открытой

круглой скобкой. Также мне нравится разделять операнды выражений, хотя это вопрос предпочтений...

Отступы

Последнее предложение по использованию пробелов относится к типичному стилю лингвистического форматирования Паскаля, первоначально известному как красивая печать (pretty-printing), но теперь обычно называемому отступом.

примечание Правила отступов подчиняются личному вкусу, и я не хочу вступать в битву табуляции против пробелов. Здесь я просто указываю, что это "самый распространенный" или "стандартный" стиль форматирования в мире Object Pascal, который используется в исходном коде библиотек Delphi. Исторически в мире Pascal этот набор правил обозначался как красиво распечатанный, термин теперь довольно необычный.

Это правило простое: Каждый раз, когда вам нужно написать составное утверждение, отступайте на два пробела (не табуляция, как это обычно делает программист на Си) справа от текущего утверждения. Составной оператор внутри другого составного оператора отступает на четыре пробела и так далее:

```
if ... then
  statement;

if ... then
begin
  statement1;
  statement2;
end;

if ... then
begin
  if ... then
    statement1;
  statement2;
end;
```

Опять же, программисты по-разному интерпретируют это общее правило. Одни отступают от `begin` и `end` утверждений до уровня внутреннего кода, другие ставят `begin` в конце строки

предыдущего утверждения (по-си-подобному). В основном это дело личного вкуса.

Подобный формат отступов часто используется для списков переменных или типов данных:

```
type
  Letters = ('A', 'B', 'C');
  AnotherType = ...

var
  Name: string;
  I: Integer;
```

Заметка В приведенном выше коде можно задаться вопросом, почему два разных типа, string и Integer, написаны с разным регистром для начальной буквы. Как было сказано в оригинальном руководстве Object Pascal Styles Guide, "такие типы, как Целый, являются просто идентификаторами и появляются с первой буквой; строки, однако, объявляются с зарезервированной строкой слова, которая должна быть полностью строчной"

Раньше при декларировании пользовательских типов и переменных также было принято использовать отступ на основе столбцов, но теперь это происходит реже. В таком случае код, приведенный выше, будет выглядеть как *не рекомендуемый* код, приведенный ниже:

```
type
  Letters     = ('A', 'B', 'C');
  AnotherType = ...


var
  Name : string;
  I    : Integer;
```

Отступы также обычно используются для выражений, продолжающихся от предыдущей строки, или для параметров функции (если не ставить каждый параметр в отдельную строку):

```
MessageDlg ('This is a message',
  mtInformation, [mbOk], 0);
```

Выделение синтаксиса

Чтобы облегчить чтение и написание кода Object Pascal, в редакторе IDE есть функция подсветки синтаксиса. В зависимости от значения в языке вводимых слов, они отображаются с использованием различных цветов и стилей шрифта. По умолчанию ключевые слова выделены жирным шрифтом, строки и комментарии - цветом (и часто курсивом), и так далее.

Зарезервированные слова, комментарии и строки, вероятно, являются тремя элементами, которые извлекают наибольшую пользу из этой особенности. С первого взгляда можно увидеть неправильно написанное ключевое слово, строку с неправильным завершением и длину многострочного комментария.

Вы можете легко настроить параметры подсветки синтаксиса, используя страницу Редактор Цветов в диалоге Options (Параметры) IDE. Если вы единственный человек, использующий компьютер для поиска исходного кода Object Pascal, выберите нужные цвета. Если вы работаете в тесном контакте с другими программистами, вы все должны договориться о стандартной цветовой схеме. Я часто обнаруживал, что работа на компьютере с другой синтаксической раскраской, чем та, которую я обычно использую, действительно сбивает с толку.

Понимание ошибок и понимание кода

Редактор IDE имеет много других возможностей, которые помогут вам написать корректный код. Наиболее очевидным из них является Error Insight, который помещает красную волнистую линию под элементы исходного кода, которые он не

понимает, таким же образом, как текстовый процессор отмечает орфографические ошибки.

примечание Иногда необходимо скомпилировать программу в первый раз, чтобы избежать признаков Error Insight для совершенно легитимного кода. Кроме того, сохранение файла, например, формы, может привести к включению соответствующих модулей, необходимых для текущих компонентов, что позволит устранить проблемы, связанные с ошибками в Error Insight. Эти проблемы были в значительной степени решены с помощью нового LSD (Language Server Protocol) Code Insight в Delphi 10.4.

Другие функции, такие как Code Completion, помогают вам писать код, предоставляя список допустимых символов прямо в том месте, где вы пишете. Когда функция или метод имеет параметры, вы увидите их в списке по мере того, как будете набирать текст. Также вы можете навести курсор на символ, чтобы увидеть его определение. Однако, это специфические возможности редактора, которые я не хочу вдаваться в подробности, так как я хочу остаться сосредоточенным на языке и не обсуждать редактор IDE в деталях (даже если это, безусловно, наиболее распространенные инструменты, используемые для написания кода Object Pascal).

Ключевые слова языка

Ключевые слова — это все идентификаторы, зарезервированные языком. Это символы, которые имеют предопределённое значение и роль, и вы не можете использовать их в другом контексте. Формально существует различие между зарезервированными словами и директивами: Зарезервированные слова не могут быть использованы в качестве идентификаторов, в то время как директивы имеют специальное значение, но могут быть использованы и в другом контексте (хотя рекомендуется не делать этого). На практике не

следует использовать какие-либо ключевые слова в качестве идентификатора.

Если вы напишете подобный код (где `свойство` действительно является ключевым словом):

```
var
  property: string
```

вы увидите сообщение об ошибке типа:

```
E2029 Identifier expected but 'PROPERTY' found
```

В общем, при неправильном употреблении ключевого слова, в зависимости от ситуации, вы получите различные сообщения об ошибках, так как компилятор распознает ключевое слово, но не понимает его положении в коде или в следующих элементах.

Здесь я не хочу показывать вам полный список ключевых слов, так как некоторые из них довольно непонятны и редко используются, а лишь перечислю несколько, группируя их по ролям. Мне понадобится несколько глав, чтобы исследовать все эти ключевые слова, а другие я пропущу в этом списке.

примечание Обратите внимание, что некоторые ключевые слова могут быть использованы в разных контекстах, и здесь я, как правило, имею в виду только наиболее распространенный контекст (хотя пара ключевых слов перечислена дважды). Одна из причин в том, что на протяжении многих лет разработчики компиляторов хотели избежать введения новых ключевых слов, так как это может нарушить работу существующих приложений, поэтому они *переработали* некоторые из существующих.

Итак, давайте начнем исследование ключевых слов с тех, которые вы уже видели в исходных демо-кодах и которые используются для определения **структуры проекта приложения**:

program	Указывает название проекта приложения

library	Показывает имя проекта библиотеки
package	Указывает имя проекта пакетной библиотеки
unit	Указывает имя модуля, файл с исходным кодом.
uses	ссылки к другим модулям, которые использует код
interface	Часть модуля с декларациями
implementation	Часть модуля с выполняемым кодом
initialization	Код, выполняемый при запуске программы
finalization	Код, выполненный по завершению программы
begin	Начало блока кода
end	Конец блока кода

Другой набор ключевых слов связан с объявлением различных базовых **типов данных и переменных** таких типов данных:

type	Вводит блок описания типа

var	Вводит блок объявлений переменных
const	Вводит блок деклараций констант
set	Определяет мощный тип данных *набор*
string	Определяет строковую переменную или пользовательский тип строки
array	Определяет тип массива
record	Определяет тип записи
integer	Определяет целочисленную переменную
real, single, double	Определяет переменные с плавающей точкой
file	Определяет файл

примечание Есть много других типов данных, определенных в Object Pascal, о которых я расскажу позже.

Третья группа включает в себя ключевые слова, используемые для **основных операторов языка**, такие условия и циклы, включая также функции и процедуры:

if	Вводит условный оператор

then	Отделяет условие от кода для выполнения
else	Показывает возможный альтернативный код
case	Вводит условный оператор с несколькими опциями
of	Отделяет условие от опций
for	Вводит цикл с фиксированным числом повторений
to	Показывает конечное верхнее значение для повторов цикла.
downto	Показывает конечное нижнее значение для повторов цикла.
in	Показывает коллекцию для итерации в цикле
while	Вводит условный повторяющийся цикл
do	Отделяет условие цикла от кода
repeat	Вводит повторяющийся цикл с условием в конце
until	Показывает условие окончания цикла

with	Показывает структуру данных для работы с ней
function	Подпрограмма или группа операторов, возвращающих результат
procedure	Подпрограмма или группа операторов, которые не возвращают результат
inline	Обращается к компилятору с просьбой оптимизировать функцию или процедуру
overload	Позволяет повторно использовать название функции или процедуры.

Многие другие ключевые слова относятся к **классам и объектам**:

class	Указывает тип класса
object	Используется для указания более старого типа класса (в настоящее время устаревшего).
abstract	Класс, который не полностью определен

sealed	Класс, от которого другие классы не могут наследовать
interface	Указывает тип интерфейса (перечислен также в первой группе).
constructor	Метод инициализации объекта или класса
destructor	Объект или метод очистки класса
virtual	Виртуальный метод
override	Модифицированная версия виртуального метода
inherited	Относится к методу базового класса
private	Часть класса, недоступная снаружи.
protected	Часть класса с ограниченным доступом извне
public	Часть класса полностью доступна снаружи
published	Часть класса, специально доступная для пользователей
strict	Более строгое ограничение для секций private и protected

property	Символ, сопоставленный значению или методу.
read	способ для получения значение свойства.
write	способ для установки значения свойства
nil	Значение *нулевого* объекта (используется также для других объектов)

Для **работы с исключениями** используется меньшая группа ключевых слов (см. главу 11):

try	Начало блока обработки исключений
finally	Вводит код, который должен быть выполнен независимо от исключений
except	Вводит код, который должен быть выполнен в случае исключения
raise	Используется, чтобы вызвать исключение

Другая группа ключевых слов используется для **операций** и рассматривается в разделе "Выражения и операторы" далее в этой главе (кроме некоторых продвинутых операторов, рассмотренных только в более поздних главах):

```
as                    and                   div
is                    in                    mod
not                   or                    shl
shr                   xor
```

Наконец, вот неполный список других, **менее распространенных ключевых слов**, включая несколько старых, которые действительно следует избегать использования. Опять же, здесь лишь краткое указание на то, где они используются, как многие из них или описаны позже в книге:

default	Показывает значение свойства по умолчанию
dynamic	Виртуальный метод с другой реализацией
export	Устаревшее ключевое слово, используемое при экспорте, заменено на приведенное ниже
exports	В проекте DLL список функций для экспорта
external	Относится к внешней DLL-функции, которую вы хотите привязать к
file	Используется для устаревшего типа *file*, редко используется в наши дни.
forward	Указывает на декларирование функции, определенной ниже

goto	Перепрыгивает на метку в определенном месте кода. Настоятельно рекомендуется избегать "goto".
index	Используется для проиндексированных свойств и (редко) при импорте или экспорте функций
label	Определяет метку, к которой возможен переход по goto. Настоятельно рекомендуется избегать goto.
message	Альтернативное ключевое слово для виртуальных функций, связанных с сообщениями платформы.
name	Используется для ‘привязки’ внешних функций
nodefault	Показывает, что свойства не имеют значения по умолчанию
on	Используется, чтобы вызвать исключение
out	Альтернатива var, указывающая на параметр, переданный по ссылке, но не инициализированный

packed	Изменение компоновки памяти записи или структуры данных
reintroduce	Позволяет повторно использовать имя виртуальной функции
requires	В пакете указывает зависимость от пакетов

Обратите внимание, что в список ключевых слов языка Object Pascal за последние годы было внесено очень мало дополнений, так как любое дополнительное ключевое слово подразумевает потенциальное внесение ошибок компиляции в некоторые существующие программы, предотвращая случайное использование одного из новых ключевых слов в качестве символа. Большинство недавних дополнений к языку не требовали нового ключевого слова, как, например, generic и анонимные методы.

Структура программы

Вы почти никогда не пишете весь код в одном файле, хотя так было с первым простым консольным приложением, которое я показывал ранее в этой главе. Как только вы создаете визуальное приложение, вы получаете как минимум один вторичный файл с исходным кодом рядом с файлом проекта. Этот вторичный файл называется *unit* (юнит) и обозначается расширением .PAS (для исходного модуля на Pascal), в то время как основной файл проекта использует расширение .DPR (для

файла Delphi Project). Оба файла содержат исходный код на Object Pascal.

В Object Pascal широко используются модули или программные модули. Модули, по сути, могут использоваться для обеспечения модульности и инкапсуляции даже без использования объектов, и они действуют как пространства имен. Приложение Object Pascal обычно состоит из нескольких модулей, в том числе модулей описания форм и модулей данных. Фактически, когда вы добавляете новую форму в проект, IDE фактически добавляет новый модуль, который определяет код новой формы.

Для модулей unit не требуются определения форм; они могут просто определить и сделать доступным набор процедур или один из нескольких типов данных (включая классы). Если вы добавите новый пустой unit в проект, он будет содержать только ключевые слова, используемые для разделения секций, на которые разделен модуль:

```
unit Unit1;

interface

implementation

end.
```

Структура модуля достаточно проста, как показано выше:

Во-первых, модуль имеет уникальное имя, соответствующее его имени файла (т.е. приведенный выше пример модуля должен храниться в файле *Unit1.pas*).

Во-вторых, модуль имеет раздел `interface`, декларирующий то, что видно другим модулям.

В-третьих, модуль имеет секцию `implementation` с деталями реализации, фактическим кодом и, возможно, другими локальными декларациями, которые не видны за пределами модуля.

Имена модулей и программ

Как я уже упоминал, имя модуля должно соответствовать имени файла этого модуля. То же самое относится и к программе. Чтобы переименовать модуль, вы выполняете операцию Save As в IDE, и эти две операции будут синхронизированы. Конечно, вы также можете изменить имя файла на уровне файловой системы, но если вы также не измените объявление в начале модуля, вы увидите ошибку при компиляции модуля (или даже при его загрузке в IDE). Это пример сообщения об ошибке, которое вы получите, если вы измените объявление модуля без обновления также и имени файла:

```
[DCC Error] E1038 Unit identifier 'Unit3' does not match file name
```

Подразумевается, что имя модуля или программы должно быть не только действительным идентификатором Pascal, но и действительным именем файла в файловой системе. Например, оно не может содержать пробела, а не специальных символов рядом с подчеркиванием (_), о чем говорилось ранее в данной главе в разделе об идентификаторах. Учитывая, что модули и программы должны быть названы с использованием идентификатора Object Pascal, они автоматически приводят к действительным именам файлов, поэтому не стоит беспокоиться об этом. Исключением, конечно, является использование символов Юникода, которые не являются действительными именами файлов на уровне файловой системы.

Имена модулей с точками

Существует расширение основных правил для идентификаторов единиц: имя модули может использовать

точечную нотацию. Таким образом, все нижеследующие правила являются действительными именами единиц:

```
unit1
myproject.unit1
mycompany.myproject.unit1
```

В соответствии с общими правилами эти модули необходимо сохранять в файлах с одинаковым именем (т.е. модуль под названием `MyProject.Unit1` должен храниться в файле *MyProject.Unit1.pas*).

Причина такого расширения заключается в том, что имена модулей должны быть уникальными, а с учетом того, что все больше и больше модулей поставляется и Embarcadero, и сторонними производителями, это становится все более сложным. Все RTL модули и различные другие модули, поставляемые в составе библиотек продуктов, теперь следуют правилу имен модулей с точками, с определенными префиксами, обозначающими, например, область:

- `System` для базовой RTL
- `Data` для доступа к базам данных и тому подобное
- `FMX` для платформы FireMonkey, кроссплатформенной архитектуры для настольных приложений и мобильных устройств.
- `VCL` для библиотеки визуальных компонентов для Windows

примечание Обычно вы ссылаетесь на точечные имена модулей, включая библиотечные модули, с полным именем. Также возможно использовать в ссылке только последнюю часть имени (что позволяет обеспечить обратную совместимость со старым кодом), установив соответствующее правило в опциях проекта. Эта настройка называется “Unit scope names” и представляет собой список, разделенный точкой с запятой. Заметьте, однако, что использование этой функции замедляет компиляцию по сравнению с использованием полных имен модулей.

Подробнее о структуре модулей

Помимо секций интерфейса и реализации, модуль может иметь дополнительную секцию `инициализации` с некоторым кодом запуска, который будет выполняться при первой загрузке программы в память. Если имеется секция initialization, то можно также иметь секцию finalization, которая будет выполняться при завершении программы.

примечание Вы также можете добавить код инициализации в конструкторе класса, недавняя возможность языка, описанная в Гл. 12. Использование конструкторов классов помогает компоновщику удалить ненужный код, поэтому рекомендуется использовать конструкторы и деструкторы классов, а не старые секции инициализации и доработки. Как историческая справка, компилятор по-прежнему поддерживает использование `begin` ключевого слова вместо ключевого слова `initialization`. Аналогичное использование `begin` до сих пор является стандартным в исходных текстах проекта

Другими словами, общая структура модуля со всеми ее возможными секциями и некоторыми примерами элементов выглядит следующим образом:

```
unit unitName;

interface

// other units we refer to in the interface section
uses
  unitA, unitB, unitC;

// exported type definitions
type
  newType = TypeDefinition;

// exported constants
const
  Zero = 0;

// global variables
var
  Total: Integer;

// list of exported functions and procedures
procedure MyProc;
```

```
implementation

// other units we refer to in the implementation
uses
  unitD, unitE;

// hidden global variable
var
  PartialTotal: Integer;

// all the exported functions must be coded
procedure MyProc;
begin
  // ... code of procedure MyProc
end;
initialization
  // optional initialization code

finalization
  // optional clean-up code

end.
```

Назначение интерфейсной части модуля состоит в том, чтобы подробно описать, что содержится в модуле и может быть использовано основными программами и другими модулями, которые будут использовать этот модуль. В то же время секция реализации содержит гайки и болты модуля, которые скрыты от посторонних зрителей. Таким образом, Object Pascal может обеспечить так называемую инкапсуляцию даже без использования классов и объектов.

Как видите, `interface` модуля может объявлять несколько различных элементов, включая процедуры, функции, глобальные переменные и типы данных. Как правило, чаще всего используются типы данных. IDE автоматически помещает новый тип данных класса в модуль каждый раз, когда вы создаете визуальную форму. Однако, наличие определений формы, безусловно, не единственное применение для модулей в Object Pascal. Можно иметь только модули, с функциями и процедурами (традиционным образом) и с классами, которые не относятся к формам или другим визуальным элементам.

В пределах секций интерфейса или реализации, объявления для типов, переменных, констант и т.п. могут быть записаны в любом порядке и могут быть повторены несколько раз. Можно иметь несколько констант, некоторые типы, затем больше констант, другие переменные и секции для других типов. Единственное правило заключается в том, что для обращения к символу, он должен быть объявлен перед обращением к нему. Это причина, по которой вам часто приходится иметь несколько секций.

Раздел Uses

Раздел `uses` в начале секции интерфейса указывает на то, к каким другим модулям нам нужно получить доступ в разделе интерфейса. Сюда относятся модули, определяющие типы данных, на которые мы ссылаемся в определении типов данных этого блока, например, компоненты, используемые в форме, которую мы определяем.

Второй раздел `uses`, в начале секции реализации, указывает на дополнительные модули, доступ к которым нам нужен только в коде реализации. Когда вам нужно обратиться к другим модулям из кода процедур и методов, вы должны добавить элементы в это второе выражение use вместо первого. Все юниты, на которые вы ссылаетесь, должны присутствовать в каталоге проекта или в пути поиска каталогов.

совет Путь поиска для проекта можно задать в Параметрах проекта (Project Options). Система также учитывает модули в пути Библиотеки (Library path), который является глобальной настройкой IDE.

Программисты на Си++ должны знать, что строка `uses` не соответствует директиве include. Uses импортирует только предварительно скомпилированную часть интерфейса из перечисленных модулей. Секция реализации модуля учитывается только в том случае, если этот модуль

скомпилирован. Модули, на которые вы ссылаетесь, могут быть как в формате исходного кода (PAS), так и в скомпилированном формате (DCU).

Несмотря на редкое использование, в Object Pascal также была директива компилятора $INCLUDE, которая работает аналогично C/C++ include. Эти специальные включаемые файлы используются некоторыми библиотеками для обмена директивами компилятора или другими настройками между несколькими модулями, и обычно имеют расширение файла INC. Эта директива кратко рассмотрена в конце данной главы.

примечание Обратите внимание, что скомпилированные модули в Object Pascal совместимы только в том случае, если они собраны с одной и той же версией компилятора и системными библиотеками. Юнит, скомпилированный в более старой версии продукта, как правило, не совместим с более поздней версией компилятора. Обновления, являющиеся частью одного и того же релиза, поддерживают совместимость. Другими словами, модуль, собранный в версии 10.3.1, совместим со всеми версиями 10.3.x, но не с версиями 10.2 или 10.4.

Модули и область видимости

В Object Pascal модули являются ключом к инкапсуляции и видимости, и в этом смысле они, вероятно, даже более важны, чем ключевые слова public и private у класса. Область действия идентификатора (например, переменная, процедура, функция или тип данных) — это та часть кода, в которой идентификатор доступен или *видим*. Основное правило заключается в том, что идентификатор имеет значение только в пределах своей области действия, т.е. только в том модуле, функции или процедуре, в которой он объявлен. Нельзя использовать идентификатор, выходящий за его пределы.

примечание В отличие от C или C++, Object Pascal не имеет концепции общего блока кода, который может включать декларацию. Хотя вы можете использовать begin/end для создания составного высказывания, это не похоже на блок C или C++ со

фигурными скобками, который имеет свою собственную область видимости для внутренне объявленных переменных.

Обычно идентификатор виден только после его определения. Существуют техники в языке, которые позволяют декларировать идентификатор до его полного определения, но общее правило все равно применяется, если рассматривать как определения, так и декларации.

Учитывая, что записывать целую программу в один файл не имеет смысла, все же как правило, приведенное выше, меняется, когда вы используете несколько модулей? Короче говоря, при обращении к другому модулю с оператором uses, идентификаторы в разделе интерфейса этого модуля становятся видимыми для нового модуля.

С другой стороны, если вы объявляете идентификатор (тип, функцию, класс, переменную и т.д.) в интерфейсной части модули, то он становится видимым для любого другого модуля, ссылающегося на эту модуль. Если же вы объявляете идентификатор в части реализации модуля, то вместо этого он может быть использован только в этом модуле (и обычно называется *локальным идентификатором*).

Использование модулей, подобно пространствам имён

Мы видели, что оператор uses является стандартной техникой доступа к идентификаторам, объявленным в области видимости других модулей. В этот момент вы можете получить доступ к определениям модулей. Но может случиться, что два модуля, на которые вы ссылаетесь, объявят один и тот же идентификатор; то есть, у вас может быть два класса или две процедуры с одним и тем же именем.

В этом случае можно просто использовать имя юнита для префиксации названия типа или функции, определенной в устройстве. Например, вы можете обратиться к процедуре `ComputeTotal`, определенной в данном модуле `Calc` как `Calc.ComputeTotal`. Это не часто требуется, так как настоятельно не рекомендуется использовать один и тот же идентификатор для двух разных элементов одной и той же программы, если этого можно избежать.

Однако, если вы посмотрите на систему или сторонние библиотеки, вы найдете функции и классы с одинаковым именем. Хорошим примером являются визуальные элементы управления различных фреймворков пользовательского интерфейса. Когда вы видите ссылку на `TForm` или `TControl`, это может означать различные классы в зависимости от реальных единиц, на которые вы ссылаетесь.

Если один и тот же идентификатор предоставляется двумя модулями в операторе uses, то последний модуль, который используется, переопределяет символ, и будет тем, который использует компилятор. Другими словами, символы, определенные в последнем модуле в списке, выигрывают. Если вы просто не можете избежать такого сценария, рекомендуется символ писать с префиксом с именем модуля, чтобы код не зависел от порядка, в котором модули перечислены.

примечание Разработчики Delphi действительно пользуются возможностью иметь два класса с именем sme, с техникой, называемой *interposer classes,* которая объясняется далее в этой книге.

Файл программы

Как мы видели, приложение Delphi состоит из двух видов файлов с исходным кодом: один или несколько модулей и один, и только один, файл программы (сохраненный в файле

DPR). Модули можно считать вторичными файлами, на которые ссылается основная часть приложения - программа. Теоретически это так. На практике файл программы обычно представляет собой автоматически генерируемый файл с ограниченной ролью. Ему просто необходимо запустить программу, как правило, создающую и запускающую основную форму, в случае визуального приложения. Код файла программы можно отредактировать вручную, но он также изменяется автоматически с помощью некоторых опций проекта IDE (например, тех, которые относятся к объекту приложения и формам).

Структура файла программы обычно намного проще, чем структура единиц. Вот исходный код примерного файла программы (с пропущенными некоторыми необязательными стандартными единицами), который автоматически создается IDE для вас:

```
program Project1;

uses
  FMX.Forms,
  Unit1 in 'Unit1.PAS' {Form1};

begin
  Application.Initialize;
  Application.CreateForm (TForm1, Form1);
  Application.Run;
end.
```

Как видите, существует просто раздел uses и основной код приложения, ограниченный ключевыми словами begin/end. Оператор uses программы особенно важен, поскольку он используется для управления компиляцией и компоновкой приложения.

совет Список модулей в файле программы соответствует списку модулей, входящих в проект в IDE Project Manager. При добавлении модуля в проект IDE она автоматически добавляется в список в исходном файле программы. Обратное происходит, если его удалить из проекта. В любом случае, если Вы редактируете исходный код файла программы, список модулей в менеджере проектов обновляется соответствующим образом.

Директивы компилятора

Еще одним особенным элементом структуры программы (кроме ее реального кода), как уже упоминалось ранее, являются директивы компилятора. Это специальные инструкции компилятора, написанные в формате:

```
{$X+}
```

Некоторые директивы компилятора имеют один символ, как указано выше, с символом плюс или минус, указывающим, включена ли директива или неактивирована. Большинство директив также имеют более длинную и читабельную версию и используют символы `ON` и `OFF` для обозначения, если они активны. Некоторые директивы имеют только более длинный, описательный формат.

Директивы компилятора не генерируют скомпилированный код напрямую, а влияют на то, как компилятор генерирует код после встречи с директивой. Во многих случаях использование директивы компилятора является альтернативой изменению одной из настроек компилятора в Параметрах проекта IDE, хотя существуют сценарии, в которых необходимо применить определенную настройку компилятора только к модулю или фрагменту кода.

Я расскажу о конкретных директивах компилятора, когда это уместно при обсуждении особенностей языка, на которые они могут повлиять. В этом разделе я хочу упомянуть только пару директив, относящихся к потоку кода программы: условные определения и включения.

Условные определения

Определения условий позволяют указать компилятору на включение части исходного кода или проигнорировать его. В Delphi есть две разновидности условных определений. Традиционный `$IFDEF` и `$IFNDEF` и более новый, и гибкий `$IF`.

Эти условные определения могут зависеть от уже определенных символов или, для версии `$IF`, от значений констант. Определенные символы могут быть предопределены системой (как, например, символы компилятора и платформы), они могут быть определены в конкретной опции проекта, или они могут быть введены в код с помощью другой директивы компилятора, `$DEFINE`.

Традиционные `$IFDEF` и `$IFNDEF` имеют такой формат:

```
{$IFDEF TEST}
  // this is going to be compiled
{$ENDIF}

{$IFNDEF TEST}
  // this is not going to be compiled
{$ENDIF}
```

Вы также можете учесть альтернативы, используя директиву $ELSE для их разделения.

Как уже упоминалось, более гибкой является новая директива `$IF`, позволяющая использовать для условия константные выражения, такие как функции сравнения, которые могут ссылаться на любое константное значение в коде (например, проверка, не превышает ли версия компилятора заданное значение). Директива `$IF` закрывается директивой `$IFEND`:

```
{$IF (ProgramVersion > 2.0) }
  ... // this code executes if condition is true
{$ELSE}
  ... // this code execute if condition is false
{$IFEND}
```

Версии компилятора

Каждая версия компилятора Delphi имеет определенное определение, которое вы можете использовать для проверки того, компилируетесь ли вы с определенной версией продукта. Это может потребоваться, если вы используете функцию, введенную позже, но хотите убедиться, что код все еще компилируется для более старых версий.

Если вам нужен определенный код для некоторых последних версий Delphi, вы можете основывать ваши утверждения `$IFDEF` на следующих определениях:

Delphi 2007	VER180
Delphi XE	VER220
Delphi XE2	VER230
Delphi XE4	VER250
Delphi XE5	VER260
Delphi XE6	VER270
Delphi XE7	VER280
Delphi XE8	VER290
Delphi 2007	VER180
Delphi XE	VER220
Delphi XE2	VER230

Delphi XE4	VER250
Delphi XE5	VER260
Delphi XE6	VER270
Delphi XE7	VER280
Delphi XE8	VER290
Delphi 10 Seattle	VER300
Delphi 10.1 Berlin	VER310
Delphi 10.2 Tokyo	VER320
Delphi 10.3 Rio	VER330
Delphi 10.4 Sydney	VER340

Цифры десятков этих номеров версий указывают на реальную версию компилятора (например, 26 в Delphi XE5). Числовая последовательность не является специфичной для Delphi, и восходит к первому компилятору Turbo Pascal, опубликованному компанией Borland.

Можно также использовать внутреннюю константу версии в операторах `$IF`, с преимуществом использования оператора сравнения (`>=`), а не совпадения для конкретной версии. Константа версии называется `CompilerVersion` и в Delphi XE5 ей присваивается значение с плавающей точкой `26.0`. Так, например:

```
{$IF CompilerVersion >= 26)}
  // code to compile in Delphi XE5 or later
{$IFEND}
```

Аналогичным образом, вы можете использовать системные определения для различных платформ, для которых вы можете скомпилировать код, если вам нужен код, специфичный для конкретной платформы (как правило, это исключение в Object Pascal, а не общепринятая практика):

Windows (32 и 64 бита)	MSWINDOWS
Mac OS X	MACOS
iOS	IOS
Android	ANDROID
Linux	LINUX

Ниже приведен фрагмент кода с тестами, основанными на платформах, определенных выше, часть проекта `HelloPlatform`:

```
{$IFDEF IOS}
  ShowMessage ('Running on iOS');
{$ENDIF}

{$IFDEF ANDROID}
  ShowMessage ('Running on Android');
{$ENDIF}
```

Включение файлов

Другая директива, о которой я хочу рассказать здесь, это директива `$INCLUDE`, уже упоминавшаяся при обсуждении оператора uses. Эта директива позволяет ссылаться и включать фрагмент исходного кода в заданную позицию файла с исходным кодом. Иногда это используется для того, чтобы

иметь возможность включать один и тот же фрагмент в разные модули, в тех случаях, когда фрагмент кода определяет директивы компилятора и другие элементы, используемые непосредственно компилятором. При использовании модуля он компилируется только один раз. Когда вы включаете файл, этот код компилируется внутри каждого из модулей, к которым он добавляется (поэтому, как правило, следует избегать объявления нового идентификатора в включаемом файле).

Другими словами, вы, как правило, не должны добавлять никаких элементов языка и определений в включаемые файлы (в отличие от языка C), так как именно для этого и предназначаются модули. Так как же вы используете включаемый файл? Хорошим примером является набор директив компилятора, которые вы хотите включить в большинстве модулей, или некоторые дополнительные специальные определения.

Большие библиотеки часто используют для этой цели включаемые файлы, примером может служить библиотека FireDAC – библиотека для работы с базами данных, которая в настоящее время является частью системных библиотек. Другой пример, применяется в системных модулях RTL, - использование отдельных include для каждой платформы, при этом `$IFDEF` используется для условного включения только одной из них.

02: Переменные и типы данных

Object Pascal известен как язык с *сильной типизацией*. Переменные в Object Pascal объявлены как *тип данных (или тип данных, определяемый пользователем*). Тип переменной определяет значения, которые может иметь переменная, и операции, которые могут над ней выполняться. Это позволяет компилятору как выявлять ошибки в коде, так и генерировать для вас более быстрые программы.

Поэтому понятие типа в Pascal сильнее, чем в таких языках, как C или C++. В C, например, арифметические типы данных практически взаимозаменяемы.

Более поздние языки, основанные на том же синтаксисе, но нарушающие совместимость с C, такие как C# и Java, в отличие от C, приняли сильное представление Pascal о типе данных. В отличие от оригинальных версий BASIC, в них не было похожего понятия, и во многих современных скриптовых языках (JavaScript является очевидным примером) понятие типа данных очень сильно отличается.

примечание На самом деле, есть некоторые трюки для обхода безопасности типов, например, использование типов variant record. Использование этих трюков настоятельно не рекомендуется, и сегодня они применяются мало.

Как я уже упоминал, все динамические языки, начиная с JavaScript и далее, не имеют одинакового понятия типа данных, или (по крайней мере) имеют очень слабое представление о типах. В некоторых из этих языков тип выводится по значению, которое вы присваиваете переменной, и тип переменной может меняться со временем. Важно отметить, что типы данных являются ключевым элементом для обеспечения корректности работы большого приложения уже во время компиляции, а не для того, чтобы полагаться на проверки во время выполнения. Типы данных требуют большего порядка и структуры, а также некоторого планирования кода, который вы будете писать, что явно имеет свои преимущества и недостатки.

примечание Нет необходимости говорить о том, что я предпочитаю сильно типизированные языки, но в любом случае моя цель в этой книге - объяснить, как работает язык Object Pascal, а не доказывать, почему я считаю его таким замечательным языком программирования. Хотя я уверен, что вы получите это впечатление во время чтения текста.

Переменные и присвоения

Как и в других ярко языках с сильной типизацией, Object Pascal требует, чтобы все переменные были объявлены до их использования. Каждый раз, когда вы объявляете переменную, вы должны указывать тип данных. Вот некоторые объявления переменных:

```
var
  Value: Integer;
  IsCorrect: Boolean;
  A, B: Char;
```

Ключевое слово `var` может быть использовано в нескольких местах программы, например, в начале функции или процедуры, для объявления переменных локальными для этой

части кода, или внутри модуля для объявления глобальных переменных.

примечание Как мы скоро увидим, в последних версиях Delphi добавлена возможность объявлять переменную "inline", прямо вместе с операторами программирования. Это значительный отход от классического Паскаля.

После ключевого слова `var` идет список имен переменных, за которым следует двоеточие и имя типа данных. Можно написать несколько имен переменных в одной строке, как `A` и `B` в последнем выражении предыдущего фрагмента кода (стиль кодирования, который сегодня менее распространен, по сравнению с разбиением объявления на две отдельные строки: использование отдельных строк улучшает читабельность, сравнение версий и слияние).

После того, как вы определили переменную данного типа, вы можете выполнять над ней только те операции, которые поддерживаются ее типом данных. Например, можно использовать булевское значение при логической проверке, а целое значение в численном выражении. Нельзя смешивать, например, булевы значения и целые числа или любые несовместимые типы данных (даже если внутреннее представление может быть физически совместимым, как это бывает для булевых значений и целых чисел).

Самое простое присвоение – это присвоение фактического, конкретного значения. Например, вы можете захотеть, чтобы в переменной `Value` было числовое значение `10`. Но как выразить буквальные значения? Хотя это понятие может быть очевидным, на него стоит обратить внимание с некоторыми подробностями.

Литералы

Литеральное значение – это значение, которое вы вводите непосредственно в исходный код программы. Если вам нужно число со значением двадцать, вы можете просто написать:

```
20
```

Существует также альтернативное представление одного и того же числового значения, основанное на шестнадцатеричном значении, например:

```
$14
```

Это будут литеральные значения для целого числа (или одного из различных доступных типов целых чисел). Если вам нужно одно и то же числовое значение, но для литерального значения с плавающей точкой, то обычно после него добавляется нулевое десятичная цифра:

```
2.0
```

Литеральные значения не ограничиваются числами. Вы также можете иметь литералы - символы и строки. Для обоих используют одинарные кавычки (существует много других языков программирования, где надо использовать двойные кавычки для обоих, или одинарные кавычки для символов и двойные кавычки для строк):

```
// literal characters
'K'
#55

// literal string
'Marco'
```

Как видно выше, можно также указывать символы по их соответствующему цифровому значению (первоначально ASCII число, теперь значение точки кода Юникода), поставив # в начале к числу с символом, как в #32 (для пробела). Это полезно в основном для управляющих символов без текстового

представления в исходном коде, например, пробела или табуляции.

В случае, если вам нужна кавычка внутри строки, вам придется ее удвоить. Так что, если я хочу иметь свое имя и фамилию (написано с окончательной кавычкой, а не с акцентом), я могу написать:

```
'My name is Marco Cantu'''
```

Две кавычки означают кавычки внутри строки, а третья последовательная кавычка - конец строки. Также обратите внимание, что строковый литерал должен быть записан в одну строку, но вы можете соединить несколько строковых литералов, используя знак +. Если Вы хотите новую строку или перенос строки внутри строки, не пишите ее на двух строках, а системной константой `sLineBreak` (которая зависит от платформы) соединяйте эти два элемента, как в примере:

```
'Marco' + sLineBreak + 'Cantu'''
```

Оператор присваивания

Присвоения в Object Pascal используют оператор colon-equal (`:=`), необычная нотация для программистов, привыкших к другим языкам. Оператор `=`, используемый для присваиваний на многих других языках, в Object Pascal используется для проверки равенства.

исторически Оператор := происходит от предшественника Pascal, Algol, языка, о котором мало кто из современных разработчиков слышал (не говоря уже, использовать). Большинство современных языков избегают нотации := и отдают предпочтение нотации присваивания =.

Используя различные символы для присваивания и теста на равенство, компилятор Pascal (как и компилятор C) может быстрее транслировать исходный код, так как ему не нужно изучать контекст, в котором используется оператор для

определения его значения. Использование различных операторов также облегчает чтение кода. С самого начала Pascal выбрал два разных оператора, чем Си (и синтаксические производные, такие как Java, C#, JavaScript), которые используют = для присваивания и == для проверки равенства.

примечание Для полноты следует упомянуть, что в JavaScript также есть оператор === (выполняющий строгий тест на равенство типов и значений), но это то, что даже большинство JavaScript-программистов путают.

Два элемента присваивания часто называются *lvalue* и *rvalue*, для левого значения (переменной или ячейке памяти, которую Вы присваиваете) и для правого значения, для значения присваиваемых выражений. В то время как *rvalue* может быть выражением, *lvalue* должно относиться (прямо или косвенно) к ячейке памяти, которую Вы можете изменять. Существуют некоторые типы данных, которые характеризуются определенным поведением при присваивании, о котором я расскажу в свое время.

Другое правило заключается в том, что тип *lvalue* и *rvalue* должны совпадать, или должно существовать автоматическое преобразование между ними, как объяснено в следующем разделе.

Присвоение и преобразование

Используя простые присваивания, мы можем написать следующий код (который вы можете найти среди многих других фрагментов в этом разделе в проекте `VariablesTest`):

```
Value := 10;
Value := Value + 10;
IsCorrect := True;
```

Учитывая предыдущие объявления переменных, эти три присваивания корректны. Следующее утверждение, вместо

этого, некорректно, так как эти две переменные имеют разные типы данных:

```
Value := IsCorrect; // ошибка
```

Если попытаться скомпилировать этот код, компилятор выдаст ошибку с таким описанием:

```
[dcc32 Error]: E2010 Incompatible types: 'Integer' and 'Boolean'
```

Компилятор сообщает, что в коде что-то не так, а именно два несовместимых типа данных. Конечно, часто бывает возможно преобразовать значение переменной из одного типа в другой. В некоторых случаях преобразование происходит автоматически, например, если переменной с плавающей точкой присваивается целое значение (но, конечно, не противоположное). Обычно требуется вызвать специальную системную функцию, которая изменяет внутреннее представление данных.

Инициализация глобальных переменных

Для глобальных переменных можно присвоить начальное значение при объявлении переменной, используя нотацию константы присваивания, описанную ниже (=), вместо оператора присваивания (:=). Например, можно записать:

```
var
  Value: Integer = 10;
  Correct: Boolean = True;
```

Данная техника инициализации работает только для глобальных переменных, которые в любом случае инициализируются до их значений по умолчанию (например, до нуля для числа).

Инициализация локальных переменных

Переменные, объявленные в начале процедуры или функции, *не* инициализируются до значения по умолчанию и не имеют синтаксиса назначения. Для таких переменных часто стоит добавлять явный код инициализации в начале кода:

```
var
  Value: Integer;
begin
  Value := 0; // initialize
```

Опять же, если вы не инициализируете локальную переменную, но используете ее как есть, то переменная будет иметь совершенно случайное значение (в зависимости от содержимого байтов, расположенных в этих адресах памяти). В нескольких сценариях компилятор предупредит о потенциальной ошибке, но не всегда.

Другими словами, если вы пишете:

```
var
  Value: Integer;
begin
  ShowMessage (Value.ToString); // Value is undefined
```

Вывод будет полностью случайным значением, содержимым байт, расположенных в памяти переменной `Value`, рассматриваемой как целое число.

Inline переменные

Последние версии Delphi (начиная с 10.3 Rio) имеют дополнительную концепцию, которая революционизирует подход к объявлению переменных, использовавшийся со времен ранних компиляторов Pascal и TurboPascal, - объявления переменных inline.

Новый синтаксис декларации inline-переменной позволяет объявлять переменную непосредственно в блоке кода (позволяя также использовать несколько символов, как и для традиционных деклараций переменных):

```
procedure Test;
begin
  var I, J: Integer;
  I := 22;
  J := I + 20;
  ShowMessage (J.ToString);
end;
```

Хотя это может показаться незначительным отличием, есть несколько побочных эффектов этого изменения, связанных с инициализацией, наследованием типа и временем жизни переменной. Я расскажу об этом в следующих разделах этой главы.

Инициализация inline-переменных

Первое существенное изменение по сравнению со старой моделью декларирования заключается в том, что в одном операторе можно выполнить встраиваемое объявление и инициализацию переменной. Это делает вещи более читабельными по сравнению с инициализацией нескольких переменных в начале функции, как было указано в предыдущем разделе:

```
procedure Test;
begin
  var I: Integer := 22;
  ShowMessage (I.ToString);
end;
```

Основное преимущество здесь заключается в том, что если значение переменной становится доступным только позже в блоке кода, а не устанавливается начальное значение (например, `0` или `nil`) и позднее присваивается действительное значение, то можно отложить объявление переменной до

момента, когда можно вычислить правильное начальное значение, как показано ниже, как показано `K ниже`:

```
procedure Test1;
begin
  var I: Integer := 22;
  var J: Integer := 22 + I;
  var K: Integer := I + J;
  ShowMessage (K.ToString);
end;
```

Другими словами, если раньше все локальные переменные, были видимы во всем блоке кода, то теперь inline переменная видна только с позиции ее объявления и до конца блока кода. Другими словами, переменная K не может быть использована в первых двух строках блока кода, прежде чем ей будет присвоено соответствующее значение.

Вывод типа для inline-переменных

Еще одним существенным преимуществом встроенных переменных является то, что компилятор теперь может при определенных условиях сделать вывод о типе встроенной переменной, на основании типа присвоенного выражения или значения. Вот очень простой пример:

```
procedure Test;
begin
  var I := 22;
  ShowMessage (I.ToString);
end;
```

Для определения типа переменной анализируется тип выражения *rvalue* (то есть то, что написано после `:=`). При присвоении строковой константы, переменная будет строкового типа, но тот же самый анализ происходит и при присваивании результата функции или сложного выражения.

Обратите внимание, что переменная по-прежнему сильно типизирована, так как компилятор определяет тип данных и присваивает его при компиляции, и его нельзя изменить

присвоением нового значения. Вывод типа — это только удобство, чтобы вводить меньше кода (что-то, относящееся к сложным типам данных, например, тип generic), но не меняющее статическую и сильно типизированную природу языка и не вызывающее замедления во время выполнения.

примечание При выведении типа некоторые из типов данных "расширяются" до более крупного типа, как в случае, описанном выше, когда числовое значение 22 (а `ShortInt`) расширяется до `Integer`. Как правило, если правый тип выражения является целочисленным и меньше 32 бит, то переменная будет объявлена как 32-битное `Integer`. Вы можете использовать явный тип, если хотите конкретный, меньший, числовой тип.

Константы

Object Pascal также позволяет декларировать константы. Это позволяет давать осмысленные имена значениям, которые не изменяются во время выполнения программы (и, возможно, уменьшает размер, не дублируя константы в скомпилированном коде).

Для объявления константы не требуется указывать тип данных, а только присваивать начальное значение. Компилятор посмотрит на значение и автоматически выведет нужный тип данных. Приведем несколько примеров деклараций (также из проекта приложения `VariablesTest`):

```
const
  Thousand = 1000;
  Pi = 3.14;
  AuthorName = 'Marco Cantu';
```

Компилятор определяет тип константных данных по их значению. В приведенном выше примере константа Thousand принимается за тип `SmallInt`, наименьший интегральный тип, который может ее содержать. Если вы хотите указать компилятору использовать определенный тип, вы можете просто добавить имя типа в объявление, как ниже:

```
const
  Thousand: Integer = 1000;
```

предупреждение Рассчитывать в программе на предположения, как правило, плохо, и компилятор со временем может измениться и не совпадать с предположениями разработчика. Если у вас есть способ выразить код более четко и без предположений, сделайте это!

Когда вы объявляете константу, компилятор может выбрать, присваивать ли ей место в памяти и сохранять ли там ее значение, или дублировать действительное значение каждый раз, когда используется константа. Такой второй подход имеет смысл особенно для простых констант.

Объявив константу, вы можете использовать ее почти как любую другую переменную, но не можете присвоить ей новое значение. Если вы попытаетесь, то получите ошибку компилятора.

примечание Удивительно, но Object Pascal позволяет изменять значение типизированной константы во время выполнения программы, как будто это была переменная, но только если включена директива компилятора $J, или используется соответствующая опция компилятора Assignable typed constants. Такое необязательное поведение включено для обратной совместимости кода, который был написан со старым компилятором. Очевидно, что это не рекомендуемый стиль кодирования, и я отметил его в большей степени в качестве исторического анекдота о подобных техниках программирования.

Inline-константы

Как мы видели ранее для переменных, теперь можно также встроить объявление константы. Это может быть применено к типизированным константам или нетиповым константам, в этом случае тип выводится (возможность, которая была доступна для констант уже долгое время). Простой пример приведен ниже:

```
begin
  // some code
  const M: Integer = (L + H) div 2; // identifier with type specifier
  // some code
  const M = (L + H) div 2; // identifier without type specifier
```

Обратите внимание, что в то время, как объявление обычной константы позволяет присвоить только значение константы, для объявления inline константы можно использовать любое выражение.

Константы Resource String

Хотя это несколько более продвинутая тема, когда вы определяете строковую константу, вместо написания стандартного константного объявления вы можете использовать специальную директиву, `resourcestring`, которая указывает компилятору и компоновщику обращаться со строкой как с ресурсом Windows (или эквивалентной структурой данных на платформах, не относящихся к Windows, поддерживаемых Object Pascal):

```
const
  AuthorName = 'Marco';

resourcestring
  StrAuthorName = 'Marco';

begin
  ShowMessage (StrAuthorname);
```

В обоих случаях вы определяете константу, т.е. значение, которое не меняется во время выполнения программы. Разница заключается только во внутренней реализации. Строковая константа, определенная директивой `resourcetring`, хранится в ресурсах программы, в таблице строк.

Вкратце, преимущества использования ресурсов — это более эффективная работа с памятью под управлением Windows, соответствующая реализация Delphi для других платформ, а также лучший способ локализации программы (трансляция строк на другой язык) без необходимости изменения ее исходного кода. Как правило, вы должны использовать `Resourcestring` для любого текста, который показывается пользователям и может потребовать перевода, и внутренние

константы для каждой другой внутренней строки программы, например, фиксированное имя конфигурационного файла.

совет Редактор IDE имеет автоматический *рефакторинг,* с помощью которого можно заменить строковую константу в коде на соответствующую декларацию `Resourcestring`. Поместите курсор редактора внутри строкового литерала и нажмите Ctrl+Shift+L, чтобы активировать этот рефакторинг.

Жизненный цикл и видимость переменных

В зависимости от того, как вы определяете переменную, она будет использовать различные места размещения в памяти и оставаться доступной в течение разного времени (то, что обычно называется *время жизни* переменной) и будет доступна в разных частях вашего кода (функция, называемая термином *область видимости*).

Пока мы не можем дать полного описания всех вариантов на этой ранней стадии книги, но мы, безусловно, можем рассмотреть наиболее релевантные случаи:

- **Глобальные переменные**: Если вы объявляете переменную (или любой другой идентификатор) в интерфейсной части модуля, то сфера ее действия распространяется на любой другой модуль, использующий объявленную переменную. Память для этой переменной выделяется при запуске программы и существует до ее окончания. Ей можно присвоить значение по умолчанию или использовать секцию инициализации модули, если требуется вычислить начальное значение более сложным способом.
- **Глобальные скрытые переменные**: Если вы объявляете переменную в части реализации модуля, вы не можете

использовать ее за пределами этого модуля, но вы можете использовать ее в любом блоке кода и процедуры, определенной внутри модуля, начиная с позиции объявления. Такая переменная использует глобальную память и имеет то же время жизни, что и первая группа; единственное отличие заключается в ее видимости. Инициализация такая же, как и у глобальной переменной.

- **Локальные переменные**: Если вы объявляете переменную внутри блока, определяющего функцию, процедуру или метод, вы не можете использовать эту переменную вне этого блока кода. Область действия идентификатора охватывает всю функцию или метод, включая вложенные процедуры (если только идентификатор с таким же именем во вложенной процедуре не скрывает внешнее определение). Память для этой переменной выделяется на стеке, когда программа выполняет определяющую ее рутину. По окончании выполнения рутины память на стеке автоматически освобождается.
- **Локальные встроенные переменные**: Если вы объявили встроенную inline переменную в блоке, определяющем функцию, процедуру или метод, то дополнительным ограничением, по сравнению с традиционной локальной переменной, является то, что видимость начинается со строки, в которой объявлена переменная, и продолжается до конца функции или метода.
- **Блочные встроенные переменные**: Если вы объявите inline переменную внутри блока кода (т.е. вложенного начального оператора), то видимость переменной будет ограничена этим вложенным блоком. Это соответствует тому, что происходит в большинстве других языков

программирования, но было введено в Object Pascal только с недавним синтаксисом объявления inline переменных.

примечание Для блочно-скопических встроенных переменных время жизни не только видимости, но и переменной ограничено блоком. Управляемый тип данных (например, интерфейс, строка или управляемая запись) будет освобождена в конце подблока, а не в конце процедуры или метода. Это справедливо и для временных переменных, созданных для хранения результатов выражения.

Любые объявления в интерфейсной части модуля доступны из любой части программы, которая включает модуль в пункт uses. Переменные классов формы объявляются аналогичным образом, так что вы можете ссылаться на форму (и ее публичные поля, методы, свойства и компоненты) из кода любой другой формы. Конечно, это плохая практика программирования - объявлять все глобально. Кроме очевидных проблем с потреблением памяти, использование глобальных переменных усложняет поддержание и обновление программы. Одним словом, следует использовать как можно меньшее количество глобальных переменных.

Типы данных

В Pascal существует несколько предопределенных типов данных, которые можно разделить на три группы: *порядковые типы*, *реальные типы* и *строки*. Порядковые и реальные типы мы обсудим в следующих разделах, а строки будут специально рассмотрены в Главе 6.

Delphi также включает в себя *нетипизированный* тип данных, называемый *вариантом (Variant)*, и другие "гибкие" типы, такие как `TValue` (часть расширенной поддержки RTTI). Некоторые из этих более продвинутых типов данных будут рассмотрены ниже в главе 5.

Порядковый и цифровой типы

Типы заказов основаны на концепции заказа или последовательности. Вы можете не только сравнить два значения, чтобы увидеть, какое из них выше, но и запросить следующие или предыдущие значения любого значения и вычислить самые низкие и самые высокие из возможных значений, которые может представлять тип данных.

Три наиболее важных предопределенных ординарных типа - Integer, Boolean, и Char (символ). Тем не менее, существуют и другие родственные типы, которые имеют одинаковое значение, но отличаются внутренним представлением и поддерживают другой диапазон значений. В следующей таблице перечислены ординарные типы данных, используемые для представления чисел:

Размер	Со знаком (Signed)	Без знака (Unsigned)
8 bits	`ShortInt`: -128 до 127	Byte: 0 до 255
16 bits	`SmallInt`: -32768 до 32767 (-32K до 32K)	Word: 0 до 65,535 (0 до 64K)
32 bits	`Integer`: -2,147,483,648 до 2,147,483,647 (-2GB до +2GB)	Cardinal: 0 до 4,294,967,295 (0 до 4 GB)
64 bits	Int64: -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807	UInt64: 0 до 18,446,744,073,709,551,615 (вы сможете его прочитать?!)

Как видите, эти типы соответствуют различным представлениям чисел, в зависимости от количества битов, используемых для выражения значения, и наличия или отсутствия знакового бита. Значения со знаком могут быть

положительными или отрицательными, но иметь меньший диапазон значений (половина соответствующего беззнакового значения), так как для хранения самого значения доступен на один бит меньше.

Тип `Int64` представляет собой целое число, вмещающее до 18 цифр. Этот тип полностью поддерживается некоторыми подпрограммами порядкового типа (такими как `High` и `Low`), цифровыми подпрограммами (такими как `Inc` и `Dec`) и подпрограммами преобразования строк (такими как `IntToStr`) библиотеки времени выполнения.

Псевдонимы целых типов

Если вам трудно запомнить разницу между `ShortInt` и `SmallInt` (включая тот, который фактически меньше), вместо фактического типа, вы можете использовать один из предопределенных псевдонимов, объявленных в модуле `System`:

```
type
  Int8   = ShortInt;
  Int16  = SmallInt;
  Int32  = Integer;
  UInt8  = Byte;
  UInt16 = Word;
  UInt32 = Cardinal;
```

Опять же, эти типы не добавляют ничего нового, но, вероятно, проще в использовании, так как легко запомнить реальный размер Int16, а не SmallInt. Эти псевдонимы типов также проще использовать разработчикам, пришедшим с языка Си и других языков, использующих сходные имена типов.

Целочисленный тип, 64-разрядный, NativeInt и LargeInt.

В 64-битных версиях Object Pascal вы можете удивиться, узнав, что тип Integer все еще 32-битный. Это так, потому что это наиболее эффективный тип для числовой обработки на уровне процессора.

А вот тип Pointer (подробнее об указателях позже) и другие связанные с ним опорные типы, являются 64-битными. Если вам нужен цифровой тип, который адаптируется к размеру указателя и родной платформе процессора, вы можете использовать два специальных алиасных типа `NativeInt` и `NativeUInt`. Они имеют одинаковый размер с указателями на конкретной платформе (т.е. 32 бита на 32-битных платформах и 64 бита на 64-битных платформах).

Немного другой сценарий происходит с типом `LargeInt`, который часто используется для сопоставления с нативными функциями API платформы. Он имеет 32 бита на 32-битных платформах и на Windows 32 бита, в то время как на 64-битной платформе ARM он составляет 64 бита. Лучше держаться от него подальше, если он не нужен специально для нативного кода, чтобы он адаптировался к базовой операционной системе.

Integer Type Helper

Хотя в языке Object Pascal типы Integer рассматриваются отдельно от объектов, с переменными (и значениями констант) этих типов можно работать с операциями, которые применяются с помощью "точечной нотации". Эта нотация обычно используется для применения методов к объектам.

примечание Технически эти операции с нативными типами данных определяются с помощью "intrinsic record helpers". Помощники классов и записей описаны в Главе 12. Короче говоря, вы можете настроить операции, применимые к основным типам данных. Разработчики-эксперты заметят, что операции типа определяются как статические методы класса в хелпере соответствующих внутренних записей.

Вы можете увидеть пару примеров в следующем коде, извлеченном из `IntegerTest`

```
var
  N: Integer;
begin
  N := 10;
  Show (N.ToString);

  // display a constant
  Show (33.ToString);

  // type operation, show the bytes required to store the type
  Show (Integer.Size.ToString);
```

примечание Функция `Show`, используемая в этом фрагменте кода, является простой процедурой, используемой для отображения вывода некоторой строки в памяти, чтобы избежать необходимости закрывать несколько диалогов `ShowMessage`. Побочным преимуществом этого подхода является то, что он упрощает копирование вывода и вставку в текст (как я сделал это ниже). Вы увидите, что этот подход используется в большинстве демок этой книги.

Программа выдает следующее:

```
10
33
4
```

Учитывая, что эти операции очень важны (больше, чем другие, входящие в библиотеку времени выполнения), их стоит перечислить здесь:

ToString	Преобразование в число в строке с использованием десятичного формата
ToBoolean	Преобразование в булевский тип

ToHexString	Преобразование в строку с использованием шестнадцатеричного формата
ToSingle	Преобразование в единственный тип данных с плавающей запятой
ToDouble	Преобразование в тип данных с двойной плавающей точкой
ToExtended	Преобразование в расширенный тип данных с плавающей точкой

Первая и третья операции преобразуются в число в строку с помощью десятичной или шестнадцатеричной операции. Вторая операция - приведение к булеву, а последние три операции - приведение к типам с плавающей точкой, описанным

Существуют и другие операции, которые можно применить к типу Целого числа (и большинству других числовых типов), например:

Size	Количество байт, требуемое для хранения переменной данного типа
Parse	Преобразовывать строку в числовое значение, которое она представляет, вызывая исключение, если строка не представляет собой число
TryParse	Попытка преобразовать строку в число

Стандартные операции порядкового типа

Помимо операций, определенных помощниками типа Integer и перечисленных выше, существует несколько стандартных и "классических" функций, которые можно применять к любому порядковому типу (а не только к числовому). Классическим примером является запрос информации о самом типе с помощью функций SizeOf, High и Low. Результатом работы системной функции SizeOf (которую можно применить к любому типу данных языка) является целое число, указывающее на количество байт, необходимое для представления значений данного типа (точно так же, как показанная выше вспомогательная функция Size).

Системные процедуры, которые работают с ординарными типами, показаны в следующей таблице:

Dec	Уменьшает переменную, передаваемую в качестве параметра, на один или на значение дополнительного второго параметра.
Inc	Увеличивает переменную, передаваемую в качестве параметра, на единицу или на указанное значение
Odd	Возвращает True, если аргумент является нечетным числом. Для проверки четных чисел следует использовать выражение not (`not Odd`)
Pred	Возвращает значение перед аргументом в порядке, определяемом типом данных, предшественник

Succ	Возвращает значение после аргумента, преемник
Ord	Возвращает номер, указывающий порядок следования аргумента в наборе значений типа данных (используется для нечисловых порядковых типов)
Low	Возвращает наименьшее значение в диапазоне порядкового типа, переданного в качестве параметра.
High	Возвращает наивысшее значение в диапазоне ординарного типа данных

Примечание Программисты на языках Си и Си++ должны обратить внимание, что два варианта процедуры Inc, с одним или двумя параметрами, соответствуют операторам ++ и += (то же самое относится и к процедуре Dec, которая соответствует операторам -- и -=). Компилятор Object Pascal оптимизирует эти операции инкремента и декремента, аналогично тому, как это делают компиляторы С и С++.

Тем не менее, Delphi предлагает только преинкремент и предекремент, а не постинкремент и постдекремент, и операция не возвращает значения.

Обратите внимание, что некоторые из этих подпрограмм автоматически вычисляются компилятором и заменяются на их значение. Например, если вы вызываете `High(X)`, где `X` определено как `Integer`, компилятор заменяет выражение на максимально возможное значение типа данных `Integer`.

В проект приложения `IntegersTest` я добавил событие с некоторыми из этих функций ординарного типа:

```
var
  N UInt16;
begin
  N := Low (UInt16);
  Inc (N);
```

```
  Show (N.ToString);
  Inc (N, 10);
  Show (N.ToString);
  if Odd (N) then
    Show (N.ToString + ' is odd');
```

Вот, что вы должны увидеть на выходе:

```
1
11
11 is odd
```

Вы можете изменить тип данных с `UInt16` на Integer или другие порядковые типы, чтобы посмотреть, как изменяется выходной сигнал.

Операции Out-of-Range (вне диапазона)

Переменная типа `n` выше имеет только ограниченный диапазон допустимых значений. Если присваиваемое ей значение отрицательное или слишком большое, это приводит к ошибке. На самом деле существует три различных типа ошибок, с которыми можно столкнуться при выполнении операций вне диапазона.

Первый тип ошибки — это ошибка компилятора, которая возникает при присвоении константного значения (или константного выражения), находящегося вне диапазона. Например, если добавить в код выше:

```
N := 100 + High (N);
```

то компилятор выдаст ошибку:

```
[dcc32 Error] E1012 Constant expression violates subrange bounds
 (Постоянное выражение нарушает границы поддиапазона).
```

Второй сценарий происходит, когда компилятор не может предвидеть состояние ошибки, так как это зависит от выполнения программы. Предположим, что мы пишем (в одном и том же куске кода):

```
  Inc (N, High (N));
  Show (N.ToString);
```

Компилятор не даст ошибку, так как происходит вызов функции, а компилятор заранее не знает, как это повлияет (а ошибка также будет зависеть от исходного значения `n`). В этом случае есть две возможности. По умолчанию, если вы скомпилируете и запустите это приложение, то в переменной получите совершенно нелогичное значение (в этом случае операция приведет к вычитанию 1!). Это самый худший сценарий, так как вы не получите ошибку, но ваша программа не корректна.

Что можно сделать (и настоятельно рекомендуется), так это включить опцию компилятора "Overflow check" (`{$Q+}` или `{$OVERFLOWCHECKS ON}`), которая защитит от аналогичной операции *переполнения* и вызовет ошибку, в данном конкретном случае "Целочисленное переполнение" (“Integer overflow”).

Boolean

Логические значения `True` и `False` представлены с помощью булевого типа. Это также тип условия в условных операторах, как мы увидим в следующей главе. Тип `Boolean` может иметь только одно из двух возможных значений `True` или `False`.

внимание Для совместимости с COM и OLE автоматизацией от Microsoft, типы данных `ByteBool`, `WordBool` и `LongBool` представляют значение True с значением -1, в то время как значение False по-прежнему равно 0. Опять же, в общем случае, следует игнорировать эти типы и избегать всех низкоуровневых булевых манипуляций и числового отображения, если только в этом нет абсолютной необходимости.

В отличие от языка Си и некоторых его производных языков, Boolean является перечисляемым типом в Object Pascal, нет прямого приведения к числовому значению, представляющему `boolean` переменную, и вы не должны злоупотреблять прямым приведением типов при попытке преобразования `boolean` в числовое значение. Однако, помощники `boolean` типов (Boolean

type helpers) включают в себя функции `ToInteger` и `ToString`. О перечисляемых типах я расскажу позже в этой главе.

Обратите внимание, что использование `ToString` возвращает строку с числовым значением булевой переменной. В качестве альтернативы можно использовать глобальную функцию `BoolToStr`, установив второй параметр в True, чтобы указать на использование булевых строк (*'True'* и *'False'*) для вывода. (См. пример в разделе "Операции типа char" ниже).

Символы

Символьная переменная определяется с помощью типа Char. В отличие от старых версий, сегодня язык использует тип Char для представления двухбайтовых символов Unicode (также представленных типом `WideChar`).

примечание В компиляторе Delphi до сих пор существует различие между `AnsiChar` для однобайтных ANSI-символов и `WideChar` для Unicode-символов, при этом тип `Char` определяется как псевдоним последнего. Рекомендуется ориентироваться на `WideChar` и использовать тип данных `Byte` для однобайтовых элементов. Однако, действительно, начиная с Delphi 10.4, тип `AnsiChar` стал доступен на всех компиляторах и платформах для лучшей совместимости с существующим кодом.

Для знакомства с символами в Юникоде, включая определение точки кода и определение суррогатных пар (наряду с другими дополнительными темами), вы можете прочитать Главу 6. В этой главе я просто сосредоточусь на основных понятиях типа Char.

Как я уже упоминал ранее при рассмотрении буквенных значений, символы-константы могут быть представлены с символической нотацией, как `'k'`, или с числовой нотацией, как `#78`. Последняя также может быть выражена с помощью системной функции `Chr`, как `Chr(78)`. Противоположное преобразование может быть сделано с помощью функции Ord.

Обычно лучше использовать символьную нотацию при указании букв, цифр или символов.

При обращении к специальным символам, например, к управляющим символам, в таблице символов ниже `#32`, обычно используется числовая нотация. Следующий список включает некоторые из наиболее часто используемых специальных символов:

`#8`	backspace
`#9`	tab
`#10`	new line
`#13`	CR, возврат каретки
`#27`	escape

Операции с типом данных Char

Как и другие порядковые типы, тип Char имеет несколько предопределенных операций, которые можно применить к переменным этого типа с помощью точечной нотации. Эти операции опять-таки определяются внутренним хелпером.

Однако сценарий использования совершенно другой. Во-первых, чтобы использовать эту функцию, необходимо *включить* ее, поместив модуль Character в оператор `uses`. Во-вторых, вместо нескольких функций преобразования, помощник для типа Char включает в себя пару дюжин.

Специфические для Юникода операции, включают такие тесты, как `IsLetter`, `IsNumber` и `IsPunctuation`, и преобразования, такие как `ToUpper` и `ToLower`. Пример взят из прикладного проекта `CharsTest`:

```
uses
  Character;
...
var
  Ch: Char;
begin
```

```
  Ch := 'a';
  Show (BoolToStr(Ch.IsLetter, True));
  Show (Ch.ToUpper);
```

Вывод этого кода:

```
True
A
```

примечание Операция ToUpper хелпера типа Char полностью работает с Unicode. Это означает, что если вы передадите букву с акцентом типа ù, то результат будет Ù. Некоторые из традиционных RTL-функций не столь умны и работают только для простых ASCII-символов. До сих пор они не модифицировались, так как соответствующие операции в Юникоде выполняются значительно медленнее.

Char как порядковый тип данных

Тип данных Char достаточно большой, но все равно это порядковый тип, поэтому вы можете использовать на нем функции `Inc` и `Dec` (чтобы перейти к следующему или предыдущему символу или двигаться вперед на заданное количество элементов, как мы видели в разделе "Операции стандартных порядковых типов") и использовать в циклах for со счетчиком типа Char (подробнее `для` циклов в следующей главе).

Вот простой фрагмент, используемый для отображения нескольких символов, полученных путем увеличения значения с начальной точки:

```
var
  Ch: Char;
  Str1: string;
begin
  Ch := 'a';
  Show (Ch);
  Inc (Ch, 100);
  Show (Ch);

  Str1 := '';
  for Ch := #32 to #1024 do
    Str1 := Str1 + Ch;
  Show (Str1)
```

Цикл for для проекта приложения CharsTest добавляет много текста в строку, что делает вывод довольно длинным. Он начинается со следующих строк текста:

```
a
Å
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abc
defghijklmnopqrstuvwxyz{|}~
// few more lines omitted...
```

// еще несколько строк опущены.../

Преобразование с помощью Chr

Мы видели, что существует функция Order, которая возвращает числовое значение (точнее, точку кода Юникода) символа. Существует также противоположная функция, с помощью которой можно получить символ, соответствующий точке кода, т.е. специальная функция Chr.

32-битные символы

Хотя тип Char по умолчанию теперь представляет WideChar, стоит отметить, что в Delphi также определен 4-байтовый символьный тип UCS4Char, определенный в модуле System как:

```
type
  UCS4Char = type LongWord;
```

Это определение типа и соответствующее ему определение для UCS4String (определяемого как массив UCS4Char) мало используются, но они являются частью среды исполнения языка и используются в некоторых функциях модуля Character.

Типы с плавающей точкой

В то время как целые числа различного рода могут быть представлены порядковым набором значений, вещественные

числа с плавающей точкой не являются порядковыми (они имеют понятие порядка, но не понятие последовательности элементов) и представлены некоторой приблизительной величиной, с некоторой погрешностью в их представлении.

Числа с плавающей точкой реализуются в различных форматах, в зависимости от количества байтов, используемых для их представления, и качества аппроксимации. Ниже приведен список типов данных с плавающей точкой в Object Pascal:

Single	Наименьший размер хранилища задается числами single, которые реализуются с 4-байтовым значением. Имя указывает на значение с плавающей точкой одинарной точности, а на других языках тот же тип указывается с помощью имени `float`.
Double	Это числа с плавающей точкой, реализованные с 8 байтами. Имя указывает на значение с плавающей точкой двойной точности и присутствует во многих языках. `Double` является наиболее часто используемым типом данных с плавающей точкой, а также псевдонимом старого типа Pascal, называемого `Real`.
Extended	Это числа, реализованные 10 байтами в оригинальном компиляторе Delphi Win32, но этот тип доступен не на всех платформах (на некоторых, например, Win64, он возвращается в `Double`, а на Mac OS X - в 16 байтов). Другие языки программирования называют этот тип данных Long Double

Все они представляют собой типы данных с плавающей запятой с различной точностью, которые соответствуют стандартным представлениям чисел с плавающей запятой IEEE и напрямую поддерживаются процессором (или, если быть точным, FPU, устройством работы с плавающей запятой), для максимальной скорости.

Существует также два специфических неординальных числовых типа данных, которые можно использовать для представления чисел с точным, а не приблизительным представлением:

Comp	Описывает очень большие целые числа, использующие 8 байт или 64 бита (которые могут содержать числа с 18-ю цифрами после запятой). Идея заключается в том, чтобы представлять большие числа без потери точности, в отличие от соответствующих значений с плавающей точкой.
Currency	Указывает на десятичное значение с фиксированной точкой с четырьмя знаками после запятой и тем же 64-битным представлением, что и тип Comp. Как следует из названия, тип данных Currency был добавлен для работы с очень точными денежными значениями, с четырьмя знаками после запятой (опять же без потери точности при расчетах).

Все эти не-ординальные типы данных не имеют понятий Low, High или `Order`. Вещественные типы представляют (теоретически) бесконечное множество чисел; порядковые типы представляют собой фиксированное множество значений.

Чем отличаются значения с плавающей точкой?

Позвольте мне продолжить объяснение. Когда у вас есть целое число 23, вы можете определить, что является следующим значением. Целые числа конечные (они имеют определенный диапазон и порядок). Числа с плавающей точкой бесконечны даже в небольшом диапазоне и не имеют порядка: сколько же на самом деле значений между 23 и 24? И какое число следует за 23,46? 23.47, 23.461 или 23.4601? Это действительно невозможно узнать!

По этой причине, хотя имеет смысл задать порядковую позицию символа `'w'` в диапазоне типа данных `Char`, нет никакого смысла задавать тот же самый вопрос о 7143.1562 в диапазоне типа данных с плавающей точкой. Хотя вы действительно можете знать, имеет ли одно вещественное число большее значение, чем другое, нет смысла задавать вопрос о том, сколько вещественных чисел существует перед данным числом (в этом и заключается смысл функции `Order`).

Другой ключевой концепцией, стоящей за значениями с плавающей точкой, является то, что их реализация не может точно отражать все числа. Часто бывает так, что результат вычисления, который вы ожидаете получить в виде конкретного числа (иногда целого), на самом деле может быть его приблизительным значением. Рассмотрим этот код, взятый из проекта приложения `FloatTest`:

```
var
  S1: Single;
begin
  S1 := 0.5 * 0.2;
  Show (S1.ToString);
```

Можно ожидать, что результат будет `0.1`, в то время как на самом деле вы получите что-то вроде

`0.100000001490116`. Это близко к ожидаемому значению, но не совсем. Нет нужды говорить, что, если округлить результат, вы получите ожидаемое значение. Если вместо этого использовать переменную типа Double, то результат будет `0.1`, как показывает прикладной проект `FloatTest`.

Примечание Здесь у меня нет времени на углубленное обсуждение математики с плавающей точкой на компьютерах, поэтому я сокращаю это обсуждение, но если вас интересует эта тема с точки зрения языка Object Pascal, я могу порекомендовать вам отличную статью с покойного Руди Велтхайса на сайте http://rvelthuis.de/articles/articles-floats.html.

Хелперы для вещественных типов и модуль Math

Как видно из приведенного выше фрагмента кода, типы данных с плавающей точкой также имеют хелперов, позволяющих применять операции непосредственно к переменным, как если бы они были объектами. На самом деле, список операций для чисел с плавающей точкой на самом деле довольно длинный.

Вот список операций над экземплярами для типа single (некоторые операции очевидны из их названий, другие немного более загадочны, описание вы можете найти в документации):

```
Exponent              Fraction         Mantissa
Sign                  Exp              Frac
SpecialType           BuildUp          ToString
IsNan                 IsInfinity       IsNegativeInfinity
IsPositiveInfinity    Bytes            Words
```

Библиотека времени исполнения также имеет `модуль Math`, который определяет продвинутые математические процедуры, охватывающие тригонометрические функции (такие как функция `ArcCosh`), финансы (такие как функция `InterestPayment`) и статистику (такие как процедура `MeanAndStdDev`). Существует несколько таких процедур, некоторые из которых мне кажутся

довольно странными, например, функция `MomentSkewKurtosis` (узнайте сами, что это такое).

Модуль `System.Math` очень богат возможностями, но вы также найдете множество дополнительных коллекций математических функций для Object Pascal от внешних поставщиков.

Простые пользовательские типы данных

Наряду с понятием типа, одной из великих идей, введенных Никлаусом Виртом в исходный язык Pascal, была возможность определения новых типов данных в программе (то, что мы сегодня воспринимаем как нечто само собой разумеющееся, но не очевидное в его время). Вы можете определять свои собственные типы данных с помощью *определений типов, таких как* поддиапазонные типы, типы массивов, типы записей, перечисляемые типы, типы указателей и множественные типы. Наиболее важным пользовательским типом данных является класс, который является частью объектно-ориентированных возможностей языка, рассмотренных во второй части этой книги.

Если вы считаете, что конструкторы типов распространены во многих языках программирования, то вы правы, но Pascal был первым языком, который представил идею формально и очень точно. Object Pascal все еще обладает некоторыми довольно уникальными возможностями, такими как определение поддиапазона, перечислений и множеств, которые рассматриваются в следующих разделах. Более сложные

конструкторы типов данных (такие как массивы и записи) рассматриваются в Главе 5.

Именованные против Безымянных Типов.

Определенные пользователем типы данных могут иметь имя для последующего использования или применены непосредственно к переменной. В Object Pascal договорились использовать префикс – букву `T` для обозначения любого типа данных, включая классы, но не ограничиваясь ими. Я настоятельно рекомендую придерживаться этого правила, даже если поначалу вы можете не чувствовать себя естественным, если вы начинали на Java или C#.

Когда вы даете имя типу, вы должны сделать это в разделе Type вашей программы (вы можете добавить столько типов, сколько вы хотите в каждом юните). Ниже приведен простой пример нескольких объявлений типов:

```
type
  // subrange definition
  TUppercase = 'A'..'Z';

  // enumerated type definition
  TMyColor = (Red, Yellow, Green, Cyan, Blue, Violet);

  // set definition
  TColorPalette = set of TMyColor;
```

С помощью этих типов теперь можно определить некоторые переменные:

```
var
  UpSet: TUpperLetters;
  Color1: TMyColor;
```

В вышеприведенном сценарии я использую *именованный* тип. В качестве альтернативы, определение типа может быть

использовано непосредственно для определения переменной без явного имени типа, как в следующем коде:

```
var
  Palette: set of TMyColor;
```

В целом, следует избегать использования *безымянных* типов, как в коде выше, так как нельзя передавать их в качестве параметров в подпрограммы или объявлять другие переменные того же типа. Учитывая, что язык, в конечном счете, прибегает к *эквивалентности имени типа,* а не к эквивалентности структурного типа, наличие единого определения для каждого типа действительно важно. Также помните, что определения типов в `интерфейсной` части модуля можно увидеть в коде любого другого модуля с помощью оператора `uses`.

Что означают приведенные выше определения типов? Я предоставлю некоторые описания для тех, кто не знаком с традиционными конструкциями описания типа языка Pascal. Также я постараюсь подчеркнуть отличия от аналогичных конструкций в других языках программирования, так что в любом случае вам может быть интересно прочитать следующие разделы.

Псевдонимы Типа

Как мы видели, язык Delphi при проверке совместимости типов использует имя типа (а не его фактическое определение). Два идентично определенных типа с разными именами, это два разных типа.

Частично это также верно, когда вы определяете псевдоним типа, то есть новое имя типа, основанное на существующем типе. Что вводит в заблуждение, так это то, что есть две вариации одного и того же синтаксиса, которые производят

слегка разные эффекты. Посмотрите на этот код в примере TypeAlias:

```
type
  TNewInt = Integer;
  TNewInt2 = type Integer;
```

Оба новых типа остаются совместимыми с типом Integer (посредством автоматического преобразования), однако тип `newint2` не будет точно совпадать и, например, не может быть передан в качестве параметра ссылки функции, ожидающей алиасного типа:

```
procedure Test (var N: Integer);
begin

end;

procedure TForm40.Button1Click(Sender: TObject);
var
  I: Integer;
  NI: TNewInt;
  NI2: TNewInt2;
begin
  I := 10;
  NI := I; // works
  NI2 := I; // works

  Test(I);
  Test (NI);
  Test (NI2); // error
```

The last line produces the error:

```
E2033 Types of actual and formal var parameters must be identical
```

(Типы фактических и формальных параметров var должны быть идентичны)

Что-то похожее происходит с хелперами типов, так как хелпер типа Integer может быть использован для `newint`, но не для `newint2`. Об этом специально пойдет речь в следующем разделе при обсуждении помощников.

Типы поддиапазонов

Тип поддиапазона определяет диапазон значений в пределах диапазона другого типа (отсюда и название *поддиапазона*). Например, вы можете определить поддиапазон типа Integer, от 1 до 10 или от 100 до 1000, или вы можете определить поддиапазон типа `Char` только с английскими заглавными символами, как в тексте:

```
type
  TTen = 1..10;
  TOverHundred = 100..1000;
  TUppercase = 'A'..'Z';
```

В определении поддиапазона нет необходимости указывать имя базового типа. Вам просто нужно указать две константы этого типа. Исходный тип должен быть порядковым, а результирующий тип будет другим порядковым. Когда вы определили переменную как поддиапазон, вы можете присвоить ей любое значение в пределах этого диапазона. Этот код работает:

```
var
  UppLetter: TUpperCase;

begin
  UppLetter := 'F';
```

Но этот не работает:

```
var
  UppLetter: TUpperCase;

begin
  UppLetter := 'e'; // compile-time error
```

При написании кода, приведенного выше, возникает ошибка компиляции *"Литерал нарушает границы поддиапазона"*. Если вместо этого написать следующий код, компилятор его примет:

```
var
  UppLetter: TUppercase;
  Letter: Char;

begin
  Letter :='e';
  UppLetter := Letter;
```

Зато во время выполнения, если вы включили опцию Компилятор проверки диапазона (на странице Компилятор диалогового окна Project Options), вы получите сообщение об *ошибке проверки диапазона*, как и ожидалось. Это похоже на ошибки переполнения целочисленного типа, которые я описывал ранее.

Я рекомендую включить эту опцию компилятора во время разработки кода программы, чтобы он был более надежным и простым в отладке, так как в случае ошибок вы получите явное сообщение, а не неопределенное поведение. В конце концов, вы можете отключить этот вариант для окончательной сборки программы, чтобы она работала немного быстрее. Однако увеличение скорости почти ничтожно, поэтому я предлагаю оставить все эти проверки времени исполнения включенными, даже при передаче программы потребителям.

Перечисляемые типы

Перечисляемые типы (обычно называемые "перечислениями") представляют собой другой определяемый пользователем порядковый тип. Вместо указания диапазона существующего типа, в перечислении перечисляются все возможные значения для данного типа. Другими словами, перечисление - это список (постоянных) значений. Приведем несколько примеров:

```
type
  TColors = (Red, Yellow, Green, Cyan, Blue, Violet);
  TSuit = (Club, Diamond, Heart, Spade);
```

Каждое значение в списке имеет соответствующую *последовательность*, начинающуюся с нуля. При применении функции `Ord` к значению перечисляемого типа, вы получаете это значение. Например, `Ord(Diamond)` возвращает 1.

Перечислимые типы могут иметь различные внутренние представления. По умолчанию Delphi использует 8-битное

представление, если только не существует более 256 различных значений, в этом случае используется 16-битное представление. Также существует 32-битное представление, которое иногда полезно для совместимости с библиотеками C или C++.

примечание С помощью директивы компилятора `$Z` можно изменить представление перечисленных типов по умолчанию, запрашивая больший тип, независимо от количества элементов в перечислении. Это довольно редкая настройка.

Перечисления с ограниченной областью действия

Конкретные постоянные значения перечисляемого типа можно рассматривать как глобальные константы, и имели место случаи конфликта названий между различными перечисляемыми значениями. Поэтому язык поддерживает перечисления с ограниченной областью действия - функцию, которую можно активировать с помощью специальной директивы компилятора `$SCOPEDENUMS` и которая требует обращаться к перечисляемому значению, используя имя типа в качестве префикса:

```
// classic enumerated value
S1 := Club;

// "scoped" enumerated value
S1 := TSuit.Club;
```

Когда эта функция была введена, стиль кодирования по умолчанию оставался традиционным, чтобы избежать прекращения работы существующего кода. Перечисления с ограниченной областью действия, по сути, изменяют поведение перечислений, делая обязательным обращение к ним с полностью квалифицированным префиксом типа.

Наличие *абсолютного* имени для обозначения перечисляемых значений устраняет риск конфликта, позволяет избежать использования исходного префикса перечисляемых значений

как способа дифференциации с другими перечислениями, а также делает код более читабельным, даже если писать его гораздо дольше.

В качестве примера, модуль `System.IOUtils` определяет этот тип:

```
{$SCOPEDENUMS ON}
type
  TSearchOption = (soTopDirectoryOnly, soAllDirectories);
```

Это означает, что вы не можете ссылаться на второе значение как на `soAllDirectories`, но вы должны ссылаться на перечисленное значение с его полным именем:

```
TSearchOption.soAllDirectories
```

Платформенная библиотека FireMonkey использует достаточно большое количество scoped enumerators, также требующих типа в качестве префикса к используемым значениям, в то время как более старая библиотека VCL, как правило, основана на более традиционной модели. RTL — это смесь обеих.

примечание Перечисляемые значения в библиотеках Object Pascal часто используют два или три символа инициала типа в начале значения, как "so" для Search Options в примере выше. При использовании типа в качестве префикса, это может показаться немного избыточным, но, учитывая общность подхода, я не думаю, что это скоро исчезнет.

Типы наборов (Set types)

Типы набора обозначают группу значений, где список доступных значений обозначается порядковым типом, на котором основан набор. Эти порядковые типы обычно ограничены и довольно часто представлены перечислением или поддиапазоном.

Если взять поддиапазон 1..3, то возможные значения набора на его основе включают только 1, только 2, только 3, как 1, так и 2, как 1, так и 3, как 2, так и 3, все три значения, или ни одно из них.

Переменная обычно содержит одно из возможных значений диапазона своего типа. Вместо этого переменная типа set может содержать ни одного, одно, два, три и более значений диапазона. Она даже может включать все значения.

Вот пример набора:

```
type
  TSuit = (Club, Diamond, Heart, Spade);
  TSuits = set of TSuit;
```

Теперь я могу определить переменную этого типа и присвоить ей некоторые значения исходного типа. Чтобы указать некоторые значения в наборе, пишите список, разделенный запятыми, заключенный в квадратные скобки. Следующий код показывает присваивание переменной нескольких значений, одного значения и пустого значения:

```
var
  Cards1, Cards2, Cards3: TSuits;

begin
  Cards1 := [Club, Diamond, Heart];
  Cards2 := [Diamond];
  Cards3 := [];
```

В Object Pascal набор обычно используется для указания нескольких неисключительных флагов. Например, значение, основанное на типе набора, является стилем шрифта. Возможные значения указывают на полужирный, курсивный, подчеркнутый и сквозной шрифт. Конечно, один и тот же шрифт может быть и курсивным, и жирным, не иметь атрибутов или иметь их все. По этой причине он объявляется набором. Значения этому множеству можно присвоить в коде программы следующим образом:

```
Font.Style := []; // no style
Font.Style := [fsBold]; // bold style only
Font.Style := [fsBold, fsItalic]; // two styles active
```

Операторы наборов (Set operators)

Мы видели, что наборы — это очень Pascal-специфический тип данных, определяемый пользователем. Поэтому стоит подробнее рассмотреть операторы множеств. К ним относятся объединение (+), разница (-), пересечение (*), тест на членство (in), плюс некоторые реляционные операторы.

Чтобы добавить элемент в набор, вы можете сделать объединение набора с другим, в котором есть только нужные вам элементы. Вот пример, связанный со стилями шрифтов:

```
// add bold
Style := Style + [fsBold];

// add bold and italic, but remove underline if present
Style := Style + [fsBold, fsItalic] - [fsUnderline];
```

В качестве альтернативы можно использовать стандартные процедуры `Include` и `Exclude`, которые гораздо эффективнее (но не могут использоваться со свойствами компонентов заданного типа):

```
Include (Style, fsBold);
Exclude (Style, fsItalic);
```

Выражения и операторы

Мы видели, что переменной можно присваивать литеральное значение, совместимое с типом, константное значение или значение другой переменной. Во многих случаях то, что вы присваиваете переменной, является результатом выражения, включающего одно или несколько значений и один или несколько операторов. Выражения являются еще одним ключевым элементом языка.

Использование операторов

Общего правила для построения выражений не существует, так как они в основном зависят от используемых операторов. В Object Pascal есть целый ряд операторов. Есть логические, арифметические, булевые, реляционные, заданные операторы, а также некоторые другие специальные:

```
// sample expressions
20 * 5 // multiplication
30 + n // addition
a < b        // less than comparison
-4           // negative value
c = 10       // test for equality (like == in C syntax)
```

Выражения являются общими для большинства языков программирования, и большинство операторов одинаковы. Выражение — это любая корректная комбинация констант, переменных, литеральных значений, операторов и результатов работы функций. Выражения могут использоваться для определения значения, которое необходимо присвоить переменной, для вычисления параметра функции или процедуры, или для проверки условия. Каждый раз, когда вы выполняете операцию со значением идентификатора, а не используете идентификатор сам по себе, вы используете выражение.

примечание Результат выражения обычно хранится во временной переменной соответствующего типа данных, автоматически генерируемой компилятором от вашего имени. Возможно, вам захочется использовать эксплицитную переменную, когда необходимо вычислить одно и то же выражение более одного раза в одном и том же фрагменте кода. Обратите внимание, что сложные выражения могут потребовать наличия нескольких временных переменных для хранения промежуточных результатов, опять же то, о чем заботится компилятор, и что обычно можно игнорировать.

Показать результат выражения.

Если вы хотите провести несколько экспериментов с выражениями, то нет ничего лучше, чем написать простую программу. Как и для большинства первоначальных примеров этой книги, создайте простую программу, основанную на форме, и используйте пользовательскую функцию `Show`, чтобы показать что-то пользователю. В случае, если информация, которую вы хотите показать, является не строковым сообщением, а числом или булевым логическим значением, вам необходимо преобразовать ее, например, вызвав функцию `IntToStr` или `BoolToStr`.

примечание В Object Pascal параметры, передаваемые функции или процедурам, заключены в круглые скобки. Некоторые другие языки (в частности, Rebol и, в некоторой степени, Ruby) позволяют передавать параметры, просто записывая их после имени функции или процедуры. Возвращаясь к Object Pascal, вызовы вложенных функций используют вложенную скобку, как показано в коде ниже.

Вот пример фрагмента кода из проекта `ExpressionsTest`

```
Show (IntToStr (20 * 5));
Show (IntToStr (30 + 222));
Show (BoolToStr (3 < 30, True));
Show (BoolToStr (12 = 10, True));
```

Результат выдачи очень прост:

```
100
252
True
False
```

Я привел этот пример в качестве скелета, чтобы вы могли попробовать различные типы выражений и операторов и посмотреть соответствующий вывод.

примечание Выражения, которые вы пишете в Object Pascal, разбираются компилятором и генерируют ассемблерный код. Если Вы хотите изменить одно из этих выражений, Вам необходимо изменить исходный код и перекомпилировать приложение. Системные библиотеки, однако, имеют поддержку динамических выражений,

---

вычисляемых во время выполнения, особенность, привязанная к отражению и описанная в Главе 16

Операторы и Приоритет

Выражения состоят из операторов, применяемых к значениям. Как я уже упоминал, большинство операторов общие с различными языками программирования и достаточно интуитивно понятны, например, основные операторы сопоставления и сравнения. В этом разделе я остановлюсь только на специфических элементах операторов Object Pascal.

Ниже вы можете увидеть список операторов языка, сгруппированных по приоритетам и сравниваемых с операторами на C#, Java и Objective-C (а также на большинстве языков, основанных на синтаксисе языка C, в любом случае).

Операторы сравнения и сопоставления (самый низкий приоритет)

=	Проверка равенства (в C это ==)
<>	Проверка неравенства (в C это !=).
<	Проверка на меньше
>	Проверка на больше
<=	Проверка меньше или равно, или на подмножество множества.
>=	Тестирование, является ли оно большим, равным или расширением набора

in	Проверка того, является ли данный элемент частью комплекта
is	Проверьте, совместим ли объект с данным типом (описанным в разделе Глава 8) или реализует заданный интерфейс (рассматривается в главе 11).

Дополнительные операторы

+	Арифметическое добавление, объединение наборов, конкатенация строк, добавление смещения указателя
-	Арифметическое вычитание, заданная разность, вычитание смещения указателя
Or	Логическое или побитовое “или” (в Си это либо `||` или `|`)
Xor	Логическое или побитовое эксклюзивное “или” (в C побитовое эксклюзивное “или” или ^)

Мультипликативные и побитовые операторы

*	Арифметическое умножение или заданное пересечение
/	Деление с плавающей точкой

div	Целочисленное деление (в C это также используется `/`)
mod	Modulo (остаток от целочисленного деления) (в C это `%`)
as	Позволяет выполнить преобразование с проверкой типа во время выполнения (рассматривается в Главе 8)
and	Логическое или побитовое “и” (в C это либо `&&` либо `&`)
shl	Побитовое смещение влево (в C это <<)
shr	Побитовое смещение вправо (в C это `>>`)

Унарные операторы (высший приоритет)

@	Адрес памяти переменной или функции (возвращает указатель, в C это &)
not	Логическое или побитовое - нет (в C это !)

В отличие от многих других языков программирования, логические операторы (включая `and`, or, not) имеют больший приоритет, чем операторы сравнения (включая < на >). Поэтому, если вы напишете:

```
a < b and c < d
```

компилятор выполнит операцию `and` первой, что, как правило, приводит к ошибке компилятора совместимости типов в

выражении. Если необходимо проверить оба сравнения, то следует заключить каждое из них в круглые скобки:

```
(a < b) and (c < d)
```

Напротив, для математических операций применяются общие правила, в которых умножение и деление имеют приоритет над сложением и вычитанием. Первые два приведенных ниже выражения эквивалентны, а третье отличается:

```
10 + 2 * 5      // результат 20
10 + (2 * 5)    // результат 20
(10 + 2) * 5    // результат 60
```

совет Хотя в некоторых случаях скобки не нужны, учитывая, что можно рассчитывать на правила приоритета операторов языка, настоятельно рекомендуется в любом случае добавлять их, так как эти правила варьируются в зависимости от языка программирования, и всегда лучше быть более понятным для любого, кто читает или модифицирует код в будущем.

Некоторые из операторов имеют разное значение при использовании с разными типами данных. Например, оператор + можно использовать для сложения двух чисел, сцепления двух строк, объединения двух наборов и даже добавления смещения к указателю (если для конкретного типа указателя включена *математика указателей*):

```
10 + 2 + 11
10.3 + 3.4
'Hello' + ' ' + 'world'
```

Однако вы не можете сложить два символа, как это возможно в C.

Необычный оператор - `div`. В Object Pascal с помощью оператора `/` можно выполнить деление любых двух чисел (вещественные или целые), и вы всегда получите результат в виде вещественного числа. Если вам нужно деление двух целых чисел и вам нужен целочисленный результат, используйте вместо него оператор `div`. Вот два примера присваивания (этот код станет более понятным, когда мы рассмотрим типы данных в следующей главе):

```
realValue := 123 / 12;
integerValue := 123 div 12;
```

Чтобы убедиться, что целочисленное деление не имеет остатка, можно воспользоваться оператором `mod` и проверить, что результат равен нулю, как в следующем логическом выражении:

```
(x mod 12) = 0
```

Дата и время

Хотя в ранних версиях языка Pascal не существовало родного типа для даты и времени, Object Pascal имеет встроенный тип для даты и времени. Он использует представление с плавающей точкой для обработки информации о дате и времени. Более конкретно `модуль System` определяет для этой цели конкретный тип данных `TDateTime`.

Это тип с плавающей точкой, потому что он должен быть достаточно широким, чтобы хранить годы, месяцы, дни, часы, минуты и секунды, вплоть до миллисекундного разрешения в одной переменной:

Даты хранятся как количество дней с 1899-12-30 (с отрицательными значениями, указывающими на даты до 1899 года) в целой части значения TDateTime

Времена хранятся как доли дня в десятичной части значения

исторически Если вам интересно, откуда взялась эта странная дата, то за этим стоит довольно длинная история. Она привязана к Excel и представлениям дат в приложениях Windows. Идея состояла в том, чтобы считать днем номер 1 первое января 1900 года, так что канун Нового Года 1899 года был бы день номер 0. Однако первоначальный разработчик этого представления даты должным образом забыл, что 1900 год не был високосным, и поэтому расчеты были впоследствии скорректированы на 1 день, превратив первое января 1900 года в день номер 2.

Как уже упоминалось, `TDateTime` не является предопределенным типом, который понимает компилятор, но определяется в модуле System как:

```
type
  TDateTime = type Double;
```

примечание Модуль System можно иногда образом рассматривать почти как часть основного языка, поскольку он всегда автоматически включается в каждую компиляцию без оператора uses (на самом деле добавление системного блока в секцию uses приведет к ошибке компиляции). Технически, однако, этот модуль рассматривается как часть ядра библиотеки времени исполнения (RTL), и об этом будет сказано в Гл. 17.

Существует также два взаимосвязанных типа для работы с частями времени и даты в структуре `TDateTime`, определенными как `TDate` и `TTime`. Эти специфические типы являются псевдонимами полного `TDateTime`, но они обрабатываются системными функциями, обрезающими неиспользуемую часть данных.

Использовать типы данных для даты и времени довольно просто, так как Delphi включает в себя ряд функций, которые работают с этим типом. В модуле `System.SysUtils имеется` несколько основных функций, а в модуле `System.DateUtils -` множество специфических функций (которые, несмотря на название, включают в себя также функции для управления временем).

Здесь вы можете найти краткий список часто используемых функций манипулирования датой/временем:

Now	Возвращает текущую дату и время в значение даты/времени
Date	Возвращает только текущую дату.
Time	Возвращает только текущее время.

DateTimeToStr	преобразует значение даты и времени в строку, используя форматирование по умолчанию; чтобы иметь больше контроля над преобразованием, используйте параметр Вместо этого используется функция FormatDateTime.
DateToStr	Преобразование части даты/времени из значения даты/времени в строку.
TimeToStr	Преобразует временную часть значения даты/времени в строку.
FormatDateTime	Форматирует дату и время, используя указанный формат; вы можете указать, какие значения вы хотите видеть и какой формат использовать, предоставив сложную строку формата
StrToDateTime	Преобразует строку с информацией о дате и времени в дату/время значение, вызывающее исключение в случае ошибки в формате строки. Сопутствующая ей функция StrToDateTimeDef возвращает значение по умолчанию в случае ошибки, а не при вызове исключения
DayOfWeek	Возвращает число, соответствующее дню недели значения даты/времени, переданного в качестве параметра (с использованием локальной конфигурации).

DecodeDate	Получает значения года, месяца и дня из значения даты
DecodeTime	Восстанавливает часы, минуты, секунды и миллисекунды от значения даты
EncodeDate	Преобразует значения года, месяца и дня в значение даты/времени
EncodeTime	Преобразует значения часов, минут, секунд и миллисекунд в значения даты/времени.

Чтобы показать вам, как использовать этот тип данных и некоторые связанные с ним процедуры, я создал простой прикладной проект под названием `TimeNow`. При запуске программы он автоматически вычисляет и отображает текущее время и дату.

```
var
  StartTime: TDateTime;
begin
  StartTime := Now;
  Show ('Time is ' + TimeToStr (StartTime));
  Show ('Date is ' + DateToStr (StartTime));
```

Первое утверждение - вызов функции `Now`, которая возвращает текущую дату и время. Это значение хранится в переменной `StartTime`.

примечание При вызове функции Object Pascal без параметров, в отличие от языков стиля C, нет необходимости печатать пустые скобки.

Следующие два оператора отображают временную часть значения `TDateTime`, преобразованную в строку, и временную часть того же значения. Это вывод программы (который будет зависеть от конфигурации вашей локали):

```
Time is 6:33:14 PM
```

```
Date is 10/7/2020
```

Для компиляции этой программы необходимо обратиться к функциям, входящим в состав модуля `System.SysUtils` (сокращенное название "системных утилит"). Кроме вызова `TimeToStr` и `DateToStr` можно использовать более мощную функцию `FormatDateTime`.

Обратите внимание, что значения времени и даты преобразуются в строки в зависимости от международных настроек системы. Информация о форматировании даты и времени считывается из системы, в зависимости от операционной системы и локали, заполняя структуру данных `TFormatSettings`. При необходимости индивидуального форматирования можно создать пользовательскую структуру такого типа и передать ее в качестве параметра большинству функций форматирования даты и времени.

примечание В проекте TimeNow также есть вторая кнопка, которую можно использовать для включения таймера. Это компонент который автоматически выполняет обработчик события с течением времени (вы указываете интервал). В примере, если вы включите таймер, вы будете видеть текущее время, добавляемое в список каждую секунду. Более полезным пользовательским интерфейсом было бы обновление метки с текущим временем каждую секунду, в основном построение часов.

Приведение и преобразование типов

Как мы видели, нельзя присвоить переменную одного типа данных другому типу. Причина в том, что в зависимости от фактического представления данных, вы можете получить что-то бессмысленное.

Теперь это не так для каждого типа данных. Например, числовые типы всегда можно безопасно продвигать. Здесь "продвинутый" означает, что вы всегда можете безопасно присвоить значение типу с большим представлением. Таким образом, вы можете присваивать слово целому, а целое - значению Int64. Противоположная операция, называемая "понижение", разрешена компилятором, но он выдаст предупреждение, так как в результате вы можете получить частичные данные. Остальные автоматические преобразования возможны только в одном случае: например, числу с плавающей точкой можно присвоить целое число, но обратная операция недопустима.

Есть сценарии, в которых нужно изменить тип значения, и операция имеет смысл. Когда вам нужно это сделать, есть два варианта. Первый - выполнить прямое приведение типа, которое скопирует физические данные и может дать правильное преобразование или нет в зависимости от типов. Когда вы выполняете приведение типов, вы говорите компилятору: "Я знаю, что делаю, поверь мне". Если вы используете приведение типов, но на самом деле не уверены в том, что вы делаете, вы можете попасть в беду, так как потеряете защитную сеть компилятора для проверки типов.

При приведении типа используется простая функциональная нотация, имя типа данных назначения используется в качестве функции:

```
var
  I: Integer;
  C: Char;
  B: Boolean;

begin
  I := Integer ('X');
  C := Char (I);
  B := Boolean (I);
```

Вы можете безопасно переходить между типами данных, имеющими одинаковый размер (т.е. одинаковое количество байт для представления данных - в отличие от приведенного выше фрагмента кода!). Как правило, можно конвертация безопасна между порядковыми типами, но можно также переходить и между указателями (и объектами), если вы знаете, что делаете.

Прямое приведение типов - опасная практика программирования, так как позволяет получить доступ к значению, как если бы оно представляло что-то другое. Так как внутренние представления типов данных, как правило, не совпадают (и могут даже изменяться в зависимости от целевой платформы), вы рискуете случайно создать трудно отслеживаемые ошибки. По этой причине, *как правило, следует избегать приведения типов*.

Вторым вариантом присвоения переменной одного из различных типов является использование функции приведения типов. Ниже приведен список функций, позволяющих осуществлять преобразование между различными базовыми типами (а некоторые из этих функций я уже использовал в демонстрационных примерах этой главы):

Chr	Преобразует порядковый номер в символ.
Ord	Преобразует значение ординарного типа в номер, указывающий его порядок.
Round	Преобразует значение вещественного типа в значение целочисленного типа, округляя его (см. также следующее примечание).

Trunc	Преобразует значение вещественного типа в значение целочисленного типа, усекая его значение.
Int	Возвращает Целочисленную часть аргумента значения с плавающей точкой.
`FloatToDecimal`	Преобразует значение с плавающей точкой в запись, включая его десятичное представление (экспонента, цифры, знак).
`FloatToStr`	Преобразует значение с плавающей точкой в его строковое представление с помощью форматирования по умолчанию
`StrToFloat`	Преобразует строку в значение с плавающей точкой

примечание Реализация функции Round основана на собственной реализации, предлагаемой CPU. Современные процессоры обычно применяют так называемое "Банковское округление", которое округляет средние значения (например, 5,5 или 6,5) вверх и вниз, в зависимости от того, следуют ли они за нечетным или четным числом. Существуют и другие функции округления, такие как RoundTo, которые обеспечивают больший контроль над фактической операцией.

Как уже упоминалось ранее в этой главе, некоторые из этих функций преобразования также доступны в виде прямых операций с типом данных (благодаря механизму хелперов типов). Несмотря на то, что существуют классические преобразования типа `IntToStr`, для большинства числовых типов можно применять операцию `ToString`, чтобы преобразовать их в строковое представление. Существует множество преобразований, которые можно применять непосредственно к переменным с помощью хелперов типов, и это должен быть предпочтительный стиль кодирования.

Некоторые из этих подпрограмм работают с типами данных, которые мы рассмотрим в следующих разделах. Обратите внимание, что в таблицу не включены процедуры для специальных типов (таких как `TDateTime` или Variant) или процедуры, специально предназначенные для форматирования большего, чем преобразование, например, мощные подпрограммы `Format` и `FormatFloat`.

03: языковые утверждения

Если понятие типа данных было одним из прорывов языка программирования Pascal, то другая сторона представлена кодом или операторами программирования. В то время эта идея была описана в выдающейся книге Никлауса Вирта "Алгоритмы + структуры данных = программы", изданной Prentice Hall в феврале 1976 года (классическая книга, до сих пор переиздаваемая и имеющаяся в продаже). Хотя эта книга вышла много лет до распространения объектно-ориентированного программирования, ее можно считать одной из основ современного программирования, базирующейся на сильном представлении о типе данных, и, таким образом, основой концепций, которые приводят к появлению объектно-ориентированных языков программирования.

Операции языка программирования основываются на ключевых словах и других элементах, которые позволяют указать компилятору последовательность выполняемых операций. Операторы часто вложены в процедуры или функции, как мы начнем более подробно рассматривать в следующей главе. А пока мы остановимся лишь на основных видах инструкций, которые можно написать для создания программы. Как мы видели в Главе 1 (в разделе, посвященном свободному пространству и форматированию кода), реальный код программы можно писать достаточно свободно. Я также рассказал о комментариях и некоторых других специальных элементах, но так и не представил полностью некоторые основные понятия, например, утверждения.

Простые и составные утверждения

Инструкции в программировании обычно называются *утверждениями*. Программный блок может состоять из *нескольких* утверждений. Существует два типа утверждений, простые и составные.

Утверждение называется *простым*, когда оно не содержит никаких других под-утверждений. Примерами простых операторов являются операторы присваивания и вызова процедур. В Object Pascal простые операторы *разделяются* точкой с запятой:

```
X := Y + Z;  // assignment
Randomize;   // procedure call
...
```

Чтобы определить *составное* утверждение или *составной* оператор, вы можете заключить еще одно из утверждений в ключевые слова "`begin`" и "`end`", которые выступают в качестве контейнеров множества утверждений и имеют роль, похожую, но не идентичную фигурным скобкам в C-образных языках. Составной оператор может появиться в любом месте, где может появиться простой объектный паскальный оператор. Приведем пример:

```
begin
  A := B;
  C := A * 2;
end;
```

Точка с запятой после последнего утверждения составного оператора (т.е. до `end`) не требуется, как указано ниже:

```
begin
  A := B;
  C := A * 2
end;
```

Обе версии верны. Первая версия имеет бесполезную (но безобидную) конечную точку с запятой. Эта точка с запятой на самом деле является нулевым или пустым выражением, то есть выражением без кода. Это существенно отличается от многих других языков программирования (например, основанных на синтаксисе языка Си), в которых точка с запятой является *терминатором* оператора (а не разделителем) и всегда требуется в конце оператора.

Обратите внимание, что иногда нулевой оператор может быть специально использован внутри циклов или в других особых случаях вместо фактического утверждения, как, например, в случае с нулевым оператором:

```
while condition_with_side_effect do
  ; // null or empty statement
```

Хотя эти конечные точки с запятой не служат никакой цели, большинство разработчиков склонны их использовать, и я предлагаю вам сделать то же самое. Иногда после того, как вы

написали пару строк, вы можете добавить еще одно утверждение. Если последняя точка с запятой отсутствует, Вы должны не забыть ее добавить, так что обычно лучше добавить ее сначала. Как мы сразу увидим, есть исключение из этого правила добавления лишних точек с запятой, а именно, когда следующий элемент является `другим` выражением внутри условия.

Условное выражение IF

Условный оператор используется для выполнения либо одного из операторов, которые он содержит, либо ни одного из них, в зависимости от конкретного теста (или условия). Существует два основных варианта условных операторов: операторы `if` и операторы `case`.

Оператор `if` может быть использован для выполнения оператора только при выполнении определенного условия (`if-then`) или для выбора между двумя различными альтернативами (`if-then-else`). Условие определяется логическим выражением.

На простом примере на Object Pascal, называемом `IfTest`, покажем, как записывать условные операторы. В этой программе мы воспользуемся элементом checkbox для получения пользовательского ввода, прочитав его свойство `IsChecked` (и сохранив его во временной переменной, хотя, строго говоря, это не обязательно, так как можно было бы напрямую проверить значение свойства в условном выражении):

```
var
  IsChecked: Boolean;
begin
```

```
IsChecked := CheckBox1.IsChecked;
if IsChecked then
  Show ('Checkbox is checked');
```

Если флажок установлен, программа выдаст простое сообщение. В противном случае ничего не произойдет. Для сравнения, один и тот же оператор, использующий синтаксис языка Си, будет выглядеть следующим образом (где условное выражение *должно* быть заключено в круглые скобки):

```
if (IsChecked)
 Show ("Checkbox is checked");
```

Некоторые другие языки имеют понятие элемента `endif`, позволяющего записывать несколько операторов, но в синтаксисе Object Pascal условный оператор по умолчанию является одним оператором. Используйте `begin-end` блок для выполнения более чем одного оператора для обработки части одного условия.

Если вы хотите выполнять различные операции в зависимости от выполнения условия, вы можете использовать оператор `if-then-else` (в данном случае я использовал прямое выражение, чтобы прочитать статус флажка):

```
// if-then-else statement
if CheckBox1.IsChecked then
  Show ('Checkbox is checked')
else
  Show ('Checkbox is not checked');
```

Обратите внимание, что нельзя ставить точку с запятой после первого утверждения и перед `другим` ключевым словом, иначе компилятор выдаст синтаксическую ошибку. Причина в том, что оператор `if-then-else` является единственным оператором, поэтому Вы не можете поставить точку с запятой посередине.

`Утверждение if` может быть довольно сложным. Условие может быть превращено в ряд условий (с помощью булевых операторов `and`, `or`, и `not`), или в оператор `if` может быть вложен второй оператор `if`. Помимо вложенности операторов `if`, когда существует несколько различных условий, часто встречаются

последовательные операторы `if-then-else-ifthen`. Вы можете вкладывать сколько угодно из этих условий, `если` хотите.

Третья кнопка проекта приложения `IfTest` демонстрирует эти сценарии, используя первый символ поля редактирования (который может отсутствовать, отсюда и внешний тест) в качестве входного:

```
var
  AChar: Char;
begin
  // multiple nested if statements
  if Edit1.Text.Length > 0 then
  begin
    AChar := Edit1.Text.Chars[0];

    // checks for a lowercase char (two conditions)
    if (AChar >= 'a') and (AChar <= 'z') then
      Show ('char is lowercase');

    // follow up conditions
    if AChar <= Char(47) then
      Show ('char is lower symbol')
    else if (AChar >= '0') and (AChar <= '9') then
      Show ('char is a number')
    else
      Show ('char is not a number or lower symbol');
  end;
```

Внимательно посмотрите на код и запустите программу, чтобы увидеть, понимаете ли вы его (и поиграйте с похожими программами, которые вы можете написать, чтобы узнать больше). Вы можете рассмотреть больше опций и логических выражений и увеличить сложность этого небольшого примера, делая любую проверку, которая вам нравится.

Оператор Case

Если ваши операторы if становятся очень сложными, иногда Вы можете заменить их утверждениями `case`. Оператор case состоит из выражения, используемого для выбора значения

(селектор) и списка возможных значений, или диапазона значений. Эти значения являются константами, они должны быть уникальными и иметь порядковый тип. В самом конце может появиться `else – одно` выражение, которое выполняется, если ни одно из указанных вами значений не соответствует значению селектора. Хотя конкретного оператора `endcase не существует`, `case` всегда завершается `end` (который в данном случае не является блочным терминатором, так как нет совпадающего `begin`).

примечание Создание case требует перечисления для селектора. Оператор case, основанный на строковом значении, в настоящее время не разрешен. В этом случае необходимо использовать вложенные операторы if или другую структуру данных, например, dictionary (как я покажу позже в книге в главе 14).

Приведем пример (часть проекта `CaseTest`), который использует в качестве входа интегральную часть числа, введенного в контрол NumberBox, элемент управления цифровым вводом:

```
var
  Number: Integer;
  AText: string;
begin
  Number := Trunc(NumberBox1.Value);
  case Number of
    1: AText := 'One';
    2: AText := 'Two';
    3: AText := 'Three';
  end;
  if AText <> '' then
    Show(AText);
```

Другим примером является расширение предыдущего комплексного оператора if, выраженное в ряде различных условий проверки:

```
case AChar of
  '+' : AText := 'Plus sign';
  '-' : AText := 'Minus sign';
  '*', '/': AText := 'Multiplication or division';
  '0'..'9': AText := 'Number';
  'a'..'z': AText := 'Lowercase character';
  'A'..'Z': AText := 'Uppercase character';
  #12032..#12255: AText := 'Kangxi Radical';
else
```

```
    AText := 'Other character: ' + aChar;
  end;
```

примечание Как видно из предыдущего фрагмента кода, диапазон значений определяется с тем же синтаксисом поддиапазонного типа данных. Напротив, несколько значений для одной ветви разделены запятой. Для секции Kangxi Radical я использовал численное значение, а не фактические символы, потому что большинство шрифтов фиксированного размера, используемых редактором IDE, не отображают символы должным образом.

Считается хорошей практикой включать `else`, чтобы сигнализировать о неопределенном или неожиданном состоянии. Оператор case в Object Pascal выбирает один путь выполнения, он не позиционирует себя в качестве точки входа. Другими словами, он выполнит оператор или блок после двоеточия выбранного значения и перейдет к следующему оператору после Case.

Это очень отличается от языка C (и некоторых его производных языков), который рассматривает ветви оператора `switch` как точки входа и будет выполнять все последующие операторы, если только вы специально не используете `break` (хотя это специфический сценарий, в котором Java и C# на самом деле отличаются в своей реализации). Синтаксис языка Си выглядит следующим образом:

```
switch (aChar) {
  case '+': aText = "plus sign"; break;
  case '-': aText = "minus sign"; break;
  ...
  default: aText = "unknown"; break;
}
```

Цикл For

Язык Object Pascal имеет стандартные повторяющиеся или циклические операторы большинства языков

программирования, в том числе и операторы `for`, `while и repeat`, а также более современный цикл `for-in` (или *for-each*). Большинство этих циклов будет знакомо, если вы использовали другие языки программирования, поэтому я расскажу о них лишь вкратце (с указанием основных отличий от других языков).

Цикл `for` в Object Pascal строго основан на счетчике, который может быть как увеличен, так и уменьшен каждый раз при выполнении цикла. Приведем простой пример цикла `for`, используемого для добавления первых десяти чисел (часть примера `ForTest`).).

```
var
  Total, I: Integer;
begin
  Total := 0;
  for I := 1 to 10 do
    Total := Total + I;
  Show(Total.ToString);
```

Для любопытных результат равен `55`. Другой способ записать цикл for после введения inline переменных — это объявить внутри объявления переменную-счетчик цикла (с синтаксисом, который несколько похож на синтаксис для циклов на языке Си и производных языках, рассмотренных позже):

```
  for var I: Integer := 1 to 10 do
    Total := Total + I;
```

В этом случае вы также можете воспользоваться выводом по типу и опустить спецификацию типа. Полный фрагмент кода, приведенный выше, становится:

```
var
  Total: Integer;
begin
  Total := 0;
  for var I := 1 to 10 do
    Total := Total + I;
  Show(Total.ToString);
```

Одно из преимуществ использования inline счетчика цикла состоит в том, что область его действия будет ограничена только циклом: Использование ее после выражения `for приведет к` ошибке, в то время как в общем случае вы получите только предупреждение при использовании счетчика цикла вне цикла.

Цикл `для` цикла в Pascal менее гибкий, чем в других языках (нельзя указать приращение, отличное от одного), но простой и понятный. Для сравнения: то же самое относится и `к` циклу, написанному в синтаксисе языка Си:

```
int total = 0;
for (int i = 1; i <= 10; i++) {
  total = total + i;
}
```

В этих языках инкремент — это выражение, которое может задать любую последовательность, которая может дать код, который многие посчитают нечитаемым:

```
int total = 0;
for (int i = 10; i > 0; total += i--) {
  ...
}
```

В Object Pascal вместо этого можно использовать только одношаговый инкремент. Если вы хотите протестировать более сложное условие, или если вы хотите предоставить настраиваемый счетчик, вам нужно будет использовать `утверждения while` или `repeat`, а не цикл for.

Единственная альтернатива одиночному инкременту - одиночный декремент или обратный цикл с ключевым словом `downto`:

```
var
  Total, I: Integer;
begin
  Total := 0;
  for I := 10 downto 1 do
    Total := Total + I;
```

примечание Обратный подсчет полезен, например, когда вы влияете на структуру данных, основанную на списке, через которую вы перебираете данные. При удалении

некоторых элементов, вы часто возвращаетесь назад, так как при прямом цикле вы можете повлиять на последовательность, в которой вы работаете (то есть, если вы удаляете третий элемент списка, четвертый элемент становится третьим: теперь вы находитесь на третьем, переходите к следующему (четвертому), но на самом деле вы работаете с тем, что было пятым элементом, пропуская один).

В Object Pascal счетчик цикла `for` не обязательно должен быть числом. Это может быть значение любого порядкового типа, например, символ или перечисляемый тип. Это помогает писать более читабельный код. Приведем пример с циклом `for`, основанным на типе `Char`:

```
var
  AChar: Char;
begin
  for AChar := 'a' to 'z' do
    Show (AChar);
```

Этот код (часть программы `ForTest`) показывает все буквы английского алфавита, каждую в отдельной строке выходного Memo control.

примечание Я уже показывал похожую демонстрацию, но на основе целочисленного счетчика, как часть примера `CharsTest` из Главы 2. В этом случае, однако, символы были собраны в одной выходной строке.

Вот еще один фрагмент кода, который показывает цикл for по пользовательскому перечислению:

```
type
  TSuit = (Club, Diamond, Heart, Spade);
var
  ASuit: TSuit;
begin
  for ASuit := Club to Spade do
    ...
```

Цикл в конце циклически повторяется на всех элементах типа данных. Может быть, лучше написать так, чтобы явно оперировать с каждым элементом типа (что сделает его более гибким к изменениям определения), чем конкретно указывать первый и последний элемент, записывая:

```
  for ASuit := Low (TSuit) to High (TSuit) do
```

Аналогичным образом, довольно распространена запись для цикла по всем элементам структуры данных, такие как строка. В этом случае можно использовать данный код (из проекта ForTest):

```
var
  S: string;
  I: Integer;
begin
  S := 'Hello world';
  for I := Low (S) to High (S) do
    Show(S[I]);
```

Если вы предпочитаете не указывать первый и последний элемент структуры данных, вы можете использовать цикл `for-in`, специализированный `вид цикла for`, о котором мы поговорим в следующем разделе.

примечание То, как компилятор относится к прямому чтению строковых данных с помощью операторов `[]` и определяет нижнюю и верхнюю границу строки, остается довольно сложной темой в Object Pascal, даже если значения по умолчанию теперь одинаковы для всех платформ. Это будет рассмотрено в Главе 6.

Для структур данных, использующих zerobyte индексацию (начиная с 0), необходимо выполнить цикл от нулевого индекса до индекса, предшествующего размеру или длине структуры данных. Распространенными способами написания такого кода являются:

```
  for I := 0 to Count-1 do ...
  for I := 0 to Pred(Count) do ...
```

Заключительное замечание относительно циклов про то, что происходит со счетчиком циклов после окончания цикла. Коротко говоря, значение *не определено*, и компилятор выдаст предупреждение, если вы попытаетесь использовать счетчик цикла for после завершения цикла. Одно из преимуществ использования inline переменной для счетчика циклов состоит в том, что переменная определяется только внутри самого цикла, и после его завершения она будет недоступна, что

приводит к ошибке компилятора (что является более надежной защитой):

```
begin
  var Total := 0;
  for var I: Integer := 1 to 10 do
    Inc (Total, I);
  Show (Total.ToString);
  Show (I.ToString); // compiler error: Undeclared Identifier 'I'
```

Цикл for-in

В Microsoft's Visual Basic всегда была специфическая конструкция цикла для циклической обработки всех элементов списка или коллекции, вызываемых *для каждого из них*. Та же идея была позже представлена в C#, где механизм `foreach` достаточно открыт и основан на использовании интерфейса `IEnumerator` и стандартного шаблона кодирования, в то время как Java использует ключевое слово `for для` выражения обоих типов циклов.

Объект Pascal имеет аналогичный цикл, вызываемый `for-in`. В этом цикле `для` каждого элемента массива, списка, строки или контейнера какого-то другого типа действует цикл for. В отличие от C#, Object Pascal не требует реализации интерфейса `IEnumerator`, но внутренняя реализация несколько похожа.

примечание Технические подробности поддержки цикла `for-in` в классе, добавление поддержки пользовательского перечисления, можно найти в Гл. 10.

Начнем с очень простого контейнера - строки, которую можно рассматривать как набор символов. В конце предыдущего раздела мы видели, как использовать цикл для работы со всеми элементами строки. Тот же самый эффект можно получить и при следующем `for-in` цикле на основе строки, где переменная `Ch` по очереди получает в качестве значения каждый из элементов строки:

```
var
```

```
  S: string;
  Ch: Char;
begin
  S := 'Hello world';
  for Ch in S do
    Show(Ch);
```

Этот фрагмент также является частью проекта приложения `ForTest`. Преимущество перед использованием традиционного цикла for заключается в том, что вам не нужно запоминать, какой первый элемент строки и как извлечь позицию последнего. Этот цикл проще в написании и обслуживании и имеет аналогичную эффективность.

Как и в традиционных циклах for, для циклов for-in также может быть полезно использовать внутристрочное объявление переменных. Мы можем переписать код, приведенный выше, используя следующий эквивалентный код:

```
var
  S: string;
begin
  S := 'Hello world';
  for var Ch: Char in S do
    Show(Ch);
```

цикл for-in может быть использован для доступа к элементам нескольких различных структур данных:

- Символы в строке (см. предыдущий фрагмент кода)
- Активные значения в наборе
- Элементы в статическом или динамическом массиве, включая двухмерные массивы (рассматривается в главе 5)
- Объекты, на которые ссылаются классы с поддержкой GetEnumerator, в том числе многие предопределенные, такие как строки в строковом списке, элементы различных контейнерных классов, компоненты, принадлежащие форме, и многие другие. Как это реализовать, будет рассмотрено в Гл. 10.

Пока в книге немного сложно описать эти расширенные шаблоны использования, поэтому я вернусь к примерам этого цикла позже в книге.

Примечание Цикл `for-in` в некоторых языках (например, JavaScript) имеет плохую репутацию, так как он очень медленно выполняется. Это не так в Object Pascal, где время цикла занимает примерно то же самое время, что и у соответствующего стандарта. Чтобы доказать это, я добавил в проект приложения `LoopsTest код замера времени`, который сначала создает строку из 30 миллионов элементов, а затем сканирует ее с обоими типами циклов (делая очень простую операцию на каждой итерации. Разница в скорости составляет около 10% в пользу классического для циклов (62 миллисекунды против 68 миллисекунд на моей машине Windows).

Циклы While и Repeat

Идея циклов `while-do` и `repeat-until` заключается в том, чтобы повторять выполнение блока кода снова и снова до тех пор, пока не будет выполнено заданное условие. Разница между этими двумя циклами заключается в том, что условие проверяется в начале или в конце цикла. Другими словами, блок кода оператора repeat всегда выполняется хотя бы один раз.

примечание Большинство других языков программирования имеют только один тип открытого оператора цикла, обычно называемого и ведущего себя как while. Синтаксис языка C имеет те же два варианта, что и синтаксис языка Pascal, с циклами `while` и `do-while`. Однако обратите внимание, что они используют одно и то же логическое условие, отличное от цикла repeat, имеющего обратное условие.

Легко понять, почему цикл repeat всегда выполняется хотя бы один раз, можно взглянув на простой пример кода:

```
while (I <= 100) and (J <= 100) do
begin
  // use I and J to compute something...
  I := I + 1;
  J := J + 1;
end;

repeat
  // use I and J to compute something...
  I := I + 1;
```

```
  J := J + 1;
until (I > 100) or (J > 100);
```

примечание Вы заметили, что как в условиях "`while`", так и в условиях "`repeat`" я заключил "подусловия" в круглые скобки. Это необходимо в данном случае, так как компилятор выполнит `or` до выполнения сравнений (о чем я рассказывал в разделе об операторах Главы 2).

Если начальное значение `I` или `J` больше 100, то цикл `while` полностью пропускается, в то время как утверждения внутри цикла repeat все равно выполняются один раз.

Другое ключевое различие между этими двумя циклами заключается в том, что цикл repeat имеет *обратное* состояние. Этот цикл выполняется *до тех пор, пока условие не будет выполнено*.

При выполнении условия цикл завершается. Это противоположно циклу `while-do`, который выполняется при выполнении условия true. По этой причине для получения аналогичного эффекта пришлось перевернуть условие в коде выше.

примечание “Обратное условие" формально известно как "законы Де Моргана" (описано, например, в Википедии по адресу http://en.wikipedia.org/wiki/De_Morgan%27s_laws).

Примеры циклов

Чтобы рассмотреть некоторые детали циклов, давайте рассмотрим небольшой практический пример. Программа LoopsTest подчёркивает разницу между циклом с фиксированным счетчиком и циклом с открытым счетчиком. Первый цикл с фиксированным счетчиком, а именно цикл `for`, отображает числа в последовательности:

```
var
  I: Integer;
begin
```

```
  for I := 1 to 20 do
    Show ('Number ' + IntToStr (I));
end;
```

То же самое можно было бы получить и с `временным` циклом, с внутренним приростом в единицу (обратите внимание, что вы увеличиваете значение после использования текущего). Однако, с помощью `некоторого` цикла Вы можете задать свой собственный инкремент, например, на 2:

```
var
  I: Integer;
begin
  I := 1;
  while I <= 20 do
  begin
    Show ('Number ' + IntToStr (I));
    Inc (I, 2)
  end;
end;
```

Этот код показывает все нечетные числа от одного до 19.

Эти циклы с фиксированным приращением логически эквивалентны и выполняются предопределенное количество раз. Это не всегда так. Существуют циклы, которые более неопределенны в своем выполнении, например, в зависимости от внешних условий.

примечание При написании цикла `while` всегда необходимо учитывать случай, когда условие никогда не выполняется. Например, если Вы напишете цикл выше, но забудете увеличить счетчик цикла, то это приведет к бесконечному циклу (который навсегда затормозит программу, потребляя процессор на 100%, до тех пор, пока операционная система не убьет ее).

Чтобы показать пример менее детерминированного цикла, я написал цикл `while`, все еще основанный на счетчике, но тот, который увеличивается случайным образом. Для этого я вызвал функцию `Random` со значением диапазона 100. Результатом работы этой функции является число от 0 до 99, выбранное случайным образом. Ряд случайных чисел контролирует, сколько раз выполняется цикл: :

```
var
```

```
  I: Integer;
begin
  Randomize;
  I := 1;
  while I < 500 do
  begin
    Show ('Random Number: ' + IntToStr (I));
    I := I + Random (100);
  end;
end;
```

Если вы не забыли добавить вызов процедуры `Randomize`, которая сбрасывает генератор случайных чисел в разных точках выполнения каждой программы, то каждый раз, когда вы запускаете программу, цифры будут отличаться. Ниже приведен вывод двух отдельных исполнений, отображаемых бок о бок:

```
Random Number: 1                Random Number: 1
Random Number: 40               Random Number: 47
Random Number: 60               Random Number: 104
Random Number: 89               Random Number: 201
Random Number: 146              Random Number: 223
Random Number: 198              Random Number: 258
Random Number: 223              Random Number: 322
Random Number: 251              Random Number: 349
Random Number: 263              Random Number: 444
Random Number: 303              Random Number: 466
Random Number: 349
Random Number: 366
Random Number: 443
Random Number: 489
```

Обратите внимание, что каждый раз не только сгенерированные номера отличаются друг от друга, но и количество элементов. Цикл While здесь выполняется случайное число раз. Если вы выполните программу несколько раз подряд, то увидите, что на выходе разное количество строк.

Прерывание выполнения цикла с

break и continue

Несмотря на различия, каждый из циклов позволяет выполнять блок операторов несколько раз, основываясь на некоторых правилах. Тем не менее, существуют сценарии, в которых Вы, возможно, захотите добавить некоторое дополнительное поведение. Предположим, в качестве примера, у Вас есть цикл for, в котором Вы ищете появление заданной буквы (этот код является частью прикладного проекта `FlowTest`):

```
var
  S: string;
  I: Integer;
  Found: Boolean;
begin
  S := 'Hello World';
  Found := False;
  for I := Low (S) to High (S) do
    if (S[I]) = 'o' then
      Found := True;
```

В конце можно проверить значение `Found`, чтобы узнать, была ли данная буква частью строки. Проблема в том, что программа продолжает повторять цикл и проверять на наличие данного символа даже после того, как нашла его появление (что было бы проблемой с очень длинной строкой).

Классической альтернативой было бы превратить это в цикл и проверить оба условия (счетчик цикла и значение Found):

```
var
  S: string;
  I: Integer;
  Found: Boolean;
begin
  S := 'Hello World';
  Found := False;
  I := Low (S);
  while not Found and (I <= High(S)) do
  begin
    if (S[I]) = 'o' then
      Found := True;
    Inc (I);
```

```
  end;
```

Несмотря на то, что этот код логичный и читабельный, он длиннее, а если условия станут многократными и более сложными, то сочетание всех различных вариантов сделает код очень сложным.

Поэтому в языке (или, точнее, в его поддержке времени исполнения) есть системные процедуры, которые позволяют изменять стандартный ход выполнения цикла:

- Процедура Break прерывает цикл, перепрыгивая непосредственно на первое следующее за ним утверждение, прекращая дальнейшее выполнение цикла
- Процедура continue переходит к проверке условия или приращению счетчика цикла, продолжая следующую итерацию цикла (до тех пор, пока условие больше не соответствует действительности или счетчик не достиг своего максимального значения).

Используя операцию `Break`, мы можем изменить предыдущий цикл для подбора символа следующим образом:

```
var
  S: string;
  I: Integer;
  Found: Boolean;
begin
  S := 'Hello World';
  Found := False;
  for I := Low (S) to High (S) do
    if (S[I]) = 'o' then
    begin
      Found := True;
      Break; // jumps out of the for loop
    end;
```

Еще две системные процедуры, `Exit` и `Halt`, позволяют немедленно вернуться к текущей функции или процедуре или прекратить работу программы. Я расскажу о `exit` в следующей главе, в то время как в принципе нет причин когда-либо вызывать `Halt` (поэтому я не буду обсуждать это в книге).

А вот и GOTO? Ни за что!

На самом деле имеется другой способ, чтобы прервать выполнение, кроме четырех системных процедур, описанные выше. Оригинальный язык Pascal имел среди своих возможностей *печально известный* оператор `goto`, позволяющий прикрепить метку к любой строке исходного кода и перепрыгнуть на эту строку из другого места. В отличие от условных и циклических операторов, которые показывают, почему вы хотите отклониться от последовательного выполнения кода, операторы `goto`, как правило, выглядят как беспорядочные прыжки, и на самом деле полностью разочаровывают. Упоминал ли я, что они не поддерживаются в Object Pascal? Нет, не упоминал, и не собираюсь показывать пример кода. Для меня `"goto"` давно уже не существует.

примечание Есть и другие языковые высказывания, которые я до сих пор не освещал, но которые являются частью определения языка. Одно из них утрверждение with, которое специально привязано к записям, поэтому я расскажу об этом в Главе 5. With является еще одной "спорной" особенностью языка, но не так сильно осуждаемой, как goto.

04: Процедуры и функции

Другая важная идея, подчеркнутая в языке Object Pascal (наряду с аналогичными характеристиками языка C), — это концепция подпрограммы, по сути, серия высказываний с уникальным названием, которая может быть активирована много раз. подпрограммы (или функции) вызываются по их названию. Таким образом, Вы избегаете необходимости писать один и тот же код снова и снова, и получаете единую версию кода, используемую во многих местах через программу. С этой точки зрения, вы можете думать о подпрограммах, как о базовом механизме инкапсуляции кода.

Процедуры и функции

В объекте Паскаль подпрограмма может принимать две формы: процедуру и функцию. Теоретически, процедура — это операция, которую вы запрашиваете у компьютера, а функция — это вычисление, возвращающее значение. Это различие подчеркивается тем, что функция имеет результат, возвращаемое значение или тип, а процедура - нет. Синтаксис

языка Си предусматривает единый механизм, функции, а в языке Си процедура — это функция с `ничтожным` (*void* или *null*) результатом.

Оба типа подпрограмм могут иметь несколько параметров заданных типов данных. Как мы увидим позже, процедуры и функции также являются основой методов класса, и в этом случае сохраняется различие между этими двумя формами. Фактически, в отличие от C, C++, Java, C# или JavaScript, при объявлении функции или метода необходимо использовать одно из этих двух ключевых слов.

На практике, даже если существуют два отдельных ключевых слова, разница между функциями и процедурами очень ограничена: вы можете вызвать функцию для выполнения некоторой работы, а затем проигнорировать результат (это может быть необязательный код ошибки или что-то в этом роде), или вы можете вызвать процедуру, которая передает обратно результат в одном из параметров (подробнее о справочных параметрах позже в этой главе).

Вот определение процедуры с использованием синтаксиса языка Object Pascal, который использует специальное ключевое слово procedure и является частью проекта FunctionTest:

```
procedure Hello;
begin
  Show ('Hello world!');
end;
```

Для сравнения, это была бы та же самая функция, написанная с синтаксисом языка Си, которая не имеет ключевого слова, требует наличия скобок даже в случае отсутствия параметров, и имеет `пустое` или пустое возвращаемое значение, указывающее на отсутствие результата:

```
void Hello ()
{
  Show ("Hello world!");
};
```

Фактически, в синтаксисе языка Си нет разницы между процедурой и функцией. В синтаксисе языка Pascal вместо этого функция имеет определенное ключевое слово и должна иметь возвращаемое значение (или тип возврата).

примечание Существует еще одно очень специфическое синтаксическое различие между Object Pascal и другими языками, а именно наличие точки с запятой в конце определения функции или процедуры, перед ключевым словом `begin`.

Существует два способа указать результат вызова функции, присвоить значение имени функции или использовать ключевое слово Result:

```
// классический стиль Паскаля
function DoubleOld (Value: Integer) : Integer;
begin
  DoubleOld := Value * 2;
end;


// современная альтернатива
function Double (Value: Integer) : Integer;
begin
  Result := Value * 2;
end;
```

примечание В отличие от классического синтаксиса языка Pascal, современный Object Pascal на самом деле имеет три способа указания результата функции, включая механизм выхода, обсуждаемый в этой главе в разделе "Выход с результатом".

Использование `Result` вместо имени функции для присвоения возвращаемого значения функции является наиболее распространенным синтаксисом и имеет тенденцию делать код более читабельным. Использование имени функции является классической паскальной нотацией, которая сейчас используется редко, но все еще поддерживается.

Опять же, для сравнения, ту же самую функцию можно было бы написать с синтаксисом языка Си, как показано ниже:

```
int Double (int Value)
{
  return Value * 2;
};
```

примечание Оператор return в языках, основанных на синтаксисе C, показывает результат работы функции, но также прекращает выполнение, возвращая управление вызываемому элементу. В Object Pascal, вместо этого, присвоение значения результату функции не завершает его. Поэтому результат часто используется как обычная переменная, например, для присвоения начального значения по умолчанию или даже для модификации результата в алгоритме. В то же время, если вам необходимо остановить выполнение, вам также необходимо использовать `Exit` или какой-нибудь другой оператор управления потоком. Более подробно все это описано в следующем разделе "Выход с результатом".

Если определяются подпрограммы именно так, то синтаксис вызова относительно прост, так как вы вводите идентификатор, за которым следуют параметры в круглых скобках. В случае отсутствия параметров, пустые круглые скобкй могут быть опущены (в отличие от языков, основанных на синтаксисе C). Данный фрагмент кода и несколько следующих являются частью проекта FunctionsTest данной главы:

```
//вызовите процедуру
Hello;

// вызов функции
X := Double (100);
Y := Double (X);
Show (Y.ToString);
```

Это концепция инкапсуляции кода, которую я описал. Когда вы вызываете функцию `Double`, вам не нужно знать алгоритм, используемый для ее реализации. Если в дальнейшем вы найдете лучший способ удвоения чисел, вы легко сможете изменить код функции, но код вызова останется неизменным (хотя его выполнение может стать более быстрым).

Тот же принцип может быть применен к процедуре `Hello`: мы можем модифицировать вывод программы, изменяя код этой процедуры, а основной код программы автоматически изменит свой эффект. Вот как мы можем изменить код реализации процедуры:

```
procedure Hello;
begin
  Show ('Hello world, again!');
```

```
end;
```

Предварительные декларации

Когда необходимо использовать идентификатор (любого вида), компилятор должен был его уже видеть, чтобы знать, к какому элементу он относится. По этой причине вы обычно даете полное определение перед тем, как использовать какую-либо подпрограмму. Однако, бывают случаи, когда это невозможно. Если процедура A вызывает процедуру B, а процедура B вызывает процедуру A, то, когда Вы начинаете писать код, Вам нужно будет вызвать подпрограмму, для которой компилятор до сих пор не видел определения.

В этих случаях (и во многих других) можно объявить о существовании процедуры или функции с определенным именем и заданными параметрами, не указывая ее реального кода. Один из способов объявить процедуру или функцию без ее определения — это написать ее имя и параметры (называемые сигнатурой функции), за которым следует ключевое слово `forward`:

```
procedure NewHello; forward;
```

Позже в коде должно быть полное определение процедуры (которое должно быть в том же самом модуле), но теперь процедуру можно вызвать до того, как она будет полностью определена. Вот пример, просто чтобы дать вам идею:

```
procedure DoubleHello; forward;

procedure NewHello;
begin
  if MessageDlg ('Do you want a double message?',
      TMsgDlgType.mtConfirmation,
      [TMsgDlgBtn.mbYes, TMsgDlgBtn.mbNo],
      0) = mrYes then
    DoubleHello
  else
    ShowMessage ('Hello');
```

```
end;

procedure DoubleHello;
begin
  NewHello;
  NewHello;
end;
```

примечание Функция MessageDlg, вызываемая в предыдущем фрагменте, является относительно простым способом запросить подтверждение пользователя в фреймворке FireMonkey (аналогичная функция существует и в VCL-фреймворке). Параметрами являются сообщение, тип диалогового окна и кнопки, которые необходимо отобразить. Результатом является идентификатор выбранной пользователем кнопки.

Этот подход (который также применяется в части прикладного проекта FunctionTest) позволяет написать взаимную рекурсию: `DoubleHello` вызывает `Hello`, но `Hello` может вызвать и `DoubleHello`. Другими словами, если Вы продолжите нажимать кнопку Yes, программа продолжит показывать сообщение и будет показывать каждое из них дважды для каждого *Yes*. В рекурсивном коде должно быть условие для завершения рекурсии, чтобы избежать условия, известного как переполнение стека.

примечание Вызовы функций используют *stack*: *стековую* часть памяти приложения для параметров, возвращаемого значения, локальных переменных и многого другого. Если функция продолжает вызывать себя в бесконечном цикле, область памяти для стека (которая, как правило, имеет фиксированный и предопределённый размер, определяемый компоновщиком и настроенный в опциях проекта) завершится ошибкой, известной как переполнение стека. Нет необходимости говорить о том, что популярный сайт поддержки разработчиков (www.stackoverflow.com) взял свое название от этой программной ошибки.

Хотя предварительное объявление процедуры не очень распространено в Object Pascal, есть похожий случай, который встречается гораздо чаще. Когда вы объявляете процедуру или функцию в секции интерфейса устройства, она автоматически считается предварительным объявлением, даже если ключевое слово `forward` отсутствует. На самом деле, вы не можете записать тело процедуры в разделе интерфейса модуля. Обратите

внимание, что вы должны указать фактическую реализацию каждой подпрограммы, которую вы декларировали в одном и том же модуле.

Рекурсивная функция

Учитывая, что я упомянул рекурсию и привел довольно своеобразный пример (с двумя вызовами процедур друг друга), позвольте мне также показать вам классический пример вызова рекурсивной функции самой себя. Использование рекурсии часто является альтернативным способом кодирования цикла.

Чтобы придерживаться классического примера, предположим, что вы хотите вычислить степень числа, и вам не хватает соответствующей функции (которая, конечно же, уже доступна в библиотеке run-time). Из курса математики вы помните, что 2 в степени 3 соответствует умножению 2 само на себя 3 раза, то есть 2*2*2.

Одним из способов выразить это в коде было бы написать цикл `for`, который выполняется 3 раза (или значение экспоненты) и умножает 2 (или значение основания) на текущий результат, начиная с 1:

```
function PowerL (Base, Exp: Integer): Integer;
var
  I: Integer;
begin
  Result := 1;
  for I := 1 to Exp do
    Result := Result * Base;
end;
```

Альтернативный подход заключается в многократном умножении основания на степень того же числа, с уменьшением значения показателя, до значения показателя 0,

в этом случае результат всегда равен 1. Это можно выразить рекурсивным вызовом одной и той же функции снова и снова:

```
function PowerR (Base, Exp: Integer): Integer;
var
  I: Integer;
begin
  if Exp = 0 then
    Result := 1
  else
    Result := Base * PowerR (Base, Exp - 1);
end;
```

Скорее всего, рекурсивная версия программы не быстрее, чем версия, основанная на цикле `for`, и не более читабельная. Однако существуют такие сценарии, как синтаксический разбор структур кода (например, древовидной структуры), в которых нет фиксированного количества элементов для обработки, и, следовательно, написание цикла близко к невозможному, в то время как рекурсивные функции приспосабливаются к этой роли.

В общем, рекурсивный код мощный, но имеет тенденцию быть более сложным. После многих лет, в течение которых рекурсия была почти забыта, по сравнению с ранними версиями программирования, новые функциональные языки, такие как Haskell, Erlang и Elixir, активно используют рекурсию и возвращают популярность этой идеи.

В любом случае, код этих двух степенных функций можно найти в проекте приложения FunctionTest.

примечание Обе степенные функции демо-версии не справляются со случаем отрицательного экспонента. Рекурсивная версия в таком случае будет зацикливаться вечно (до тех пор, пока программа не столкнется с физическим ограничением). Также при использовании целых чисел относительно быстро достигается максимальный размер типа данных и происходит его переполнение. Я написал эти функции с присущими им ограничениями, чтобы попытаться сохранить их код простым.

Что такое Метод?

Мы видели, как можно написать предварительное объявление в разделе интерфейса единицы измерения, используя ключевое слово `forward`. Объявление метода внутри типа класса также считается предварительным объявлением.

Но что именно является методом? Метод — это особый вид функции или процедуры, который связан с одним из двух типов данных, записью или классом. В Object Pascal каждый раз, когда мы обрабатываем событие для визуального компонента, нам необходимо определить метод, как правило, процедуру, но термин метод используется для обозначения как функций, так и процедур, связанных с классом или записью.

Вот пустой метод, автоматически добавляемый в исходный код формы (это действительно класс, как мы рассмотрим намного позже в книге):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // here goes your code
end;
```

Параметры и возвращаемые значения

При вызове функции или процедуры необходимо передать правильное количество параметров и убедиться, что они соответствуют ожидаемому типу. В противном случае компилятор выдаст сообщение об ошибке, аналогичное несоответствию типа при присваивании переменной значения неправильного типа. Учитывая предыдущее определение

функции `Double`, при вызове которой принимается параметр Integer:

```
Double (10.0);
```

Компилятор покажет ошибку:

```
[dcc32 Ошибка] E2010 Несовместимые типы: "Целочисленный" и
"Расширенный".
```

совет Редактор поможет вам, предложив список параметров функции или процедуры с подсказкой «на лету», как только вы наберете ее имя и открытую круглую скобку. Эта функция называется Code Parameters и является частью технологии Code Insight (известной в других IDE как *IntelliSense*). Code Insight, начиная с Delphi 10.4, работает через LSP Server (Language Server Protocol).

Есть сценарии, в которых, как и в случае с присваиваниями, допускается ограниченное приведение типов, но в общем случае следует стараться использовать параметры конкретного типа (это обязательно для ссылочных параметров, как мы увидим через некоторое время).

При вызове функции можно передать параметр как выражение, а не как значение. Выражение вычисляется и его результат присваивается параметру. В более простых случаях достаточно передать имя переменной. В этом случае значение переменной копируется в параметр (который, как правило, имеет другое имя). Я настоятельно не рекомендую использовать одно и то же имя для параметра и для переменной, передаваемой в качестве значения этого параметра, поскольку это может ввести в заблуждение.

предупреждение В Delphi, как правило, не следует полагаться на порядок вычисления параметров, переданных функции, так как он изменяется в зависимости от соглашения о вызове, и в тех же случаях он не определен, хотя наиболее распространенным случаем является вычисление справа налево. Дополнительная информация: `http://docwiki.embarcadero.com/RADStudio/en/Procedures_and_Functions_(Delphi)#Calling_Conventions`

Наконец, обратите внимание, что вы можете иметь функцию или процедуру с разными версиями (функция, называемая

перегрузкой), а некоторые параметры вы можете пропустить, чтобы позволить использовать предопределенное значение (функция, называемая *параметрами по умолчанию*). Эти две ключевые особенности для функций и процедур подробно описаны в конкретных разделах ниже в этой главе.

Выход с результатом

Мы видели, что при возвращении результата от функции используется совершенно другой синтаксис по сравнению с семейством языков C. Не только синтаксис отличается, но и поведение. Присвоение значения Result (или имени функции) не завершает работу функции, как это делает оператор `return`. Разработчики Object Pascal часто пользуются этой возможностью, используя `Result в` качестве временного хранилища. Вместо того, чтобы писать:

```
function ComputeValue: Integer;
var
  Value: Integer;
begin
  Value := 0;
  while ...
    Inc (Value);
  Result := Value;
end;
```

Вы можете опустить временную переменную и вместо нее использовать `Result`. Каким бы ни было значение `Result` при завершении работы функции, это значение, возвращаемое функцией:

```
function ComputeValue: Integer;
begin
  Result := 0;
  while ...
    Inc (Result);
end;
```

С другой стороны, бывают ситуации, когда вы хотите присвоить значение и сразу же выйти из процедуры, например, в

определенной ветке `if`. Если вам нужно присвоить результат функции *и* остановить текущее выполнение, то вы должны использовать два отдельных оператора, присвоить `Result`, а затем использовать ключевое слово `Exit`.

Если вы помните код проекта приложения FlowTest из последней главы (описанной в разделе "Breaking the Flow with Break and Continue"), то его можно переписать как функцию, заменив вызов на Break вызовом на Exit. Я внес это изменение в следующий фрагмент кода, часть проекта приложения ParamsTest:

```
function CharInString (S: string; Ch: Char): Boolean;
var
  I: Integer;
begin
  Result := False;
  for I := Low (S) to High (S) do
    if (S[I]) = Ch then
    begin
      Result := True;
      Exit;
    end;
end;
```

В Object Pascal можно заменить два оператора `if` блока специальным вызовом `Exit`, передав ему возвращаемое значение функции, аналогично оператору `возврата` языка C. Таким образом, можно более компактно написать код, приведенный выше (также потому, что одним оператором можно избежать блока `begin-end`):

```
function CharInString2 (S: string; Ch: Char): Boolean;
var
  I: Integer;
begin
  Result := False;
  for I := Low (S) to High (S) do
    if (S[I]) = Ch then
      Exit (True);
end;
```

примечание `Exit` в Object Pascal является функцией, поэтому вы должны заключить возвращаемое значение в круглые скобки, в то время как `возвращаемое` в C-style

языках значение является ключевым словом компилятора, не требующим круглых скобок.

Ссылочные параметры

В Object Pascal процедуры и функции позволяют передавать параметры по значению и по ссылке. По умолчанию передача параметров по значению: значение копируется в стек, и подпрограмма использует и манипулирует этой копией данных, а не исходным значением (как я описывал ранее в разделе "Параметры функции и возвращаемые значения"). Передача параметра по ссылке означает, что его значение не копируется в стек в формальном параметре подпрограммы. Вместо этого программа обращается к исходному значению, также в коде подпрограммы. Это позволяет процедуре или функции изменять фактическое значение переменной, переданной в качестве параметра. Передача параметра по ссылке выражается ключевым словом var.

Эта методика также доступна на большинстве языков программирования, так как отказ от копирования часто означает, что программа выполняется быстрее. Ее нет в Си (где можно просто использовать указатель), но она была введена в Си++ и других языках, основанных на синтаксисе Си, где используется символ `&` (pass by reference). Приведем пример передачи параметра по ссылке с использованием ключевого слова `var`:

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
End;
```

В этом случае параметр используется как для передачи значения в процедуру, так и для возврата нового значения в код вызова. При написании:

```
var
  X: Integer;
begin
  X := 10;
  DoubleTheValue (X);
  Show (X.ToString);
```

значение переменной X становится 20, так как функция использует ссылку на исходное место в памяти X, что влияет на ее исходное значение.

По сравнению с общими правилами передачи параметров, передача значений ссылочным параметрам подчиняется более строгим правилам, поскольку передаваемое значение не является значением, а фактической переменной. Нельзя передавать константу в качестве параметра ссылки, выражения, результата функции или свойства. Другое правило - нельзя передавать переменную несколько иного типа (требующую автоматического преобразования). Тип переменной и параметр должны точно совпадать, или как говорится в сообщении компилятора об ошибке:

```
[dcc32 Ошибка] E2033 Типы фактических и формальных параметров var должны
быть идентичными.
```

Это сообщение об ошибке, которое вы получите, например, если напишете (это тоже часть проекта приложения ParamsTest, но прокомментировано):

```
var
  C: Cardinal;
begin
  C := 10;
  DoubleTheValue (C);
```

Передача параметров по ссылке имеет смысл как для порядковых типов, так и для записей (как мы увидим в следующей главе). Эти типы часто называют *типами значений*, так как по умолчанию они имеют семантику "передача по значению" и "присваивание по значению".

Объекты и строки Object Pascal имеют несколько иное поведение, которое мы рассмотрим подробнее позже.

Объектные переменные являются ссылками, поэтому вы можете изменять фактические данные объекта, передаваемые в качестве параметра. Эти типы входят в различную группу, часто обозначаются как типы *ссылок*.

Помимо стандартных и ссылочных (`var`) типов параметров, Object Pascal также имеет очень необычный тип спецификатора параметров, `out`. Параметр `out` не имеет начального значения и используется только для возврата значения. За исключением отсутствия начального значения, параметры `out` ведут себя как параметры `var`.

примечание Out параметры были введены для поддержки соответствующей концепции в объектной модели Windows Component Object Model (или COM). Они могут быть использованы для выражения намерения разработчика ожидать неинициализированных значений.

Постоянные параметры

В качестве альтернативы опорным параметрам можно использовать параметр `const`. Так как нельзя присвоить новое значение константному параметру внутри подпрограммы, компилятор может оптимизировать передачу параметра. Компилятор может выбрать подход, аналогичный опорным параметрам (или константной ссылке в терминах языка Си++), но поведение останется аналогичным параметрам значения, так как исходное значение не может быть модифицировано функцией.

Фактически, если попытаться скомпилировать следующий код (доступный, но прокомментированный в проекте ParamsTest), то система выдаст ошибку:

```
function DoubleTheValue (const Value: Integer): Integer;
begin
  Value := Value * 2;      // compiler error
  Result := Value;
```

```
end;
```

Сообщение об ошибке, которое вы увидите, может быть не сразу интуитивно понятным, как говорится:

```
[dcc32 Error] E2064 Left side cannot be assigned to
     ([dcc32 Ошибка] E2064 Левая сторона не может быть назначена )
```

Постоянные параметры достаточно распространены для строк, так как в этом случае компилятор может отключить механизм подсчета ссылок, получив небольшую оптимизацию. Это наиболее распространенная причина использования константных параметров - особенность, которая имеет ограниченный смысл для порядкового и скалярного типов.

примечание Существует еще одна малоизвестная альтернатива передаче параметра `const`, `которая` заключается в добавлении к нему атрибута `ref`, как в " `const [ref]`". Этот атрибут заставляет компилятор передавать константный параметр по ссылке, где по умолчанию компилятор будет выбирать передачу константного параметра по значению или по ссылке в зависимости от размера параметра, при этом результат варьируется в зависимости от целевого процессора и платформы.

Перегрузка функций

Иногда может понадобиться две очень похожие функции с разными параметрами и разной реализацией. В то время как традиционно для каждой из них нужно было придумывать своё имя, современные языки программирования позволяют *перегружать* символ множеством различных определений.

Идея перегрузки проста: компилятор позволяет определить две или более функции, или процедуры с одним и тем же именем, при условии, что параметры различаются. Фактически, проверяя параметры, компилятор может определить, какую из версий функции вы хотите вызвать.

Рассмотрим ряд функций, взятых из модуля System.Math библиотеки runtime:

```
function Min (A,B: Integer): Integer; overload;
```

```
function Min (A,B: Int64): Int64; overload;
function Min (A,B: Single): Single; overload;
function Min (A,B: Double): Double; overload;
function Min (A,B: Extended): Extended; overload;
```

Когда вы вызываете `Min (10, 20)`, компилятор определяет, что вы вызываете первую функцию группы, поэтому возвращаемое значение также будет целочисленным.

Есть два основных правила перегрузки:

- За каждой версией перегруженной функции (или процедуры) должно следовать ключевое слово `overload` (включая первую).
- Среди перегруженных функций должна быть разница в количестве или типе параметров. Имена параметров не учитываются, так как при вызове они не указываются. Кроме того, тип возврата не может быть использован для различения двух перегруженных функций.

примечание Существует исключение из правила, в котором вы не можете различать функции на возвращаемых значениях, и оно касается операторов `Implicit` и `Explicit` преобразования, описанных в Главе 5.

Вот три перегруженных версии процедуры `ShowMsg`, которые я добавил в пример OverloadTest (приложение, демонстрирующее как перегрузку, так и параметры по умолчанию):

```
procedure ShowMsg (Str: string); overload;
begin
  Show ('Message: ' + Str);
end;

procedure ShowMsg (FormatStr: string;
  Params: array of const); overload;
begin
  Show ('Message: ' + Format (FormatStr, Params));
end;

procedure ShowMsg (I: Integer; Str: string); overload;
begin
  Show (I.ToString + ' ' + Str);
end;
```

Три функции показывают окно сообщения со строкой, после дополнительного форматирования строки различными способами. Вот три вызова программы:

```
ShowMsg ("Привет");
ShowMsg ('Total = %d.', [100]);
ShowMsg (10, 'MBytes');
```

И это их эффект:

```
Message: Привет
Message: Total = 100.
Message: 10 MBytes
```

совет Технология Code Parameters IDE очень хорошо работает с перегруженными процедурами и функциями. При вводе открытой круглой скобки после имени подпрограммы перечисляются все доступные альтернативы. При вводе параметров технология Code Insight использует их тип для определения того, какие из альтернатив все еще доступны.

Что делать, если попытаться вызвать функцию с параметрами, которые не соответствуют ни одной из доступных перегруженных версий? Конечно, вы получите сообщение об ошибке. Предположим, вы хотите вызвать так:

```
ShowMsg (10.0, "Привет");
```

Ошибка, которую вы увидите в данном случае, очень специфична:

```
[dcc32 Error] E2250 There is no overloaded version of 'ShowMsg' that can
be called with these arguments
[dcc32 Ошибка] E2250 Не существует перегруженной версии 'ShowMsg',
которую можно вызвать с этими аргументами.
```

Тот факт, что каждая версия перегруженной подпрограммы должна быть правильно отмечена, подразумевает, что Вы не можете перегрузить существующую подпрограмму того же устройства, которое не отмечено ключевым словом `overload`.

Сообщение об ошибке, которое вы получаете при попытке:

```
Предыдущее объявление '<name>' не было помечено директивой 'перегрузка'.
```

Однако вы можете создать подпрограмму с тем же именем, которое было объявлено в *другом юните*, учитывая, что модули выступают в качестве пространств имен. В этом случае вы не перегружаете функцию новой версией, а заменяете

функцию новой версией, скрывая оригинальную (на которую можно ссылаться, используя префикс имени модуля). Поэтому компилятор не сможет выбрать версию, основываясь на параметрах, но будет пытаться подобрать единственную версию, которую видит, выдавая ошибку, если типы параметров не совпадают.

Перегрузка и неоднозначные вызовы

При вызове перегруженной функции компилятор, как правило, находит совпадение и работает корректно или выдает ошибку, если ни одна из перегруженных версий не имеет нужных параметров (как мы только что видели).

Но есть и третий сценарий: Учитывая, что компилятор может выполнять некоторые приведения типов для параметров функции, возможны различные приведения для одного вызова. Когда компилятор находит несколько версий функции, которую он может вызвать, и нет ни одной, которая бы идеально соответствовала типу (который был бы выбран), он выдает сообщение об ошибке, указывающее на то, что вызов функции *неоднозначен (ambiguous)*.

Это не обычный сценарий, и мне пришлось построить довольно нелогичный пример, чтобы показать его вам, но его стоит рассмотреть (поскольку это иногда случается в реальном мире).

Предположим, вы решили реализовать две перегруженные функции для сложения целых чисел и чисел с плавающей точкой:

```
function Add (N: Integer; S: Single): Single; overload;
begin
  Result := N + S;
end;
```

```
function Add (S: Single; N: Integer): Single; overload;
begin
  Result := N + S;
end;
```

Эти функции находятся в проекте приложения OverloadTest. Теперь вы можете вызывать их, передавая два параметра в любом порядке:

```
Show (Add (10, 10.0).ToString);
Show (Add (10.0, 10).ToString);
```

Однако дело в том, что в общем случае функция может принять параметр другого типа при преобразовании, например, принять целое число, когда функция ожидает параметр типа с плавающей точкой. Так что же происходит при вызове:

```
Show (Add (10, 10).ToString);
```

Компилятор может вызвать первую версию перегруженной функции, но также может вызвать и вторую версию. Не зная, что вы просите (и зная, что вызов той или иной функции даст тот же эффект), он выдаст ошибку:

```
[dcc32 Error] E2251 Ambiguous overloaded call to 'Add'
  Related method: function Add(Integer; Single): Single;
  Related method: function Add(Single; Integer): Single;
```

совет В панели ошибок IDE вы увидите сообщение об ошибке с первой строкой вверху, а знак плюс сбоку можно развернуть, чтобы увидеть следующие две строки, в которых подробно описаны перегруженные функции, которые компилятор считает неоднозначными.

Если это реальный сценарий, и вам нужно сделать вызов, вы можете добавить ручной вызов преобразования типа, чтобы решить проблему и указать компилятору, какую из версий функции вы хотите вызвать:

```
Show (Add (10, 10.ToSingle).ToString);
```

Особый случай неоднозначных вызовов может произойти, если использовать variants, довольно своеобразный тип данных, о котором я расскажу несколько позже в этой книге.

Параметры по умолчанию

Другой особенностью, связанной с перегрузкой, является возможность предоставления значения по умолчанию некоторым параметрам функции, чтобы можно было вызывать функцию с этими параметрами или без них. Если параметр отсутствует в вызове, то ему будет присвоено значение по умолчанию.

Приведу пример (все еще часть проекта приложения OverloadTest). Мы можем определить следующую инкапсуляцию вызова `Show`, предоставляя два параметра по умолчанию:

```
procedure NewMessage (Msg: string; Caption: string = 'Message';
  Separator: string = ': ');
begin
  Show (Caption + Separator + Msg);
end;
```

С таким определением мы можем назвать процедуру по каждому из следующих способов:

```
NewMessage ('Something wrong here!');
NewMessage ('Something wrong here!', 'Attention');
NewMessage ('Hello', 'Message', '--');
```

Это выход:

```
Message: Something wrong here!
Attention: Something wrong here!
Message--Hello
```

Обратите внимание, что компилятор не генерирует специальный код, поддерживающий параметры по умолчанию, а также не создает нескольких (перегруженных) копий функций или процедур. Пропущенные параметры просто добавляются компилятором в вызывающий код. Есть одно важное ограничение, влияющее на использование параметров по умолчанию: нельзя "пропускать" параметры. Например,

нельзя передать третий параметр в функцию после пропуска второго.

Есть еще несколько правил для определения и вызова функций и процедур (и методов) с параметрами по умолчанию:

- В вызове можно пропустить параметры только начиная с последнего. Другими словами, если вы опускаете параметр, вы должны опустить и следующие.
- По определению, параметры со значениями по умолчанию должны находиться в конце списка параметров.
- Значения по умолчанию должны быть константами. Очевидно, что это ограничивает типы, которые можно использовать с параметрами по умолчанию. Например, динамический массив или тип интерфейса не может иметь параметр по умолчанию, отличный от `nil`; записи нельзя использовать вообще.
- Параметры со значениями по умолчанию должны передаваться по значению или как `const`. Параметр со ссылкой (`var`) не может иметь значение по умолчанию.

Использование параметров по умолчанию и перегрузка в то же время увеличивает вероятность попадания в ситуацию, которая сбивает компилятор с толку, вызывая ошибку *неоднозначного вызова*, как уже говорилось в предыдущем разделе. Например, если я добавлю следующую новую версию процедуры `NewMessage` к предыдущему примеру:

```
procedure NewMessage (Str: string; I: Integer = 0); overload;
begin
  Show (Str + ': ' + IntToStr (I))
end;
```

то компилятор не будет жаловаться, так как это законное определение. Однако, если вы напишете вызов:

```
NewMessage ('Hello');
```

компилятор пометит это как:

```
[dcc32 Error] E2251 Ambiguous overloaded call to 'NewMessage'
```

```
Related method: procedure NewMessage(string; string; string);
Related method: procedure NewMessage(string; Integer);
```

Обратите внимание, что эта ошибка проявляется в строке кода, которая корректно компилировалась перед новым перегруженным определением. На практике мы не имеем возможности вызвать процедуру `NewMessage` с одним строковым параметром, так как компилятор не знает, будем ли мы вызывать версию только со строковым параметром или со строковым и целочисленным параметром со значением по умолчанию. Когда у него возникает подобное сомнение, компилятор останавливается и просит программиста более четко изложить свои намерения.

Встраивание

Встраивание функций и методов Object Pascal является низкоуровневой функцией языка, которая может привести к значительным оптимизациям. Как правило, при вызове метода компилятор генерирует некоторый код, чтобы позволить программе перейти к новой точке выполнения. Это подразумевает установку фрейма стека и выполнение еще нескольких операций и может потребовать дюжины или около того машинных инструкций. Однако, выполняемый метод может быть очень коротким, возможно, даже метод доступа, который просто устанавливает или возвращает некоторое приватное поле. В таком случае имеет смысл скопировать реальный код в место вызова, избегая установки фрейма стека и всего остального. Устранив эти накладные расходы, ваша программа будет работать быстрее, особенно когда вызов происходит в цикле, выполняемом тысячи раз.

Для некоторых очень маленьких функций результирующий код может быть даже меньше, так как вставленный код может быть меньше кода, необходимого для вызова функции. Однако обратите внимание, что, если вставлена более длинная функция и эта функция вызывается во многих различных местах вашей программы, вы можете столкнуться с раздуванием кода, что является ненужным увеличением размера исполняемого файла.

В Object Pascal можно попросить компилятор вставить функцию (или метод) в строку с директивой inline, помещенной после объявления функции (или метода). Нет необходимости повторять эту директиву в определении. Всегда помните, что директива `inline` — это всего лишь подсказка компилятору, который может решить, что функция не является хорошим кандидатом для встраивания, и пропустить запрос (без предупреждения в любом случае). Компилятор также может встраивать некоторые, но не обязательно все вызовы функции после анализа вызывающего кода и в зависимости от состояния директивы $INLINE в месте вызова. Эта директива может принимать три различных значения (обратите внимание, что эта возможность не зависит от ключа компилятора оптимизации):

- С помощью `{$INLINE OFF}` можно подавить встраивание в программу, в часть программы или для конкретного места вызова, независимо от наличия директивы `inline` в вызываемых функциях.
- При значении по умолчанию `{$INLINE ON}` для функций, помеченных директивой `inline`, встраивание включено.
- С помощью `{$INLINE AUTO}` компилятор, как правило, встраивает функции, отмеченные директивой, плюс автоматически встраивает в строку очень короткие функции. Обратите

внимание, что эта директива может привести к раздуванию кода.

В библиотеке Object Pascal Run-Time Library имеется множество функций, которые были помечены как кандидаты на встраивание. Например, функция Max в System.Math имеет определения типа:

```
function Max(const A, B: Integer): Integer; overload; inline;
```

Для проверки фактического эффекта встраивания этой функции я написал следующий цикл в проекте InliningTest:

```
var
  Sw: TStopWatch;
  I, J: Integer;
begin
  J := 0;
  Sw := TStopWatch.StartNew;
  for I := 0 to LoopCount do
    J := Max (I, J);
  Sw.Stop;
  Show ('Max ' + J.ToString +
      ' [' + Sw.ElapsedMilliseconds.ToString + '] ');
```

В этом коде запись `TStopWatch` из модуля System.Diagnostics - структура, которая отслеживает время (или системные импульсы), прошедшее между вызовами `Start` (или `StartNew`) и `Stop`.

Форма имеет две кнопки, обе вызывающие в точности один и тот же код, но одна из них имеет отключенную вставку на месте вызова. Обратите внимание, что вам необходимо скомпилировать с конфигурацией Release, чтобы увидеть разницу (так как встраивание — это Release оптимизация). С двадцатью миллионами взаимодействий (значение константы `LoopCount`), на моем компьютере я получаю следующие результаты:

```
// on Windows (running in a VM)
Max on 20000000 [17]
Max off 20000000 [45]

// on Android (on device)
Max on 20000000 [280]
```

```
Max off 20000000 [376]
```

Как мы можем прочитать эти данные? На Windows, встраивание более чем в два раза увеличивает скорость выполнения, в то время как на Android это делает программу примерно на 35% быстрее. Однако, на устройстве программа работает намного медленнее (на порядок), поэтому в то время как на Windows мы экономим 30 миллисекунд, на моем Android-устройстве эта оптимизация экономит около 100 миллисекунд.

Эта же программа делает второй подобный тест с функцией `Length`, компилятор-магической функцией, которая была специально модифицирована для встраивания. Опять-таки внутренняя версия значительно быстрее как на Windows, так и на Android:

```
// on Windows (running in a VM)
Length inlined 260000013 [11]
Length not inlined 260000013 [40]

// on Android (on device)
Length inlined 260000013 [401]
Length not inlined 260000013 [474]
```

Вот код, используемый этим вторым циклом тестирования:

```
var
  Sw: TStopWatch;
  I, J: Integer;
  Sample: string;
begin
  J := 0;
  Sample:= 'sample string';
  Sw := TStopWatch.StartNew;
  for I := 0 to LoopCount do
    Inc (J, Length(Sample));
  Sw.Stop;
  Show ('Length not inlined ' + IntToStr (J) +
      ' [' + IntToStr (Sw.ElapsedMilliseconds) + '] ');
end;
```

Компилятор Object Pascal не определяет четкого ограничения на размер функции, которая может быть вставлена в функцию, или конкретный список конструкций (`for` циклов или `while`

циклов, условных операторов), которые могли бы препятствовать встраиванию. Однако, так как встраивание большой функции дает мало преимуществ, но содержит риск некоторых реальных недостатков (с точки зрения раздувания кода), то этого следует избегать.

Одним из ограничений является то, что метод или функция не могут ссылаться на идентификаторы (такие как типы, глобальные переменные или функции), определенные в секции реализации устройства, так как они не будут доступны в месте вызова. Однако, если вы вызываете локальную функцию, которая, по случаю, также будет встроена, то компилятор примет ваш запрос, чтобы вставить вашу подпрограмму в подпрограмму.

Недостатком является то, что встраивание требует более частой перекомпиляции блоков, так как при изменении встраиваемой функции код каждого из вызывающих мест также необходимо будет перекомпилировать. Внутри модуля можно написать код встраиваемых функций перед их вызовом, но лучше поместить их в начало раздела реализации.

примечание В Delphi однопроходной компилятор, поэтому он не может ссылаться на код функции, которую он еще не видел.

Внутри различных модулей необходимо специально добавлять другие модули с встроенными функциями к операторам `uses`, даже если вы не вызываете эти методы напрямую. Предположим, ваш юнит A вызывает внутреннюю функцию, определенную в юните B. Если эта функция, в свою очередь, вызывает другую внутреннюю функцию в юните C, то в вашем юните A также должна быть ссылка на C. Если нет, то вы увидите предупреждение компилятора, указывающее на то, что вызов не был включен из-за отсутствующей ссылки на юнит. Связанный с этим эффект заключается в том, что при наличии

кольцевых ссылок на модули (через их разделы реализации) функции никогда не будут встраиваться.

Расширенные возможности функций

Если то, о чем я говорил до сих пор, включает в себя основные возможности, есть также несколько расширенных возможностей, которые стоит изучить. Однако, если вы пока новичок в области разработки программного обеспечения, вы можете пропустить оставшуюся часть этой главы и перейти к следующей.

Конвенции по вызову в Object Pascal

Всякий раз, когда ваш код вызывает функцию, обе стороны должны договориться о реальном практическом способе передачи параметров от вызывающего абонента к вызывающему абоненту, что-то вроде *соглашения о вызове*. Обычно вызов функции происходит путем передачи параметров (и ожидания возвращаемого значения) через область памяти стека. Однако порядок размещения параметров и возвращаемого значения на стеке меняется в зависимости от языка и платформы программирования, при этом большинство языков способны использовать несколько различных соглашений по вызову.

Давным-давно 32-битная версия Delphi ввела новый подход к передаче параметров, известный как "fastcall": всегда, когда это возможно, в регистрах процессора может быть передано до трех параметров, что значительно ускоряет вызов функции. Объект Pascal по умолчанию использует это соглашение быстрого вызова, хотя его также можно запросить, используя ключевое слово register.

Fastcall является соглашением по умолчанию, и функции, использующие его, не совместимы с внешними библиотеками, такими как функции Windows API в Win32. Функции Win32 API должны быть объявлены с использованием соглашения о вызовах stdcall (*стандартный вызов*), смеси оригинального соглашения о вызовах pascal API Win16 и соглашения о вызовах cdecl языка Си. Все эти соглашения по вызову поддерживаются в Object Pascal, но вы редко будете использовать что-то, отличное от стандартного, если только вам не нужно вызвать библиотеку, написанную на другом языке, например, системную библиотеку.

Типичный случай, когда вам нужно отойти от стандартного соглашения о быстром вызове, это когда вам нужно вызвать родной API платформы, который требует другого соглашения о вызове в зависимости от операционной системы. Даже Win64 использует другую модель, чем Win32, поэтому Object Pascal поддерживает множество различных опций, не стоит здесь подробно останавливаться. Мобильные операционные системы, напротив, склонны к публикации классов, а не родных функций, хотя вопрос соблюдения заданного соглашения о вызовах должен быть учтен даже в этом сценарии.

Процедурные типы

Еще одной особенностью Object Pascal является наличие процедурных типов. Это действительно продвинутая языковая тема, которой будут регулярно пользоваться лишь немногие программисты. Однако, так как мы будем обсуждать смежные темы в последующих главах (в частности, указатели на методы - техника, интенсивно используемая окружением для определения обработчиков событий, и анонимные методы), то здесь стоит дать краткий обзор.

В Object Pascal (но не в более традиционном языке Pascal) есть понятие процедурного типа (которое похоже на концепцию указателя на функцию в языке Си - другие языки вроде C# и Java исключили, потому что оно привязано к глобальным функциям и указателям).

Объявление процедурного типа указывает перечень параметров и, в случае функции, тип возврата. Например, этим кодом можно объявить новый процедурный тип, при котором параметр Integer передается по ссылке:

```
type
  TIntProc = procedure (var Num: Integer);
```

Этот тип процедуры совместим с любой подпрограммой, имеющей точно такие же параметры (или ту же сигнатуру функции при использовании жаргона C). Приведем пример совместимой подпрограммы:

```
procedure DoubleTheValue (var Value: Integer);
begin
  Value := Value * 2;
end;
```

Процедурные типы могут использоваться для двух различных целей: можно объявить переменные процедурного типа или передать в качестве параметра другой подпрограмме процедурный тип (то есть указатель на функцию). Учитывая

предшествующие объявления типа и процедуры, можно написать данный код:

```
var
  IP: TIntProc;
  X: Integer;
begin
  IP := DoubleTheValue;
  X := 5;
  IP (X);
end;
```

Этот код имеет тот же эффект, что и следующая укороченная версия:

```
var
  IP: TIntProc;
  X: Integer;
begin
  IP := DoubleTheValue;
  X := 5;
  IP (X);
end;
```

Первая версия явно сложнее, так зачем и когда ее использовать? Бывают случаи, когда возможность решить, какую функцию вызывать, а на самом деле вызывать ее позже, может быть очень мощной. Можно построить сложный пример, демонстрирующий такой подход. Однако я предпочитаю, чтобы вы рассмотрели достаточно простой проект приложения, называемый ProcType.

Этот пример основан на двух процедурах. Одна процедура используется для удвоения значения параметра, как я уже показывал. Вторая процедура используется для утроения значения параметра, и поэтому называется `TripleTheValue`:

```
procedure TripleTheValue (var Value: Integer);
begin
  Value := Value * 3;
end;
```

Вместо прямого вызова этих функций та или иная функция сохраняется в переменной процедурного типа. Переменная изменяется по мере того, как пользователь выбирает флажок, и текущая процедура вызывается таким образом, как будто

пользователь нажимает на кнопку. Программа использует две инициализированные глобальные переменные (вызываемая процедура и текущее значение), так что эти значения сохраняются во времени. Вот полный код, за исключением определений фактических процедур, уже показанных выше:

```
var
  IntProc: TIntProc = DoubleTheValue;
  Value: Integer = 1;

procedure TForm1.CheckBox1Change(Sender: TObject);
begin
  if CheckBox1.IsChecked then
    IntProc := TripleTheValue
  else
    IntProc := DoubleTheValue;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  IntProc (Value);
  Show (Value.ToString);
end;
```

Когда пользователь меняет статус флажка, все последующие нажатия кнопок вызовут активную функцию. Таким образом, если вы дважды нажмете на кнопку, измените выбор и еще раз нажмете на кнопку дважды, вы сначала удвоите, а затем утроите текущее значение, получив следующий вывод:

```
2
4
12
36
```

Другой практический пример использования процедурных типов — это когда вам нужно передать функцию в операционную систему типа Windows (где они обычно называются "функциями обратного вызова"). Как упоминалось в начале этого раздела, в дополнение к процедурным типам Object Pascal разработчики используют указатели на методы (рассматриваются в Гл. 10) и анонимные методы (рассматриваются в Гл. 15).

примечание Наиболее распространенным объектно-ориентированным механизмом получения позднего вызова функции (т.е. вызова функции, который может измениться во время выполнения) является использование виртуальных методов. Хотя виртуальные методы очень распространены в Object Pascal, процедурные типы используются редко. Техническая основа, однако, в некоторой степени схожа. Виртуальные функции и полиморфизм рассматриваются в главе 8.

Декларация внешних функций

Другой важный элемент для системного программирования представлен внешними декларациями. Первоначально использованные для связи кода с внешними функциями, которые были написаны на ассемблере, внешние декларации стали обычным явлением в программировании Windows для вызова функции из DLL (динамическая библиотека). Декларация внешней функции подразумевает возможность вызова функции, не полностью доступной компилятору или компоновщику, но требующей возможности загрузить внешнюю динамическую библиотеку и вызвать одну из ее функций.

примечание Всякий раз, когда вы вызываете в коде Object Pascal API для определенной платформы, вы теряете возможность перекомпилировать приложение для любой другой платформы, кроме конкретной. Исключение составляют случаи, когда вызов окружен директивами компилятора $IFDEF для конкретной платформы.

Вот, например, как можно вызывать функции Windows API из приложения Delphi. Если вы откроете модуль Winapi.Windows, вы найдете множество деклараций функций и определений:

```
// forward declaration
function GetUserName(lpBuffer: LPWSTR;
  var nSize: DWORD): BOOL; stdcall;

// external declaration (instead of actual code)
function GetUserName; external advapi32
  name 'GetUserNameW';
```

Вам редко придется писать такие декларации, как только что проиллюстрированные, поскольку они уже есть в списке модулей `Windows` и многих других системных модулей. Единственная причина, по которой вам может понадобиться написать этот код внешней декларации — это вызов функций из пользовательской DLL или вызов функций Windows, не транслированных в API платформы.

Это объявление означает, что код функции GetUserName хранится в динамической библиотеке advapi32 (advapi32 - константа, связанная с полным именем DLL, *'advapi32.dll'*) с именем `GetUserNameW`, так как эта функция API имеет версию как ASCII, так и WideString. Внутри внешнего объявления, на самом деле, мы можем указать, что наша функция относится к функции DLL, которая изначально имела другое имя.

Поздняя загрузка DLL-функций

В операционной системе Windows есть два способа вызова функции API из Windows SDK (или любой другой DLL): вы можете позволить загрузчику приложения разрешить все ссылки на внешние функции или вы можете написать конкретный код, который ищет функцию и выполняет ее, если она доступна.

Код выше проще написать, как мы видели в предыдущем разделе: все, что вам нужно — это объявление внешней функции. Однако, если библиотека или даже одна из функций, которые вы хотите вызвать, недоступны, ваша программа не сможет запуститься на версиях операционной системы, которые не предоставляют эту функцию.

Динамическая загрузка обеспечивает большую гибкость, но подразумевает загрузку библиотеки вручную, используя `GetProcAddress` API для поиска вызываемой функции, и вызов ее

после приведения указателя к нужному типу. Такой код достаточно громоздкий и склонный к ошибкам.

Поэтому хорошо, что компилятор Object Pascal и компоновщик имеют специальную поддержку функции, которая сейчас доступна на уровне операционной системы Windows и уже используется некоторыми компиляторами C++, отложенная загрузка функций до момента их вызова. Цель этого объявления - не избежать неявной загрузки DLL, которая все равно происходит, а позволить отложенную привязку этой конкретной функции внутри DLL.

В основном код пишется таким образом, что он очень похож на классическое выполнение функции DLL, но адрес функции разрешается при первом вызове функции, а не во время загрузки. Это означает, что, если функция недоступна, вы получаете исключение во время выполнения, EExternalException. Однако, как правило, можно проверить текущую версию операционной системы или версию конкретной библиотеки, которую вы вызываете, и заранее решить, хотите ли вы сделать вызов или нет.

примечание Если вы хотите что-то более конкретное и легкое в обращении на глобальном уровне, чем исключение, вы можете подключиться к механизму ошибок для задержки вызова загрузки, как объяснил Аллен Бауэр в своем посте в блоге: https://blog.therealoracleatdelphi.com/2009/08/exceptional-procrastination_29.html.

С точки зрения языка Object Pascal единственное отличие заключается в объявлении внешней функции. Вместо того, чтобы писать:

```
function MessageBox;
  external user32 name 'MessageBoxW';
```

Теперь вы можете писать (опять же, из реального примера в модуле `Windows`):

```
function GetSystemMetricsForDpi(nIndex: Integer; dpi: UINT): Integer;
  stdcall; external user32 name 'GetSystemMetricsForDpi' delayed;
```

С учетом того, что этот API был впервые добавлен в Windows 10, версия 1607, вы, возможно, на время работы захотите написать код, как показано ниже:

```
if (TOSVersion.Major >= 10) and (TOSVersion.Build >= 14393)
begin
  NMetric := GetSystemMetricsForDpi (SM_CXBORDER, 96);
```

Это намного, намного меньше кода, чем нужно было написать без задержки загрузки, чтобы иметь возможность запускать одну и ту же программу на старых версиях Windows.

Еще одним важным замечанием является то, что вы можете использовать один и тот же механизм при создании собственных DLL и вызове их в Object Pascal, предоставляя один исполняемый файл, который может привязываться к нескольким версиям одной и той же DLL в случае, если вы используете отложенную загрузку новых DLL.

05: Массивы и записи

Когда я представлял типы данных в главе 2, я ссылался на тот факт, что в Object Pascal есть как встроенные типы данных, так и конструкторы типов. Простой пример конструктора типов – перечисляемый тип, рассмотренный в этой главе.

Реальная сила определения типа приходит с более продвинутыми механизмами, такими как массивы, записи и классы. В этой главе я расскажу о первых двух, которые по своей сути восходят к раннему определению Паскаля, но были настолько изменены с годами (и сделаны настолько мощными), что едва ли похожи на конструкторов своих предковых типов с тем же именем.

В конце главы я также кратко представлю некоторые продвинутые типы данных Object Pascal в качестве указателей. Реальная сила пользовательских типов данных, однако, будет раскрыта в главе 7, где мы начнем рассматривать классы и объектно-ориентированное программирование.

Типы массивов данных

Типы массивов определяют списки с элементами определенного типа. Эти списки могут иметь фиксированное количество элементов (статические массивы) или переменное количество элементов (динамические массивы). Как правило, для доступа к одному из элементов массива используется *индекс* в квадратных скобках. Квадратные скобки также используются для указания количества значений массива фиксированного размера.

Язык Object Pascal поддерживает различные типы массивов, от традиционных статических до динамических. Использование динамических массивов рекомендуется, в частности, для мобильных версий компилятора. Сначала я представлю статические массивы, а затем остановлюсь на динамических.

Статические массивы

Массивы традиционного языка Pascal определяются статическим или фиксированным размером. Пример приведен в следующих фрагментах кода, которые определяют список из 24 целых чисел, представляющих температуры в течение 24 часов в сутки:

```
type
  TDayTemperatures = array [1..24] of Integer;
```

В этом классическом определении массива вы можете использовать в квадратных скобках поддиапазонный тип, фактически определяя новый специфический поддиапазонный тип, используя две константы порядкового типа. Этот поддиапазон указывает на действительные индексы массива. Так как вы указываете и верхний, и нижний индекс массива,

индексы не обязательно должны быть нулевыми, как это происходит в C, C++, Java и большинстве других языков (хотя массивы, основанные на 0, также довольно распространены в Object Pascal). Заметьте также, что статические индексы массивов в Object Pascal могут быть числами, а также другими порядковыми типами, такими как символы, перечисленные типы и многое другое. Нечисловые индексы встречаются довольно редко.

примечание Существуют языки типа JavaScript, которые интенсивно используют ассоциативные массивы. Массивы Object Pascal ограничены порядковыми индексами, поэтому вы не можете напрямую использовать строку в качестве индекса. Есть другие готовые структуры данных в RTL, реализующие словари и другие подобные структуры данных, предоставляющие такие возможности. Я расскажу о них в главе о дженериках, в третьей части книги.

Так как индексы массивов основаны на поддиапазонах, компилятор может проверить их диапазон. Недопустимый постоянный поддиапазон приводит к ошибке во время компиляции; а индекс вне диапазона, используемый во время выполнения, приводит к ошибке во время выполнения, но только если включена соответствующая опция компилятора.

примечание Это опция Проверка диапазона (Range) группы *Runtime errors (Ошибки выполнения) на* странице *Compiling (Компиляция)* диалогового окна Project Options (Параметры проекта) IDE. Я уже упоминал об этой опции в Главе 2, в разделе "Типы поддиапазонов".

Используя приведенное выше определение массива, можно установить значение переменной `DayTemp1` типа `TDayTemperatures` следующим образом (и так же, как я сделал это в прикладном проекте ArraysTest, из которого были извлечены следующие фрагменты кода):

```
type
  TDayTemperatures = array [1..24] of Integer;

var
  DayTemp1: TDayTemperatures;

begin
```

```
  DayTemp1 [1] := 54;
  DayTemp1 [2] := 52;
  ...
  DayTemp1 [24] := 66;

  // The following line causes:
  // E1012 Constant expression violates subrange bounds
  // DayTemp1 [25] := 67;
```

Теперь стандартным способом работы с массивами, учитывая их природу, является использование циклов `for`. Вот пример цикла, используемого для отображения всех температур за день:

```
var
  I: Integer;
begin
  for I := 1 to 24 do
    Show (I.ToString + ': ' + DayTemp1[I].ToString);
```

Хотя этот код работает, наличие жестко закодированных границ массивов (`1` и `24`) далеко от идеала, так как само определение массива со временем может измениться. В этом случае вы можете захотеть перейти к использованию динамического массива.

Размеры и границы массива

При работе с массивом всегда можно проверить его границы с помощью стандартных функций `Low` и `High`, которые возвращают нижнюю и верхнюю границу. Использование функций `Low` и `High` при работе с массивом настоятельно рекомендуется, особенно в циклах, так как это делает код независимым от текущего диапазона массива (который может идти от 0 до длины массива минус один, может начинаться с 1 и достигать длины массива, или иметь любое другое определение поддиапазона). Если в дальнейшем следует изменить объявленный диапазон индексов массивов, то код, использующий `Low` и `High`, все равно будет работать. Если вы

написали цикл с жестким кодированием диапазона массива, то при изменении размера массива необходимо обновить код цикла. Low и High делают ваш код более простым в обслуживании и более надежным.

примечание Кстати, нет дополнительной нагрузки во время выполнения при использовании `Low` и `High` со статическими массивами. Они разрешаются во время компиляции в постоянные выражения, а не при реальных вызовах функций. Такое разрешение выражений и вызовов функций во время компиляции происходит и для многих других системных функций.

Еще одной релевантной функцией является `Length`, которая возвращает количество элементов массива. Я объединил эти три функции в следующем коде, который вычисляет и отображает среднюю температуру за день:

```
var
  I: Integer;
  Total: Integer;
begin
  Total := 0;
  for I := Low(DayTemp1) to High(DayTemp1) do
    Inc (Total, DayTemp1[I]);
  Show ((Total / Length(DayTemp1)).ToString);
```

Этот код также является частью прикладного проекта ArraysTest.

Многомерные статические массивы

Массив может иметь более одного измерения, выражающего матрицу или куб, а не список. Вот два примера определений:

```
type
  TAllMonthTemps = array [1..24, 1..31] of Integer;
  TAllYearTemps = array [1..24, 1..31, 1..12] of Integer;
```

Вы можете получить доступ к элементу как:

```
var
  AllMonth1: TAllMonthTemps;
```

```
  AllYear1: TAllYearTemps;
begin
  AllMonth1 [13, 30] := 55; // hour, day
  AllYear1 [13, 30, 8] := 55; // hour, day, month
```

примечание Статические массивы сразу же занимают много памяти (в случае, описанном выше, в стеке), чего следует избегать. Для переменной `AllYear1` требуется 8928 целых чисел, занимающих по 4 байта, то есть почти 35 КБ. Выделение такого большого блока в глобальной памяти или на стеке (как в демонстрационном коде) - действительно ошибка. Вместо этого динамический массив использует кучу памяти и предлагает гораздо большую гибкость в плане выделения памяти и управления ею.

Поскольку эти два типа массивов построены на одних и тех же основных типах, то лучше объявлять их, используя предыдущие типы данных, как в следующем коде:

```
type
  TMonthTemps = array [1..31] of TDayTemperatures;
  TYearTemps = array [1..12] of TMonthTemps;
```

Эта декларация инвертирует порядок индексов, в отличие от примера выше, но она также позволяет присваивать целые блоки между переменными. Посмотрим, как можно назначить отдельные значения:

```
  Month1 [30][14] := 44;
  Month1 [30, 13] := 55; // day, hour
  Year1 [8, 30, 13] := 55; // month, day, hour
```

Теоретически, следует использовать первую строку, выделяя один из массивов, а затем элемент результирующего массива. Однако вариант с двумя индексами в квадратных скобках также допускается. Или с тремя индексами, в примере "куб".

Важность использования промежуточных типов заключается в том, что массивы совместимы с типами только в том случае, если они ссылаются на одно и то же точное имя типа (т.е. точно такое же определение типа), а не в том случае, если их определения типа случайно ссылаются на одну и ту же реализацию. Это правило совместимости типов одно и то же для всех типов в Object Pascal, лишь с некоторыми специфическими исключениями.

Например, в следующем заявлении копируются месячные температуры третьего месяца года:

```
Year1[3] := Month1;
```

Вместо этого, аналогичное утверждение основано на самостоятельных определениях массивов (которые не совместимы с типами):

```
AllYear1[3] := AllMonth1;
```

приведет к ошибке:

```
Error: Incompatible types: 'array[1..31] of array[1..12] of Integer' and
'TAllMonthTemps'
Ошибка: Несовместимые типы: 'array[1..31] of array[1..12] of Integer' и
'TAllMonthTemps'.
```

Как я уже упоминал, статические массивы страдают от проблем с управлением памятью, а именно, когда вы хотите передать их в качестве параметров или выделить память только для части большого массива. Более того, их размер нельзя изменить за время жизни переменной массива. Поэтому предпочтительнее использовать динамические массивы, даже если они требуют немного больше дополнительного управления, например, в отношении выделения памяти.

Динамические массивы

В традиционном языке Pascal массивы имели фиксированный размер, и вы указывали количество элементов массива при объявлении типа данных. Объект Pascal также поддерживает прямую и нативную реализацию динамических массивов.

примечание "Прямая реализация динамических массивов" здесь контрастирует с использованием указателей и динамическим выделением памяти для получения аналогичного эффекта... с очень сложным и подверженным ошибкам кодом. Кстати, динамические массивы — это единственный вид конструкции в большинстве современных языков программирования.

Динамические массивы динамически выделяются и подсчитываются ссылки (что делает передачу параметра намного быстрее, так как передается только ссылка, а не копия полного массива). После этого можно очистить массив, присвоив его переменной `nil` или установив его длину равной нулю, но учитывая, что они являются ссылочными, в большинстве случаев компилятор автоматически освобождает память за вас.

В динамическом массиве вы объявляете тип массива без указания количества элементов, а затем выделяете его с заданным размером с помощью процедуры `SetLength`:

```
var
  Array1: array of Integer;
begin
  // this would cause a runtime Range Check error
  // Array1 [1] := 100;
  SetLength (Array1, 10);
  Array1 [1] := 100; // this is OK
```

Вы не можете использовать массив, пока не назначите его длину, выделив необходимую память на куче. Без этого вы увидите либо ошибку Range Check (если активна соответствующая опция компилятора), либо Access Violation (на Windows), либо подобную ошибку доступа к памяти на другой платформе. Вызов `SetLength` устанавливает все значения элементов в ноль. Код инициализации позволяет сразу же начать чтение и запись значений массива, не опасаясь ошибок работы с памятью (если только вы не нарушаете границы массива).

Хотя вам действительно нужно выделить память явно, вам не требуется освобождать ее напрямую. В приведенном выше фрагменте кода, когда код завершается и переменная `Array1` выходит за рамки видимости, компилятор автоматически освобождает память (в данном случае выделенные десять целых чисел). Таким образом, хотя можно присвоить

переменной динамического массива значение nil или вызвать `SetLength` со значением `0`, в этом, как правило, нет необходимости (и делается это редко).

Обратите внимание, что процедура SetLength также может быть использована для изменения размера массива без потери его текущего содержимого (если вы его увеличиваете), но вы потеряете некоторые элементы (если вы его уменьшаете). Так как в начальном вызове `SetLength` вы указываете только количество элементов массива, то индекс динамического массива неизменно начинается с `0` и возрастает до количества элементов минус 1. Другими словами, динамические массивы не поддерживают две особенности классических статических паскальных массивов - ненулевой нижний предел и нечисловые индексы. В то же время, они более точно соответствуют тому, как массивы работают в большинстве языков, основанных на синтаксисе C.

Подобно статическим массивам, чтобы узнать текущий размер динамического массива, можно воспользоваться функциями Length, High и Low. Для динамических массивов, однако, функция `Low` всегда возвращает `0`, а функция `High` всегда возвращает длину минус единица. Это означает, что для пустого массива `High` возвращает `-1` (что, если подумать, является странным значением, так как оно меньше значения, возвращаемого функцией `Low`).

Так, например, в проекте приложения DynArray я заполнил и извлек информацию из динамического массива с помощью адаптируемых циклов. Это определение типа и переменной:

```
type
  TIntegersArray = array of Integer;
var
  IntArray1: TIntegersArray;
```

Массив выделяется и заполняется значениями, соответствующими индексу, с помощью следующего цикла:

```
var
  I: Integer;
begin
  SetLength (IntArray1, 20);
  for I := Low (IntArray1) to High (IntArray1) do
    IntArray1 [I] := I;
end;
```

Вторая кнопка имеет код как для отображения каждого значения, так и для вычисления среднего, аналогичного предыдущему примеру, но в одном цикле:

```
var
  I: Integer;
  Total: Integer;
begin
  Total := 0;
  for I := Low(IntArray1) to High(IntArray1) do
  begin
    Inc (Total, IntArray1[I]);
    Show (I.ToString + ': ' + IntArray1[I].ToString);
  end;
  Show ('Average: ' + (Total / Length(IntArray1)).ToString);
end;
```

Вывод этого кода вполне очевиден (и в основном пропущен):

```
0: 0
1: 1
2: 2
3: 3
...
17: 17
18: 18
19: 19
Average: 9.5
```

Помимо `Length`, `SetLength`, `Low` и `High`, существуют и другие распространенные процедуры, которые можно использовать на массивах, такие как функция Copy, которую можно использовать для копирования части массива (или всего массива). Обратите внимание, что вы также можете присваивать массив из одной переменной другой, но в этом случае вы не делаете полную копию, а имеете в памяти две переменные, ссылающиеся на один и тот же массив.

Единственный более сложный код находится в заключительной части проекта приложения DynArray, которое копирует один массив в другой двумя разными способами:

используя функцию Copy, которая дублирует данные массива в новой структуре данных, используя отдельную область памяти

используя оператор присваивания, который фактически создает псевдоним, новую переменную, ссылающуюся на тот же массив в памяти

На этом этапе, если вы модифицируете один из элементов нового массива, вы повлияете на исходную версию или нет в зависимости от того, как вы сделали копию. Вот полный код:

```
var
  IntArray2: TIntegersArray;
  IntArray3: TIntegersArray;
begin
  // alias
  IntArray2 := IntArray1;

  // separate copy
  IntArray3 := Copy (IntArray1, Low(IntArray1), Length(IntArray1));

  // modify items
  IntArray2 [1] := 100;
  IntArray3 [2] := 100;

  // check values for each array
  Show (Format ('[%d] %d -- %d -- %d',
    [1, IntArray1 [1], IntArray2 [1], IntArray3 [1]]));
  Show (Format ('[%d] %d -- %d -- %d',
    [2, IntArray1 [2], IntArray2 [2], IntArray3 [2]]));
```

Выдача, которую вы получите, будет выглядеть следующим образом:

```
[1] 100 -- 100 -- 1
[2] 2 -- 2 -- 100
```

Изменения в `IntArray2` распространяются на `IntArray1`, потому что это всего лишь две ссылки на один и тот же физический массив; изменения в `IntArray3` - отдельные, потому что у него есть отдельная копия данных.

Внутренние операции на динамических массивах

Динамические массивы имеют поддержку присвоения константных массивов и конкатенирования.

примечание Эти расширения для динамических массивов были добавлены в Delphi XE7 и помогают разработчику почувствовать ценность динамических массивов и очевидный выбор (забыв о своем статическом аналоге).

На практике можно написать код, который будет значительно проще по сравнению с более ранними фрагментами кода:

```
var
  DI: array of Integer;
  I: Integer;
begin
  DI := [1, 2, 3];    // initialization
  DI := DI + DI;      // concatenation
  DI := DI + [4, 5];  // mixed concatenation

  for I in DI do
  begin
    Show (I.ToString);
  end;
```

Обратите внимание на использование оператора for-in для сканирования элементов массива в данном коде, который является частью проекта приложения DynArrayConcat. Обратите внимание, что эти массивы могут быть основаны на любом типе данных, от простых целых чисел, как в коде выше, до записей и классов.

Есть второе дополнение, которое было сделано наряду с присвоением и конкатендацией, но это больше часть RTL, чем языка. Теперь можно использовать на динамических массивах функции, которые были обычными для строк, такие как Insert и Delete.

Это означает, что теперь вы можете писать код, подобный следующему (часть одного и того же проекта):

```
var
```

```
  DI: array of Integer;
  I: Integer;
begin
  DI := [1, 2, 3, 4, 5, 6];
  Insert ([8, 9], DI, 4);
  Delete (DI, 2, 1); // remove the third (0-based)
```

Открытый массив параметров

Существует совершенно особый сценарий использования массивов, который заключается в передаче функции гибкого списка параметров. Помимо непосредственной передачи массива, в этом и следующем разделе объясняются две специальные синтаксические структуры. Примером такой функции, кстати, является функция `Format`, которую я вызывал в последнем фрагменте кода, и которая имеет массив значений в квадратных скобках в качестве второго параметра.

В отличие от языка C (и некоторых других языков, основанных на синтаксисе C), в традиционном языке Pascal функция или процедура всегда имеет фиксированное количество параметров. Однако в Object Pascal существует способ передачи в подпрограмму различного числа параметров, используя в качестве параметра массив, метод, известный как *открытый массив параметров*.

примечание Исторически сложилось так, что *открытые массивы параметров* предшествовали динамическим массивам, но сегодня эти две возможности выглядят настолько похожими в своей работе, что в наши дни они практически неотличимы друг от друга. Поэтому я остановился на открытых массивах параметров только после обсуждения динамических массивов.

Основное определение открытого массива параметров такое же, как и для типизированного типа динамического массива, префикс которого задан спецификатором `const`. Это означает, что вы указываете тип параметра(ов), но вам не нужно указывать, сколько элементов этого типа будет иметь массив.

Приведем пример такого определения, извлеченного из проекта приложения OpenArray:

```
function Sum (const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(A) to High(A) do
    Result := Result + A[I];
end;
```

Эту функцию можно вызвать, передав ей константное выражение *array-of-Integer* (которое также может включать переменные как часть выражений, используемых для вычисления отдельных значений):

```
X := Sum ([10, Y, 27*I]);
```

Получив динамический `array of Integer`, можно передать его непосредственно в подпрограмму, требующую параметра – открытого массива того же базового типа (в данном случае Целого). Вот пример, где в качестве параметра передается полный массив:

```
var
  List: array of Integer;
  X, I: Integer;
begin
  // initialize the array
  SetLength (List, 10);
  for I := Low (List) to High (List) do
    List [I] := I * 2;
  // call
  X := Sum (List);
```

Это если у вас есть динамический массив. Если у вас есть статический массив соответствующего базового типа, вы также можете передать его функциям, ожидающим параметр открытого массива, или вызвать функцию Slice для передачи только части существующего массива (как указывает его второй параметр). Следующий фрагмент (также часть проекта приложения OpenArray) показывает, как передать статический массив или его часть функции `Sum`:

```
var
  List: array [1..10] of Integer;
  X, I: Integer;
begin
  // initialize the array
  for I := Low (List) to High (List) do
    List [I] := I * 2;

  // call
  X := Sum (List);
  Show (X.ToString);

  // pass portion of the array
  X := Sum (Slice (List, 5));
  Show (X.ToString);
```

Параметры открытых массивов типа variant

Помимо этих типизированных открытых массивов параметров, язык Object Pascal позволяет определять variant или не типизированные открытые массивы. Этот специальный тип массива имеет не только неопределенное количество элементов, но и неопределенный тип данных для этих элементов, а также возможность передачи элементов различных типов. Это одна из ограниченных областей языка, которая не является полностью безопасной в работе с типами.

Технически, можно определить параметр `array of const` для передачи в функцию массива с неопределенным количеством элементов различных типов. Например, вот определение функции `Format` (мы посмотрим, как использовать эту функцию в главе 6, рассматривая строки, но я уже использовал ее на некоторых демонстрациях):

```
function Format (const Format: string;
  const Args: array of const): string;
```

Вторым параметром является открытый массив, который получает неопределенное количество значений. Фактически, эту функцию можно вызвать следующими способами:

```
N := 20;
```

```
S := 'Total:';
Show (Format ('Total: %d', [N]));
Show (Format ('Int: %d, Float: %f', [N, 12.4]));
Show (Format ('%s %d', [S, N * 2]));
```

Обратите внимание, что параметр может передаваться либо как постоянная величина, либо как значение переменной, либо как выражение. Объявить такую функцию просто, но как ее закодировать? Как узнать типы параметров? Значения параметра open array типа variant совместимы с элементами типа `TVarRec`.

примечание Не путайте запись TVarRec с записью TVarData, используемой типом Вариант. Эти две структуры имеют разную цель и не совместимы. Даже список возможных типов отличается, поскольку `TVarRec` может содержать типы данных Object Pascal, в то время как `TVarData` может содержать типы данных Windows OLE. Варианты рассматриваются ниже в этой главе.

Ниже приведены типы данных, поддерживаемые значением элемента открытого массива типа variant и записью TVarRec:

```
vtInteger      vtBoolean      vtChar
vtExtended     vtString       vtPointer
vtPChar        vtObject       vtClass
vtWideChar     vtPWideChar    vtAnsiString
vtCurrency     vtVariant      vtInterface
vtWideString   vtInt64        vtUnicodeString
```

Структура записи имеет поле с типом (`VType`) и поле варианта, которое можно использовать для доступа к реальным данным (подробнее о записях через несколько страниц, даже если это расширенное использование для данной конструкции).

Типичный подход заключается в использовании оператора case для работы с различными типами параметров, которые вы можете получить при таком вызове. В примере функции `SumAll` я хочу иметь возможность суммировать значения различных типов, преобразовывая строки в целые числа, символы в соответствующее порядковое значение и добавляя 1 для True Boolean значений. Код, конечно, достаточно продвинутый (и в

нем используются разыменования указателей), так что не волнуйтесь, если вы пока не до конца его понимаете:

```
function SumAll (const Args: array of const): Extended;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(Args) to High (Args) do
    case Args [I].VType of
      vtInteger:
        Result := Result + Args [I].VInteger;
      vtBoolean:
        if Args [I].VBoolean then
          Result := Result + 1;
      vtExtended:
        Result := Result + Args [I].VExtended^;
      vtWideChar:
        Result := Result + Ord (Args [I].VWideChar);
      vtCurrency:
        Result := Result + Args [I].VCurrency^;
    end; // case
end;
```

Я добавил эту функцию в проект приложения OpenArray, который вызывает ее следующим образом:

```
var
  X: Extended;
  Y: Integer;
begin
  Y := 10;
  X := SumAll ([Y * Y, 'k', True, 10.34]);
  Show ('SumAll: ' + X.ToString);
end;
```

Вывод этого вызова добавляет квадрат `Y`, порядковое значение *K* (которое равно 107), 1 для булева значения, и расширенное число, в результате чего выводится:

```
218.34
```

Типы данных record

В то время как массивы определяют списки идентичных элементов, на которые ссылается цифровой индекс, записи (`record`) определяют группы элементов различных типов, на которые ссылаются по имени. Другими словами, запись представляет собой список названных элементов или полей, каждый из которых имеет определенный тип данных. Определение типа записи перечисляет все эти поля, присваивая каждому из них название, используемое для ссылки на него. Если в начале Паскаля записи имели только поля, то теперь они могут иметь и методы, и операторы, как мы увидим в этой главе.

примечание Записи доступны в большинстве языков программирования. Они определяются ключевым словом `struct` в языке Си, в то время как в C++ есть расширенное определение, включающее методы, точно так же, как в Object Pascal. Некоторые более "чистые" объектно-ориентированные языки имеют только понятие класса, а не записи или структуры, но C# недавно вновь ввел это понятие.

Вот небольшой фрагмент кода (из прикладного проекта RecordsDemo) с определением типа записи, объявлением переменной этого типа и несколькими операторами, использующими эту переменную:

```
type
  TMyDate = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;

var
  BirthDay: TMyDate;
begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
  Show ('Born in year ' + BirthDay.Year.ToString);
```

примечание Термин "записи" иногда используется довольно вольно для обозначения двух различных элементов языка: определения типа записи и переменной типа записи (или экземпляра записи). Запись используется как синоним как типа записи, так и экземпляра записи, в отличие от типов классов, в которых экземпляр называется объектом.

В Object Pascal эта структура данных гораздо больше, чем простой список полей, как будет показано в оставшейся части этой главы, но давайте начнем с этого *традиционного* подхода к записям. Память для записи обычно выделяется в стеке для локальной переменной и в глобальной памяти для глобальной. Это подчеркивается вызовом SizeOf, который возвращает количество байт, требуемое переменной или типу, как в данном операторе:

```
Show ('Record size is ' + SizeOf (BirthDay).ToString);
```

который возвращает 8 (почему он возвращает 8, а не 6, 4 байта для Целого и по два для каждого байтового поля, я расскажу в разделе "Выравнивание полей").

Другими словами, записи — это типы значений. Это означает, что, если вы присваиваете запись другому, вы делаете полную копию. Если вы внесете изменения в копию, это не повлияет на исходную запись. Этот фрагмент кода объясняет это понятие с точки зрения кода:

```
var
  BirthDay: TMyDate;
  ADay: TMyDate;
begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;

  ADay := Birthday;
  ADay.Year := 2008;

  Show (MyDateToString (BirthDay));
  Show (MyDateToString (ADay));
```

Вывод (в японском или международном формате даты):

```
1997.2.14
2008.2.14
```

Та же самая операция копирования происходит, когда вы передаете запись в качестве параметра в функцию, как в MyDateToString, которую я использовал выше:

```
function MyDateToString (MyDate: TMyDate): string;
begin
  Result := MyDate.Year.ToString + '.' +
    MyDate.Month.ToString + '.' +
    MyDate.Day.ToString;
end;
```

Каждый вызов этой функции включает в себя полную копию данных записи. Чтобы избежать копирования и, возможно, внести изменения в исходную запись, необходимо явно использовать параметр ссылки. Это выделяется следующей процедурой, которая вносит некоторые изменения в запись, переданную в качестве параметра:

```
procedure IncreaseYear (var MyDate: TMyDate);
begin
  Inc (MyDate.Year);
end;

var
  ADay: TMyDate;
begin
  ADay.Year := 2020;
  ADay.Month := 3;
  ADay.Day := 18;

  Increaseyear (ADay);
  Show (MyDateToString (ADay));
```

Учитывая, что поле Year первоначального значения записи увеличивается при вызове процедуры, конечный результат выводится на год позже ввода:

```
2021.3.18
```

Использование массивов записей

Как я уже упоминал, массивы представляют собой структуру данных, повторяющуюся несколько раз, в то время как record – одна структура с различными элементами. Учитывая, что эти

два типа конструкторов ортогональны, очень часто их используют вместе, определяя массивы записей (в то время как массив в записи можно увидеть, но редко).

Код массива подобен коду любого другого массива, при этом каждый элемент массива принимает размер определенного типа записи. Хотя позже мы увидим, как использовать более сложные классы коллекций или контейнеров (для списков элементов), с точки зрения управления данными можно достичь многого с помощью массивов записей.

В проект приложения RecordsTest я добавил массив типа `TMyDate`, который можно выделить, инициализировать и использовать с таким кодом:

```
var
  DatesList: array of TMyDate;
  I: Integer;
begin
  // allocate array elements
  SetLength (DatesList, 5);

  // assign random values
  for I := Low(DatesList) to High(DatesList) do
  begin
    DatesList[I].Year := 2000 + Random (50);
    DatesList[I].Month := 1 + Random (12);
    DatesList[I].Day := 1 + Random (27);
  end;

  // display the values
  for I := Low(DatesList) to High(DatesList) do
    Show (I.ToString + ': ' +
      MyDateToString (DatesList[I]));
```

Учитывая, что приложение использует случайные данные, выходные данные будут каждый раз отличаться, и могут быть похожими на следующие данные, которые я получил:

```
0: 2014.11.8
1: 2005.9.14
2: 2037.9.21
3: 2029.3.12
4: 2012.7.2
```

примечание Запись в массиве может быть автоматически инициализирована в случае управляемой записи, функция, введенная в Delphi 10.4 Sydney, о которой я расскажу позже в этой главе.

Вариантные записи

Начиная с ранних версий языка, типы record могут также иметь вариантную часть, т.е. несколько полей могут быть отображены в одну и ту же область памяти, даже если они имеют разные типы данных. (Это соответствует *объединению union* в языке Си.) Альтернативно, вы можете использовать эти варианты полей или групп полей для доступа к одной и той же области памяти внутри записи, но с учетом этих значений с точки зрения типов данных. Основными применениями этого типа были хранение похожих, но разных данных и получение эффекта, схожего с приведением типов (это использовалось в первые годы существования языка, когда прямое типовое приведение не разрешалось). Использование вариантных типов записей было в значительной степени заменено объектно-ориентированными и другими современными методами, хотя некоторые системные библиотеки в особых случаях используют их внутри себя.

Использование вариантных типов записи не является безопасным и не является рекомендуемой практикой программирования, особенно для новичков. В любом случае, вам не придется заниматься ими до тех пор, пока вы не станете специалистом по Object Pascal... и поэтому я решил не показывать вам реальные примеры и не освещать эту возможность более подробно. Если вам действительно нужна подсказка, посмотрите на использование `TvarRec,` которое я сделал в примере раздела "Параметры в открытых массивах Type-Variant".

Выравнивание полей

Другая продвинутая тема, связанная с записями, это способ выравнивания их полей, который также помогает понять реальный размер записи. Если вы посмотрите на библиотеки, то часто увидите использование ключевого слова `packed`, применяемого к записям: это подразумевает, что запись должна использовать минимально возможное количество байт, даже если это приводит к замедлению операций доступа к данным.

Традиционно разница связана с 16-битным или 32-битным выравниванием различных полей, так что байт, за которым следует целое число, может в итоге занять 32 бита, даже если используется только 8. Это связано с тем, что доступ к следующему целому значению на 32-битной границе ускоряет выполнение кода.

примечание Размер полей и их выравнивание зависят от размера типа. Для любого типа с размером, не равным 2 (или 2^N), размер — это следующая более высокая степень 2. Так, например, тип `Extended`, использующий 10 байт, в записи занимает 16 байт (если только запись не `упаковывается`).

В общем выравнивание полей используется структурами данных, такими как записи, для повышения скорости доступа к отдельным полям для некоторых процессорных архитектур. Существуют различные параметры, которые можно применить к директиве компилятора $ALIGN для ее изменения.

С помощью `{$ALIGN 1}` компилятор будет экономить на использовании памяти, используя все возможные байты, как при использовании спецификатора `packed` для записи. С другой стороны, `{$ALIGN 16}` будет использовать наибольшее выравнивание. Дополнительные опции используют 4 и 8.

Например, если я вернусь в проект RecordsTest и добавлю в определение записи ключевое слово `packed`:

```
type
  TMyDate = packed record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;
```

В результате вызов SizeOf теперь вернет 6, а не 8.

В качестве более продвинутого примера, который можно пропустить, если вы еще не являетесь опытным разработчиком Object Pascal, рассмотрим следующую структуру (доступную в проекте приложения AlignTest):

```
type
  TMyRecord = record
    c: Byte;
    w: Word;
    b: Boolean;
    I: Integer;
    d: Double;
  end;
```

При {$ALIGN `1}` структура занимает 16 байт (значение, возвращаемое `sizeOf`), а поля будут находиться по следующим относительным адресам памяти:

```
c: 0 w: 1 b: 3 i: 4 d: 8
```

примечание Относительные адреса вычисляются путем размещения записи и вычисления разницы между числовым значением указателя на структуру и значением указателя на данное поле, с выражением типа: `UIntPtr(@MyRec.w) - UintPtr(@MyRec1)`. Понятие `pointer` и оператора `address of` (@) будут рассмотрены далее в этой главе.

Напротив, если изменить выравнивание на 4 (что может привести к оптимизации доступа к данным), то размер будет 20 байт и относительные адреса:

```
c: 0 w: 2 b: 4 i: 8 d: 12
```

Если вы перейдете к экстремальной опции и используете {$ALIGN `16}`, то структура требует 24 байта и отображает поля следующим образом:

```
c: 0 w: 2 b: 4 i: 8 d: 16
```

Особенности оператора With

Еще одним традиционным языковым оператором, используемым для работы с записями или классами, является оператор `with`. Раньше всего это ключевое слово было свойственно синтаксису Pascal, но позже оно было введено в JavaScript и Visual Basic. Это ключевое слово может оказаться очень удобным для написания меньшего количества кода, но оно также может стать очень опасным, так как делает код намного менее читабельным.

Вы найдете много споров вокруг оператора `with`, и я склонен согласиться, что его следует использовать экономно, если использовать вообще. В любом случае, я считаю важным включить его в эту книгу (в отличие от утверждений "`goto`").

примечание В настоящее время ведутся дебаты о том, имеет ли смысл удалять операторы `goto` из языка Object Pascal, а также обсуждается вопрос о том, следует ли удалять `with` из мобильной версии этого языка. Несмотря на то, что существуют некоторые законные способы использования, учитывая, какие проблемы обработки операторы `with` могут вызвать, есть веские причины для прекращения использования этой функции (или изменения ее таким образом, чтобы требовался псевдоним, как в C#).

Утверждение `with` — это всего лишь сокращение. Когда вам нужно обратиться к переменной типа `record` (или объекту), вместо того, чтобы повторять ее имя каждый раз, вы можете использовать выражение `with`.

Например, представляя тип записи, я написал этот код:

```
var
  BirthDay: TMyDate;
begin
  BirthDay.Year := 2008;
  BirthDay.Month := 2;
  BirthDay.Day := 14;
```

Используя утверждение `with`, я мог бы модифицировать заключительную часть этого кода следующим образом:

```
 with BirthDay do
```

```
  begin
    Year := 2008;
    Month := 2;
    Day := 14;
  end;
```

Этот подход может быть использован в программах Object Pascal для обращения к компонентам и другим классам. При работе с компонентами или классами в целом, оператор `with` позволяет пропустить написание некоторого кода, в частности, для вложенных структур данных.

Так почему же я не поощряю использование с заявлением? Причина в том, что это может вызвать тонкие ошибки, которые очень трудно «поймать». Хотя некоторые из этих трудноуловимых ошибок нелегко объяснить на данном этапе книги, давайте рассмотрим простой сценарий, который все же может заставить вас почесать в затылке. Вот тип записи и какой-то код, использующий его:

```
type
  TMyRecord = record
    MyName: string;
    MyValue: Integer;
  end;

procedure TForm1.Button2Click(Sender: TObject);
var
  Record1: TMyRecord;
begin
  with Record1 do
  begin
    MyName := 'Joe';
    MyValue := 22;
  end;

  with Record1 do
    Show (Name + ': ' + MyValue.ToString);
```

Все верно? Приложение компилируется и запускается, но его вывод не тот, что вы могли бы ожидать (по крайней мере, на первый взгляд):

```
Form1: 22
```

Строковая часть вывода не является тем значением записи, которое было установлено ранее. Причина заключается в том, что второй оператор with ошибочно использует поле `Name`, которое является не полем записи, а еще одним полем, которое случайно попадает в область видимости (в частности, имя объекта формы, частью которого является метод `Button2Click`).

Если бы ты написал:

```
Show (Record1.Name + ': ' + Record1.MyValue.ToString);
```

компилятор выдал бы сообщение об ошибке, указывающее на то, что данная структура записи не имеет поля `Name`.

В общем, можно сказать, что поскольку утверждение `with` вводит новые идентификаторы в текущем scope, мы можем скрыть существующие идентификаторы, или неправомерно получить доступ к другому идентификатору в том же scope. Это является хорошей причиной для того, чтобы отказаться от использования выражения `with`. Более того, следует избегать использования нескольких утверждений `with`, например:

```
with MyRecord1, MyDate1 do...
```

Следующий за этим код, вероятно, будет сильно нечитаемым, так как для каждого поля, используемого в блоке, нужно будет подумать, к какой записи он относится.

Методы записей

В Object Pascal записи более мощные, чем в оригинальном языке Паскаль или чем структуры в языке С. Записи, на самом деле, могут иметь процедуры и функции (называемые методами), связанные с ними. Они могут даже переопределять операторы языка настраиваемыми способами (функция,

называемая *перегрузкой операторов*), как вы увидите в следующем разделе.

Запись с методами несколько похожа на класс, как мы узнаем позже, с самым главным отличием в том, как эти две структуры управляют памятью. Запись в Object Pascal имеет две фундаментальные особенности современных языков программирования:

Методы, которые представляют собой функции и процедуры, связанные со структурой данных записи и имеющие прямой доступ к полям записи. Другими словами, методы — это функции и процедуры, объявленные (или имеющие прямое объявление) в рамках определения типа записи.

Инкапсуляция — это возможность скрыть прямой доступ к некоторым полям (или методам) структуры данных от остальной части кода. Инкапсуляцию можно получить с помощью спецификатора доступа `private`, в то время как поля и методы, видимые извне, помечены как `public`. Спецификатор по умолчанию для записи является `public`.

Теперь, когда у вас есть основные понятия вокруг расширенных записей, давайте посмотрим на определение образца записи, взятого из прикладного проекта RecordMethods:

```
type
  TMyRecord = record
  private
    Name: string;
    Value: Integer;
    SomeChar: Char;
  public
    procedure Print;
    procedure SetValue (NewString: string);
    procedure Init (NewValue: Integer);
  end;
```

Вы можете видеть, что структура записи разделена на две части: частную и публичную. Вы можете иметь несколько разделов, так как ключевые слова private и public могут

повторяться столько раз, сколько вам нужно, но четкое разделение этих двух разделов, безусловно, помогает читабельности. Методы перечислены в определении записи (как в определении класса) без их полного кода. Другими словами, метод имеет предварительное объявление.

Как написать реальный код метода, его полное определение? Почти так же, как вы бы кодировали глобальную функцию или процедуру. Различия заключаются в том, как вы пишете имя метода, которое представляет собой комбинацию имени типа записи и фактического имени записи, а также в том, что вы можете напрямую обращаться к полям и другим методам записи, без необходимости записывать имя записи:

```
procedure TMyRecord.SetValue (NewString: string);
begin
  Name := NewString;
end;
```

совет Хотя сначала может показаться утомительным написание объявления метода, а затем его полное определение, можно использовать комбинацию Ctrl+Shift+C в редакторе IDE для автоматической генерации одного из другого. Также можно использовать клавиши Ctrl+Shift+Up/Down Arrow для перехода от объявления метода к соответствующему определению и наоборот.

Ниже приведен код других методов данного типа записи:

```
procedure TMyRecord.Init(NewValue: Integer);
begin
  Value := NewValue;
  SomeChar := 'A';
end;

function TMyRecord.ToString: string;
begin
  Result := Name + ' [' + SomeChar + ']: ' + Value.ToString;
end;
```

Вот пример того, как вы можете использовать эту запись:

```
var
  MyRec: TMyRecord;
begin
  MyRec.Init(10);
  MyRec.SetValue ('hello');
  Show (MyRec.ToString);
```

Как вы могли догадаться, результат будет:

```
hello [A]: 10
```

Что будет, если вы хотите использовать поля из кода, который использует запись, как в фрагменте выше:

```
 MyRec.Value := 20;
```

На самом деле это компилируется и работает, что может быть удивительно, так как мы объявили поле в приватной секции, так что только методы записи могут получить доступ к нему.

Дело в том, что в Object Pascal спецификатор приватного доступа на самом деле включен только между различными модулями, так что эта строка не будет легальной в другом модуле, но может быть использована в модуле, который изначально определил тип данных. Как мы увидим, это также верно и для классов.

Self: Магия записей

Предположим, у вас есть две записи, например, `myrec1` и `myrec2` одного типа. Когда вы вызываете метод и выполняете его код, как метод узнает, с какой из двух копий записи он должен работать? За кулисами, когда вы определяете метод, компилятор добавляет к нему скрытый параметр, ссылку на запись, к которой вы применили метод.

Другими словами, вызов вышеприведенного метода преобразуется компилятором в нечто подобное:

```
// you write
MyRec.SetValue ('hello');

// the compiler generates
SetValue (@MyRec, 'hello');
```

В этом псевдокоде, @ — это оператор получения адреса, используемый для получения места в памяти экземпляра записи.

примечание Опять же, оператор @ вкратце рассматривается в конце этой главы в (расширенном) разделе "Что насчет указателей?".

Так транслируется код вызова, но как реальный вызов метода может ссылаться и использовать этот скрытый параметр? Неявно используя специальное ключевое слово, называемое `self`. Таким образом, код метода может быть написан как:

```
procedure TMyRecord.SetValue (NewString: string);
begin
  self.Name := NewString;
end;
```

Пока этот код компилируется, нет смысла использовать `self` явно, если только не нужно ссылаться на запись в целом, например, передавать запись в качестве параметра в другую функцию. Это чаще всего происходит с классами, которые имеют один и тот же скрытый параметр для методов и одно и то же ключевое слово `self`.

Одна из ситуаций, в которой использование явного параметра `self` может сделать код более читабельным (даже если это не требуется) — это когда вы манипулируете второй структурой данных того же типа, как в случае, если вы тестируете значение из другого экземпляра:

```
function TMyRecord.IsSameName (ARecord: TMyRecord): Boolean;
begin
  Result := (self.Name = ARecord.Name);
end;
```

примечание «Скрытый» параметр self называется this в C++ и Java, но в Objective-C (и, конечно же, в Object Pascal) он называется `self`.

Инициализация записей

Когда вы определяете переменную типа записи (или экземпляр записи) как глобальную переменную, ее поля инициализируются, но когда вы определяете ее на стеке (как локальную переменную функции или процедуры, это не так). Таким образом, если вы пишете подобный код (также являющийся частью проекта RecordMethods):

```
var
  MyRec: TMyRecord;
begin
  Show (MyRec.ToString);
```

его выход будет более или менее случайным. Пока строка инициализируется на пустую строку, символьное и целое поля будут содержать данные, которые случайно оказались в заданной ячейке памяти (так же, как это происходит в общем случае для символьной или целочисленной переменной на стеке). В общем случае, в зависимости от фактической компиляции или выполнения, вы получите различные результаты:

```
 [□]: 1637580
```

Поэтому важно инициализировать запись (как и большинство других переменных) перед ее использованием, чтобы избежать риска чтения нелогичных данных, которые даже потенциально могут привести к краху приложения.

Есть два радикально разных подхода к этому сценарию. Первый — это использование конструкторов для записи, о чем будет сказано в следующем разделе. Второй - использование управляемых записей, новая возможность в Delphi 10.4, о которой я расскажу позже в этой главе.

Записи и конструкторы

Давайте начнем с обычных конструкторов. Записи поддерживают специальный тип методов, называемых конструкторами, с помощью которых можно инициализировать данные записи. В отличие от других методов, конструкторы также могут быть применены к типу записи для определения нового экземпляра (но они все равно могут быть применены к существующему экземпляру).

Так можно добавить конструктор к записи:

```
type
  TMyNewRecord = record
  private
    ...
  public
    constructor Create (NewString: string);
    function ToString: string;
    ...
```

Конструктор - это метод с кодом:

```
constructor TMyNewRecord.Create (NewString: string);
begin
  Name := NewString;
  Init (0);
end;
```

Теперь вы можете инициализировать запись любым из двух следующих стилей кодирования:

```
var
  MyRec, MyRec2: TMyNewRecord;
begin
  MyRec := TMyNewRecord.Create ('Myself'); // class-like
  MyRec2.Create ('Myself'); // direct call
```

Обратите внимание, что конструкторы записей должны иметь параметры: Если вы попробуете использовать `Create()`, то получите сообщение об ошибке "Конструкторы без параметров запрещены для типов записей".

примечание Согласно документации определение безпараметрического конструктора для записей зарезервировано за системой (которая имеет свой способ инициализации

некоторых полей записей, таких как строки и интерфейсы). Поэтому любой определяемый пользователем конструктор должен иметь хотя бы один параметр. Конечно, можно также иметь несколько перегруженных конструкторов или несколько конструкторов с разными именами. Подробнее об этом я расскажу при обсуждении конструкторов для классов. Управляемые записи, как мы увидим вкратце, используют другой синтаксис и вводят не конструкторы без параметров, а метод класса `Initialize`.

Операторы получают новую основу

Другой особенностью языка Object Pascal, связанной с записями, является перегрузка операторов, то есть возможность определить собственную реализацию для стандартных операций (сложение, умножение, сравнение и т.д.) над вашими типами данных. Идея заключается в том, что вы можете реализовать оператор добавления (специальный метод `Add`), а затем использовать знак + для его вызова. Для определения оператора вы используете комбинацию ключевых слов оператора класса.

примечание Повторно используя существующие зарезервированные слова, дизайнеры языка сумели не повлиять на существующий код. Это то, что они довольно часто делали в последнее время в комбинациях ключевых слов, таких как `strict private`, `class operator` и `class var`.

Термин *класс* здесь относится к методам класса, понятие, которое мы рассмотрим намного позже (в Гл. 12). После директивы вы пишете имя оператора, например `Add`:

```
type
  TPointRecord = record
  public
    class operator Add (
      A, B: TPointRecord): TPointRecord;
```

The operator `Add` is then called with the + symbol, as you'd expect:

```
var
  A, B, B: TPointRecord;
```

```
begin
  ...
  B := A + B;
```

Так какие операторы доступны? В основном весь набор операторов языка, так как вы не можете определить совершенно новые операторы языка:

- **Операторы приведения**: Неявные и явные
- **Унарные операторы**: `Positive`, `Negative`, `Inc`, `Dec`, `LogicalNot`, `BitwiseNot`, `Trunc` и `Round`
- **Операторы сравнения**: `Equal`, `NotEqual`, `GreaterThan`, `GraterThanOrEqual`, `LessThan` и `LessThenOrEqual`
- **Двоичные операторы**: `Add`, `Subtract`, `Multiply`, `Divide`, `IntDivide`, `Modulus`, `ShiftLeft`, `ShiftRight`, `LogicalAnd`, `LogicalOr`, `LogicalXor`, `BitwiseAnd`, `BitwiseOr` и `BitwiseXor`.
- **Операторы управляемых записей**: `Initialize`, `Finalize`, `Assign` (см. следующий раздел "Операторы и пользовательские управляемые записи" для получения конкретной информации об этих 3 операторах, добавленных в Delphi 10.4).

В коде, вызывающем оператора, эти названия не используются, а используется соответствующий символ. Эти специальные имена используются только в определении, с префиксом `class operator`, чтобы избежать конфликта имен. Например, можно получить запись с помощью метода `Add` и добавить к ней оператор `Add`.

Когда вы определяете эти операторы, вы пишете параметры, и оператор применяется только в том случае, если параметры точно совпадают. Чтобы добавить два значения разных типов, необходимо указать две различные операции `Add`, так как каждый операнд может быть первой или второй записью выражения. На самом деле, определение операторов не предусматривает автоматической коммутативности. Более того, необходимо очень точно указать тип, так как автоматическое

приведение типов не применяется. Во многих случаях это подразумевает определение множества перегруженных версий оператора, с различными типами параметров.

Еще одним важным фактором, который следует отметить, является то, что существуют два специальных оператора, которые вы можете определить для преобразования данных - Implicit и Explicit. Первый используется для определения неявного приведения типа (или молчаливого преобразования), которое должно быть идеальным и не с потерями. Второй, `Explicit`, может быть вызван только при явном приведении переменной типа к другому данному типу. Вместе эти два оператора определяют преобразования, которые разрешено приводить к и от данного типа данных.

Обратите внимание, что и операторы `Implicit и Explicit` могут быть перегружены на основе типа возврата функции, что обычно невозможно для перегруженных методов. В случае приведения типа, на самом деле, компилятор знает ожидаемый результирующий тип и может вычислить, к какому типу следует применить операцию приведения типа. В качестве примера я написал проект приложения OperatorsOver, который определяет запись с несколькими операторами:

```
type
  TPointRecord = record
  private
    X, Y: Integer;
  public
    procedure SetValue (X1, Y1: Integer);
    class operator Add (A, B: TPointRecord): TPointRecord;
    class operator Explicit (A: TPointRecord): string;
    class operator Implicit (X1: Integer): TPointRecord;
  end;
```

Вот реализация методов записи:

```
class operator TPointRecord.Add(
  A, B: TPointRecord): TPointRecord;
begin
  Result.X := A.X + B.X;
  Result.Y := A.Y + B.Y;
end;
```

```
class operator TPointRecord.Explicit(
  A: TPointRecord): string;
begin
  Result := Format('(%d:%d)', [A.X, A.Y]);
end;

class operator TPointRecord.Implicit(
  X1: Integer): TPointRecord;
begin
  Result.X := X1;
  Result.Y := 10;
end;
```

Использование такой записи очевидно, вы можете написать такой код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  A, B, C: TPointRecord;
begin
  A.SetValue(10, 10);
  B := 30;
  C := A + B;
  Show (string(C));
end;
```

Второе присваивание (`B:=30;`) выполняется с помощью неявных операторов, из-за отсутствия кастинга, в то время как вызов `Show` использует явную нотацию для активации явного приведения типа. Считайте также, что оператор `Add` не изменяет своих параметров, а возвращает совершенно новое значение.

примечание Тот факт, что операторы возвращают новые значения, затрудняет мысль о перегрузке операторов для классов. Если оператор создает новые временные объекты, то кто будет им распоряжаться?

Перегрузка операторов изнутри

Это довольно продвинутая, короткая секция, вы можете пропустить при первом чтении.

Малоизвестно, что технически возможно вызвать оператор, используя его полное внутреннее имя (например, `&&op_Addition`), `прикрепив` его двойным &, вместо использования символа

оператора. Например, сумму записей можно переписать следующим образом (полный листинг см. в примере):

```
C := TPointRecord.&& op_Addition(A, B);
```

хотя я могу придумать очень мало маргинальных случаев, в которых вы могли бы захотеть это сделать. (Вся цель определения операторов заключается в том, чтобы иметь возможность использовать более дружественную нотацию, чем имя метода, а не более уродливую, как предыдущий прямой вызов).

Коммутативность реализации

Предположим, вы хотите реализовать возможность добавления целого числа к одной из ваших записей. Вы можете определить следующий оператор (который доступен в коде проекта приложения OperatorsOver, для немного другого типа записи):

```
class operator TPointRecord2.Add (A: TPointRecord2;
  B: Integer): TPointRecord2;
begin
  Result.X := A.X + B;
  Result.Y := A.Y + B;
end;
```

примечание Причина, по которой я определил этот оператор на новом, а не на существующем типе, заключается в том, что та же самая структура уже определяет `Implicit` приведение целого к типу записи, поэтому я уже могу добавлять целые числа и записи, не определяя конкретный оператор. Этот вопрос более подробно объясняется в следующем разделе.

Теперь вы можете законно добавить к записи значение с плавающей точкой:

```
var
  A: TPointRecord2;
begin
  A.SetValue(10, 20);
  A := A + 10;
```

Однако, если вы попытаетесь поменять слагаемые местами:

```
 A := 30 + A;
```

это не сработает с ошибкой:

```
[dcc32 Error] E2015 Operator not applicable to this operand type
[dcc32 Ошибка] E2015 Оператор не применим к данному типу операнда.
```

На самом деле, как я уже упоминал, коммутативность не является автоматической для операторов, применяемых к переменным разных типов, а должна быть специально реализована либо повторением вызова, либо вызовом (как показано ниже) другой версии оператора:

```
class operator TPointRecord2.Add(B: Integer;
  A: TPointRecord2): TPointRecord2;
begin
  Result := A + B; // implement commutativity
end;
```

Неявное приведение и продвижение типов

Важно отметить, что правила, связанные с разрешением вызовов с участием операторов, отличаются от традиционных правил с участием методов. При автоматическом продвижении типов существует вероятность того, что одно и то же выражение вызовет разные версии перегруженного оператора и приведет к неоднозначности вызова. Именно поэтому при написании неявных операторов необходимо с большой осторожностью подходить к их написанию.

Рассмотрим эти выражения присвоения C из предыдущего примера:

```
A := 50;
C := A + 30;
C := 50 + 30;
C := 50 + TPointRecord(30);
```

Они все легальны! В первом случае компилятор преобразует 30 в соответствующий тип записи, во втором преобразование происходит после присваивания, а в третьем явное приведение приводит к неявному приведению первого значения, так что выполняемое сложение является пользовательским среди записей. Другими словами, результат второй операции

отличается от двух других, что выделено в выводе и в расширенном варианте этих выражений:

```
// вывод
(80:20)
(80:10)
(80:20)

// expanded statements
C := A + TPointRecord(30);
// that is: (50:10) + (30:10)

C := TPointRecord (50 + 30);
// that is 80 converted into (80:10)

C := TPointRecord(50) + TpointRecord(30);
// that is: (50:10) + (30:10)
```

Операторы и управляемые записи

Существует специальный набор операторов, которые вы можете использовать для записей на языке Delphi, чтобы определить пользовательскую управляемую запись. Перед тем, как мы туда доберемся, позвольте мне описать правила инициализации памяти записей, а также различия между обычными и управляемыми записями.

Записи в Delphi могут иметь поля любого типа данных. Когда запись имеет простые (неуправляемые) поля, например, числовые или другие перечисленные значения, компилятору делать нечего. Создание и уничтожение записи состоит в выделении памяти или освобождении ячейки памяти. (Обратите внимание, что по умолчанию Delphi не инициализирует записи нулевым значением).

Если запись имеет поле типа, управляемое компилятором (например, строка или интерфейс), компилятору необходимо ввести дополнительный код для управления инициализацией или доработкой. Строка, например, считается ссылкой, поэтому, когда запись выходит за пределы области видимости,

необходимо уменьшить количество ссылок внутри записи, что может привести к перераспределению памяти под строку. Поэтому при использовании такой управляемой записи в одном из участков кода компилятор автоматически добавляет блок try-finally вокруг этого кода и следит за тем, чтобы данные были очищены даже в случае исключения. Так происходит уже давно. Другими словами, управляемые записи являются частью языка Delphi.

Теперь, начиная с 10.4, тип записи Delphi поддерживает пользовательскую инициализацию и завершение, помимо операций по умолчанию, которые компилятор выполняет для управляемых записей. Вы можете объявить запись с пользовательским кодом инициализации и завершения независимо от типа данных ее полей, и вы можете написать такой пользовательский код инициализации и завершения. Эти записи обозначаются как "*пользовательские управляемые записи*".

Разработчик может превратить запись в пользовательскую управляемую запись, добавив к типу записи один или несколько новых операторов:

- Оператор `Initialize` вызывается после выделения памяти для записи, что позволяет написать код для установки начального значения для полей
- Operator `Finalize` вызывается до того, как память для записи будет удалена, и позволяет выполнить любую необходимую очистку.
- Оператор `Assign` вызывается, когда данные записи копируются в другую запись того же типа, так что вы можете скопировать информацию из одной записи в другую определенным, пользовательским способом.

примечание Учитывая, что finalization управляемых записей выполняется даже в случае исключения (компилятор автоматически генерирует блок try-finally), они часто используются как альтернативный способ защиты выделения ресурсов или реализации операций по очистке. Пример такого использования мы рассмотрим в разделе "Восстановление курсора с управляемой записью" Главы 9.

Записи с операторами Initialize и Finalize

Давайте начнем с инициализации и завершения. Ниже приведен простой фрагмент кода:

```
type
  TMyRecord = record
    Value: Integer;
    class operator Initialize (out Dest: TMyRecord);
    class operator Finalize(var Dest: TMyRecord);
  end;
```

Конечно, необходимо написать код для двух методов класса, например, записать их выполнение в журнал или инициализировать значение записи. В этом примере (часть демонстрационного проекта ManagedRecords_101) я также протоколирую ссылку на область памяти, чтобы увидеть, какая запись выполняет каждую отдельную операцию:

```
class operator TMyRecord.Initialize (out Dest: TMyRecord);
begin
  Dest.Value := 10;
  Log('created' + IntToHex (Integer(Pointer(@Dest))));
end;

class operator TMyRecord.Finalize(var Dest: TMyRecord);
begin
  Log('destroyed' + IntToHex (Integer(Pointer(@Dest))));
end;
```

Разница между этим механизмом построения и тем, что ранее было доступно для записей, заключается в автоматическом вызове. Если вы пишете нечто подобное коду, приведенному ниже, то вы можете вызвать как инициализацию, так и код завершения и в конечном итоге получить try-finally блок, сгенерированный компилятором для вашего управляемого экземпляра записи:

```
procedure LocalVarTest;
var
  My1: TMyRecord;
begin
  Log (My1.Value.ToString);
end;
```

С таким кодом вы получите такой журнал:

```
created 0019F2A8
10
destroyed 0019F2A8
```

Другой сценарий - использование встроенных переменных, как в случае с in:

```
begin
  var T: TMyRecord;
  Log(T.Value.ToString);
```

что дает вам ту же последовательность в журнале.

Оператор Assign

В общем, присвоение (:=) просто копирует все данные полей записи. Для управляемых записей (со строками и другими управляемыми типами) они обрабатываются компилятором корректно.

Когда у вас есть пользовательские поля данных и пользовательская инициализация, вы, возможно, захотите изменить поведение по умолчанию. Вот почему для пользовательских управляемых записей вы также можете определить оператор присваивания. Новый оператор вызывается с синтаксисом :=, но определяется как `Assign`:

```
  class operator Assign (var Dest: TMyRecord;
      const [ref] Src: TMyRecord);
```

Определение оператора должно соответствовать очень точным правилам, включая то, что первый параметр как параметр передается по ссылке (`var`), а второй как параметр `const` передается по ссылке. Если этого не сделать, компилятор выдает сообщения об ошибках следующего вида:

```
[dcc32 Error] E2617 First parameter of Assign operator must be a var
parameter of the container type
[dcc32 Hint] H2618 Second parameter of Assign operator must be a
const[Ref] or var parameter of the container type

[dcc32 Ошибка] E2617 Первый параметр оператора Assign должен быть
параметром var типа контейнера.
[подсказка dcc32] H2618 Второй параметр оператора Assign должен быть
параметром const[Ref] или var типа контейнера.
```

Пример вызыва оператора `Assign`:

```
var
  My1, My2: TMyRecord;
begin
  My1.Value := 22;
  My2 := My1;
```

который производит такой журнал (в который я также добавляю порядковый номер к записи):

```
created 5 0019F2A0
created 6 0019F298
5 copied to 6
destroyed 6 0019F298
destroyed 50019F2A0
```

Обратите внимание, что последовательность освобождения отличается от последовательности создания, причем последняя созданная запись является первой освобожденной.

Передача управляемых записей как параметр

Управляемые записи могут работать иначе, чем обычные записи, также при передаче в качестве параметров или возврате функцией. Вот несколько процедур, показывающих различные сценарии:

```
procedure ParByValue (Rec: TMyRecord);
procedure ParByConstValue (const Rec: TMyRecord);
procedure ParByRef (var Rec: TMyRecord);
procedure ParByConstRef (const [ref] Rec: TMyRecord);
function ParReturned: TMyRecord;
```

Теперь, не просматривая каждый лог по одному (их можно увидеть, запустив демонстрационную версию ManagedRecords_101), вот краткое изложение информации:

- `ParByValue` создает новую запись и вызывает оператор присваивания (если доступно) для копирования данных, уничтожая временную копию при выходе из процедуры.
- `ParByConstValue` не делает никаких копий и вообще вызовов.
- `ParByRef` не делает ни копий, ни вызовов.
- `ParByConstRef` не делает ни копий, ни вызовов.
- `ParReturned` создает новую запись (через `Initialize`) и при возврате вызывает оператор `Assign`, если вызов похож на `my1 := ParReturned`, и удаляет временную запись после назначения.

Исключения и управляемые записи

Когда возникает исключение, записи в целом очищаются даже при отсутствии явного блока try-finally, в отличие от объектов. Это фундаментальное различие и ключ к реальной полезности управляемых записей.

```
procedure ExceptionTest;
begin
  var A: TMRE;
  var B: TMRE;
  raise Exception.Create('Error Message');
end;
```

В рамках этой процедуры происходит два вызова конструкторов и два вызова деструкторов. Опять же, это принципиальное отличие и ключевая особенность управляемых записей.

Массивы управляемых записей

При определении статического массива управляемых записей они инициализируются вызовом оператора `Initialize` при объявлении:

```
var
  A1: array [1..5] of TMyRecord; // call here
begin
  Log ('ArrOfRec');
```

Они все уничтожаются, когда выходят из поля зрения. При определении динамического массива управляемых записей вызывается код инициализации когда устанавливается размер массива (с `SetLength`):

```
var
  A2: array of TMyRecord;
begin
  Log ('ArrOfDyn');
  SetLength(A2, 5); // call here
```

Тип данных Variant

Первоначально представленный на языке для обеспечения полной поддержки Windows OLE и COM, Object Pascal имеет концепцию *нетипизированного* нативного типа данных, называемого `Variant`. Несмотря на то, что название напоминает разновидность записи (упоминалось ранее) и реализация имеет некоторое сходство с параметрами открытого массива, это отдельная особенность с очень специфической реализацией (редко встречающейся в языках, не входящих в мир разработки Windows).

В этом разделе я не буду ссылаться на OLE и другие сценарии, в которых используется этот тип данных (например, доступ к

полям для датасетов), я просто хочу обсудить этот тип данных с общей точки зрения.

Я вернусь к динамическим типам, RTTI, и к отражению в Главе 16, где я также расскажу о связанном (но безопасном и гораздо более быстром) типе данных RTL, называемом `TValue`.

Варианты не имеют типа

В общем случае можно использовать переменную типа варианта для хранения любого из базовых типов данных, а также для выполнения многочисленных операций и приведения типов. Автоматическое приведение типов идет вразрез с общим типобезопасным подходом языка Object Pascal и является реализацией типа динамического набора типов, первоначально введенного такими языками, как Smalltalk и Objective-C, и недавно ставшего популярным в скриптовых языках, включая JavaScript, PHP, Python и Ruby.

Вариант проверяется по типу и вычисляется во время выполнения. Компилятор не предупредит о возможных ошибках в коде, которые могут быть пойманы только при тщательной проверке. В целом, порции кода, использующие варианты, можно считать интерпретируемым кодом, так как, как и в случае с известными интерпретаторами, многие операции не могут быть разрешены до момента выполнения. В частности, это влияет на скорость работы кода.

Теперь, когда я предостерег вас от использования типа `Variant`, пришло время посмотреть, что вы можете с ним сделать. В основном, как только вы объявили переменную типа Вариант, например, следующую:

```
var
  V: Variant;
```

ему можно присваивать значения нескольких различных типов:

```
V := 10;
V := "Привет, мир";
V := 45.55;
```

После того, как у вас есть значение варианта, вы можете скопировать его на любой совместимый или несовместимый тип данных. Если вы присваиваете значение несовместимому типу данных, компилятор, как правило, не будет отмечать его ошибкой, а выполнит преобразование во время выполнения, если в этом есть смысл. В противном случае он выдаст ошибку во время выполнения. Технически вариант хранит информацию о типе вместе с реальными данными, позволяя выполнять ряд удобных, но медленных и опасных операций во время выполнения.

Рассмотрим следующий код (часть проекта приложения VariantTest), который является расширением приведенного выше кода:

```
var
  V: Variant;
  S: string;
begin
  V := 10;
  S := V;
  V := V + S;
  Show (V);

  V := 'Hello, World';
  V := V + S;
  Show (V);

  V := 45.55;
  V := V + S;
  Show (V);
```

Забавно, правда? Вот результаты (не удивительно):

```
20
Hello, World10
55.55
```

Кроме присвоения переменной `s` варианта, содержащего строку, можно присвоить ей вариант, содержащий целое число или число с плавающей точкой. Еще *хуже* то, что с помощью вариантов можно вычислять значения с помощью операции `V := V + S,` которая интерпретируется по-разному в зависимости от данных, хранящихся в варианте. В приведенном выше коде в эту же строку можно добавлять целые числа, значения с плавающей точкой или конкатенировать строки.

Написание выражений, включающих варианты, по меньшей мере, рискованно. Если строка содержит число, то все работает. Если нет, то поднимается исключение. Без веских оснований не стоит использовать тип `Variant`; придерживайтесь стандартных типов данных Object Pascal и подхода к проверке типов.

Варианты в подробностях

Для тех, кто заинтересован в лучшем понимании вариантов, позвольте мне добавить некоторую техническую информацию о том, как работают варианты и как вы можете получить больше контроля над ними. В RTL включен тип записи варианта, TVarData, который имеет ту же самую раскладку памяти, что и тип `Variant`. Вы можете использовать его для доступа к действительному типу варианта. Структура `TVarData` включает тип варианта, обозначенный как VType, некоторые зарезервированные поля и фактическое значение.

примечание Для получения более подробной информации обратитесь к определению TVarData в исходном коде RTL, в Системном блоке. Это далеко не простая структура, и я рекомендую только разработчикам, имеющим некоторый опыт работы, рассматривать детали реализации типа варианта.

Возможные значения поля `VType` соответствуют типам данных, которые можно использовать в OLE автоматизации и которые часто называют типами OLE или *вариантами*. Полный

алфавитный список доступных типов вариантов приведен ниже:

```
varAny              varArray            varBoolean
varByte             varByRef            varCurrency
varDate             varDispatch         varDouble
varEmpty            varError            varInt64
varInteger          varLongWord         varNull
varOleStr           varRecord           varShortInt
varSingle           varSmallint         varString
varTypeMask         varUInt64           varUnknown
varUString          varVariant          varWord
```

Большинство из этих констант имен типов вариантов легко понять. Обратите внимание, что существует понятие *нулевого значения*, которое вы получаете, присваивая NULL (а не nil).

Существует также множество функций для работы с вариантами, которые можно использовать для выполнения специфических приведений типов или для запроса информации о типе варианта (см., например, функцию `VarType`). Большинство из этих функций приведения и присваивания типов на самом деле вызываются автоматически при написании выражений с использованием вариантов. Другие вспомогательные процедуры поддержки вариантов фактически работают на массивах вариантов, опять же структура, используемая почти исключительно для OLE-интеграции в Windows.

Варианты - медленные

Код, использующий тип `Variant`, работает медленно не только при преобразовании типов данных, но даже при простом добавлении двух значений варианта, содержащих целые числа. Они почти так же медленны, как и интерпретируемый код. Чтобы сравнить скорость работы алгоритма, основанного на вариантах, со скоростью работы того же самого кода,

основанного на целых числах, можно посмотреть вторую кнопку проекта VariantTest.

Эта программа выполняет цикл, синхронизируя его скорость и отображая состояние в индикаторе прогресса. Вот первый из двух очень похожих циклов, основанных на Int64 и вариантах:

```
const
  maxno = 10000000;  // 10 million

var
  time1, time2: TDateTime;
  n1, n2: Variant;
begin
  time1 := Now;
  n1 := 0;
  n2 := 0;

  while n1 < maxno do
  begin
    n2 := n2 + n1;
    Inc (n1);
  end;

  // we must use the result
  time2 := Now;
  Show (n2);
  Show ('Variants: ' + FormatDateTime (
    'ss.zzz', Time2-Time1) + ' seconds');
```

На код измерения времени стоит посмотреть, потому что его можно легко адаптировать для любого теста производительности. Как видите, программа использует функцию `Now` для получения текущего времени и функцию FormatDateTime для вывода разницы во времени, показывая только секунды (*"ss"*) и миллисекунды (*"zzz"*).

В этом примере разница в скорости на самом деле настолько велика, что вы заметите ее даже без точного времени:

```
49999995000000
Variants: 01.169 seconds
49999995000000
Integers: 00.026 second
```

Это числа, которые я получаю на своей виртуальной машине Windows, и это дает замедление примерно в 50 раз для кода,

основанного на варианте. Реальные значения зависят от компьютера, на котором вы запускаете эту программу, но относительная разница не сильно изменится. Даже на моем Android телефоне я получаю похожую пропорцию (но в целом намного больше):

```
49999995000000
Варианты: 07.717 секунд
49999995000000
Целые: 00.157 секунды
```

На моем телефоне этот код выполняется в 6 раз больше времени, чем на Windows, но факт остается в том, что среднее время выполнения отличается более чем на 7 секунд, что делает вариантную реализацию заметно более медленной для пользователя, в то время как реализация на базе Int64 все еще очень быстрая (пользователь вряд ли заметил бы десятую долю секунды).

А что насчет указателей?

Другой фундаментальный тип данных языка Object Pascal представлен указателями. Некоторые объектно-ориентированные языки проделали большой путь, чтобы скрыть эту мощную, но опасную конструкцию языка, в то время как Object Pascal позволяет программисту использовать его при необходимости (что, как правило, не очень часто).

Но что такое указатель, и откуда его название? В отличие от большинства других типов данных, указатель не содержит действительного значения, но содержит косвенную ссылку на переменную, которая, в свою очередь, имеет значение. Более технический способ выразить это заключается в том, что тип указателя определяет переменную, которая содержит адрес

памяти другой переменной данного типа данных (или неопределенного типа).

примечание Это продвинутый раздел книги, добавленный здесь, потому что указатели являются частью языка Object Pascal и должны быть частью основных знаний любого разработчика, хотя это и не основная тема, и если вы новичок в языке, вы можете пропустить этот раздел при первом прочтении книги. Опять же, есть шанс, что вы использовали языки программирования без (явных) указателей, так что этот краткий раздел может быть интересным для чтения!

Определение типа указателя не основано на конкретном ключевом слове, а использует специальный символ - каретку (^, знак вставки) . Например, можно определить тип, представляющий собой указатель на переменную типа Integer со следующим объявлением:

```
type
  TPointerToInt = ^Integer;
```

Определив переменную-указатель, можно присвоить ей адрес другой переменной того же типа, используя оператор @:

```
var
  P: ^Integer;
  X: Integer;
begin
  X := 10;
  P := @X;
  // change the value of X using the pointer
  P^ := 20;
  Show ('X: ' + X.ToString);
  Show ('P^: ' + P^.ToString);
  Show ('P: ' + UIntPtr(P).ToHexString (8));
```

Данный код является частью прикладного проекта PointersTest. Учитывая, что указатель `P` ссылается на переменную `X`, можно использовать `P^`, чтобы ссылаться на значение переменной, а также читать или изменять его. Также можно отобразить значение самого указателя, то есть адрес памяти `X`, приведя указатель к числу с помощью специального типа `UIntPtr` (подробнее см. примечание ниже). Вместо того, чтобы показывать простое числовое значение, этот код показывает шестнадцатеричное представление, которое чаще встречается

для адресов памяти. Вот результат (где адрес указателя может зависеть от конкретной компиляции):

```
X: 20
P^: 20
P: 0018FC18
```

предупреждение Вывод указателя на Целое число является корректным кодом только на 32-битных платформах, когда он ограничен 2 Гб. Если вы включите использование большего объема памяти, то вам придется использовать тип Cardinal. Для 64-битных платформ лучшим вариантом будет использование . Однако существует псевдоним этого типа, специально предназначенный для указателей и называемый UIntPtr, что является лучшим вариантом для данного сценария, так как при его использовании вы четко указываете компилятору Delphi свои намерения.

Позвольте мне подытожить, для ясности. Когда у тебя есть указатель `P`:

Используя указатель напрямую (с выражением `P`) вы ссылаетесь на адрес ячейки памяти, на которую ссылается указатель

При разыменовании указателя (с выражением `P^`) вы ссылаетесь на фактическое содержимое этой ячейки памяти

Вместо того, чтобы ссылаться на существующую память, указатель может также ссылаться на новый и специфический блок памяти, динамически выделяемый на куче с помощью процедуры `New`. В этом случае, когда вам больше не нужно значение, к которому обращается указатель, вам также придется избавиться от динамически выделяемой памяти, вызвав процедуру `Dispose`.

примечание Управление памятью в целом и то, как работает куча, в частности, рассматриваются в главе 13. Короче говоря, куча — это (большая) область памяти, в которой можно выделять и освобождать блоки памяти не в заданном порядке. В качестве альтернативы New and Dispose можно использовать GetMem и FreeMem, которые требуют от разработчика указать размер выделения (в то время как компилятор определяет размер выделения автоматически в случае New and Dispose). В тех случаях, когда размер выделения не известен во время компиляции, `GetMem` и `FreeMem` становятся удобными.

Вот фрагмент кода, который динамически выделяет память:

```
var
  P: ^Integer;
begin
  // initialization
  New (P);
  // operations
  P^ := 20;
  Show (P^.ToString);
  // termination
  Dispose (P);
```

Если вы не распорядитесь памятью после ее использования, ваша программа может в конечном итоге израсходовать всю доступную память и аварийно завершить работу. Ошибка с освобождением памяти, которая больше не нужна, известна как *утечка памяти*.

предупреждение Чтобы быть более безопасным, приведенный выше код действительно должен использовать обработку исключений с помощью блока `try-finally`, тему, которую я решил не вводить на данном этапе книги, но о ней я расскажу позже в Главе 9.

Если указатель не имеет значения, можно присвоить ему значение `nil`. Можно проверить, равен ли указатель `nil`, чтобы узнать, ссылается ли он в данный момент на значение с помощью прямого сравнения на равенство или с помощью специальной функции Assigned, как показано ниже.

Такой тест часто используется, так как разыменование (то есть обращение к значению по адресу памяти, хранящемуся в указателе) недействительного указателя приводит к нарушению доступа к памяти (с несколько различными эффектами в зависимости от операционной системы):

```
var
  P: ^Integer;
begin
  P := nil;
  Show (P^.ToString);
```

Пример эффекта этого кода можно посмотреть, запустив проект приложения PointersTest. Ошибка, которую вы увидите (на Windows) должна быть аналогичной:

```
Access violation at address 0080B14E in module 'PointersTest.exe'. Read
of address 00000000.

Нарушение доступа по адресу 0080B14E в модуле 'PointersTest.exe'.
Считывание адреса 00000000.
```

Одним из способов сделать доступ к данным по указателям безопаснее, является добавление проверки безопасности "указатель не является нулевым", как показано ниже:

```
if P <> nil then
  Show (P^.ToString);
```

Как я упоминал ранее, альтернативный способ, который, как правило, предпочтительнее из соображений удобочитаемости, это использование псевдо-функции `Assigned (Назначенная)`:

```
if Assigned (P) then
  writeln (P^.ToString);
```

примечание `Assigned` не является реальной функцией, так как она "разрешается" компилятором, генерирующим соответствующий код. Кроме того, ее можно использовать поверх процедурной переменной типа (или ссылки на метод), не вызывая ее на самом деле, а только проверяя, назначена ли она.

Объект Pascal также определяет тип данных `Pointer`, который указывает на указатели без типа (например, `void*` в языке C). Если вы используете указатель без типа, то вместо `New` следует использовать `GetMem` (указание на количество выделяемых байт, учитывая, что это значение не может быть выведено из самого типа). Процедура `GetMem` требуется каждый раз, когда размер выделяемой переменной памяти не определен.

То, что указатели редко нужны в Object Pascal, является интересным преимуществом этого языка. Тем не менее, наличие этой возможности может помочь в реализации некоторых особо эффективных низкоуровневых функций и при вызове API операционной системы.

В любом случае, понимание указателей важно для продвинутого программирования и для полного понимания

объектной модели языка, который использует указатели (обычно называемые ссылками) в реализации.

предупреждение Когда переменная содержит указатель на вторую переменную и эта вторая переменная выходит за пределы области видимости или освобождается (если динамически выделяется), указатель будет просто ссылаться на область памяти либо неопределенную, либо содержащую некоторые другие данные. Это может привести к очень трудно обнаруживаемым ошибкам.

Типы файлов, кто-нибудь?

Последний конструктор типа данных Object Pascal, рассмотренный (кратко) в этой главе, является типом *файла*. Типы файлов представляют собой физические дисковые файлы, что, безусловно, является особенностью оригинального языка Pascal, учитывая, что очень немногие старые или современные языки программирования включают понятие файла как примитивного типа данных. Язык Object Pascal имеет ключевое слово `file`, которое является спецификатором типа, как `array` или `record`. Вы используете `file` для определения нового типа, а затем можете использовать новый тип данных для объявления новых переменных:

```
type
  IntFile = file of Integers;
var
  IntFile1: IntFile;
```

Также можно использовать ключевое слово `file` без указания типа данных, для указания нетипизированного файла. В качестве альтернативы можно использовать тип TextFile, заданный в модуле System библиотеки Run Time Library для объявления файлов с символами ASCII (или, вернее в нынешнее время, файлов с байтами).

Прямое использование файлов, хотя все еще поддерживается, все менее распространено в наши дни, так как библиотека Run Time Library включает в себя множество классов для управления бинарными и текстовыми файлами на гораздо более высоком уровне (включая поддержку Unicode кодировок для текстовых файлов, например).

Приложения Delphi обычно используют потоки RTL (`TStream` и производные классы) для обработки любых сложных операций чтения и записи файлов. Потоки представляют собой виртуальные файлы, которые могут быть привязаны к физическим файлам, блоку памяти, сокету или любой другой непрерывной серии байт.

Одна из областей, когда вам все еще нужны некоторые из старых процедур управления файлами, это когда вы пишете консольные приложения, где вы можете использовать функции `write`, `writeln`, `read`, и связанные с ними функции для работы со специальным файлом, который является стандартным входом и стандартным выходом (C и C++ имеют похожую поддержку входа и выхода из консоли, и многие другие языки предлагают аналогичные услуги).

06: Все о строках

Строки символов - один из наиболее часто используемых типов данных в любом языке программирования. Object Pascal делает обработку строк довольно простой, но в то же время очень быстрой и чрезвычайно мощной. Даже если основы строк легко понять, и я использовал строки для вывода в предыдущих главах, внутри ситуация немного сложнее, чем может показаться на первый взгляд. Манипулирование текстом включает в себя несколько тесно связанных между собой тем, которые стоит изучить: чтобы полностью понять обработку строк, необходимо знать о представлениях в Unicode, понять, как строки отображаются в массивы символов, и узнать о некоторых наиболее актуальных операциях со строками в библиотеке времени выполнения, в том числе о сохранении строк в текстовые файлы и их загрузке.

Object Pascal имеет несколько опций для манипулирования строками и делает доступными различные типы данных и подходы. Основное внимание в главе будет уделено стандартному строковому типу данных, но я также уделю немного времени более старым строковым типам, таким как AnsiString, которые до сих пор можно использовать в компиляторах Delphi. Но прежде, чем мы к этому приступим, позвольте мне начать с самого начала: с представления Юникода.

Юникод: Алфавит для всего мира

Управление строками Object Pascal сосредоточено вокруг набора символов Юникода и, в частности, использование одного из его представлений, называемого UTF-16. Прежде чем перейти к техническим подробностям реализации, стоит посвятить несколько разделов, чтобы полностью понять стандарт Юникода.

Идея Юникода (что делает его одновременно простым и сложным) заключается в том, что каждый символ во всех известных алфавитах мира имеет свое собственное описание, графическое представление и уникальное числовое значение (называемое *кодовой точкой* Юникода - Unicode *code point*).

примечание Справочный веб-сайт консорциума "Юникод" - http://www.unicode.org, на котором представлено большое количество документов. Последняя ссылка - книга "Стандарт Юникода", которую можно найти по адресу http://www.unicode.org/book/aboutbook.html.

Не все разработчики знакомы с Юникодом, и многие до сих пор думают о символах с точки зрения старых, ограниченных представлений, таких как ASCII, и с точки зрения кодирования ISO. Дальше - небольшой раздел, описывающий эти старые стандарты, чтобы вы лучше оценили особенности (и сложность) Юникода.

Символы из прошлого: от ASCII до ISO кодировки

Представление символов началось с Американского стандартного кода для информационного обмена (ASCII), который был разработан в начале 60-х

годов как стандартное кодирование компьютерных символов, охватывающее 26 букв английского алфавита, как строчных, так и заглавных, 10 цифровых цифр, общих знаков препинания, а также ряд управляющих символов (используемых и по сей день).

ASCII использует 7-битную систему кодирования для представления 128 различных символов. Только символы между #32 (Space) и #126 (Tilde) имеют визуальное представление, как показано на рисунке 6.1 (извлечено из приложения Object Pascal, работающего под Windows).

Рисунок 6.1: Таблица с набором символов ASCII для печати

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th></tr>
<tr><td>0</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>16</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>32</td><td></td><td>!</td><td>"</td><td>#</td><td>$</td><td>%</td><td>&</td><td>'</td><td>(</td><td>)</td><td>*</td><td>+</td><td>,</td><td>-</td><td>.</td><td>/</td></tr>
<tr><td>48</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>:</td><td>;</td><td><</td><td>=</td><td>></td><td>?</td></tr>
<tr><td>64</td><td>@</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td><td>H</td><td>I</td><td>J</td><td>K</td><td>L</td><td>M</td><td>N</td><td>O</td></tr>
<tr><td>80</td><td>P</td><td>Q</td><td>R</td><td>S</td><td>T</td><td>U</td><td>V</td><td>W</td><td>X</td><td>Y</td><td>Z</td><td>[</td><td>₩</td><td>]</td><td>^</td><td>_</td></tr>
<tr><td>96</td><td>`</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td><td>h</td><td>i</td><td>j</td><td>k</td><td>l</td><td>m</td><td>n</td><td>o</td></tr>
<tr><td>112</td><td>p</td><td>q</td><td>r</td><td>s</td><td>t</td><td>u</td><td>v</td><td>w</td><td>x</td><td>y</td><td>z</td><td>{</td><td>|</td><td>}</td><td>~</td><td>▯</td></tr>
</table>

Хотя ASCII, безусловно, был основой (с его базовым набором из 128 символов, которые до сих пор являются частью ядра Юникода), вскоре он был заменен расширенными версиями, которые использовали 8-й бит для добавления еще 128 символов в набор.

Теперь проблема в том, что при таком большом количестве языков по всему миру не было простого способа выяснить, какие еще символы включить в набор (иногда обозначается как ASCII-8). Короче говоря, Windows приняла другой набор символов, называемый *кодовой страницей*, с набором символов в зависимости от вашей локальной конфигурации и версии Windows. Помимо кодовых страниц Windows существует множество других стандартов, основанных на аналогичном «страничном» подходе, и эти страницы стали частью международных стандартов ISO.

Самым актуальным, безусловно, был стандарт ISO 8859, который определяет несколько *региональных* наборов. Наиболее часто

используемым набором (ну, тот, который используется в большинстве западных стран, если быть немного точнее) является латинский набор, называемый ISO 8859-1.

примечание Даже если кодовая страница Windows 1252 частично похожа, она не полностью соответствует набору ISO 8859-1. Windows добавляет дополнительные символы, такие как символ €, дополнительные кавычки и т.д. в области от 128 до 150. В отличие от всех других значений латинского набора, эти расширения Windows не соответствуют кодовым точкам Юникода.

Unicode Code Points и Graphemes

Если я действительно хочу быть точным, я должен включить еще одно понятие, выходящее за рамки пунктов кода. Иногда, на самом деле, несколько точек кода могут быть использованы для представления одной *графемы* (визуального символа). Обычно это не буква, а комбинация букв или символов. Например, если у вас есть последовательность точек кода, представляющих латинскую букву *a* (`#$0061`), за которой следует точка кода, представляющая собой большой акцент (`#$0300`), то она должна отображаться как один символ с акцентом.

В терминах кодирования Object Pascal, если написать следующее (часть прикладного проекта CodePoints), то сообщение будет иметь один единственный символ с акцентом, как показано на рисунке 6.2.

```
var
  Str: String;
begin
  Str := #$0061 + #$0300;
  ShowMessage (Str);
```

Рисунок 6.2: Одна графема может быть результатом множества кодовых точек

В данном случае мы имеем два символа, представляющие две точки кода, но только одну графему (или визуальный элемент). Дело в том, что если в латинском алфавите для представления данной графемы можно использовать определенную кодовую точку Юникода (*буква a с сильным акцентом — это кодовая точка $00E0*), то в других алфавитах комбинирование кодовых точек Юникода - единственный способ получить заданную графему (и правильный вывод).

Даже если на экране отображается символ с акцентом, автоматической нормализации или трансформации значения (только правильное отображение) не происходит, поэтому строка внутри остается отличной от строки с одним символом à.

примечание Отрисовка графем из нескольких точек кода может зависеть от конкретной поддержки со стороны операционной системы и используемой техники отрисовки текста, так что вы можете обнаружить, что для некоторых графем не все операционные системы предлагают корректный вывод.

От кодовых точек к байтам (UTF)

В то время как ASCII использовал прямое и простое отображение символов в их числовом представлении, Юникод использует более сложный подход. Как я уже упоминал, каждый элемент алфавита Юникод имеет ассоциированную

точку кода, но сопоставление с реальным представлением часто бывает более сложным.

Один из элементов путаницы в Юникоде заключается в том, что существует множество способов представления одной и той же точки кода (или числового значения символа Юникода) с точки зрения фактического хранения, физических байтов, в памяти или в файле.

Проблема заключается в том, что единственный способ представить все точки кода Юникода простым и единообразным способом — это использовать четыре байта для каждой точки кода. Это приводит к представлению фиксированной длины (для каждого символа всегда требуется одно и то же количество байт), но большинство разработчиков воспримут это как слишком дорогостоящий способ представления в памяти и с точки зрения обработки.

примечание В Object Pascal кодовые точки Юникода могут быть представлены непосредственно в 4-байтовом представлении с помощью типа данных UCS4Char.

Поэтому стандарт Юникода определяет другие представления, обычно требующие меньше памяти, но в которых количество байт для каждого символа различно, в зависимости от его точки кода. Идея состоит в том, чтобы для наиболее распространенных элементов использовать представление меньшего размера, а для редко встречающихся - большего.

Различные физические представления точек кода Юникода называются Unicode Transformation Formats (или UTF). Это алгоритмические *отображения*, являющиеся частью стандарта Юникода, которые сопоставляют каждую точку кода (абсолютное числовое представление символа) с уникальной последовательностью байтов, представляющих данный символ. Обратите внимание, что отображения могут использоваться в

> обоих направлениях, преобразуя туда и обратно между различными представлениями.
>
> Стандарт определяет три из этих форматов в зависимости от того, сколько бит используется для представления начальной части набора (начальные 128 символов): 8, 16 или 32. Интересно отметить, что все три вида кодировок требуют не более 4 байт данных для каждой точки кода.

UTF-8 преобразует символы в кодировку переменной длины от 1 до 4 байт. UTF-8 популярна для HTML и подобных протоколов, потому что она достаточно компактна, когда большинство символов (например, теги в HTML) попадают в подмножество ASCII.

UTF-16 популярен во многих операционных системах (включая Windows и MacOS) и средах разработки. Она достаточно удобна, так как большинство символов помещается в два байта, достаточно компактна и быстро обрабатывается.

UTF-32 имеет большой смысл для обработки (все точки кода имеют одинаковую длину), но он потребляет память и имеет ограниченное использование на практике.

> Существует распространенное заблуждение, что UTF-16 может отображать непосредственно все точки кода двумя байтами, но так как Юникод определяет более 100,000 точек кода, вы можете легко понять, что они не поместятся в 64K элементов. Правда, иногда разработчики используют только подмножество Юникода, чтобы сделать его вписывающимся в представление с фиксированной длиной в 2 байта на символ. Раньше это подмножество Юникода называлось UCS-2, теперь его часто называют Basic Multilingual Plane (BMP). Однако, это только подмножество Юникода (одна из многих *плоскостей*).

примечание Проблема, связанная с многобайтовыми представлениями (UTF-16 и UTF-32), заключается в том, какой из байтов приходит первым? Согласно стандарту, разрешены все формы, поэтому вы можете иметь UTF-16 BE (big-endian) или LE (little-endian), и то же самое для UTF-32. Сериализация байтов в big-endian ставит

первым более важный байт, в little-endian байт сериализации ставит первым наименее важный. Сериализация байтов часто отмечается в файлах вместе с представлением UTF заголовком, называемым Byte Order Mark (BOM).

Отметка Byte Order Mark

При наличии текстового файла, в котором хранятся символы Юникода, есть способ указать, какой формат UTF используется для точек кода. Информация хранится в заголовке или маркере в начале файла, который называется Byte Order Mark (BOM). Это сигнатура, указывающая на используемый формат Unicode и форму порядка следования байтов (little или big endian - LE или BE). В следующей таблице приведена краткая информация о различных записях BOM, длина которых может составлять 2, 3 или 4 байта:

`00 00 FE FF`	UTF-32, BE
`ФФ ФЕ 00 00`	UTF-32, little-endian
`ИП ПФ`	UTF-16, BE
`ФФ ФЭ`	UTF-16, LE
`EF BB BF`	UTF-8

Позже в этой главе мы увидим, как Object Pascal управляет BOM внутри своих потоковых классов. BOM появляется в самом начале файла, причем актуальные данные Unicode идут сразу после него. Таким образом, UTF-8 файл с содержимым *AB* содержит пять шестнадцатеричных значений (3 для BOM, 2 для букв):

```
EF BB BF 41 42
```

Если текстовый файл не имеет ни одной из этих сигнатур, он обычно считается текстовым файлом в формате ASCII, но с тем же успехом может содержать текст в любой кодировке.

примечание С другой стороны, когда вы получаете данные из веб-запроса или через другие интернет-протоколы, у вас может быть определенный заголовок (часть протокола), указывающий на кодировку, и можете не полагаться на BOM.

Рассматривая Юникод

Как создать таблицу символов Юникода, подобных тем, что я отображал ранее, для ASCII символов? Мы можем начать с отображения кодовых точек в базовой многоязычной плоскости (BMP), исключая то, что называется суррогатными парами.

примечание Не все числовые значения являются истинными кодовыми точками UTF-16, так как существуют некоторые недействительные числовые значения для символов (называемых суррогатами), которые используются для формирования парного кода и представляют кодовые точки выше 65535. Хорошим примером суррогатной пары является символ, используемый в музыкальных партитурах для F (или баса) clef□. Это кодовая точка 1D122, которая представлена в UTF-16 двумя значениями, D834, за которой следует DD22.

Для отображения всех элементов BMP потребуется сетка 256 * 256, которую трудно разместить на экране. Поэтому в проекте приложения ShowUnicode есть закладка с двумя страницами: на первой закладке находится основной селектор из 256 блоков, а на второй странице отображается сетка с реальными элементами Unicode, по одному разделу за раз. Эта программа имеет немного больше пользовательского интерфейса, чем большинство других в книге, и вы можете просто пропустить ее код, если вас интересует только ее вывод (а не внутренности).

Когда программа запускается, она заполняет элемент управления ListView на первой странице TabControl 256-ю элементами, каждый из которых указывает первый и

последний символ группы из 256. Вот реальный код обработчика события OnCreate формы и простая функция, используемая для отображения каждого элемента, выдающая результат, показанный на рисунке 6.3:

```
// helper function
function GetCharDescr (nChar: Integer): string;
begin
  if Char(nChar).IsControl then
    Result := 'Char #' + IntToStr (nChar) + ' [  ]'
  else
    Result := 'Char #' + IntToStr (nChar) +
      ' [' + Char (nChar) + ']';
end;

procedure TForm2.FormCreate(Sender: TObject);
var
  I: Integer;
  ListItem: TListViewItem;
begin
  for I := 0 to 255 do // 256 pages * 256 characters each
  begin
    ListItem := ListView1.Items.Add;
    ListItem.Tag := I;
    if (I < 216) or (I > 223) then
      ListItem.Text :=
        GetCharDescr(I*256) + '/' + GetCharDescr(I*256+255)
    else
      ListItem.Text := 'Surrogate Code Points';
  end;
end;
```

Рисунок 6.3: Первая страница прикладного проекта `ShowUnicode` содержит длинный список разделов символов Юникода

Обратите внимание, как код сохраняет номер "страницы" в свойстве `Tag` элементов ListView, информацию, используемую позже для заполнения страницы. По мере того, как пользователь выбирает один из элементов, приложение переходит на вторую страницу TabControl, заполняя ее строковую сетку 256 символами раздела:

```
procedure TForm2.ListView1ItemClick(const Sender: TObject;
  const AItem: TListViewItem);
var
  I, NStart: Integer;
begin
  NStart := AItem.Tag  * 256;
  for I := 0 to 255 do
  begin
    StringGrid1.Cells [I mod 16, I div 16] :=
      IfThen (not Char(I + NStart).IsControl, Char (I + NStart), '');
  end;
  TabControl1.ActiveTab := TabItem2;
```

Функция IfThen, используемая в приведенном выше коде, является проверкой альтернативы: Если условие, переданное в

первом параметре, истинно, то функция возвращает значение второго параметра, если нет, то возвращает значение третьего параметра. В тесте по первому параметру используется метод `IsControl` хелпера типа Char для отфильтровывания непечатаемых управляющих символов.

примечание Функция `IfThen` более-менее похожа на оператор `? :` большинства языков программирования, основанный на синтаксисе языка Си. Есть версия для строк и отдельная для целых чисел. Для версии для строк необходимо включить модуль System.StrUtils, для версии для целых чисел `IfThen` - System.SysUtils.

Сетка символов Юникода, созданная приложением, видна на рисунке 6.4. Обратите внимание, что вывод изменяется в зависимости от способности выбранного шрифта и конкретной операционной системы отображать заданный символ Юникода.

Рисунок 6.4: на второй странице прикладного проекта `ShowUnicode` есть некоторые из актуальных символов Юникода

Вернемся к типу Char

После этого введения в Юникод, давайте вернемся к реальной теме этой главы, которая заключается в том, как язык Object Pascal управляет символами и строками. Я представил тип данных Char в главе 2 и упомянул о некоторых функциях помощника типа, доступных в модуле Character. Теперь, когда вы лучше понимаете Юникод, стоит вернуться к этому разделу и рассказать о нем подробнее.

Прежде всего, тип Char не всегда представляет собой точку кода Юникода. Фактически, тип данных использует 2 байта для каждого элемента. Хотя он и представляет собой кодовую точку для элементов в Unicode'e Basic Multi-language Plane (BMP), Char также может быть частью пары суррогатных значений, представляя собой кодовую точку.

Технически, существует другой тип, который можно использовать для непосредственного представления любой точки кода Юникода, и это тип UCS4Char, который использовал 4 байта для представления значения). Этот тип используется редко, так как требуемую дополнительную память, как правило, трудно обосновать, но вы можете заметить, что модуль Character (рассматривается далее) также включает в себя несколько операций для этого типа данных.

Возвращаясь к типу Char, помните, что это порядковый тип (даже если он достаточно большой), поэтому он имеет понятие последовательности и предлагает базовые операции, такие как Ord, Inc, Dec, High и Low. Большинство расширенных операций, в том числе специальный помощник типа, не входят в базовую систему модулей RTL, но требуют включения модуля Character.

Операции в Юникод из модуля Character

Большинство специфических операций для символов Юникода (а также, разумеется, строк Юникода) определяются в специальных юнитах под названием System.Character. В этом модуле определяется помощник TCharHelper для типа Char, который позволяет применять операции непосредственно к переменным этого типа.

примечание Модуль Character также определяет запись TCharacter, которая в сущности представляет собой набор статических функций класса, плюс ряд глобальных процедур, отображенных в этом методе. Это более старые, устаревшие функции, учитывая, что теперь предпочтительным способом работы с типом Char на уровне Unicode является использование помощника класса.

Модуль также определяет два интересных перечисленных типа. Первый называется TUnicodeCategory и отображает различные символы в широких категориях, таких как управляющие, пробелы, прописная или строчная буква, десятичное число, знаки препинания, математический символ и многое другое. Второе перечисление называется TUnicodeBreak и определяет семейство различных пробелов, дефисов и разрывов. Если вы привыкли к ASCII операциям, то это большое отличие.

Цифры в Юникоде — это не только символы между 0 и 9; пробелы не ограничиваются символом #32; и так далее для многих других соглашений (гораздо более простого) 256-элементного алфавита.

Помощник типа Char имеет более 40 методов, которые включают в себя множество различных тестов и операций. Их можно использовать для :

Получение числового представления символа (GetNumericValue).

Запрос категории (GetUnicodeCategory) или проверка ее на соответствие одной из различных категорий (IsLetterOrDigit, IsLetter, IsDigit, IsNumber, IsControl, IsWhiteSpace, IsPunctuation, IsSymbol и IsSeparator). Я использовал операцию IsControl в предыдущей демонстрации.

Проверка, является ли она строчной или заглавной (IsLower и IsUpper) или преобразование (ToLower и ToUpper).

Проверка, является ли она частью суррогатной пары UTF-16 (IsSurrogate, IsLowSurrogate и IsHighSurrogate) и преобразование суррогатных пар различными способами.

Преобразование в и из UTF32 (ConvertFromUtf32 и ConvertToUtf32) и типа UCS4Char (ToUCS4Char).

Проверка, является ли она частью заданного списка символов (IsInArray).

> Обратите внимание, что некоторые из этих операций могут быть применены к типу в целом, а не к конкретной переменной. При этом их можно вызывать, используя в качестве префикса тип Char, как во втором фрагменте кода ниже.
>
> Чтобы немного поэкспериментировать с этими операциями над символами Юникода, я создал прикладной проект под названием CharTest. Одним из примеров этого примера является эффект вызова прописных и строчных операций над элементами Юникода. На самом деле, классическая функция RTL UpCase работает только с базовыми 26 символами английского языка ANSI-представления, в то время как в ней не получается какой-то символ Юникода, который имеет конкретное заглавное представление (не все алфавиты имеют понятие заглавного, поэтому это не универсальное понятие).

Для проверки этого сценария в проект приложения CharTest я добавил следующий фрагмент, который пытается преобразовать букву с акцентом в заглавный:

```
var
  Ch1: Char;
begin
  Ch1 := 'ù';
  Show ('UpCase ù: ' + UpCase(Ch1));
  Show ('ToUpper ù: ' + Ch1.ToUpper);
```

Традиционный `Upcase` вызов не преобразует символ с латинским акцентом, в то время как функция `ToUpper` работает правильно:

```
UpCase ù: ù
ToUpper ù: Ù
```

В справке по типу Char есть много функций, связанных с Юникодом, например, выделенные в коде ниже, которые определяют строку как включающую в себя также и символ вне BMP (первые 64K точек кода Юникода). В фрагменте кода, также являющемся частью проекта приложения CharTest, есть несколько тестов на различных элементах строки, все из которых возвращают True:

```
var
  Str1: string;
begin
  Str1 := '1.' + #9 + Char.ConvertFromUtf32 (128) +
    Char.ConvertFromUtf32($1D11E);
  ShowBool (Str1.Chars[0].IsNumber);
  ShowBool (Str1.Chars[1].IsPunctuation);
  ShowBool (Str1.Chars[2].IsWhiteSpace);
  ShowBool (Str1.Chars[3].IsControl);
  ShowBool (Str1.Chars[4].IsSurrogate);
end;
```

Функция показа, используемая в данном случае, является адаптированной версией:

```
procedure TForm1.ShowBool(value: Boolean);
begin
  Show(BoolToStr (Value, True));
end;
```

примечание Unicode code point $1D11E - *музыкальный символ G clef*.

Символьные литералы Юникода

В нескольких примерах мы видели, что переменной строкового типа можно присвоить индивидуальный символьный или строковый литерал. В целом, использование числового представления символа с префиксом # достаточно простое. Однако есть некоторые исключения. Для обратной совместимости, обычные символьные литералы преобразуются в зависимости от их контекста. Возьмем следующее простое присвоение числового значения 128, которое, вероятно, указывает на использование символа валюты евро (€):

```
var
  Str1: string;
begin
  Str1 := #$80;
```

Данный код не совместим с Юникодом, так как точка кода для этого символа 8364. Значение, на самом деле, исходит не из официальных ISO кодовых страниц, а из специфической реализации Microsoft для Windows. Чтобы облегчить перенос существующего кода в Юникод, компилятор Object Pascal может трактовать 2-значные строковые литералы как ANSI-символы (что может зависеть от вашей реальной кодовой страницы). Удивительно, но если вы возьмете это значение, конвертируете его в символ Char, и отобразите его снова... числовое представление изменится на правильное. Таким образом, выполняя операцию:

```
Show (Str1 + ' - ' + IntToStr (Ord (Str1[1])));
```

Я получу на выходе:

```
€ - 8364
```

Учитывая, что вы, возможно, предпочтете полностью мигрировать свой код и избавиться от старых литеральных значений на основе ANSI, вы можете изменить поведение компилятора, используя специальную директиву

$HIGHCHARUNICODE. Эта директива определяет, как компилятор обрабатывает литеральные значения между `#$80` и `#$FF`. То, о чем я говорил ранее, это действие опции по умолчанию (OFF). Если ее включить, то та же самая программа выдаст этот вывод:

```
 - 128
```

Число интерпретируется как фактическая точка кода Юникода, а вывод будет содержать непечатаемый управляющий символ. Другим вариантом выражения этой конкретной точки кода (или любой точки кода Юникода ниже `#$FFFF`) является использование четырехзначной нотации:

```
str1 := #$0080;
```

Это не интерпретируется как символ Евро валюты независимо от установки директивы $HIGHCHARUNICODE.

примечание Вышеуказанный код и соответствующая демонстрационная версия работают только для американской или западноевропейской локали. В других локалях символы от 128 до 255 интерпретируются по-разному.

Приятно то, что вы можете использовать четырехзначную нотацию для обозначения дальневосточных иероглифов, как следующие два японских иероглифа:

```
Str1 := #$3042#$3044;
 Show (Str1 + ' - ' + IntToStr (Ord (Str1.Chars[0])) +
   ' - ' + IntToStr (Ord (Str1.Chars[1])));
```

отображаются как (вместе с их Целочисленным представлением):

```
あい - 12354 - 12356
```

Вы также можете использовать литеральные элементы поверх `#$FFFF`, которые будут преобразованы в соответствующую суррогатную пару.

Как насчет 1-байтовых Char?

Как я упоминал ранее, язык Object Pascal реализует тип Char как `WideChar`, но по-прежнему определяет тип `AnsiChar`, в основном для совместимости с существующим кодом. Общей рекомендацией является использование байтового типа для однобайтовой структуры данных, но это правда, что `AnsiChar` может быть удобен при обработке 1-байтовых символов.

Для нескольких предыдущих версий `AnsiChar` не был доступен на мобильных платформах, зато начиная с 10.4 этот тип данных работает одинаково на всех компиляторах Delphi. При привязке данных к API платформы или сохранении в файлы, как правило, следует держаться подальше от старых однобайтовых символов, даже если они поддерживаются. Использование кодировки Unicode, безусловно, является предпочтительным подходом. Правда, 1-байтовая обработка символов может быть быстрее и использовать меньше памяти, чем 2-байтовый аналог.

Тип данных String

Строковый тип данных в Object Pascal намного сложнее, чем простой массив символов, и имеет возможности, которые выходят далеко за рамки того, что большинство языков программирования делают со схожими типами данных. В этом разделе я расскажу о ключевых понятиях, лежащих в основе этого типа данных, а в следующих разделах мы рассмотрим некоторые из этих возможностей более подробно.

В следующем списке я собрал ключевые понятия для понимания того, как строки работают в языке (помните, вы

можете использовать строки, не зная многого из этого, так как внутреннее поведение очень прозрачно):

Память для строкового типа ***динамически выделяется*** на куче. Строковая переменная — это просто ссылка на фактические данные. Не то, чтобы вам приходилось много беспокоиться об этом, так как компилятор работает с этим прозрачно. Как и для динамического массива, когда вы объявляете новую строку, она пуста.

Хотя вы можете присваивать данные строке различными способами, вы также можете ***назначить определенную область памяти*** заданного размера, вызвав функцию `SetLength`. Параметр — это количество символов (по 2 байта каждый), которое должна иметь строка. При расширении строки существующие данные сохраняются (но могут быть перенесены в новую физическую область памяти). При уменьшении размера, часть содержимого, скорее всего, будет потеряна. Установка длины строки редко бывает необходима. Единственный распространенный случай — это когда необходимо передать буфер строки в функцию операционной системы для данной платформы.

- Если вы хотите ***увеличить размер*** строки в памяти (объединив ее с другой строкой), но в соседней памяти есть что-то еще, то строка не может расти в той же самой области памяти, и поэтому полная копия строки должна быть сделана в другой области.
- Чтобы очистить строку, вы не работаете с самой ссылкой, а можете просто установить ее в пустую строку, то есть в `''`. Или можно использовать константу Empty, которая соответствует этому значению.

Согласно правилам Object Pascal, ***длина строки*** (которую можно получить, вызвав Length) - это количество действительных элементов, а не количество выделенных. В отличие от C, который имеет понятие терминатора строки (`#0`), все версии Pascal с самого начала склонны отдавать предпочтение использованию

определенной области памяти (части строки), в которой хранится информация о фактической длине. Однако иногда можно встретить строки, в которых также присутствует терминатор.

Строки Object Pascal используют механизм подсчета ссылок, который отслеживает, сколько строковых переменных ссылается на данную строку в памяти. Подсчет ссылок освобождает память, когда строка больше не используется, то есть, когда больше нет строковых переменных, ссылающихся на данные... и счетчик ссылок достигает нуля.

Строки используют технологию копирования copy-on-write, которая является высокоэффективной. Когда вы назначаете строку другой или передаете строку строковому параметру, данные не копируются, а счетчик ссылок увеличивается. Однако, если вы все же измените содержимое одной из ссылок, система сначала сделает копию, а затем изменит только эту копию, при этом остальные ссылки останутся без изменений.

Использование конкатенирования строк для добавления содержимого в существующую строку, как правило, очень быстро и не имеет существенного недостатка. Несмотря на то, что существуют альтернативные подходы, конкатенация строк быстрая и мощная. В наши дни это не так для многих языков программирования.

> Я догадываюсь, что это описание может быть немного запутанным, так что давайте рассмотрим использование строк на практике. Через некоторое время я перейду к демонстрации некоторых из вышеперечисленных операций, включая подсчет ссылок и копирование на запись. Перед этим, однако, позвольте мне вернуться к вспомогательным операциям со строками и некоторым другим фундаментальным RTL-функциям для управления строками.
>
> Прежде чем продолжить, позвольте мне рассмотреть некоторые элементы предыдущего списка с точки зрения реального кода. Учитывая, что операции со строками довольно

ясные, трудно полностью понять, что происходит, если только вы не начнете заглядывать внутрь структуры памяти строк, что я и сделаю позже в этой главе, так как пока это было бы слишком продвинутой темой. Итак, давайте начнем с простых строковых операций из проекта приложения Strings101:

```
var
  String1, String2: string;
begin
  String1 := 'hello world';
  String2 := String1;
  Show ('1: ' + String1);
  Show ('2: ' + String2);
  String2 := String2 + ', again';
  Show ('1: ' + String1);
  Show ('2: ' + String2);
end;
```

Когда выполняется этот фрагмент, видно, что если вы назначите две строки одному и тому же содержимому, то изменение одной строки не повлияет на другую. То есть на `String1` изменения в `String2` не влияют:

```
1: hello world
2: hello world
1: hello world
2: hello world, again
```

Тем не менее, как мы увидим яснее в более поздней демонстрации, начальное присвоение не вызывает полного копирования строки, копирование задерживается (опять же, функция под называется "копирование на запись").

Другая важная особенность, которую следует понимать, это то, как управляется размер. Если вы запрашиваете длину строки, вы получаете фактическое значение (которое хранится в строковых метаданных, что делает операцию очень быстрой). Но если вы вызываете `SetLength`, вы выделяете память, которая чаще всего не инициализируется. Обычно это используется при передаче строки в качестве буфера во внешнюю системную функцию. Если вам нужна пустая строка, вместо нее вы можете использовать псевдо-конструктор (`Create`). Наконец, вы можете

использовать `SetLength` для обрезки строки. Все это демонстрируется следующим кодом:

```
  SetLength (String1, 100);
  Show (String1);
  Show ('Length: ' + String1.Length.ToString);

  String1 := 'hello world';
  Show (String1);
  Show ('Length: ' + String1.Length.ToString);

  String1 := string1 + String.Create(' ', 100);
  SetLength (String1, 100);
  Show (String1);
  Show ('Length: ' + String1.Length.ToString);
```

Выходные данные более или менее следующие:

```
hello world
Length: 11
hello
world~˘~~̰~˜~´~H~ʋ~ȝ~Ј~Щ~Ы~Ѭ~ƥ~ɣ~Ә~Ҳ~ҡ~ς~h~�~ࢬ~ݦ~ݐ~ِ~ݜ~۵~ۓ~گ~ݓ~٦~ڡ~؛~�~ו~ݕ~�~�~□~□~ؘ~I~~�~ࢭ~⟨~
Length: 100
hello world
Length: 11
hello world
Length: 100
```

Третья концепция, которуе я хочу подчеркнуть в этом разделе, - это пустая строка. Строка пустая, когда ее содержимое - пустая строка. Как для присваивания, так и для сравнения можно использовать две последовательные кавычки или специальные функции:

```
var
  String1: string;
begin
  String1 := 'hello world';
  if String1 = '' then
    Show('Empty')
  else
    Show('Not empty');

  String1 := ''; // or String1.Empty;
  if String1.IsEmpty then
    Show('Empty')
  else
    Show('Not empty');
```

С этим простым выходом:

```
Not empty
Empty
```

Передача строк как праметры

Как я уже объяснял, если вы присваиваете строку другой, вы просто копируете ссылку, в то время как реальная строка в памяти не дублируется. Однако, если вы пишете код, который изменяет эту строку, то сначала строка копируется (только в этот момент), а затем модифицируется.

Что-то очень похожее происходит, когда вы передаете строку в качестве параметра в функцию или процедуру. По умолчанию вы получаете новую ссылку, и если вы изменяете строку в функции, изменение не влияет на исходную строку. Если вы хотите другое поведение, то есть возможность модифицировать исходную строку в функции, вам нужно передать строку по ссылке, используя ключевое слово `var` (как это происходит для большинства других простых и управляемых типов данных).

Но что, если вы не измените строку, переданную в качестве параметра? В этом случае можно применить существенную оптимизацию, используя модификатор `const` для параметра. В этом случае компилятор не позволит изменить строку в функции или процедуре, но и оптимизирует операцию передачи параметра. В самом деле, `const` string не требует от функции увеличения количества ссылок на строку при старте и уменьшения при завершении. Хотя эти операции очень быстрые, их выполнение в тысячи или миллионы раз добавит немного накладных расходов вашим программам. Поэтому передача строки как `const` рекомендуется в тех случаях, когда функции не нужно изменять значение строкового параметра

(хотя есть потенциальные проблемы, описанные в примечании ниже).

В терминах кодирования это декларации трех процедур со строковыми параметрами, передаваемыми разными способами:

```
procedure ShowMsg1 (Str: string);
procedure ShowMsg2 (var Str: string);
procedure ShowMsg3 (const Str: string);
```

примечание В последние годы наблюдается сильный тренд к передаче всех строковых параметров как `const`, если только функции и методы не модифицируют эти строки. Однако есть очень большое предостережение. Для константного строкового параметра компилятор берет строковую ссылку и не "управляет" ею (нет подсчета ссылок и т.д.), рассматривая ее как указатель на ячейку памяти. Компилятор корректно проверяет, что код рутины не изменяет строковый параметр. Однако он не контролирует, что происходит с исходной строкой, на которую ссылается параметр. Изменения этой строки могут повлиять на ее расположение и расположение в памяти, что-то, с чем может справиться обычный строковый параметр (строки с несколькими ссылками делают операцию копирования на запись автоматически), в то время как константный строковый параметр будет игнорироваться. Другими словами, изменение исходной строки делает недействительным const параметр, на который она ссылается, и его использование, скорее всего, приведет к ошибке доступа к памяти.

Использование [] и режимов подсчета строковых символов

Как вы, вероятно, знаете, если вы использовали Object Pascal или любой другой язык программирования, важнейшей операцией со строкой является обращение к одному из элементов строки, что часто достигается с помощью использования квадратных скобок (`[]`), точно так же, как и обращение к элементам массива.

В Object Pascal есть два слегка разных способа выполнения этих операций:

Операция хулпера строкового типа Chars[] (весь список находится в следующем разделе) - это доступ только для чтения, использующий индекс, начинающийся с 0.

Стандартный строковый оператор `[]` поддерживает как чтение, так и запись, и по умолчанию использует классический паскалевский индекс, начинающийся с 1. Эта настройка может быть изменена с помощью директивы компилятора.

Я дам разъяснения по этому вопросу после краткой исторической справки ниже. Причина этого примечания, которое можно пропустить, если не интересно, заключается в том, что трудно понять, почему сейчас язык ведет себя так, не посмотрев на то, что произошло со временем.

примечание Позвольте мне на секунду оглянуться назад, чтобы объяснить вам, как мы пришли к настоящему состоянию. В ранние времена языка Pascal строки рассматривались как массив символов, в котором первый элемент (т.е. нулевой элемент массива) использовался для хранения количества имеющихся символов в строке, или длины строки. В те дни, когда в языке C каждый раз приходилось пересчитывать длину строки, искать NULL-терминатор, код Pascal мог просто сделать прямую проверку на этот байт. Учитывая, что для определения длины использовался нулевой номер байта, случилось так, что первый фактический символ в строке находился в первой позиции.
Со временем почти во всех других языках появились строки и массивы с индексом, начинающимся с нуля. Позже Object Pascal принял нуль-индексные динамические массивы, а большая часть в RTL и библиотек компонентов использовали нуль-индексные структуры данных, причем строки являлись существенным исключением.
При переходе в мобильный мир дизайнеры языка Object Pascal решили отдать "приоритет" нуль-индексным строкам, позволяя разработчикам по-прежнему использовать старую модель в случае, если у них есть существующий исходный код Object Pascal для миграции кода, контролируя поведение с помощью директивы компилятора. В Delphi 10.4, однако, первоначальное решение было отменено, чтобы учесть большую согласованность исходного кода, независимо от целевой платформы. Другими словами, было решено отдавать предпочтение цели "единой исходной многоплатформенности" перед целью "быть похожим на другие современные языки".

Если мы хотим провести сравнение, чтобы лучше объяснить различия в базе индекса, рассмотрим, как подсчитываются этажи в Европе и в Северной Америке (честно говоря, я не

знаю, как обстоят дела в остальном мире). В Европе первый этаж — это этаж 0, а первый этаж — это этаж выше (иногда формально обозначается как "первый этаж выше земли"). В Северной Америке первый этаж — это первый этаж, а второй — это первый этаж, расположенный выше уровня земли. Иными словами, в Америке используется индекс этажа 1, а в Европе - индекс этажа 0.

Для строк в большинстве языков программирования используется нотация с базой 0, независимо от того, на каком континенте они были изобретены. Delphi и большинство диалектов Pascal используют нотацию на основе 1.

Позвольте мне немного лучше объяснить ситуацию со строковыми индексами. Как я уже упоминал, нотация Chars[] неизменно использует нулевой индекс. Так что, если вы напишете

```
var
  String1: string;
begin
  String1 := 'hello world';
  Show (String1.Chars[1]);
```

результат будет:

```
e
```

В случае, если вы используете прямую нотацию [], то на выходе операции:

```
 Show (String1[1]);
```

По умолчанию результат будет

```
h
```

Однако это может быть `e`, если компилятор определит, что $ZEROBASEDSTRING был `ON`. На данный момент (то есть после выхода Delphi 10.4.) рекомендуется переходить на строки на базе 1 во всем коде и избегать работы с разными моделями.

Но что делать, если вы хотите написать код, который работает независимо от настройки $ZEROBASEDSTRING? Вы можете абстрагироваться от индекса, например, используя Low`(string)` как индекс первого значения и `High(string)` как индекс последнего значения. Это работает и возвращает нужное значение в зависимости от настройки локального компилятора для строковой базы:

```
var
  S: string;
  I: Integer;
begin
  S := 'Hello world';
  for I := Low (S) to High (S) do
    Show(S[I]);
```

Другими словами, в строке *всегда* есть элементы, варьирующиеся от результата функции `Low` до результата функции High, применяемого к той же самой строке.

примечание Строка — это всего лишь строка, и *концепция* нуль-индекс строки совершенно неверна. Структура данных в памяти ничем не отличается, поэтому можно передать любую строку в любую функцию, использующую нотацию с любым базовым значением, и никаких проблем не возникнет. Другими словами, если у вас есть фрагмент кода, обращающийся к строкам с нулевой нотацией, вы можете передать строку в функцию, которая компилируется с помощью настроек для нотации на базе 1.

Конкатенация строк

Я уже упоминал, что в отличие от других языков, Object Pascal имеет полную поддержку прямой конкатенации строк, что на самом деле является довольно быстрой операцией. В этой главе я просто покажу вам немного кода конкатенаций строк с одновременным тестированием быстродействия. Позже, в 18-й главе, я вкратце расскажу о классе TStringBuilder, который следует нотации .NET для сборки строки из разных фрагментов. Хотя есть причины использовать `TStringBuilder`, его

производительность не самая релевантная (как покажет следующий пример).

Итак, как нам соединить строки в Object Pascal? Просто используя оператор +:

```
var
  Str1, Str2: string;
begin
  Str1 := 'Hello,';
  Str2 := ' world';
  Str1 := Str1 + Str2;
```

Обратите внимание, как я использовал переменную `str1` как слева, так и справа от присваивания, добавляя еще больше содержимого к существующей строке, а не присваивая ее совершенно новой. Обе операции возможны, но добавление содержимого к существующей строке — это то, где можно получить неплохую производительность.

Этот тип конкатенирования может быть также выполнен в цикле, как делает следующий пример, извлеченный из проекта приложения LargeString:

```
использует
uses
  System.Diagnostics;

const
  MaxLoop = 2000000; // two million

var
  Str1, Str2: string;
  I: Integer;
  T1: TStopwatch;
begin
  Str1 := 'Marco ';
  Str2 := 'Cantu ';

  T1 := TStopwatch.StartNew;
  for I := 1 to MaxLoop do
    Str1 := Str1 + Str2;

  T1.Stop;
  Show('Length: ' + Str1.Length.ToString);
  Show('Concatenation: ' + T1.ElapsedMilliseconds.ToString);
end;
```

При выполнении того кода, я получаю следующие времена на виртуальных машинах под управлением Windows и на Android-устройстве (компьютер работает намного быстрее):

```
Length: 12000006          // Windows (in a VM)
Concatenation: 59

Length: 12000006          // Android (on device)
Concatenation: 991
```

В проекте приложения также имеется аналогичный код, основанный на классе `TStringBuilder`. Хотя я не хочу подробно останавливаться на этом коде (опять же, я опишу класс в главе 18), я хочу поделиться реальным временем для сравнения с обычным временем конкатенации, которое только что было показано.

```
Length: 12000006          // Windows (in a VM)
StringBuilder: 79

Length: 12000006          // Android (on device)
StringBuilder: 1057
```

Как видите, конкатенацию можно смело считать наиболее быстрым вариантом.

Операции хелпера строк

Учитывая важность строкового типа, неудивительно, что помощник для этого типа имеет довольно длинный список операций, которые можно выполнить. А учитывая его важность и общность этих операций в большинстве приложений, я думаю, что к этому списку стоит подойти с достаточным вниманием.

Есть ключевое отличие между использованием классических функций глобальной манипуляции строками Delphi и методов помощника строк: классические операции предполагают строки на основе 1, в то время как операция помощника строки использует логику на базе нуль-индекса!

Я логически сгруппировал операции строкового помощника (большинство из которых имеют много перегруженных версий), коротко описывая, что они делают, учитывая, что довольно часто их имена довольно интуитивно понятны:

Операции копирования или частичного копирования, такие как Copy, CopyTo, Join, и SubString

Операции по изменению строк, такие как Insert, Remove, Replace (Вставка, Удаление, Замена)

Для преобразования различных типов данных в строки можно использовать Parse и Format

Преобразование в различные типы данных, когда это возможно, может быть достигнуто с помощью ToBoolean, ToInteger, ToSingle, ToDouble и ToExtended, в то время как с помощью ToCharArray можно превратить строку в массив символов.

Заполните строку пробелами или определенными символами с помощью PadLeft, PadRight и одной из перегруженных версий Create. Напротив, вы можете удалить пробелы на одном конце строки или обоих, используя TrimRight, TrimLeft и Trim

Сравнение строк и тест на равенство (Compare, CompareOrdinal, CompareText, CompareTo и Equals) - но имейте в виду, что вы также можете в некоторой степени использовать оператор равенства и операторы сравнения.

Изменение регистра с LowerCase и UpperCase, ToLower и ToUpper и ToUpperInvariant

Проверяйте содержимое строки с помощью таких операций, как Contains, StartsWith, EndsWith. Поиск в строке может быть выполнен с помощью IndexOf для поиска позиции заданного символа (от начала или от заданной позиции), аналогичной IndexOfAny (которая ищет один из элементов массива символов), операций LastIndexOf и LastIndexOfAny, которые работают в обратном направлении от конца строки, а также операций специального назначения IsDelimiter и LastDelimiter

Доступ к общей информации о строке с функциями типа `Length`, возвращающей количество символов, CountChars, учитывающей также суррогатные пары, GetHashCode, возвращающей хэш строки, и различные тесты на "пустоту", включающие в себя IsEmpty, `IsNullOrEmpty` и IsNullOrWhiteSpace

Строковые специальные операции, такие как Split, которые разбивают строку на несколько на основе определенного символа, и удаление или добавление кавычек вокруг строки с помощью QuotedString и DeQuoted.

И, наконец, получить доступ к отдельным символам с помощью Chars[], который принимает цифровой индекс элемента строки между квадратных скобок. Это может быть использовано только для чтения значения (а не для его изменения). Используется нулевой индекс, как и во всех других операциях со строковым помощником.

> Важно заметить, что на самом деле все методы строкового хелпера строились в соответствии со строковой конвенцией, используемой многими другими языками, которая включает в себя концепцию, что строковые элементы начинаются с нуля и идут до длины строки минус единица. Другими словами, как я уже упоминал ранее, но стоит еще раз подчеркнуть, все операции со строковым помощником используют в качестве параметров нулевые индексы и возвращаемые значения.

примечание Операция Split является относительно новой для Object Pascal RTL. Ранее распространенным подходом была загрузка строки в список строк, после установки определенного разделителя строк, и последующий доступ к отдельным строкам или строкам. Операция `Split` значительно более эффективна и гибка.

Учитывая большое количество операций, которые вы можете применить непосредственно к строкам, я мог бы создать несколько проектов, демонстрирующих эти возможности. Вместо этого я остановлюсь на нескольких относительно простых, хотя и очень распространенных операциях.

Прикладной проект StringHelperTest имеет две кнопки. В каждой из них в первой части кода строится и отображается строка:

```
var
  Str1, Str2: string;
  I, NIndex: Integer;
begin
  Str1 := '';

  // create string
  for I := 1 to 10 do
    Str1 := Str1 + 'Object ';

  Str2:= string.Copy (Str1);
  Str1 := Str2 + 'Pascal ' + Str2.Substring (10, 30);
  Show(Str1);
```

Заметьте, как я использовал функцию `Copy`, чтобы создать уникальную копию данных строки, а не псевдоним... даже если бы в этой конкретной демонстрации это не имело никакого значения. Вызов `Substring` в конце используется для извлечения части строки. В результате получается текст:

```
Object Object Object Object Object Object Object Object Object Object
Pascal ect Object Object Object Objec
```

После такой инициализации первая кнопка имеет код для поиска подстроки и повторения такого поиска, с другим начальным индексом, для подсчета вхождений заданной строки (в примере один символ):

```
 // найти подстроку
  Show('Pascal at: ' +
    Str1.IndexOf ('Pascal').ToString);

  // count occurrences
  I := -1;
  NCount := 0;
  repeat
    I := Str1.IndexOf('O', I + 1); // search from next element
    if I >= 0 then
      Inc (NCount); // found one
  until I < 0;

  Show('O found: ' +
    NCount.ToString + ' times');
```

Я знаю, что цикл repeat не самый простой: он начинается с отрицательного индекса, так как любой последующий поиск начинается с индекса после текущего; он считает вхождения; и его окончание основано на том, что если элемент не найден, то возвращается `-1`. Код выводит:

```
Pascal at: 70
O found: 14 times
```

Вторая кнопка имеет код для выполнения поиска и замены одного или нескольких элементов строки на что-то другое. В первой части она создает новую строку, копируя начальную и конечную часть и добавляя новый текст посередине. Во второй она использует функцию Replace, которая может работать с несколькими вхождениями, просто передавая ей соответствующий флаг (`rfReplaceAll`).

Это код:

```
// разовая замена
  NIndex := Str1.IndexOf ('Pascal');
  Str1 := Str1.Substring(0, NIndex) + 'Object' +
    Str1.Substring(nIndex + ('Pascal').Length);
  Show (Str1);

// мульти-замена
  Str1 := Str1.Replace('O', 'o', [rfReplaceAll]);
  Show (Str1);
```

Поскольку вывод довольно длинный и не легко читается, здесь я перечислил только центральную часть каждой строки:

```
...Object Pascal ect Object Object...
...Object Object ect Object Object...
...object object ect object object...
```

Опять же, это всего лишь минимальный пример из всего богатства операций со строками, которые вы можете выполнять, используя операции, доступные для строкового типа, с помощью помощника строкового типа.

Больше RTL функций для строк

Эффект от решения реализовать строковый помощник, следуя названиям операций, распространённых в других языках программирования, заключается в том, что названия операций типа часто расходятся с традиционными из Object Pascal (которые и сегодня доступны в виде глобальных функций).

В следующей таблице приведены некоторые *несовпадающие* имена функций:

```
global              string type helper
Pos                       IndexOf
IntToStr                  Parse
StrToInt                  ToInteger
CharsOf                   Create
StringReplace             Replace
```

примечание Помните, что существует большая разница между глобальными операциями и Char-помощниками: Первая группа использует 1-based нотацию для индексирования элементов внутри строки, в то время как вторая группа использует 0-based нотацию (как объяснялось ранее).

Это только наиболее часто используемые функции строк в RTL, которые изменили имя, в то время как многие другие по прежнему используют те же самые, как `UpperCase` или `QuotedString`. Модуль System.SysUtils содержит их гораздо больше, а специальный модуль System.StrUtils также имеет множество функций, предназначенных для работы со строками, которые не являются частью помощника строки.

Некоторые примечательные функции являются частью `System.StrUtils`:

ResemblesText, которая реализует алгоритм *Soundex* (поиск слов со схожим звучанием, даже если написано по-другому);

DupeString, которая возвращает запрашиваемое количество копий заданной строки;

IfThen, которая возвращает первую переданную строку, если условие истинно, иначе он вернет вторую строку (я использовал эту функцию в фрагменте кода ранее в этой главе);

ReverseString, которая возвращает строку с обратной последовательностью символов.

Строки форматирования

Хотя соединение строк с оператором плюс (+) и использование некоторых функций преобразования, позволяет построить действительно сложные строки из существующих значений различных типов данных, но существует другой и более мощный подход к форматированию чисел, значений валют и других строк в конечную строку. Сложное форматирование строк может быть достигнуто путем вызова функции `Format`, очень традиционного, но все же чрезвычайно распространенного механизма, не только в Object Pascal, но и в большинстве языков программирования.

история Семейство функций "print format string" или `printf` восходит к ранним временам программирования и к таким языкам, как FORTRAN 66, PL/1 и ALGOL 68. Специфическая структура строки формата, используемая и сегодня (и используемая Object Pascal), близка к функции `printf` на языке C. Исторический обзор можно найти на сайте en.wikipedia.org/wiki/Printf_format_string.

Функция `Format` требует в качестве параметров строку с основным текстом и некоторыми шаблонами форматирования (помеченными символом %) и массив значений, обычно по одному для каждого шаблона. Например, для форматирования двух чисел в строку можно написать:

```
Format ('First %d, Second %d', [n1, n2]);
```

где `n1` и `n2` - два целых значения. Первое значение заменяется первым, второе совпадает со вторым и так далее. Если тип вывода шаблона (обозначается буквой после символа `%`) не

совпадает с типом соответствующего параметра, то происходит ошибка во время выполнения. Фактически, отсутствие проверки типа компиляции является самым большим недостатком использования функции `Format`. Аналогично, отсутствие передачи достаточного количества параметров приводит к ошибке во время выполнения.

Функция `Format` использует параметр в виде открытого массива (параметр, который может иметь произвольное количество значений или произвольные типы данных, как описано в Главе 5). Кроме использования `%d`, можно использовать один из многих других шаблонов форматирования, определенных этой функцией и кратко перечисленных в следующей таблице. Эти шаблоны обеспечивают вывод по умолчанию для данного типа данных. Однако вы можете использовать другие спецификаторы формата для изменения вывода по умолчанию. Спецификатор ширины, например, определяет фиксированное количество символов в выводе, в то время как спецификатор точности указывает количество десятичных цифр. Например,

```
Format ('%8d', [n1]);
```

преобразует число `n1` в восьмисимвольную строку, выравнивая текст справа (используйте символ минус (-) для указания выравнивания влево), заполняя его пробелами. Ниже приведен список шаблонов форматирования для различных типов данных:

`d` (decimal)	Соответствующее целое значение преобразуется в строку десятичных цифр.
`x` (hexadecimal)	Соответствующее целое значение преобразуется в строку шестнадцатеричных цифр.

`p` (pointer)	Соответствующее значение указателя преобразуется в строку, выраженную шестнадцатеричными цифрами.
`s` (string)	соответствующее значение строки, символа или `PChar` (указатель на символьный массив).
`e` (exponential)	Соответствующее значение с плавающей точкой преобразуется в строку, основанную на научной нотации.
`f` (floating point)	Соответствующее значение с плавающей точкой преобразуется в строку, основанную на нотации с плавающей точкой.
`g` (general)	Соответствующее значение с плавающей точкой преобразуется в кратчайшую десятичную строку с использованием либо плавающей запятой, либо экспоненциальной нотации.
`n` (number)	Соответствующее значение с плавающей точкой преобразуется в строку с плавающей точкой, но при этом используются разделители тысяч.
`m` (money)	Соответствующее значение с плавающей точкой конвертируется в строку, представляющую собой денежную сумму. Конвертация, как правило, основывается на региональных настройках.

Лучший способ увидеть примеры таких преобразований — это самостоятельно поэкспериментировать со строками форматирования. Для облегчения этого я написал проект приложения FormatString, которое позволяет пользователю предоставлять строки форматирования для нескольких предопределенных целочисленных значений.

Форма программы имеет поле редактирования над кнопками, изначально содержащее простую предопределенную строку форматирования, выступающую в качестве шаблона(`'%d - %d - %d'`). Первая кнопка приложения позволяет отобразить в окне редактирования пример более сложнй строки форматирования (код имеет простое назначение редактируемого текста строки форматирования `'Value %d, Align %4d, Fill %4.4d'`). Вторая кнопка позволяет применить строку форматирования к предопределенным значениям, используя следующий код:

```
var
  StrFmt: string;
  N1, N2, N3: Integer;
begin
  StrFmt := Edit1.Text;
  N1 := 8;
  N2 := 16;
  N3 := 256;

  Show (Format ('Format string: %s', [StrFmt]));
  Show (Format ('Input data: [%d, %d, %d]', [N1, N2, N3]));
  Show (Format ('Output: %s', [Format (StrFmt, [N1, N2, N3])]));
  Show (''); // blank line
end;
```

Если вы выводите сначала строку исходного формата, а затем строку с образцом формата (т.е. если вы нажмете вторую кнопку, первую, а затем вторую снова), то вы должны получить вывод, как показано ниже:

```
Input data: [8, 16, 256]
Output: 8 - 16 - 256

Format string: Value %d, Align %4d, Fill %4.4d
Input data: [8, 16, 256]
Output: Value 8, Align   16, Fill 0256
```

Однако идея программы состоит в том, чтобы редактировать строку форматирования и экспериментировать с ней, чтобы увидеть все различные доступные опции форматирования.

Внутренняя структура строк

Хотя в целом можно использовать строки, не зная об их внутреннем устройстве, интересно посмотреть на реальную структуру данных, стоящую за этим типом данных. В ранние времена языка Pascal строки имели максимум 255 элементов по одному байту и использовали первый байт (или нулевой байт) для хранения длины строки. С тех пор прошло много времени, но концепция хранения некоторой дополнительной информации о строке в составе ее данных остается специфическим подходом языка Object Pascal (в отличие от многих языков, которые берут свое начало от языка C и используют концепцию терминатора строк).

примечание `ShortString` - это имя традиционного типа строки Pascal, строка из одного байта символов или AnsiChar, ограниченная 255 символами. Тип ShortString по-прежнему доступен в настольных компиляторах, но не в мобильных. Можно создать подобную структуру данных с динамическим массивом байт, или `TBytes`, или обычным статическим массивом `Byte` элементов.

Как я уже упоминал, строковая переменная — это не что иное, как указатель на структуру данных, выделенную на куче. На самом деле, значение, хранящееся в строке, является не ссылкой на начало структуры данных, а ссылкой на первый из символов строки, при этом мета-данные строки доступны при отрицательном смещении от этого места. Внутреннее представление данных строкового типа выглядит следующим образом:

-12	-10	-8	-4	Ссылочный адрес строки

Кодовая страница	размер элемента	Количество ссылок	Длина	Первый символ строки

Первый элемент (отсчитывающий назад от начала самой строки) - Целое число с длиной строки, второй элемент содержит счетчик ссылок. Следующими полями (используемыми в компиляторах рабочего стола) являются размер элемента в байтах (1 или 2 байта) и кодовая страница для старых строковых типов, основанных на языке Ansi.

Довольно удивительно, что доступ к большинству из этих полей возможен с помощью специфических низкоуровневых строковых мета-функций данных, помимо достаточно очевидной функции `Length`:

```
function StringElementSize(const S: string): Word;
function StringCodePage(const S: string): Word;
function StringRefCount(const S: string): Longint;
```

В качестве примера можно создать строку и запросить некоторую информацию о ней, как я сделал это в проекте приложения StringMetaTest:

```
var
  Str1: string;
begin
  Str1 := 'F' + string.Create ('o', 2);

  Show ('SizeOf: ' + SizeOf (Str1).ToString);
  Show ('Length: ' + Str1.Length.ToString);
  Show ('StringElementSize: ' +
    StringElementSize (Str1).ToString);
  Show ('StringRefCount: ' +
    StringRefCount (Str1).ToString);
  Show ('StringCodePage: ' +
    StringCodePage (Str1).ToString);
  if StringCodePage (Str1) = DefaultUnicodeCodePage then
    Show ('Is Unicode');
  Show ('Size in bytes: ' +
    (Length (Str1) * StringElementSize (Str1)).ToString);
  Show ('ByteLength: ' +
    ByteLength (Str1).ToString);
```

примечание Причина, по которой строка *'Foo'* строится динамически, а не присваивается константа, заключается в том, что в константных строках счетчик ссылок отключен (или установлен в -1). В демонстрационном примере я предпочел

показывать правильное значение для счетчика ссылок, отсюда и динамическое построение строки.

Эта программа производит вывод, аналогичный следующему:

```
SizeOf: 4
Length: 3
StringElementSize: 2
StringRefCount: 1
StringCodePage: 1200
Is Unicode
Size in bytes: 6
ByteLength: 6
```

Кодовая страница, возвращаемая UnicodeString, равна 1200, номеру, сохраненному в глобальной переменной DefaultUnicodeCodePage. В приведенном выше коде (и его выводе) отчетливо видна разница между размером строковой переменной (обязательно 4), логической длиной и физической длиной в байтах.

Ее можно получить умножением размера в байтах каждого символа на количество символов или вызовом ByteLength. Последняя функция, однако, не поддерживает некоторые из строковых типов старого компилятора рабочего стола.

Глядя на строки в памяти

Возможность просмотра метаданных строки может быть использована для лучшего понимания того, как работает управление строковой памятью, особенно в отношении подсчета ссылок. Для этого я добавил еще немного кода в проект приложения StringMetaTest.

Программа имеет две глобальные строки: `MyStr1` и `MyStr2`. Программа присваивает динамическую строку первой из двух переменных (по причине, описанной ранее в примечании), а затем присваивает вторую переменную первой:

```
MyStr1 := string.Create(['H', 'e', 'l', 'l', 'o']);
```

```
  MyStr2 := MyStr1;
```

Кроме работы со строками, программа отображает их внутренний статус, используя следующую функцию `StringStatus`:

```
function StringStatus (const Str: string): string;
begin
  Result := 'Addr: ' +  IntToStr (Integer (Str)) +
    ', Len: ' + IntToStr (Length (Str)) +
    ', Ref: ' + IntToStr (PInteger (Integer (Str) - 8)^) +
    ', Val: ' + Str;
end;
```

В функции `StringStatus` важно передать строковый параметр в качестве параметра `const`. Передача этого параметра по копированию приведет к побочному эффекту наличия одной дополнительной ссылки на строку во время выполнения функции. Напротив, передача параметра по ссылке (`var`) или константе (`const`) не подразумевает дальнейшую ссылку на строку. В данном случае я использовал параметр `const`, так как функция не должна модифицировать строку.

Чтобы получить адрес памяти строки (полезно определить ее реальную идентичность и увидеть, когда две разные строки ссылаются на одну и ту же область памяти), я просто сделал жестко закодированное преобразование из строкового типа в целочисленный. Строки — это ссылки на практике, это указатели: Их значение содержит реальную область памяти строки, а не саму строку.

Ниже приведен код, используемый для проверки того, что происходит со строкой:

```
 Show ('MyStr1 - ' + StringStatus (MyStr1));
  Show ('MyStr2 - ' + StringStatus (MyStr2));
  MyStr1 [1] := 'a';
  Show ('Change 2nd char');
  Show ('MyStr1 - ' + StringStatus (MyStr1));
  Show ('MyStr2 - ' + StringStatus (MyStr2));
```

Изначально вы должны получить две строки с одинаковым содержанием, одинаковой ячейкой памяти и количеством ссылок 2.

```
MyStr1 - Addr: 51837036, Len: 5, Ref: 2, Val: Hello
MyStr2 - Addr: 51837036, Len: 5, Ref: 2, Val: Hello
```

По мере того, как приложение изменяет значение одной из двух строк (неважно какой из них), изменяется место в памяти обновленной строки. Таков эффект от техники копирования на запись. Это вторая часть вывода:

```
Change 2nd char
MyStr1 - Addr: 51848300, Len: 5, Ref: 1, Val: Hallo
MyStr2 - Addr: 51837036, Len: 5, Ref: 1, Val: Hello
```

Вы можете свободно расширить этот пример и использовать функцию `StringStatus` для изучения поведения длинных строк во многих других обстоятельствах, с множественными ссылками, когда они передаются в качестве параметров, присваиваются локальным переменным и многое другое.

Строки и кодирование

Как мы видели, строковый тип в Object Pascal отображен в формат Unicode UTF-16, с 2 байтами на элемент и управлением суррогатными парами для кодовых точек вне BMP (Basic Multi-language Plane - базовая многоязычная плоскость).

Однако есть много случаев, когда необходимо сохранить в файл, загрузить из файла, передать через сокет-соединение или получить текстовые данные из соединения, которое использует другое представление, например, ANSI или UTF-8.

Для конвертирования файлов и данных в памяти между различными форматами (или кодировками), Object Pascal RTL имеет удобный класс TEncoding, определенный в модуле System.SysUtils вместе с несколькими унаследованными классами.

примечание В Object Pascal RTL есть еще несколько удобных классов, которые можно использовать для чтения и записи данных в текстовых форматах. Например, классы `TStreamReader` и `TStreamWriter` обеспечивают поддержку текстовых файлов любой кодировки. Эти классы будут представлены в Главе 18.

Хотя я до сих пор не рассказал про классы и наследование, этот набор классов кодировок очень прост в использовании, так как для каждой кодировки уже есть глобальный объект, автоматически созданный для вас.

Другими словами, объект каждого из этих классов кодирования доступен в классе `TEncoding`, как свойство класса:

```
type
  TEncoding = class
    ...
  public
    class property ASCII: TEncoding read GetASCII;
    class property BigEndianUnicode: TEncoding
      read GetBigEndianUnicode;
    class property Default: TEncoding read GetDefault;
    class property Unicode: TEncoding read GetUnicode;
    class property UTF7: TEncoding read GetUTF7;
    class property UTF8: TEncoding read GetUTF8;
```

примечание Юникод-кодировка основана на классе `TUnicodeEncoding`, который использует тот же формат UTF-16 LE (Little Endian), который используется строковым типом. Вместо этого BigEndianUnicode использует менее распространенное представление BigEndian. Если вы не знакомы с порядком байтов ("Endianness") — это термины, используемые для обозначения последовательности из двух байтов, составляющих кодовую точку (или любую другую структуру данных). Little Endian (от младшего к старшему)имеет более значимый байт первым, а у Big Endian (от старшего к младшему) – более значимый байт последний. Дополнительную информацию можно найти на сайте en.wikipedia.org/wiki/Endianness.

Опять же исследовать эти классы в целом немного сложно на данном этапе книги, вместо того, давайте сосредоточимся на нескольких практических примерах. В классе `TEncoding` есть методы чтения и записи строк Unicode в массивы байт, выполняющие соответствующие преобразования.

Чтобы продемонстрировать преобразование формата UTF с помощью классов TEncoding, а также чтобы мой пример был простым и четким, и чтобы избежать работы с файловой

системой, в проекте приложения EncodingsTest я создал в памяти строку UTF-8, используя некоторые специфические данные, и преобразовал ее в UTF-16 с помощью одного вызова функции:

```
var
  Utf8string: TBytes;
  Utf16string: string;
begin
  // process Utf8data
  SetLength (Utf8string, 3);
  Utf8string[0] := Ord ('a'); // single byte ANSI char < 128
  Utf8string[1] := $c9; // double byte, reversed latin a
  Utf8string[2] := $90;
  Utf16string := TEncoding.UTF8.GetString(Utf8string);
  Show ('Unicode: ' + Utf16string);
```

Должно дать на выходе:

```
Unicode: aɐ
```

Теперь, чтобы лучше понять преобразование и разницу в представлениях, я добавил следующий код:

```
 Show ('Utf8 bytes:');
  for AByte in Utf8String do
    Show (AByte.ToString);

  Show ('Utf16 bytes:');
  UniBytes := TEncoding.Unicode.GetBytes (Utf16string);
  for AByte in UniBytes do
    Show (AByte.ToString);
```

Этот код создает дамп памяти с десятичными значениями для двух представлений строки, UTF-8 (один байт и две байтовые точки кода) и UTF-16 (обе точки кода - 2 байта):

```
Utf8 bytes:
97
201
144
Utf16 bytes:
97
0
80
2
```

Обратите внимание, что прямое преобразование символов в байты, для UTF-8, работает только для символов ANSI-7, то есть значений до 127. Для символов ANSI более высокого уровня

прямое преобразование отсутствует, и вы должны выполнить преобразование, используя специфическую кодировку (которая, однако, не будет работать на многобайтовых элементах UTF-8). Таким образом, оба нижеследующих вызова выдают неверный результат:

```
 // error: cannot use char > 128
  Utf8string[0] := Ord ('à');
  Utf16string := TEncoding.UTF8.GetString(Utf8string);
  Show ('Wrong high ANSI: ' + Utf16string);
  // try different conversion
  Utf16string := TEncoding.ANSI.GetString(Utf8string);
  Show ('Wrong double byte: ' + Utf16string);

  // output
  Wrong high ANSI:
  Wrong double byte: àÉ
```

Классы кодирования позволяют делать конвертацию в обоих направлениях, поэтому в данном случае я конвертирую из UTF-16 в UTF-8, выполняя некоторую обработку строки UTF-8 (это нужно делать с осторожностью, учитывая природу переменной длины этого формата), и конвертирую обратно в UTF-16:

```
var
  Utf8string: TBytes;
  Utf16string: string;
  I: Integer;
begin
  Utf16string := 'This is my nice string with à and Æ';
  Show ('Initial: ' + Utf16string);

  Utf8string := TEncoding.UTF8.GetBytes(Utf16string);
  for I := 0 to High(Utf8string) do
    if Utf8string[I] = Ord('i') then
      Utf8string[I] := Ord('I');
  Utf16string := TEncoding.UTF8.GetString(Utf8string);
  Show ('Final: ' + Utf16string);
```

Выход:

```
Initial: This is my nice string with à and Æ
Final: ThIs Is my nIce strIng wIth à and Æ
```

Другие типы для строк

В то время как строковый тип данных на сегодняшний день является наиболее распространенным и широко используемым типом для представления строк, компиляторы Object Pascal для рабочего стола имели и до сих пор имеют множество строковых типов. Некоторые из этих типов могут быть использованы и в мобильных приложениях, где также можно просто использовать `TBytes` непосредственно для манипулирования строкой с 1-байтовым представлением, как в проекте приложения, описанном в последнем разделе.

В то время как разработчики, использовавшие Object Pascal в прошлом, могли написать много кода, основанного на этих до-Unicode типах (или непосредственно управляющим работой с UTF-8), современные приложения реально требуют полной поддержки Unicode. Кроме того, хотя некоторые типы, такие как UTF8String, доступны в языке, их поддержка в терминах RTL ограничена. Рекомендуется использовать обычные и стандартные Unicode строки.

примечание В мобильных компиляторах Object Pascal было много дискуссий и критики по поводу изначального отсутствия нативных типов, таких как AnsiString и UTF8String. В Delphi 10.1 Berlin тип UTF8String и низкоуровневый RawByteString были официально введены вновь, а позже в Delphi 10.4 также и на мобильных устройствах были включены все десктопные строковые типы. До сих пор стоит учесть, что практически нет другого языка программирования, в котором было бы больше одного нативного или собственного строкового типа. Многочисленные строковые типы сложнее осваивать, могут вызывать нежелательные побочные эффекты (например, частые вызовы автоматического преобразования, замедляющие работу программ), а также дорого обходятся в обслуживании нескольких версий всех функций управления и обработки строк. Поэтому рекомендуется, кроме особых случаев, сосредоточиться на стандартном строковом типе, или `UnicodeString`.

Тип UCS4String

Интересным, но мало используемым строковым типом является тип UCS4String, доступный на всех компиляторах. Это просто UTF32 представление строки, и ни что другое, как массив элементов UTF32Char, или 4-байтных символов. Причина появления этого типа, как уже упоминалось ранее, заключается в том, что он предлагает непосредственное представление всех точек кода Юникода. Очевидным недостатком является то, что такая строка занимает в два раза больше памяти, чем строка UTF-16 (которая уже занимает в два раза больше, чем строка ANSI).

Хотя этот тип данных может использоваться в конкретных ситуациях, он не особенно подходит для общих обстоятельств. Кроме того, данный тип не поддерживает копирование на запись и не имеет реальных системных функций и процедур для его обработки.

примечание В то время как UCS4String гарантирует один UTF32Char на точку кода Юникода, он не может гарантировать один UTF32Char на графем, или "визуальный символ".

Старые типы строк

Как уже упоминалось, компилятор Object Pascal предлагает поддержку некоторых старых, традиционных типов строк (они доступны на всех целевых платформах, начиная с Delphi 10.4). Эти старые строковые типы включают в себя:

Тип ShortString, который соответствует оригинальному строковому типу языка Pascal. Эти строки имеют ограничение в 255 символов. Каждый элемент короткой строки имеет тип `ANSIChar`.

Тип ANSIString, который соответствует строкам переменной длины. Эти строки выделяются динамически, отсчитываются по ссылкам и

используют технику копирования на запись. Размер этих строк *практически* неограничен (в них может храниться до двух миллиардов символов!). Также этот тип строк основан на типе `ANSIChar`. Также доступен на мобильных компиляторах, даже если ANSI представление специфично для Windows и некоторые специальные символы могут обрабатываться по-разному в зависимости от платформы.

Тип WideString с точки зрения представления похож на 2-байтовую строку Unicode, основан на типе `Char`, но в отличие от стандартного строкового типа не использует копирование на запись и менее эффективен с точки зрения выделения памяти. Если вас интересует, почему он был добавлен в язык, то причина в совместимости со строковым управлением в COM-архитектуре Microsoft.

UTF8String - строка, основанная на формате переменной длины символов UTF-8. Как я уже упоминал, поддержка библиотеки исполнения для этого типа невелика.

RawByteString - это массив символов без набора кодовой страницы, на котором система никогда не выполняет преобразование символов (что логически напоминает структуру `TBytes`, но позволяет выполнять некоторые прямые строковые операции, которых в настоящее время не хватает массиву байт). Этот тип данных следует использовать редко и вне библиотек.

Механизм построения строк, позволяющий определить 1-байтовую строку, связанную с определенной кодовой страницей ISO, остаток до-юникодного прошлого.

> Опять же, все эти типы строк могут быть использованы в компиляторах для desktop-платформ, но доступны только для обратной совместимости. Поставьте цель использовать Unicode, TEncoding и другие современные методы управления строками всюду, когда это возможно.

Часть II: ООП в Object Pascal

Многие современные языки программирования поддерживают ту или иную парадигму объектно-ориентированного программирования (ООП). Многие из них используют программирование на классах, основанное на трех фундаментальных понятиях:

Классы, типы данных с публичным интерфейсом и частной структурой данных, реализующие инкапсуляцию; экземпляры этих типов данных обычно называют объектами,

Расширяемость класса или наследование — это возможность расширить тип данных с помощью новых функций без изменения исходного,

Полиморфизм или поздняя привязка, то есть возможность ссылаться на объекты различных классов с единым интерфейсом, и при этом работать с объектами так, как это определено их специфическим типом.

примечание Другие языки, такие как IO, JavaScript, Lua и Rebol, используют объектно-ориентированную парадигму, основанную на прототипах, где объекты могут быть созданы из других объектов, а не из класса, в зависимости от того, как создается объект. Они действительно предоставляют форму наследования, но от другого объекта, а не от класса, и динамическую типизацию, которая может быть

использована для реализации полиморфизма, пусть даже и совсем другим способом.

Вы можете писать приложения Object Pascal, даже не зная многого об объектно-ориентированном программировании. По мере того, как вы создаете новую форму, добавляете новые компоненты и обрабатываете события, IDE автоматически подготавливает для вас большую часть соответствующего кода. Но знание деталей языка и его реализации поможет вам точно понять, что делает система, и позволит вам полностью освоить язык.

Вы также сможете создавать сложные архитектуры в своих приложениях и даже целые библиотеки, а также использовать и расширять компоненты, поставляемые со средой разработки.

Вторая часть книги посвящена основным методам объектно-ориентированного программирования (ООП). Целью этой части книги является как обучение фундаментальным концепциям ООП, так и подробное описание того, как Object Pascal их реализует, сравнивая их с другими подобными объектно-ориентированными языками.

Резюме части II

Глава 7: Объекты

Глава 8: Наследование

Глава 9: Обработка исключений

Глава 10: Свойства и события

Глава 11: Интерфейсы

Глава 12: Манипулирование классами

296- начало

07: Объекты

Даже если вы не обладаете детальными знаниями объектно-ориентированного программирования (ООП), в этой главе будет представлена каждая из ключевых концепций. Если вы уже свободно владеете ООП, вы, вероятно, сможете относительно быстро пройтись по материалу и сосредоточиться на специфике языка Object Pascal, по сравнению с другими языками, которые вы, возможно, уже знаете.

Поддержка ООП в Object Pascal имеет много общего с такими языками, как C# и Java, а также имеет некоторое сходство с C++ и другими статическими и сильно типизированными языками. Вместо этого динамические языки, как правило, предлагают отличную интерпретацию ООП, так как они более свободно и гибко подходят к системе типов.

примечание Большое концептуальное сходство между C# и Object Pascal связано с тем, что эти два языка имеют одного и того же дизайнера, Андерса Хейлсберга. Он был первоначальным автором компиляторов Turbo Pascal, первой версии Delphi Object Pascal, а позже перешел в Microsoft и разработал C# (а совсем недавно JavaScript производную TypeScript). Подробнее об истории языка Object Pascal можно прочитать в Приложении А.

Представляем классы и

объекты

Класс и *объект* — это два термина, широко используемых в Object Pascal и других ООП языках. Однако, поскольку они часто используются не по назначению, давайте с самого начала будем уверены, что мы согласились с их определениями:

Класс — это пользовательский тип данных, который определяет состояние (или представление) и некоторые операции (или поведение). Другими словами, класс имеет некоторые внутренние данные и некоторые методы в виде процедур или функций. Класс обычно описывает характеристики и поведение ряда похожих объектов, хотя существуют классы специального назначения, которые предназначены для одного объекта.

Объект — это экземпляр класса, то есть переменная типа данных, определяемого классом. Объекты являются *реальными* сущностями. При запуске программы объекты занимают некоторое количество памяти для своего внутреннего представления.

Отношение между объектом и классом такое же, как и между любой другой переменной и ее типом данных. Только в этом случае переменные имеют специальное имя.

история Терминология ООП восходит к первым нескольким языкам, принявшим эту модель, таким как Smalltalk. Однако большая часть первоначальной терминологии, была позже изменена в пользу терминов, используемых в процедурных языках. Так что, хотя такие термины, как классы и объекты по-прежнему широко используются, вы, как правило, слышите термин вызвать метод чаще, чем оригинальный термин, послать сообщение получателю (объекту). Полное и подробное руководство по жаргону ООП и тому, как он развивался с течением времени, могло бы быть интересным, но заняло бы слишком много места в этой книге.

Определение класса

В Object Pascal вы можете использовать следующий синтаксис для определения нового типа данных класса (`TDate`), с некоторыми локальными полями данных (`Month`, `Day`, `Year`) и некоторыми методами (`SetValue`, `LeapYear`):

```
type
  TDate = class
    FMonth, FDay, FYear: Integer;
    procedure SetValue (M, D, Y: Integer);
    function LeapYear: Boolean;
  end;
```

примечание Мы уже видели подобную структуру для записей, которые по определению очень похожи на классы. Существуют различия в управлении памятью и другими областями, о чем мы поговорим позже в этой главе. Исторически, однако, в Object Pascal этот синтаксис сначала был принят для классов, а затем перенесен обратно на записи.

В Object Pascal договорились использовать букву *T в* качестве префикса для названия каждого класса, который вы пишете, как и для любого другого типа (на самом деле *T* означает *Type*). Это просто условность – для компилятора, *T* — это просто буква, как и любая другая - но она настолько распространена, что следование ей сделает ваш код более понятным для других программистов.

В отличие от других языков, определение класса в Object Pascal не включает в себя фактическую реализацию (или определение) методов, а только их сигнатуру (или объявление). Это делает код класса более компактным и значительно более читабельным.

совет Несмотря на то, что может показаться, что переход к фактической реализации метода более трудоемкий, редактор в среде позволяет использовать комбинацию клавиш Shift+Up и Shift+Down для перехода от объявлений метода к его реализации и обратно. Более того, с помощью Class Completion (нажатие клавиш Ctrl+C, когда курсор находится в пределах определения класса) редактор может сгенерировать скелет определения методов после написания определения класса.

Также имейте в виду, что кроме написания определения класса (с его полями и методами) можно написать и декларацию. При этом используется только имя класса:

```
type
  TMyDate = class;
```

Причина такого заявления заключается в том, что вам, возможно, понадобится два класса, ссылающихся друг на друга. В Object Pascal вы не можете использовать символ до тех пор, пока он не будет определен, чтобы сослаться на класс, который пока не определен, вам нужно объявление. Я написал следующий фрагмент кода только для того, чтобы показать вам синтаксис, а не для того, чтобы он имел какой-то смысл:

```
type
  THusband = class;

  TWife = class
    FHusband: THusband;
  end;

  THusband = class
    FWife: TWife;
  end;
```

В реальном коде вы столкнетесь с подобными перекрестными ссылками, поэтому этот синтаксис важно иметь в виду. Обратите внимание, что, как и для методов, класс, объявленный в юните, должен быть полностью определен в нем же.

Классы в других ООП-языках

Для сравнения, это класс `TDate`, написанный на C# и на Java (которые в данном упрощенном случае оказываются одинаковыми), использующий более подходящий набор правил именования, при этом код методов опущен:

```
// C# and Java language
```

```
class Date
{
  int   month;
  int   day;
  int   year;

  void setValue (int m, int d, int y)
  {
    // code
  }

  bool leapYear()
  {
    // code
  }
}
```

В Java и C# код методов входит в определение класса, а в Object Pascal методы, объявленные в классе, должны быть полностью определены в части реализации того же самого модуля, который включает определение класса. Другими словами, в Object Pascal класс всегда полностью определяется в одном модуле (в то время как модуль может, конечно, содержать несколько классов). Напротив, если в C++ методы реализованы отдельно, как в Object Pascal, то заголовочный файл, содержащий определение класса, не имеет строгого соответствия с кодом метода в реализационном файле. Соответствующий класс C++ будет иметь вид:

```
// C++ language

class Date
{
  int   month;
  int   day;
  int   year;

  void setValue (int m, int d, int y);
  BOOL leapYear();
}
```

Методы и классы

Как и в случае с записями, при определении кода метода необходимо указать, к какому классу он относится (в данном примере к классу `TDate`), используя имя класса в качестве префикса и точечную нотацию, как в следующем коде:

```
procedure TDate.SetValue(M, D, Y: Integer);
begin
  FMonth := M;
  FDay := D;
  FYear := Y;
end;

function TDate.LeapYear: Boolean;
begin
  // call IsLeapYear in SysUtils.pas
  Result := IsLeapYear (FYear);
end;
```

В отличие от большинства других ООП-языков, которые определяют методы как функции, Object Pascal привносит основное различие между процедурами и функциями, в зависимости от наличия возвращаемого значения, также и для методов. В языке Си++ это не так, как в случае реализации отдельно определенного метода:

```
// C++ method
void Date::setValue(int m, int d, int y)
{
  month = m;
  day = d;
  year = y;
};
```

Создание объекта

После этого сравнения с другими популярными языками, давайте вернемся к Object Pascal, чтобы посмотреть, как можно использовать класс. После того, как класс определен, мы можем создать объект такого типа и использовать его как в

следующем фрагменте кода (из проекта приложения Dates1, как и весь код в этом разделе):

```
var
  ADay: TDate;
begin
  // create
  ADay := TDate.Create;
  // use
  ADay.SetValue (1, 1, 2020);
  if ADay.LeapYear then
    Show ('Leap year: ' + ADay.Year.ToString);
```

В использованной нотации нет ничего необычного, но она очень мощная. Мы можем написать сложную функцию (например, `LeapYear`), а затем получить доступ к ее значению для каждого объекта `TDate`, как если бы это был примитивный тип данных. Обратите внимание, что `ADay.LeapYear` - это выражение, похожее на `ADay.Year`, хотя первое является вызовом функции, а второе - прямым доступом к данным. Как мы увидим в Гл. 10, нотация, используемая Object Pascal для доступа к свойствам, снова та же самая.

примечание Вызовы методов без параметров в большинстве языков программирования, основанных на синтаксисе языка Си, требуют наличия скобок, как в `ADay.LeapYear()`. Этот синтаксис легален и в Object Pascal, но используется редко. Методы без параметров обычно вызываются без скобок. Это очень отличается от многих языков, в которых ссылка на функцию или метод без скобок возвращает адрес функции. Как мы видели в разделе "Процедурные типы" в главе 4, Object Pascal использует одну и ту же нотацию для вызова функции или чтения ее адреса, в зависимости от контекста выражения.

Вывод фрагмента кода выше довольно тривиален:

```
Leap year: 2020
```

Опять же, позвольте мне сравнить создание объекта с аналогичным кодом, написанным на других языках программирования:

```
// C# and Java languages (object reference model)
Date aDay = new Date();

// C++ language (two alternative styles)
Date aDay; // local allocation
Date* aDay = new Date(); // "manual" reference
```

Объектная модель

В некоторых ООП-языках, таких как C++, объявление переменной типа класса создает экземпляр этого класса (более или менее похожее на то, что происходит с записями в Object Pascal). Память для локального объекта берется из стека и освобождается при завершении работы функции. В большинстве случаев, однако, приходится явно использовать указатели и ссылки, чтобы иметь большую гибкость в управлении временем жизни объекта, добавляя много дополнительной сложности.

Вместо этого язык Object Pascal основан на *объектной эталонной модели*, в точности как Java или C#. Идея заключается в том, что каждая переменная типа класса не содержит фактического значения объекта с его данными (например, для хранения дня, месяца и года). Вместо этого, она содержит только ссылку, или *указатель,* для указания места в памяти, где хранятся фактические данные объекта.

примечание На мой взгляд, принятие объектной эталонной модели было одним из лучших проектных решений, принятых командой компиляторов в первые годы существования языка, когда эта модель не была так распространена в языках программирования (на самом деле, в то время Java не было и C# не существовало).

Поэтому на этих языках необходимо явно создавать объект и присваивать его переменной, так как объекты не инициализируются автоматически. Другими словами, когда вы объявляете переменную, вы не создаете объект в памяти, а только резервируете место в памяти для ссылки на объект. Экземпляры объектов должны создаваться вручную и явно, по крайней мере, для объектов определенных вами классов. (В Object Pascal, однако, экземпляры компонентов, которые вы

размещаете на форме, создаются автоматически библиотекой времени выполнения).

В Object Pascal для создания экземпляра объекта можно вызвать его специальный метод `Create`, который является конструктором или другим пользовательским конструктором, определенным самим классом. Вот опять код:

```
ADay := TDate.Create;
```

Как видно, конструктор применяется к классу (типу), а не к объекту (переменной). Это потому, что вы просите класс создать новый экземпляр своего типа, и в результате вы получите новый объект, который вы обычно присваиваете переменной.

Откуда берется метод `Create`? Это конструктор класса TObject, от которого наследуют все остальные классы (тема, рассмотренная в следующей главе). Однако, как мы увидим позже в этой главе, очень распространено добавлять пользовательские конструкторы к своим классам.

Утилизация объектов

В языках, использующих объектную модель ссылок, вам нужен способ создания объекта перед его использованием, а также средство освобождения памяти, которую он занимает, когда в нем больше нет необходимости. Если вы не избавитесь от него, то в конечном итоге заполните память объектами, которые вам больше не нужны, что вызовет проблему, известную как утечка памяти. Для решения этой проблемы такие языки как C# и Java, основанные на виртуальной среде исполнения (или виртуальной машине), используют сборку мусора. Хотя это облегчает жизнь разработчика, однако этот подход подвержен некоторым сложным проблемам, связанным с

производительностью, которые на самом деле не имеют отношения к рассказу про Object Pascal. Так что, как бы ни были интересны эти вопросы, я не хочу вдаваться в них здесь.

В Object Pascal вы обычно освобождаете память объекта, вызывая его специальный метод Free (опять же, метод `TObject`, доступный в каждом классе). `Free` удаляет объект из памяти после вызова его деструктора (который может иметь специальный код очистки). Таким образом, можно завершить приведенный выше фрагмент кода как:

```
var
  ADay: TDate;
begin
  // create
  ADay := Tdate.Create;

  // use (code missing)

  // free the memory
  ADay.Free;
end;
```

В то время как это стандартный подход, библиотека компонентов добавляет такие понятия, как владение объектом, чтобы значительно уменьшить влияние ручного управления памятью, что делает этот вопрос относительно простым для решения.

примечание Как мы увидим позже, при использовании интерфейсов, ссылающихся на объекты, компилятор использует форму управления памятью Automatic Reference Counting (ARC). В течение нескольких лет она также использовалась для обычных переменных типа класса в мобильных компиляторах Delphi. Начиная с версии 10.4 Sydney, модель управления памятью была унифицирована, и использует классическое, настольное управление памятью Delphi для всех целевых платформ.

В управлении памятью есть гораздо больше, что вам нужно знать, но учитывая, что это очень важная тема, но не простая, я решил предложить здесь лишь краткое введение, а также полную главу, посвященную этой теме, а именно Главу 13. В этой главе я подробно покажу вам различные техники, которые вы можете использовать.

Что такое "nil"?

Как я уже упоминал, переменная может ссылаться на объект данного класса. Но она может быть еще не инициализирована, или объект, на который она ссылалась, может быть уже недоступен. Здесь можно использовать nil. Это константа, указывающая на то, что переменная не назначена ни одному объекту (или назначена `0` ячейке памяти). Другой язык программирования использует символ `null` для выражения того же самого понятия.

Если переменная типа класса не имеет значения, ее можно инициализировать таким образом:

```
ADay := nil;
```

Чтобы проверить, присвоена ли переменная объекту, можно записать одно из следующих выражений:

```
if ADay <> nil then ...
if Assigned (ADay) then ...
```

Не делайте ошибку присвоения объекту `nil`, чтобы удалить его из памяти. Установка ссылки на объект `nil` и его освобождение - две разные операции. Поэтому часто нужно освободить объект и установить его ссылку на `nil`, или вызвать процедуру специального назначения, которая выполняет обе операции одновременно, называемую FreeAndNil. Опять же, дополнительная информация и некоторые примеры будут приведены в Гл. 13, посвященной управлению памятью.

Отличие Record от Class в памяти

Как я уже упоминал ранее, одно из главных различий между записями и объектами связано с их моделью использования памяти. Переменные типа записи используют локальную память, по умолчанию они передаются в качестве параметров

функциям по значению, а при присваивании имеют поведение "копия по значению". Это контрастирует с переменными типа класса, которые выделяются на динамической куче памяти, передаются по ссылке и имеют поведение "копия по ссылке" при присваивании (т.е. копирование ссылки на один и тот же объект в памяти, а не реальных данных).

примечание Следствием такого различного управления памятью является отсутствие наследования записей и полиморфизмов - две особенности, на которых мы остановимся в следующей главе.

Спецификатор приватного доступа обозначает поля и методы класса, недоступные вне юнита (файла исходного кода), объявляющего этот класс.

Например, когда вы объявляете переменную записи на стеке, вы можете начать использовать ее сразу же, без необходимости вызывать ее конструктор (если только это не пользовательские управляемые записи). Это означает, что переменные записи более экономичны и эффективны на менеджере памяти, чем обычные объекты, так как они не участвуют в управлении динамической памятью. Это основные причины использования записей вместо объектов для небольших и простых структур данных.

В отношении разницы в способе передачи записей и объектов в качестве параметров отметим, что по умолчанию делается полная копия блока памяти, представляющего запись (включая все ее данные) или ссылку на объект (в то время как данные не копируются). Конечно, можно использовать параметры `var` или `const` записи, чтобы изменить поведение по умолчанию для передачи параметров типа записи, избегая любого копирования.

Private, Protected и Public

Строгий спецификатор приватного доступа обозначает поля и методы, недоступные вне любого метода этого класса, в том числе и методов других классов в том же самом модуле. Это соответствует поведению ключевого слова private в большинстве других ООП языков.

> Класс может иметь любое количество полей данных и любое количество методов. Однако для хорошего объектно-ориентированного подхода данные должны быть скрыты или *инкапсулированы* внутри используемого класса. Например, при обращении к дате нет смысла изменять значение дня самому. На самом деле, изменение значения дня может привести к недействительной дате, например, 30 февраля. Использование методов доступа к внутреннему представлению объекта ограничивает риск возникновения ошибочных ситуаций, так как методы могут проверить, является ли дата действительной, и отказаться изменять новое значение, если оно не является действительным. Надлежащая инкапсуляция особенно важна, так как она дает возможность программисту класса изменять внутреннее представление в будущей версии.
>
> Концепция инкапсуляции достаточно проста: просто подумайте о классе как о "черном ящике" с маленькой, видимой частью. Видимая часть, называемая *интерфейсом класса,* позволяет другим частям программы получать доступ и использовать объекты этого класса. Однако при использовании объектов большая часть их кода скрыта. Вы редко знаете, какими внутренними данными обладает объект, и обычно не имеете прямого доступа к данным. Скорее вы используете методы для доступа к данным объекта или действия над ним.

Инкапсуляция с использованием частных и защищенных членов является объектно-ориентированным решением классической цели программирования, известной как сокрытие информации.

Объект Pascal имеет пять основных спецификаторов доступа (или видимости): `private`, `protected`, и `public`. Шестой, `published`, будет обсуждаться в Главе 10. Вот пять основных:

Спецификатор публичного доступа public обозначает поля и методы, которые свободно доступны из любой другой части программы, а также в том блоке, в котором они определены.

Защищенный (protected) и строгий спецификатор защищенного доступа (strict protected) используется для указания методов и полей с ограниченной видимостью. Только текущий класс и его производные классы (или подклассы) могут иметь доступ к `protected` элементам, если они не находятся в одном классе или в любом случае, если указан модификатор `strict`. Мы еще раз обсудим это ключевое слово в разделе "Защищенные поля и инкапсуляция" в следующей главе.

Как правило, поля класса должны быть `private` или `strict private`; методы, как правило, являются общедоступными `public`. Однако, это не всегда так. Методы могут быть `private` или `protected`, если они необходимы только внутри класса для выполнения некоторых частичных операций. Поля могут быть `protected`, если вы уверены, что определение их типа не изменится, и вы, возможно, захотите манипулировать ими непосредственно в производных классах (как это объясняется в следующей главе), хотя это редко рекомендуется.

Как правило, вы должны неизменно избегать `public` полей, и, как правило, раскрывать какой-то прямой способ доступа к данным с помощью свойств, как мы увидим в подробностях в Гл. 10. Свойства являются расширением механизма

инкапсуляции других ООП языков и очень важны в Object Pascal.

Как уже упоминалось, спецификаторы `private` доступа ограничивают код вне юнита только доступом к определенным членам классов, объявленных в этом модуле. Это означает, что если два класса находятся в одном юните, то защита их приватных полей отсутствует, если только они не помечены как `strict private`, что, как правило, является хорошей идеей.

примечание В языке Си++ существует концепция friend классов, то есть классов, которым разрешен доступ к приватным данным другого класса. Следуя этой терминологии, можно сказать, что в Object Pascal все классы в одном модуле автоматически рассматриваются как friend классы-друзья.

Пример Private данных

В качестве примера использования этих спецификаторов доступа для реализации инкапсуляции рассмотрим новую версию класса `TDate`:

```
type
  TDate = class
  private
    Month, Day, Year: Integer;
  public
    procedure SetValue (M, D, Y: Integer);
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

В этой версии поля теперь объявлены `private`, и появились некоторые новые методы. Первый, `GetText`, это функция, которая возвращает строку с датой. Можно подумать о добавлении других функций, таких как `GetDay`, `GetMonth` и `GetYear`, которые просто возвращают соответствующие `private` данные, но аналогичные функции прямого доступа к данным не всегда нужны. Предоставление функций доступа для каждой области

может уменьшить инкапсуляцию, ослабить абстракцию и затруднить дальнейшую модификацию внутренней реализации класса. Функции доступа должны предоставляться только в том случае, если они являются частью логического интерфейса реализуемого класса, а не потому, что есть соответствующие поля.

Вторым новым методом является процедура `Increase`, которая увеличивает дату на один день. Это далеко не просто, так как необходимо учитывать разницу в продолжительности различных месяцев, а также високосные и не високосные годы. Для упрощения написания кода я изменю внутреннюю реализацию класса на использование типа Object Pascal TDateTime для внутренней реализации. Таким образом, фактический класс изменится на следующий код, который вы можете найти в проекте приложения Dates2:

```
type
  TDate = class
  private
    FDate: TDateTime;
  public
    procedure SetValue (M, D, Y: Integer);
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

Обратите внимание, что поскольку единственное изменение находится в `private` части класса, вам не придется изменять ни одну из существующих программ, которые его используют. В этом заключается преимущество инкапсуляции!

примечание В новой версии класса (единственное) поле имеет идентификатор, начинающийся с буквы "F". Это довольно распространенная конвенция в Object Pascal и та, которую я обычно использую в книге.

Завершая этот раздел, позвольте мне закончить описание проекта, приведя исходный код методов класса, которые полагаются на несколько системных функций для отображения дат во внутреннюю структуру и наоборот:

```
procedure TDate.SetValue (M, D, Y: Integer);
begin
  FDate := EncodeDate (Y, M, D);
end;

function TDate.GetText: string;
begin
  Result := DateToStr (FDate);
end;

procedure TDate.Increase;
begin
  FDate := FDate + 1;
end;

function TDate.LeapYear: Boolean;
begin
  // call IsLeapYear in SysUtils and YearOf in DateUtils
  Result := IsLeapYear (YearOf (FDate));
end;
```

Заметьте также, что код для использования класса больше не может ссылаться на значение года, а может только возвращать информацию об объекте даты, разрешенную его методами:

```
var
  ADay: TDate;
begin
  // create
  ADay := TDate.Create;

  // use
  ADay.SetValue (1, 1, 2020);
  ADay.Increase;

  if ADay.LeapYear then
    Show ('Leap year: ' + ADay.GetText);

  // free the memory (for non ARC platforms)
  ADay.Free;
```

Выход не сильно отличается от прежнего:

```
Leap year: 1/2/2020
```

Инкапсуляция и формы

Одной из ключевых идей инкапсуляции является уменьшение количества глобальных переменных, используемых

программой. Доступ к глобальной переменной можно получить из каждой части программы. По этой причине изменение глобальной переменной влияет на всю программу. С другой стороны, при изменении представления поля класса достаточно изменить код некоторых методов этого класса, использующих данное поле, и ничего больше. Поэтому можно сказать, что скрытие информации относится к *инкапсуляции изменений*.

Позвольте мне прояснить эту идею на практическом примере. Когда у вас есть программа с несколькими формами, вы можете сделать некоторые данные доступными для каждой формы, объявив ее глобальной переменной в интерфейсной части модуля формы:

```
var
  Form1: TForm1;
  NClicks: Integer;
```

Это работает, но имеет две проблемы. Во-первых, данные (`NClicks`) связаны не с конкретным экземпляром формы, а со всей программой. Если вы создадите две формы одного типа, то они будут совместно использовать данные. Если вы хотите, чтобы каждая форма одного и того же типа имела свою собственную копию данных, единственное решение - добавить ее в класс формы:

```
type
  TForm1 = class(TForm)
  public
    FNClicks: Integer;
  end;
```

Вторая проблема заключается в том, что если вы определите данные как глобальную переменную или как публичное поле формы, то в будущем вы не сможете модифицировать их реализацию, не затрагивая код, который использует данные. Например, если вам нужно только прочитать текущее значение из других форм, вы можете объявить данные приватными и предоставить метод для чтения значения:

```
type
  TForm1 = class(TForm)
    // components and event handlers here
  public
    function GetClicks: Integer;
  private
    FNClicks: Integer;
  end;

function TForm1.GetClicks: Integer;
begin
  Result := FNClicks;
end;
```

Еще лучшим решением будет добавление свойства в форму, как мы увидим в Гл. 10. Вы можете поэкспериментировать с этим кодом, открыв проект приложения ClickCount. Короче говоря, форма этого проекта имеет две кнопки и метку сверху, большая часть поверхности пуста, чтобы пользователь мог на нее нажать (или нажать). В этом случае счетчик увеличивается, а метка обновляется новым значением:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Single);
begin
  Inc (FNClicks);
  Label1.Text := FNClicks.ToString;
end;
```

Вы можете увидеть приложение в действии на рисунке 7.1. Форма проекта также имеет две кнопки, одна из которых предназначена для создания новой формы того же типа, а вторая - для ее закрытия (таким образом, вы можете вернуть фокус к предыдущей форме).

Это сделано для того, чтобы подчеркнуть, что различные экземпляры одного и того же типа формы имеют свой собственный подсчет кликов. Вот код двух методов:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewForm: TForm1;
begin
  NewForm := TForm1.Create(Application);
  NewForm.Show;
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
```

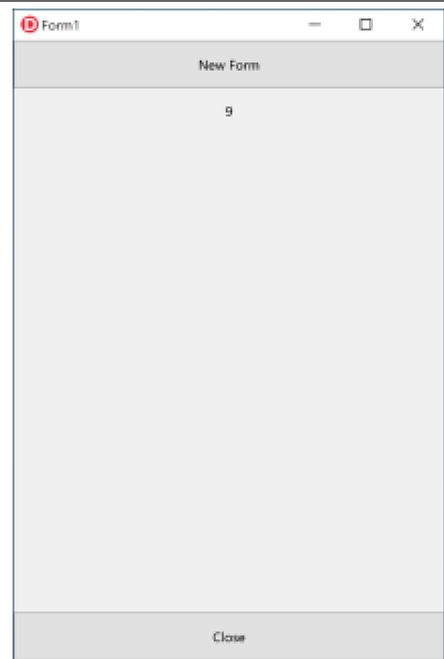


Рисунок 7.1: Форма проекта прикладной программы ClickCount, показывающая количество кликов или нажатий на форме (отслеживается с использованием данных частной формы)

Ключевое слово Self

Мы видели, что методы очень похожи на процедуры и функции. Реальная разница заключается в том, что методы имеют дополнительный, неявный параметр. Это ссылка на текущий объект, на объект, к которому применяется метод. Внутри метода можно сослаться на этот параметр - текущий объект - используя ключевое слово `Self`.

Этот дополнительный скрытый параметр необходим при создании нескольких объектов одного класса, так что каждый

раз при применении метода к одному из объектов, метод будет работать только с данными этих конкретных объектов и не затронет другие объекты того же класса.

примечание Мы уже видели роль ключевых слов `Self` в Главе 5, обсуждая записи. Концепция и ее реализация очень похожи. Опять же, исторически `Self` сначала был введен для классов, а затем расширен для записей, когда к этой структуре данных были добавлены методы.

Например, в методе `SetValue` класса `TDate`, из листинга ранее, мы просто используем `Month`, `Year`, and `Day`, чтобы ссылаться на поля текущего объекта, что вы можете выразить примерно как:

```
Self.FMonth := M;
Self.FDay := D;
```

На самом деле компилятор Object Pascal транслирует имеющийся код, а *не* так, как вы должны его писать. Ключевое слово Self — это фундаментальная конструкция языка, используемая компилятором, но иногда оно используется программистами для разрешения конфликтов имен и для того, чтобы сделать код более читабельным.

примечание Языки C++, Java, C# и JavaScript имеют схожую характеристику, основанную на ключевом слове `this`. Однако в JavaScript использование `this` в методе обращения к полям объектов является обязательным, в отличие от C++, C# и Java.

Все, что вам действительно нужно знать о `Self`, это то, что техническая реализация вызова метода отличается от реализации вызова к общей подпрограмме. Методы имеют дополнительный скрытый параметр `Self`. Поскольку все это происходит за кулисами, в данный момент вам не нужно знать, как работает `Self`.

Вторая важная вещь, которую необходимо знать, это то, что вы можете явно использовать `Self`, чтобы ссылаться на текущий объект в целом, например, передавая текущий объект в качестве параметра в другую функцию.

Динамическое создание компонентов

В качестве примера того, что я только что упомянул, ключевое слово Self часто используется, когда необходимо явно обратиться к текущей форме в одном из ее методов.

Типичным примером является создание компонента во время выполнения, где необходимо передать владельца компонента его конструктору `Create` и присвоить такое же значение его свойству `Parent`. В обоих случаях, вы должны указать объект текущей формы в качестве параметра или значения, и лучший способ сделать это - использовать ключевое слово `Self`.

примечание Владение компонентом указывает на связь жизненного цикла и управления памятью между двумя объектами. Когда владелец компонента освобождается, компонент также освобождается. Родительство относится к визуальному элементу, содержащему child элемент на своей поверхности.

Для демонстрации такого кода я создал проект приложения CreateComps. Это приложение имеет простую форму без компонентов и обработчик события `OnMouseDown`, который также получает в качестве параметра положение щелчка мыши. Мне нужна эта информация, чтобы создать компонент кнопки в этой позиции.

примечание Обработчик событий — это специальный метод, рассмотренный в Главе 10, и часть того же семейства обработчиков событий `OnClick` кнопок, которые мы уже использовали в этой книге.

Вот код метода:

```
procedure TForm1.FormMouseDown (Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  Btn: TButton;
begin
  Btn := TButton.Create (Self);
  Btn.Parent := Self;
  Btn.Position.X := X;
```

```
  Btn.Position.Y := Y;
  Btn.Height := 35;
  Btn.Width := 135;
  Btn.Text := Format ('At %d, %d', [X, Y]);
end;
```

Обратите внимание, что для компиляции этого обработчика событий вам может понадобиться добавить модуль `StdCtrls` в оператор uses.

Эффект от этого кода заключается в создании кнопок при щелчке мышью, с подписями, указывающими точное местоположение, как показано на рисунке 7.2. (Для этого проекта я отключил FMX Mobile Preview, чтобы показать кнопки Windows в родном стиле, поскольку это более наглядно). В коде выше, обратите внимание, в частности, на использование ключевого слова `Self`, в качестве параметра метода `Create` и в качестве значения свойства Parent.

При написании процедуры, подобной только что увиденному коду, может возникнуть соблазн использовать переменную `Form1 вместо Self`. В этом конкретном примере это изменение не будет иметь практического значения (хотя это и не будет хорошей практикой кодирования), но если существует несколько экземпляров формы, то использование `Form1` действительно будет ошибкой.

На самом деле, если переменная `Form1` относится к форме создаваемого типа (как правило, первой), и если вы создаете два экземпляра одного и того же типа формы, то при нажатии на любую следующую форму новая кнопка всегда будет отображаться в первой. Ее `Owner` и `Parent` будет `Form1`, а не та форма, на которую пользователь нажал.

В общем, написание метода, в котором вы ссылаетесь на конкретный экземпляр того же класса, когда требуется текущий объект, является действительно плохим стилем кодирования ООП.

Рисунок 7.2: Вывод примера проекта приложения CreateComps на мобильное устройство

Конструкторы

В приведенном выше коде для создания объекта класса (или выделения памяти для объекта) я вызвал метод `Create`. Это *конструктор*, специальный метод, который можно применить к классу для выделения памяти для нового экземпляра этого класса:

```
ADay := TDate.Create;
```

Созданный экземпляр возвращается конструктором и может быть присвоен переменной для хранения объекта и последующего использования. При создании объекта инициализируется его память. Все данные нового экземпляра

устанавливаются на ноль (или на `nil`, или на пустую строку, или на соответствующее значение "по умолчанию" для данного типа данных).

Если вы хотите, чтобы данные вашего экземпляра начинались с ненулевого значения (особенно, когда нулевое значение не имеет смысла по умолчанию), вам нужно написать пользовательский конструктор для этого. Новый конструктор может называться `Create`, или иметь любое другое имя. Что определяет его роль, так это не имя, а использование ключевого слова `constructor`.

примечание Другими словами, Object Pascal поддерживает именованные конструкторы, в то время как во многих ООП-языках конструктор должен быть назван в честь самого класса. С именованными конструкторами можно иметь более одного конструктора с одинаковыми параметрами (помимо перегрузки символа `Create` - перегрузка рассматривается в следующем разделе). Другая особенность языка, совершенно уникальная среди ООП-языков, заключается в том, что конструкторы также могут быть виртуальными. Последствия этой очень приятной особенности я покажу на нескольких примерах позже в книге после того, как в следующей главе расскажу о концепции виртуального метода.

Основной причиной добавления пользовательского конструктора в класс является инициализация его данных. Если вы создаете объекты без их инициализации, то последующий вызов методов может привести к странному поведению или даже ошибке во время выполнения. Вместо того, чтобы ждать появления этих ошибок, следует в первую очередь использовать превентивные методы, чтобы избежать их. Одной из таких методик является последовательное использование конструкторов для инициализации данных объектов. Например, после создания объекта необходимо вызвать процедуру `SetValue` класса `TDate`. В качестве альтернативы, мы можем предоставить пользовательский конструктор, который создает объект и присваивает ему начальное значение:

```
constructor TDate.Create;
begin
```

```
  FDate := Today;
end;

constructor TDate.CreateFromValues (M, D, Y: Integer);
begin
  FDate := SetValue (M, D, Y);
end;
```

Эти конструкторы можно использовать следующим образом, как я делал в проекте приложения Date3, в коде, прикрепленном к двум отдельным кнопкам:

```
ADay1 := TDate.Create;
ADay2 := TDate.CreateFromValues (12, 25, 2015);
```

Хотя в общем случае вы можете использовать любое имя для конструктора, имейте в виду, что если вы используете имя, отличное от `Create`, то конструктор `Create` базового класса TObject все равно будет доступен. Если Вы разрабатываете и распространяете код для использования другими, то программист, вызывающий этот конструктор по умолчанию `Create`, может обойти предоставленный Вами код инициализации. Определив конструктор `Create` с некоторыми параметрами (или ни одного, как в примере выше), вы заменяете определение по умолчанию на новое и делаете его использование обязательным.

Подобно тому, как класс может иметь пользовательский конструктор, он может иметь пользовательский деструктор - метод, объявленный с ключевым словом destructor и неизменно называемый Destroy. Этот метод-деструктор может выполнять некоторую очистку ресурса перед уничтожением объекта, но во многих случаях пользовательский деструктор не требуется.

Подобно тому, как вызов конструктора выделяет память для объекта, вызов деструктора освобождает память. Пользовательские деструкторы в реальности нужны только для объектов, которые приобретают ресурсы, такие как другой

объект, в своих конструкторах или в течение своего жизненного цикла.

В отличие от конструктора `Create` по умолчанию, деструктор `Destroy` по умолчанию является виртуальным, и настоятельно рекомендуется, чтобы разработчик переопределил этот виртуальный деструктор (виртуальные методы рассматриваются в следующей главе).

Это связано с тем, что вместо прямого вызова деструктора для освобождения объекта, хорошей практикой программирования Object Pascal является вызов специального метода Free класса `TObject`, который в свою очередь вызывает `Destroy` только в том случае, если объект существует, то есть, если он не `nil`. Таким образом, вы определяете деструктор с другим именем, он не будет вызываться `Free`. Опять же, больше об этом, когда мы сосредоточимся на управлении памятью в Гл. 13.

примечание Как описано в следующей главе, `Destroy` - это виртуальный метод. Вы можете заменить его базовое определение на новое в унаследованном классе, пометив его ключевым словом `override`. Кстати, наличие статического метода, вызывающего виртуальный, является очень распространенным стилем программирования, называемым *шаблоном*. В деструкторе, как правило, следует писать только код очистки ресурсов. Старайтесь избегать более сложных операций, которые могут привести к возникновению исключений или занять значительное количество времени, чтобы избежать проблем при очистке объектов, а также потому, что при завершении программы вызывается много деструкторов, и вы хотите, чтобы это происходило быстро.

Управление локальными данными класса с помощью конструкторов и деструкторов.

Хотя я расскажу более сложные сценарии позже в книге, здесь я хочу показать простой случай защиты ресурсов с помощью конструктора и деструктора. Это наиболее распространенный

сценарий использования деструктора. Предположим, что у вас есть класс со следующей структурой (также часть проекта приложения Date3):

```
type
  TPerson = class
  private
    FName: string;
    FBirthDate: TDate;
  public
    constructor Create (Name: string);
    destructor Destroy; override;
    // some actual methods
    function Info: string;
  end;
```

Этот класс имеет ссылку на другой, внутренний объект под названием `FBirthDate`. Когда создается экземпляр класса `TPerson`, этот внутренний (или дочерний) объект также должен быть создан, а когда экземпляр уничтожен, дочерний объект также должен быть утилизирован. Вот как можно написать код конструктора и переопределить деструктор, а также внутренний метод, который всегда может принять как должное, что внутренний объект существует:

```
constructor TPerson.Create (Name: string);
begin
  FName := Name;
  FBirthDate := TDate.Create;
end;

destructor TPerson.Destroy;
begin
  FBirthDate.Free;
  inherited;
end;

function TPerson.Info: string;
begin
  Result := FName + ': ' + FBirthDate.GetText;
end;
```

примечание Чтобы понять ключевое слово override, используемое для определения деструктора, и ключевое слово inherited в его определении, придется подождать до следующей главы. Пока достаточно сказать, что первое используется для указания на то, что класс имеет новое определение, заменяющее деструктор базового класса `Destroy`, в то время как второе используется для вызова деструктора этого

базового класса. Заметим также, что `override` используется в объявлении метода, но не в коде реализации метода.

Теперь вы можете использовать объект внешнего класса, как в следующем сценарии, и внутренний объект будет создан должным образом, когда объект `TPerson будет` создан и уничтожен вовремя, когда `TPerson` будет уничтожен:

```
var
  Person: TPerson;
begin
  Person := TPerson.Create ('John');
  // use the class and its internal object
  Show (Person.Info);
  Person.Free;
end;
```

Опять же, вы можете найти этот код в рамках проекта приложения Dates3.

Перегруженные методы и конструкторы

Object Pascal поддерживает перегруженные функции и методы: можно иметь несколько методов с одним и тем же именем, при условии, что параметры различны. Мы уже видели, как работает перегрузка для глобальных функций и процедур, для методов применяются одни и те же правила. Проверяя параметры, компилятор может определить, какую версию метода вы хотите вызвать.

Опять же, есть два основных правила перегрузки:

За каждой версией метода должно следовать ключевое слово `overload`.

Различия должны быть в количестве или типе параметров, или в обоих сразу. Напротив, тип возврата не может использоваться для различения двух методов.

Хотя перегрузка может быть применена ко всем методам класса, эта особенность особенно актуальна для конструкторов, так как мы можем иметь несколько конструкторов и вызывать их все `Create`, что делает их легкими для запоминания.

история Исторически сложилось так, что в C++ была добавлена перегрузка специально для того, чтобы разрешить использование нескольких конструкторов, поскольку они должны иметь одно и то же имя (имя класса). В Object Pascal эту особенность можно было бы считать ненужной просто потому, что несколько конструкторов могут иметь разные специфические имена, но она была добавлена в язык в любом случае, так как она также полезна во многих других сценариях.

В качестве примера перегрузки я добавил в класс `TDate` две различные версии метода `SetValue`:

```
type
  TDate = class
  public
    procedure SetValue (Month, Day, Year: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;

procedure TDate.SetValue (Month, Day, Year: Integer);
begin
  FDate := EncodeDate (Year, Month, Day);
end;

procedure TDate.SetValue(NewDate: TDateTime);
begin
  FDate := NewDate;
end;
```

После этого простого шага я добавил в класс два отдельных конструктора `Create`, один без параметров, который скрывает конструктор по умолчанию, а другой со значениями инициализации. Конструктор без параметров использует в качестве значения по умолчанию сегодняшнюю дату:

```
type
  TDate = class
  public
    constructor Create; overload;
    constructor Create (Month, Day, Year: Integer); overload;

constructor TDate.Create (Month, Day, Year: Integer);
begin
  FDate := EncodeDate (Year, Month, Day);
end;
```

```
constructor TDate.Create;
begin
  FDate := Date;
end;
```

Наличие этих двух конструкторов позволяет определить новый объект `TDate` двумя разными способами:

```
var
  Day1, Day2: TDate;
begin
  Day1 := TDate.Create (2020, 12, 25);
  Day2 := TDate.Create; // today
```

Данный код является частью проекта приложения Dates4.

Полный класс TDate

На протяжении всей главы я показывал вам куски и фрагменты исходного кода для разных версий класса `TDate`. Первая версия была основана на трех целых числах для хранения года, месяца и дня; вторая версия использовала поле типа `TDateTime`, предоставленное RTL. Вот полная часть интерфейса юнита, определяющая класс `TDate`:

```
unit Dates;

interface

type
  TDate = class
  private
    FDate: TDateTime;
  public
    constructor Create; overload;
    constructor Create (Month, Day, Year: Integer); overload;
    procedure SetValue (Month, Day, Year: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    procedure Increase (NumberOfDays: Integer = 1);
    procedure Decrease (NumberOfDays: Integer = 1);
    function GetText: string;
  end;
```

Цель новых методов `Increase` и `Decrease` (которые имеют значение по умолчанию для своего параметра) довольно проста

для понимания. При вызове без параметра они меняют значение даты на следующий или предыдущий день. Если параметр `NumberOfDays` является частью вызова, они добавляют или вычитают его численное значение:

```
procedure TDate.Increase (NumberOfDays: Integer = 1);
begin
  FDate := FDate + NumberOfDays;
end;
```

Метод `GetText` возвращает строку с отформатированной датой, используя функцию `DateToStr` для преобразования:

```
function TDate.GetText: string;
begin
  GetText := DateToStr (FDate);
end;
```

Большинство методов мы уже видели в предыдущих разделах, поэтому полный листинг я приводить не буду, его можно найти в коде проекта приложения ViewDate, который я написал для тестирования класса. Форма немного сложнее, чем другие в книге, и имеет поле для отображения даты и шесть кнопок, с помощью которых можно изменить значение объекта. Главную форму проекта приложения ViewDate во время выполнения можно увидеть на рисунке 7.3. Чтобы компонент Label выглядел красиво, я дал ему большой шрифт, сделал его таким же широким, как и форма, установил его свойство Alignment (Выравнивание) в `taCenter`, а свойство AutoSize (Автоматический размер) в `False`.

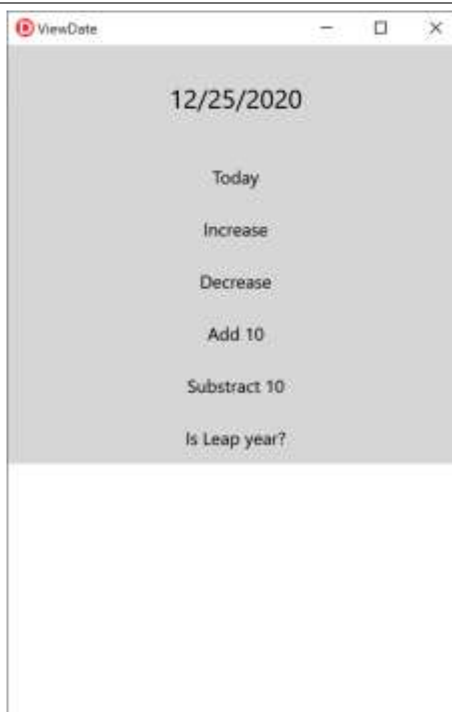


Рисунок 7.3:
Вывод приложения
ViewDate при запуске

Код, выполняемый при запуске этой программы, находится в обработчике события `OnCreate` формы. В соответствующем методе мы создаем экземпляр класса `TDate`, инициализируем этот объект, а затем отображаем его текстовое описание в свойстве `Text` метки, как показано на рисунке 7.3.

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  ADay := TDate.Create;
  LabelDate.Text := ADay.GetText;
end;
```

`ADay` является частным полем класса формы, `TDateForm`. Кстати, имя класса автоматически выбирается средой разработки при изменении свойства `Name` формы на `DateForm`.

Объект конкретной даты создается при создании формы (устанавливая ту же самую связь, которую мы видели ранее

между классом человека и его подобъектом даты), а затем уничтожается вместе с формой:

```
procedure TDateForm.FormDestroy(Sender: TObject);
begin
  ADay.Free;
end;
```

Когда пользователь нажимает одну из шести кнопок, нам нужно применить соответствующий метод к объекту `ADay`, а затем отобразить новое значение даты в метке:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
  ADay.SetValue (Today);
  LabelDate.Text := ADay.GetText;
end;
```

Альтернативный способ записи последнего метода - уничтожить текущий объект и создать новый:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
  ADay.Free;
  ADay := TDate.Create;
  LabelDate.Text := ADay.GetText;
end;
```

В данном конкретном случае, это не очень хороший подход (потому что создание нового объекта и уничтожение существующего влечет за собой большие временные затраты, когда все, что нам нужно - это изменить значение объекта), но это позволяет мне показать вам пару техник Object Pascal. Первое, что нужно заметить, это то, что мы уничтожаем предыдущий объект до назначения нового. Операция присваивания, на самом деле, заменяет ссылку, оставляя объект в памяти (даже если ни один указатель на него не ссылается). Когда вы присваиваете объект другому объекту, компилятор просто копирует ссылку на объект в памяти в ссылку нового объекта.

Одна из проблем заключается в том, как скопировать данные с одного объекта на другой. Этот случай очень прост, потому что есть только одно поле и способ его инициализации. В общем

случае, если вы хотите изменить данные внутри существующего объекта, вы должны скопировать каждое поле или предоставить специальный метод для копирования всех внутренних данных. В некоторых классах есть метод Assign, который выполняет эту операцию *глубокого копирования*.

примечание Точнее, в библиотеке исполнения все классы, наследуемые от `TPersistent`, имеют метод `Assign`, но большинство классов, наследуемых от `TComponent`, не реализуют его, что приводит к возникновению исключения при его вызове. Причина кроется в механизме потоковой передачи, поддерживаемом библиотеками времени исполнения, и поддержке свойств типов `TPersistent`, но на данном этапе это слишком сложно, чтобы вдаваться в подробности.

Вложенные типы и вложенные константы

Object Pascal позволяет объявить новые классы в секции интерфейса модуля, позволяя другим модулям программы ссылаться на них, или в секции реализации, где они доступны только из методов других классов того же модуля или из глобальных процедур, реализованных в этом модуле после определения класса.

Более свежим добавлением является возможность объявления класса (или любого другого типа данных) внутри другого класса. Как и любой другой член класса, вложенные типы могут иметь ограниченную видимость (скажем, приватную или защищенную). Релевантными примерами вложенных типов являются перечисления, используемые тем же классом и классы поддержки реализации.

Соответствующий синтаксис позволяет определить вложенную константу, постоянное значение, связанное с классом (опять-

таки используемое только внутри, если закрытое, или используемое остальной частью программы, если открытое). В качестве примера рассмотрим следующее объявление вложенного класса (извлеченное из модуля NestedClass прикладного проекта NestedTypes):

```
type
  TOne = class
  private
    FSomeData: Integer;
  public
    // nested constant
    const Foo = 12;
    // nested type
    type
      TInside = class
      public
        procedure InsideHello;
      private
        FMsg: string;
      end;
  public
    procedure Hello;
  end;

procedure TOne.Hello;
var
  Ins: TInside;
begin
  Ins := TInside.Create;
  Ins.FMsg := 'hi';
  Ins.InsideHello;
  Show ('Constant is ' + IntToStr (Foo));
  Ins.Free;
end;

procedure TOne.TInside.InsideHello;
begin
  FMsg := 'new msg';
  Show ('internal call');
  if not Assigned (InsIns) then
    InsIns := TInsideInside.Create;
  InsIns.Two;
end;

procedure TOne.TInside.TInsideInside.Two;
begin
  Show ('this is a method of a nested/nested class');
end;
```

Вложенный класс может использоваться непосредственно внутри класса (как показано в листинге) или вне класса (если он объявлен в публичном разделе), но с полным именем `TOne.TInside`. *Полное имя* класса используется также в определении метода вложенного класса, в данном случае `TOne.TInside`. Класс-хозяин может иметь поле типа вложенного класса сразу после объявления вложенного класса (как видно из кода прикладного проекта NestedClass).

Класс со вложенными классами используется следующим образом:

```
var
  One: TOne;
begin
  One := TOne.Create;
  One.Hello;
  One.Free;
```

В результате получается следующий вывод:

```
internal call
this is a method of a nested/nested class
constant is 12
```

Какая польза от использования вложенного класса на языке Object Pascal? Концепция широко используется в Java для реализации делегатов обработчиков событий и имеет смысл в C#, где вы не можете спрятать класс внутри модуля. В языке Object Pascal вложенные классы - это единственный способ получить поле типа другого приватного класса (или внутреннего класса), не добавляя его в глобальное пространство имен, что делает его глобально видимым.

Если внутренний класс используется только методом, то такого же эффекта можно добиться, объявив класс в части реализации модуля. Но если на внутренний класс есть ссылка из интерфейсной части модуля (например, потому что он используется для поля или параметра), то он должен быть объявлен в той же самой интерфейсной части и в конечном итоге будет виден. Фокус объявления такого поля общего или

базового типа с последующим приведением его к конкретному (приватному) типу гораздо менее понятен, чем использование вложенного класса.

примечание В главе 10 приведен практический пример, в котором пригодились вложенные классы, а именно реализация пользовательского итератора для цикла `for in`.

08: Наследование

Если главной причиной написания классов является инкапсуляция, то основной причиной использования наследования среди классов является гибкость. Объедините эти два понятия в то, что изначально было известно как "принцип открытого закрытия", и вы сможете иметь типы данных, которые вы можете использовать и которые не будут меняться, но давать возможность создания модифицированных версий этих типов:

> *"Программные сущности (классы, модули, функции и т.д.) должны быть открыты для расширения, но закрыты для модификации". (Бертран Мейер, "Объектно-ориентированная конструкция программного обеспечения", 1988 г.)*

Чистая правда, что наследование - это очень сильная связь, ведущая к тесно связанному коду, но это также правда, что она предлагает больше возможностей разработчику (и, да, тем больше ответственности, которая с ней приходит).

Однако вместо того, чтобы начинать дискуссию об этой функции, я хочу описать, как работает наследование типов и, в частности, как оно работает на языке Object Pascal.

Наследование от существующих типов

Нам часто нужно использовать немного другую версию существующего класса, который мы написали или который кто-то дал нам.

Например, вам может понадобиться добавить новый метод или слегка изменить существующий. Это легко сделать, модифицировав исходный код, если только вы не хотите иметь возможность использовать две разные версии класса при разных обстоятельствах. Также, если класс был изначально написан кем-то другим (и вы нашли его в библиотеке), вы можете захотеть сохранить ваши изменения отдельно.

Типичная альтернатива старой школы для двух одинаковых версий класса - сделать копию оригинального определения типа, изменить его код для поддержки новых возможностей и дать новое имя результирующему классу. Это может сработать, но это также может создать проблемы: дублируя код, вы также дублируете ошибки; когда ошибка исправлена в одной из копий кода, вы должны помнить, что применяли исправление к другой копии; и если вы хотите добавить новую возможность, вы должны добавить ее два или более раз, в зависимости от количества копий оригинального кода, которые вы сделали с течением времени. Даже если это может не замедлить вас, когда вы пишете код в первый раз, это катастрофа для обслуживания программного обеспечения. Более того, такой подход приводит к двум совершенно разным типам данных, поэтому компилятор не может помочь вам воспользоваться схожестью между этими двумя типами.

Для решения такого рода задач по выражению сходства между классами, Object Pascal позволяет определить новый класс непосредственно из существующего. Этот метод известен как *наследование* (или подкласс, или вывод типа) и является одним из фундаментальных элементов объектно-ориентированных языков программирования.

Чтобы наследовать от существующего класса, достаточно указать этот класс в начале объявления подкласса. Например, это делается автоматически каждый раз при создании новой формы:

```
type
  TForm1 = class(TForm)

  end;
```

Это простое определение указывает на то, что класс `TForm1` наследует все методы, поля, свойства и события класса `TForm`. К объекту типа `TForm1` можно применить любой публичный метод класса `TForm`. `TForm`, в свою очередь, наследует некоторые из своих методов от другого класса, и так далее, вплоть до класса `TObject` (который является базовым классом всех классов).

Для сравнения, C++, C# и Java могли бы использовать что-то вроде:

```
class Form1 : TForm
{
  ...
}
```

В качестве простого примера наследования можно немного изменить проект приложения ViewDate из последней главы, получив новый класс из `TDate` и модифицировав одну из его функций, `GetText`. Этот код можно найти в файле `DATES.PAS` прикладного проекта DerivedDates.

```
type
  TNewDate = class (TDate)
  public
     function GetText: string;
  end;
```

В данном примере `TNewDate` получен из `TDate`. Обычно говорят, что `TDate` является классом-предком или *базовым* классом, или *родительским* классом `TNewDate`, и что `TNewDate` является *подклассом,* классом-потомком или *дочерним* классом `TDate`.

Для реализации новой версии функции `GetText` я использовал функцию `FormatDateTime`, которая использует (помимо других функций) предопределенные имена месяцев. Вот метод `GetText`, где *'dddddd' означает* длинный формат данных:

```
function TNewDate.GetText: string;
begin
  Result := FormatDateTime ('dddddd', FDate);
end;
```

После того, как мы определили новый класс, нам нужно использовать этот новый тип данных в коде формы проекта DerivedDates. Просто определите объект `ADay` типа `TNewDate` и вызовите его конструктор в методе `FormCreate`:

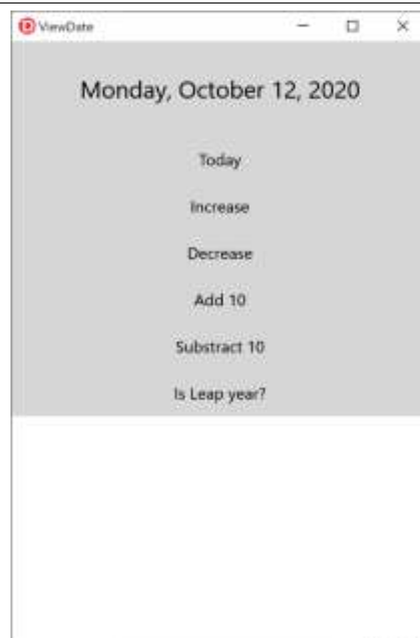
```
type
  TDateForm = class(TForm)
    ...
  private
    ADay: TNewDate; // updated declaration
  end;

procedure TDateForm.FormCreate(Sender: TObject);
begin
  ADay := TNewDate.Create; // updated line
  DateLabel.text := TheDay.GetText;
end;
```

Без необходимости каких-либо других изменений, новое приложение будет работать должным образом.

Класс `TNewDate` наследует методы увеличения даты, добавления нескольких дней и так далее. Кроме того, старый код, вызывающий эти методы, до сих пор работает. На самом деле, для вызова новой версии метода `GetText` нет необходимости менять исходный код! Компилятор Object Pascal автоматически привяжет этот вызов к новому методу.

Исходный код всех остальных обработчиков событий остается точно таким же, хотя его значение существенно меняется, как показывает новый вывод (см. рисунок 8.1).

Рисунок 8.1: Вывод программы DerivedDates с названием месяца и дня в зависимости от региональных настроек Windows



Общий базовый класс

Мы видели, что вы можете унаследовать от данного базового класса, написав:

```
type
  TNewDate = class (TDate)
    ...
  end;
```

Но что будет, если пропустить базовый класс и написать:

```
type
  TNewDate = class
    ...
```

```
end;
```

В этом случае ваш класс наследует от базового класса, называемого `TObject`. Другими словами, Object Pascal имеет иерархию классов с одним корнем, в которой все классы прямо или косвенно наследуются от общего класса-предка. Наиболее часто используемые методы TObject - Create, Free и Destroy; но есть много других, которые я буду использовать на протяжении всей книги. Полное описание этого фундаментального класса (который можно считать как частью языка, так и частью библиотеки runtime) со ссылкой на все его методы можно найти в Главе 17.

примечание Понятие общего класса предков присутствует и в языках C# и Java, где это называется просто `Object`. В языке Си++, с другой стороны, такой идеи нет, и программа на Си++, как правило, имеет несколько независимых иерархий классов.

Защищенные поля Protected и инкапсуляция

Код метода `GetText` класса `TNewDate` компилируется только в том случае, если он записан в том же самом модуле, что и класс `TDate`. Фактически он обращается к приватному полю `FDate` класса-предка. Если мы хотим поместить класс-потомка в новый юнит, то мы должны либо объявить поле `FDate protected` (или `strict protected`), либо добавить простой, возможно защищенный метод в класс-потомок, чтобы прочитать значение приватного поля.

Многие разработчики считают, что первое решение всегда лучшее, потому что объявление большинства полей защищенными сделает класс более расширяемым и облегчит

написание подклассов. Однако это нарушает идею инкапсуляции. В большой иерархии классов изменение определения некоторых защищенных полей базовых классов становится таким же сложным, как и изменение некоторых глобальных структур данных. Если десять производных классов обращаются к этим данным, то изменение их определения означает потенциальную модификацию кода в каждом из десяти классов.

Другими словами, гибкость, расширение и инкапсуляция часто становятся противоречащими друг другу целями. Когда это происходит, следует стараться отдавать предпочтение инкапсуляции. Если вы можете сделать это, не жертвуя гибкостью, то это будет еще лучше. Часто такое компромиссное решение можно получить с помощью виртуального метода, тема, которую я подробно расскажу ниже в разделе "Позднее связывание и полиморфизм". Если вы решите не использовать инкапсуляцию для получения более быстрого кодирования подклассов, то ваша конструкция может не соответствовать объектно-ориентированным принципам.

Помните также, что защищенные поля разделяют те же правила доступа, что и приватные, так что любой другой класс в том же самом модуле всегда может получить доступ к защищенным членам других классов. Как упоминалось в предыдущей главе, вы можете использовать более сильную инкапсуляцию, используя строгий спецификатор защищенного доступа.

Использование приема "Protected

Hack"

> *Если вы новичок в* Object Pascal *и в ООП, это довольно продвинутый раздел, и вы, возможно, захотите пропустить его при первом чтении этой книги, так как это может несколько запутать вас.*

Учитывая то, как работает защита модулей, даже к защищенным членам базовых классов, можно иметь прямой доступ из классов, объявленных в текущем модуле, если только вы не используете ключевое слово `strict protected`. Это является обоснованием того, что обычно называется "*protected hack*", т.е. возможностью определить производный класс, идентичный базовому классу, с единственной целью получения доступа к защищенному члену базового класса. Вот как это работает.

Мы видели, что приватные и защищенные данные класса доступны любым функциям или методам, которые появляются *в том же самом модуле, что и класс*. Например, рассмотрим этот простой класс (часть проекта приложения Protection):

```
type
  TTest = class
  protected
    ProtectedData: Integer;
  public
    PublicData: Integer;
    function GetValue: string;
  end;
```

Метод `GetValue` просто возвращает строку с двумя целыми значениями:

```
function TTest.GetValue: string;
begin
  Result := Format ('Public: %d, Protected: %d',
    [PublicData, ProtectedData]);
end;
```

Как только вы поместите этот класс в свой собственный модуль, вы не сможете получить прямой доступ к его защищенной

части из других модулей. Соответственно, если вы напишете следующий код,

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  Obj.ProtectedData := 20;  // won't compile
  Show (Obj.GetValue);
  Obj.Free;
end;
```

компилятор выдаст сообщение об ошибке, *Необъявленный идентификатор: "ProtectedData."* Тут можно подумать, что нет возможности получить доступ к защищенным данным класса, определенного в другом модуле. Однако есть способ обойти это.

Рассмотрим, что произойдет, если вы создадите, казалось бы, бесполезный производный класс, например:

```
type
  TFake = class (TTest);
```

Теперь, в том же самом модуле, где вы его объявили, вы можете вызвать любой защищенный метод класса `TFake`. Фактически можно вызывать защищенные методы класса, объявленного в том же самом модуле.

Как это помогает использовать объект класса `TTest`? Учитывая, что оба класса имеют одинаковую точную схему расположения памяти (так как различий нет), можно заставить компилятор обращаться с объектом класса, как с одним из них, с тем, что, как правило, имеет тип-небезопасное приведение:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  TFake (Obj).ProtectedData := 20; // compiles!
  Show (Obj.GetValue);
  Obj.Free;
end;
```

Этот код компилируется и работает корректно, как вы видите, запустив проект приложения Protection. Опять же, причина в том, что класс `TFake` автоматически наследует защищенные поля базового класса `TTest`, а поскольку класс `TFake` находится в том же самом модуле, что и код, который пытается получить доступ к данным в унаследованных полях, защищенные данные становятся доступными.

Теперь, когда я показал вам, как это делается, я должен предупредить вас, что такое нарушение механизма системы защиты классов может привести к ошибкам в вашей программе (от доступа к данным, которые на самом деле не должны быть доступны), и она работает вопреки хорошей практике ООП. Однако бывают редкие случаи, когда использование этой техники является лучшим решением, что можно увидеть, взглянув на исходные тексты библиотек и на код многих компонентов.

В целом, эта техника является *хакерской*, и ее следует по возможности избегать, хотя она может рассматриваться со всеми эффектами как часть спецификации языка и доступна на всех платформах и во всех настоящих и прошлых версиях Object Pascal.

От наследования к полиморфизму

Наследование - хорошая техника, позволяющая избежать дублирования кода и разделять код методов между различными классами. Ее истинная сила, однако, заключается

в возможности работать с объектами различных классов единообразно, что часто обозначается в объектно-ориентированных языках программирования термином *полиморфизм* или упоминается как *поздняя привязка*.

Чтобы полностью понять эту особенность, нужно изучить несколько элементов: совместимость типов производных классов, виртуальных методов и многое другое, о чем мы расскажем в следующих разделах.

Наследование и совместимость типов

Как мы видели в некоторой степени, Object Pascal является строго типизированным языком. Это означает, что вы не можете, например, присвоить `Boolean` переменной целое значение, по крайней мере, без явного приведения типов. Основное правило заключается в том, что два значения совместимы по типу только в том случае, если они имеют один и тот же тип данных, или (точнее), если их тип данных имеет одно и то же имя и их определение происходит из одного и того же модуля.

Важное исключение из этого правила существует в случае типов классов. Если вы объявляете класс, например `TAnimal`, и выводите из него новый класс, скажем `TDog`, то вы можете присвоить переменной типа `TAnimal` объект типа `TDog`. Это потому, что собака — это животное! Так что, хотя это может вас удивить, следующие вызовы конструкторов являются законными:

```
var
  MyAnimal1, MyAnimal2: TAnimal;
begin
  MyAnimal1 := TAnimal.Create;
  MyAnimal2 := TDog.Create;
```

Если говорить точнее, то можно использовать объект класса-потомка в любой момент времени, когда ожидается появление объекта класса-предка. Однако, обратное не является законным; нельзя использовать объект класса-предка, когда ожидается объект класса-потомка. Чтобы упростить объяснение, опять-таки в терминах кода:

```
MyAnimal := MyDog;  // Все в порядке.
MyDog := MyAnimal;  // Это ошибка!!!
```

В самом деле, хотя мы всегда можем сказать, что собака - это животное, мы не можем предполагать, что любое животное является собакой. Иногда это может быть правдой, но не всегда. Это вполне логично, и правила совместимости языковых типов следуют этой же логике.

Прежде чем мы рассмотрим последствия этой важной особенности языка, можно попробовать проект приложения Animals1, которое определяет два простых класса `TAnimal` и `TDog`, наследуя один от другого:

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
  private
    FKind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
  end;
```

Два метода `Create` просто устанавливают значение `FKind, которое` возвращается функцией `GetKind`.

Форма этого примера, показанная на рисунке 8.2, имеет две радиокнопки (расположенные на панели) для выбора объекта того или иного класса. Этот объект хранится в приватном поле `MyAnimal`, имеющем тип `TAnimal`. Экземпляр этого класса создается и инициализируется при создании и повторном

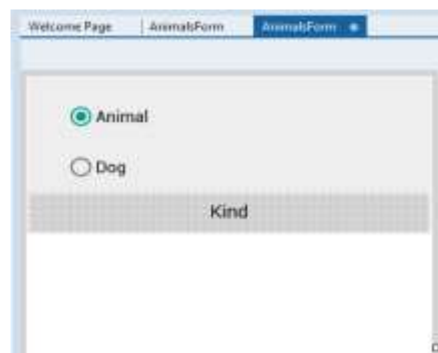
созданию формы каждый раз, когда выбирается одна из радиокнопок (здесь я показываю только код второй радиокнопки):

```
procedure TFormAnimals.FormCreate(Sender: TObject);
begin
  MyAnimal := TAnimal.Create;
end;

procedure TFormAnimals.RadioButton2Change(Sender: TObject);
begin
  MyAnimal.Free;
  MyAnimal := TDog.Create;
end;
```

Наконец, кнопка Kind вызывает метод `GetKind` для текущего животного и отображает результат в текстовой области, охватывающей нижнюю часть формы:

```
procedure TFormAnimals.BtnKindClick(Sender: TObject);
begin
  Show(MyAnimal.GetKind);
end;
```

Рисунок 8.2:
Форма заявки проекта Animals1 в среде разработки



Позднее связывание и полиморфизм

Функции и процедуры Object Pascal обычно основаны на *статической привязке,* которая также называется ранней

привязкой. Это означает, что вызов метода определяется компилятором или компоновщиком, который заменяет вызов на передачу управления в конкретное место в памяти, где находится скомпилированная функция или процедура. Объектно-ориентированные языки программирования допускают использование другой формы привязки, известной как динамическая привязка*, или* *поздняя привязка (связывание)*. В этом случае фактический адрес вызываемого метода определяется во время выполнения, исходя из типа экземпляра, использованного для вызова.

Преимущество этой техники известно, как *полиморфизм*. Полиморфизм означает, что можно записать вызов метода, применив его к переменной, но какой метод Delphi на самом деле вызывает, зависит от типа объекта, к которому относится переменная. Delphi не может определить до времени выполнения реальный класс объекта, к которому относится переменная, просто из-за правила совместимости с типом, рассмотренного в предыдущем разделе.

примечание Методы Object Pascal по умолчанию используют раннюю привязку, также как C++ и C#. Одной из причин этого является более эффективная работа. Вместо этого Java по умолчанию использует позднее связывание (и предлагает способы указать компилятору, что он может оптимизировать метод, используя раннее связывание).

Предположим, что класс и его подкласс (скажем, `TAnimal` и `TDog`, опять же) оба определяют метод, и у этого метода есть поздняя привязка. Теперь можно применить этот метод к общей переменной, такой как `MyAnimal`, которая во время выполнения может ссылаться либо на объект класса `TAnimal`, либо на объект класса `TDog`. Фактический метод для вызова определяется во время выполнения, в зависимости от класса текущего объекта.

Прикладной проект Animals2 расширяет проект Animals1 для демонстрации этой техники. В новой версии появились классы `TAnimal` и `TDog`: `Voice`, который означает вывод звука, издаваемого

выбранным животным, как в виде текста, так и в виде звука. В классе `TAnimal` этот метод определяется как `virtual` и в дальнейшем переопределяется при определении класса `TDog` с использованием ключевых слов virtual и override:

```
type
  TAnimal = class
  public
    function Voice: string; virtual;

  TDog = class (TAnimal)
  public
    function Voice: string; override;
```

Конечно, эти два метода также должны быть реализованы. Вот простой подход:

```
function TAnimal.Voice: string;
begin
  Result := 'AnimalVoice';
end;

function TDog.Voice: string;
begin
  Result := 'ArfArf';
end;
```

Каков эффект от вызова `MyAnimal.Voice`? Зависит от того. Если переменная `MyAnimal` в данный момент ссылается на объект класса `TAnimal`, то она вызовет метод `TAnimal.Voice`. Если переменная MyAnimal относится к объекту класса `TDog, то` вместо нее будет вызван метод `TDog.Voice.` Это происходит только потому, что функция виртуальная.

Вызов `MyAnimal.Voice будет` работать для объекта, который является экземпляром любого потомка класса `TAnimal`, даже для классов, которые определены после вызова этого метода или за его пределами. Компилятору не нужно знать обо всех потомках, чтобы сделать вызов совместимым с ними, нужен только класс-предшественник. Другими словами, этот вызов `MyAnimal.Voice` совместим со всеми будущими подклассами `TAnimal`.

Это главная техническая причина, по которой объектно-ориентированные языки программирования предпочтительны для реализации многократного использования. Можно писать код, использующий классы внутри иерархии, не зная о конкретных классах, входящих в эту иерархию. Другими словами, иерархия - и программа - все равно расширяема, даже если вы написали с ее помощью тысячи строк кода. Конечно, есть одно условие - классы предков иерархии должны быть очень тщательно продуманы.

Прикладной проект Animals2 демонстрирует использование этих новых классов и имеет форму, аналогичную предыдущему примеру. Этот код выполняется нажатием на кнопку, показывая вывод, а также производя некоторый звук:

```
begin
  Show (MyAnimal.Voice);
  MediaPlayer1.FileName := SoundsFolder + MyAnimal.Voice + '.wav';
  MediaPlayer1.Play;
end;
```

примечание Приложение использует компонент MediaPlayer для воспроизведения одного из двух звуковых файлов, которые поставляются вместе с приложением (звуковые файлы называются по фактическим звукам, т.е. значениям, возвращаемым методом `Voice`). Довольно случайный шум для обычного животного, и некоторый лай для собаки. Код сразу работает на Windows, пока файлы находятся в нужной папке, но требует определенных усилий для развертывания на мобильных платформах. Взгляните на фактическую демонстрацию, чтобы увидеть, как развертывание и папки структурированы.

Методы переопределения, повторного определения и повторного объявления

Как мы только что видели, для переопределения метода позднего связывания в классе-потомке, необходимо использовать ключевое слово override. Обратите внимание, что

это может сработать только в том случае, если метод был определен как виртуальный в классе-предке. В противном случае, если это был статический метод, то активировать позднее связывание, кроме изменения кода класса-предка, невозможно.

примечание Вы, наверное, помните, что я использовал то же ключевое слово, что и в прошлой главе, для переопределения деструктора по умолчанию `Destroy`, унаследованного от базового класса `TObject`.

Правила просты: Метод, определенный как статический, остается статическим в каждом подклассе, если только вы не спрячете его новым виртуальным методом с тем же именем. Метод, определенный как `virtual`, остается `поздне-связываемым` в каждом подклассе. Изменить это невозможно, так как компилятор генерирует различный код для поздних методов.

Чтобы переопределить статический метод, вы просто добавляете метод в подкласс, имеющий те же самые или отличные от исходного параметры, без каких-либо дополнительных уточнений. Чтобы переопределить виртуальный метод, необходимо указать те же самые параметры и использовать ключевое слово `override`:

```
type
  TMyClass = class
    procedure One; virtual;
    procedure Two; // static method
  end;

  TMySubClass = class (MyClass)
    procedure One; override;
    procedure Two;
  end;
```

Переопределенный метод, `Two`, не имеет поздней привязки. Поэтому, когда вы применяете его к переменной базового класса, он вызывает метод базового класса несмотря ни на что (то есть, даже если переменная относится к объекту производного класса, который имеет другую версию для этого метода).

Как правило, существует два способа отмены метода. Один из них заключается в замене метода класса-предка на новую версию. Другой - добавить дополнительный код к существующему методу. Этот второй подход может быть реализован путем использования ключевого слова `inherited` для вызова того же самого метода класса-предка. Например, можно написать

```
procedure TMySubClass.One;
begin
  // new code
  ...
  // call inherited procedure TMyClass.One
  inherited One;
end;
```

Вы можете задаться вопросом, зачем вам нужно использовать ключевое слово `override`. В других языках, когда Вы переопределяете виртуальный метод в подклассе, Вы автоматически переопределяете исходный метод. Однако наличие конкретного ключевого слова позволяет компилятору проверить соответствие между именем метода в классе-предке и именем метода в подклассе (неправильное написание переопределенной функции является распространенной ошибкой в некоторых других ООП-языках), проверить, был ли метод виртуальным в классе-предке, и так далее.

примечание Существует еще один популярный ООП-язык, который имеет такое же ключевое слово `override`, C#. Это неудивительно, учитывая тот факт, что языки придумал один дизайнер. Андерс Хейлсберг написал несколько длинных статей, объясняющих, почему ключевое слово "переопределить" является фундаментальным версионным инструментом для проектирования библиотек, как вы можете прочитать на `сайте` http://www.artima.com/intv/nonvirtual.html. Совсем недавно в языке Apple Swift было принято ключевое слово `override` для модификации методов в производных классах.

Еще одним преимуществом этого ключевого слова является то, что если вы определите статический метод в любом классе, унаследованном от класса библиотеки, то проблем не возникнет, даже если библиотека будет обновлена новым

виртуальным методом с тем же именем, что и метод, который вы определили. Поскольку ваш метод не помечен ключевым словом `override`, он будет считаться отдельным методом, а не новой версией метода, добавленной в библиотеку (что-то, что, скорее всего, сломает ваш существующий код).

Поддержка перегрузки добавляет некоторой дополнительной сложности к этой картине. Подкласс может предоставить новую версию метода по ключевому слову `overload`. Если метод имеет иные параметры, чем версия в базовом классе, то он фактически становится перегруженным методом, в противном случае он заменяет метод базового класса. Приведем пример:

```
type
  TMyClass = class
    procedure One;
  end;

  TMySubClass = class (TMyClass)
    procedure One (S: string); overload;
  end;
```

Обратите внимание, что в базовом классе метод не нужно помечать как `overload`. Однако если метод в базовом классе виртуальный, то компилятор выдает предупреждение *Метод 'One', скрывающий виртуальный метод базового типа 'TMyClass.'*

Чтобы избежать этого сообщения от компилятора и дать компилятору более точные инструкции о своих намерениях, можно использовать специальную директиву reintroduce:

```
type
  TMyClass = class
    procedure One; virtual;
  end;

  TMySubClass = class (TMyClass)
    procedure One (S: string); reintroduce; overload;
  end;
```

Вы можете найти этот код в прикладном проекте ReintroduceTest и поэкспериментировать с ним в дальнейшем.

примечание Сценарий, в котором используется ключевое слово `reintroduce` – это когда необходимо добавить пользовательский конструктор `Create` в класс компонента, который уже наследует виртуальный конструктор `Create` от базового класса `TComponent`.

Наследование и конструкторы

Как мы видели, можно использовать `inherited` ключевое слово для вызова одноименного метода (или также другого метода) в методе производного класса. То же самое справедливо и для конструкторов. В то время как в других языках, таких как C++, C# или Java, вызов конструктора базового класса является неявным и обязательным (когда необходимо передать параметры конструктору базового класса), в Object Pascal вызов конструктора базового класса не является строго обязательным.

Однако в большинстве случаев чрезвычайно важен ручной вызов конструктора базового класса. Это происходит, например, для любого класса компонента, так как инициализация компонента на самом деле осуществляется на уровне класса `TComponent`:

```
constructor TMyComponent.Create (Owner: TComponent);
begin
  inherited Create (Owner);
  // specific code...
end;
```

Это особенно важно, потому что для компонентов `Create` является виртуальным методом. Аналогично для всех классов деструктор `Destroy` является виртуальным методом, и следует помнить о вызове унаследованного в нем метода.

Остается один вопрос: если вы создаете класс, который наследуется только от TObject, то в его конструкторах нужно вызывать базовый конструктор `TObject.Create`? С технической

точки зрения ответ - "нет", учитывая, что конструктор пуст. Однако, я считаю хорошей привычкой всегда вызывать конструктор базового класса, несмотря ни на что. Однако, если вы маньяк производительности, то я допускаю, что это может неоправданно замедлить ваш код... на совершенно незаметную долю микросекунды.

Шутки в сторону, есть веские основания для обоих подходов, но особенно новичкам в языке я всегда рекомендую сделать вызов конструктор базового класса хорошей привычкой программирования, способствующей более безопасному кодированию.

Виртуальный и динамический методы

В Object Pascal есть два различных способа активации поздней привязки. Вы можете объявить метод virtual (виртуальным), как мы видели раньше, или объявить его dynamic (динамическим). Синтаксис этих двух ключевых слов точно такой же, и результат их использования также одинаков. Отличием является внутренний механизм, используемый компилятором для реализации позднего связывания.

Виртуальные методы основаны на таблице виртуальных методов (или VMT, но в разговорной форме также известной как *vtable*). Таблица виртуальных методов представляет собой массив адресов методов. Для вызова виртуального метода компилятор генерирует код для перехода к адресу, хранящемуся в *n-ом* слоте таблицы виртуальных методов объекта.

Таблицы виртуальных методов позволяют быстро выполнять вызовы методов. Их основным недостатком является то, что

они требуют записи для каждого виртуального метода для каждого класса-потомка, даже если метод не переопределен в подклассе. Иногда это приводит к распространению записей в таблицах виртуальных методов по всей иерархии классов (даже для методов, которые не переопределены). Это может потребовать много памяти просто для хранения одного и того же адреса метода несколько раз.

Вызовы динамического метода, с другой стороны, отправляются с использованием уникального номера, указывающего метод. Поиск соответствующей функции, как правило, происходит медленнее, чем простой одноступенчатый поиск виртуальных методов. Преимущество заключается в том, что записи динамического метода распространяются только в потомках, когда потомки переопределяют метод. Для больших или глубоких иерархий объектов использование динамических методов вместо виртуальных может привести к значительной экономии памяти с минимальным снижением скорости.

С точки зрения программиста, разница между этими двумя подходами заключается только в различном внутреннем представлении и незначительно различной скорости или использовании памяти. Кроме того, виртуальные и динамические методы одинаковы.

Теперь, объяснив разницу между этими двумя моделями, важно подчеркнуть, что в наибольшем числе случаев разработчики приложений используют `virtua,` а не `dynamic`.

Обработчики сообщений в Windows

Когда вы собираете приложения для Windows, для работы с системными сообщениями Windows можно использовать специальный метод поздней загрузки. Для этого Object Pascal предоставляет еще одну директиву, `message`, для определения

методов обработки сообщений, которыми должны быть процедуры с одним `var` параметром соответствующего типа. За директивой `message` следует номер сообщения Windows, с которым метод хочет обработать. Например, следующий код позволяет обрабатывать пользовательское сообщение с числовым значением, указываемым константой WM_USER Windows:

```
type
  TForm1 = class(TForm)
    ...
    procedure WmUser (var Msg: TMessage); message WM_USER;
  end;
```

Имя процедуры и фактический тип параметров зависит от вас, если физическая структура данных соответствует структуре сообщений Windows. Модули, используемые для взаимодействия с Windows API, включают ряд предопределенных типов записей для различных сообщений Windows. Эта техника может быть чрезвычайно полезной для ветеранов-программистов Windows, которые знают все о сообщениях Windows и функциях API, но она абсолютно не совместима с другими операционными системами (такими как MacOS, iOS и Android).

Абстрактные методы и классы

Когда вы создаете иерархию классов, иногда трудно определить, какой из них является базовым, так как он может не представлять собой реальную сущность, а использоваться только для поддержания некоторого общего поведения. Примером может служить базовый класс животного для чего-

то вроде класса кошки или собаки. Такой класс, для которого не предполагается создание какого-либо объекта, часто указывается как *абстрактный*, так как не имеет конкретной и полной реализации. Абстрактный класс может иметь абстрактные методы, методы, которые не имеют реальной реализации.

Абстрактные методы

Ключевое слово abstract используется для объявления виртуальных методов, которые будут определены только в подклассах текущего класса. Директива abstract полностью определяет метод; это не предварительное объявление. Если попытаться дать определение реализации метода, то компилятор ругнется.

В Object Pascal можно создавать экземпляры классов, которые имеют методы `abstract`. Однако, при попытке сделать это, компилятор выдает предупреждение: *Конструирование экземпляра <имя класса>, содержащего абстрактные методы*. Если вы случайно вызовете `abstract` метод во время выполнения, то Delphi вызовет специфическое исключение во время выполнения.

примечание C++, Java и другие языки используют более строгий подход: в этих языках нельзя создавать экземпляры абстрактных классов.

Вас может заинтересовать, зачем использовать абстрактные методы. Причина в использовании полиморфизма. Если класс `TAnimal` имеет `virtual abstract` метод `Voice`, то каждый подкласс может переопределить его. Преимущество в том, что теперь вы можете использовать общий объект `MyAnimal`, чтобы ссылаться на каждое животное, определенное подклассом, и вызывать этот метод. Если бы данный метод не присутствовал в интерфейсе

класса `TAnimal`, то его вызов не был бы разрешен компилятором, выполняющим статическую проверку типа. С помощью общего объекта `MyAnimal` можно вызвать только метод, определенный его собственным классом `TAnimal`.

Нельзя вызывать методы, предоставляемые подклассами, если только родительский класс не имеет, по крайней мере, объявления данного метода в виде `abstract` метода. Следующий проект приложения, Animals3, демонстрирует использование `абстрактных` методов и ошибку вызова абстрактного метода. Ниже приведены интерфейсы классов этого нового примера:

```
type
  TAnimal = class
  public
    constructor Create;
    function GetKind: string;
    function Voice: string; virtual; abstract;
  private
    FKind: string;
  end;

  TDog = class (TAnimal)
  public
    constructor Create;
    function Voice: string; override;
    function Eat: string; virtual;
  end;

  TCat = class (TAnimal)
  public
    constructor Create;
    function Voice: string; override;
    function Eat: string; virtual;
  end;
```

Наиболее интересной частью является определение класса `TAnimal`, включающее в себя виртуальный абстрактный метод: `Voice`. Также важно отметить, что каждый производный класс переопределяет это определение и добавляет новый виртуальный метод `Eat`. Каковы последствия этих двух разных подходов? Чтобы вызвать функцию `Voice`, можно просто написать тот же код, что и в предыдущей версии программы:

```
  Show (MyAnimal.Voice);
```

Как мы можем вызвать метод `Eat`? Мы не можем применить его к объекту класса `TAnimal`. Утверждение

```
Show (MyAnimal.Eat);
```

генерирует ошибку компилятора: Ожидается *Идентификатор поля*.

Для решения этой проблемы можно использовать динамическое и безопасное приведение типа, чтобы трактовать объект `TAnimal` как `TCat` или как объект `TDog`, но это будет очень громоздкий и склонный к ошибкам подход:

```
begin
  if MyAnimal is TDog then
    Show (TDog(MyAnimal).Eat)
  else if MyAnimal is TCat then
    Show (TCat(MyAnimal).Eat);
```

Этот код будет разъяснен далее в разделе "Безопасные операторы приведения типа". Добавление определения виртуального метода в класс `TAnimal` является типичным решением проблемы, и наличие ключевого слова `abstract` благоприятствует такому выбору. Вышеприведенный код выглядит некрасиво, и избегание такого кода как раз и является причиной использования полиморфизма.

Наконец, обратите внимание, когда класс имеет абстрактный метод, он часто рассматривается как абстрактный класс. Однако можно также специально пометить класс директивой abstract (и он будет считаться абстрактным классом, даже если у него нет абстрактных методов). Опять же, в Object Pascal это не помешает вам создать экземпляр класса, так что в этом языке полезность объявления абстрактного класса достаточно ограничена.

Запечатанные классы и

окончательные методы

Как я уже упоминал, в Java принят динамический подход с поздней привязкой (или виртуальными функциями) по умолчанию. По этой причине язык ввел такие понятия, как классы, от которых вы не можете наследовать (*sealed*), и методы, которые вы не можете переопределить в производных классах (*final methods* или невиртуальные методы).

Запечатанные (sealed) классы — это классы, от которых вы не можете далее наследовать. Это может иметь смысл, если вы распространяете компоненты (без исходного кода) или runtime-пакеты и хотите ограничить возможности других разработчиков модифицировать ваш код. Одной из первоначальных целей также было повышение безопасности во время выполнения, что, как правило, не требуется на полностью скомпилированном языке, таком как Object Pascal.

Окончательные (final) методы являются виртуальными методами, которые в дальнейшем невозможно переопределить в наследуемых классах. Опять же, хотя они и имеют смысл в Java (где все методы по умолчанию виртуальны, а конечные методы существенно оптимизированы), они были приняты в C#, где виртуальные функции явно помечены и имеют гораздо меньшее значение. Точно так же они были добавлены в Object Pascal, но они используются редко.

С точки зрения синтаксиса, вот код опечатанного класса:

```
type
  TDeriv1 = class sealed (TBase)
    procedure A; override;
  end;
```

Попытка наследования от него приводит к ошибке "*Cannot extended seal class TDeriv1*". Это синтаксис конечного метода:

```
type
  TDeriv2 = class (TBase)
    procedure A; override; final;
```

```
end;
```

Наследование от этого класса и переопределение метода `A` приводит к ошибке компилятора "*Cannot override a final method*".

Безопасные операторы приведения типа

Как мы видели ранее, правило совместимости языковых типов для классов-потомков позволяет использовать класс-потомок там, где ожидается класс-предок. Как я уже упоминал, обратное невозможно.

Теперь предположим, что класс `TDog` имеет метод `Eat,` которого нет в классе `TAnimal`. Если переменная `MyAnimal` относится к собаке, то, возможно, вы захотите иметь возможность вызвать функцию. Но если попробовать, и переменная ссылается на другой класс, то в результате получится ошибка. Выполняя явное приведение типа, мы можем привести к неприятной ошибке во время выполнения (или, что еще хуже, к трудной для обнаружения проблеме перезаписи памяти), так как компилятор не может определить, правильный ли тип объекта и действительно ли существуют вызываемые нами методы.

Для решения проблемы можно использовать методы, основанные на информации о типе исполнения. Потому что каждый объект во время выполнения "знает" свой тип и родительский класс. Эту информацию можно запросить с помощью оператора is или используя некоторые методы класса `TObject`. Параметрами оператора is являются объект и тип класса, а возвращаемое значение является Boolean:

```
if MyAnimal is TDog then
  ...
```

Выражение `is` вычисляется как `True` только в том случае, если объект `MyAnimal` в данный момент ссылается на объект класса `TDog` или потомка типа от и совместим с `TDog`. Это означает, что если вы проверите, действительно ли объект `TDog`, хранящийся в переменной `TAnimal`, является объектом `TDog`, то тест будет успешным. Другими словами, это выражение вычисляется как `True`, если вы можете безопасно присвоить объект (`MyAnimal`) переменной типа данных (`TDog`).

примечание Реализацию оператора is обеспечивает метод `InheritsFrom` класса `TObject`. Таким образом, можно написать то же выражение, что и `MyAnimal.InheritsFrom(TDog)`. Причина использования этого метода напрямую связана с тем, что он может быть применен и к ссылкам на класс и другим типам специального назначения, чем не поддерживает оператор `is`.

Теперь, когда вы точно знаете, что животное — это собака, вы можете использовать прямое приведение (что в целом было бы небезопасно), написав следующий код:

```
if MyAnimal is TDog then
begin
  MyDog := TDog (MyAnimal);
  Text := MyDog.Eat;
end;
```

Эта же операция может быть выполнена непосредственно другим связанным оператором приведения типа `as`, так как он преобразует объект только в том случае, если запрашиваемый класс совместим с реальным. Параметрами оператора as являются объект и тип класса, в результате чего объект "*преобразуется*" в новый тип класса. Можно написать следующий фрагмент:

```
MyDog := MyAnimal as TDog;
Text := MyDog.Eat;
```

Если мы хотим вызвать только функцию `Eat`, мы можем использовать еще более короткую нотацию:

```
(MyAnimal as TDog).Eat;
```

Результатом данного выражения является объект типа данных класса TDog, поэтому к нему можно применить любой метод этого класса. Разница между традиционным приведением и использованием выражения as заключается в том, что второй проверяет фактический тип объекта и вызывает исключение, если тип не совместим с типом, к которому вы пытаетесь его привести. Поднимается исключение EInvalidCast (исключения описаны в следующей главе).

предупреждение Напротив, в языке C# выражение as вернет nil, если объект не совместим с типом, в то время как прямое приведение типа вызовет исключение. Таким образом, по сравнению с Object Pascal, эти две операции, по сути, инвертируются.

Чтобы избежать этого исключения, используйте оператор is и, в случае успеха, сделайте прямое приведение типов (на самом деле, нет причин использовать is и as последовательно, делая проверку типа дважды - хотя вы часто увидите комбинированное использование is и as):

```
if MyAnimal is TDog then
  TDog(MyAnimal).Eat;
```

Оба оператора приведения типов очень полезны в Object Pascal, так как часто хочется написать общий код, который можно использовать с несколькими компонентами одного и того же типа или даже разных типов. Например, когда компонент передается в качестве параметра методу реакции на события, используется общий тип данных (TObject), поэтому часто требуется привести его обратно к исходному типу компонента:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Sender is TButton then
    ...
end;
```

Это обычная техника, которую я буду использовать в некоторых последующих примерах (события представлены в Главе 10).

Два типа операторов кастинга `is` и `as` являются чрезвычайно мощными, и у вас может возникнуть соблазн рассматривать их в качестве стандартных конструкций программирования. Несмотря на то, что они действительно мощные, вы, вероятно, должны ограничить их использование особыми случаями. Когда необходимо решить сложную проблему, включающую несколько классов, попробуйте сначала использовать полиморфизм. Только в особых случаях, когда полиморфизм сам по себе не может быть применен, попробуйте дополнить его операторами приведения типов.

примечание Использование операторов приведения типов оказывает некоторое негативное влияние на производительность, так как оно должно пройти по иерархии классов, чтобы убедиться в правильности приведения типов. Как мы видели, вызовы виртуальных методов просто требуют поиска памяти, что намного быстрее.

Наследование визуальной формы

Наследование используется не только в библиотечных классах или в классах, которые вы пишете, но и достаточно широко распространено в среде разработки, основанной на Object Pascal. Как мы видели, когда вы создаете форму в IDE, то создается экземпляр класса, который наследуется от TForm. Так что любое визуальное приложение имеет структуру, основанную на наследовании, даже в тех случаях, когда вы в конечном итоге пишете большую часть своего кода в простых обработчиках событий.

Что менее известно, даже более опытным разработчикам, так это то, что от уже созданной формы можно унаследовать новую - функцию, обычно называемую *наследованием визуальной*

формы (и нечто весьма специфическое для среды разработки Object Pascal).

Интересным элементом здесь является то, что вы можете визуально увидеть силу наследования в действии, и непосредственно разобраться в его правилах! Полезно ли это на практике? Ну, в основном это зависит от того, какое приложение вы строите. Если у него есть несколько форм, некоторые из которых очень похожи друг на друга или просто включают общие элементы, то вы можете поместить общие компоненты и общие обработчики событий в базовую форму и добавить специфическое поведение и компоненты в подклассы. Другим распространенным сценарием является использование наследования визуальных форм для настройки некоторых форм приложений для определенных компаний без дублирования исходного кода (что является основной причиной использования наследования в первую очередь).

Вы также можете использовать наследование визуальных форм для настройки приложения для различных операционных систем и форм-факторов (например, от телефона к планшетам), не дублируя исходный код или код определения формы; просто унаследуйте конкретные версии для клиента от стандартных форм. Помните, что основное преимущество визуального наследования заключается в том, что в дальнейшем вы можете изменять исходную форму и автоматически обновлять все производные формы. Это известное преимущество наследования в объектно-ориентированных языках программирования. Но есть и положительный побочный эффект: полиморфизм. Можно добавить виртуальный метод к базовой форме и переопределить его в подклассе. Затем можно обратиться к обеим формам и вызвать этот метод для каждой из них.

примечание Другой подход в строительстве форм с теми же элементами заключается в том, чтобы опираться на фреймы, то есть на визуальную композицию панелей формы. В обоих случаях во время проектирования можно работать над двумя вариантами формы. Однако, в визуальном наследовании формы вы определяете два различных класса (родительский и производный), в то время как в случае с рамками вы работаете над классом фрейма и экземпляром этого фрейма, размещенным на форме.

Наследование от базовой формы

Правила, регулирующие наследование визуальных форм, довольно просты, как только у вас есть четкое представление о том, что такое наследование. В принципе, форма подкласса имеет те же компоненты, что и родительская форма, а также некоторые новые компоненты. Вы не можете удалить компонент базового класса, хотя (если это визуальный элемент управления) вы можете сделать его невидимым. Важно то, что вы можете легко изменить свойства компонентов, которые вы наследуете.

Обратите внимание, что если вы измените свойство компонента в унаследованной форме, то любое изменение этого же свойства в родительской форме не будет иметь никакого эффекта. Изменение других свойств компонента также повлияет на унаследованные версии. Синхронизировать значения свойств можно с помощью команды локального меню "Revert to Inherited" программы Object Inspector. То же самое можно сделать, установив два свойства в одно и то же значение и перекомпилировав код. После модификации нескольких свойств можно синхронизировать их все с базовой версией, применив команду Revert to Inherited локального меню компонента.

Помимо наследования компонентов, новая форма наследует все методы базовой формы, включая обработчики событий. Вы

можете добавлять новые обработчики в унаследованную форму, а также переопределять существующие обработчики.

Чтобы продемонстрировать, как работает наследование визуальных форм, я построил очень простой пример под названием VisualInheritTest. Я опишу шаг за шагом, как его построить. Сначала запустите новый Multi-device проект, выберите пустой шаблон проекта и добавьте две кнопки к его основной форме. Затем выберите File New Others, и в диалоговом окне New Items (Новые элементы) выберите страницу "Inheritable Items" (Унаследованные элементы) (см. Рисунок 8.3). Здесь вы можете выбрать форму, от которой хотите унаследовать.

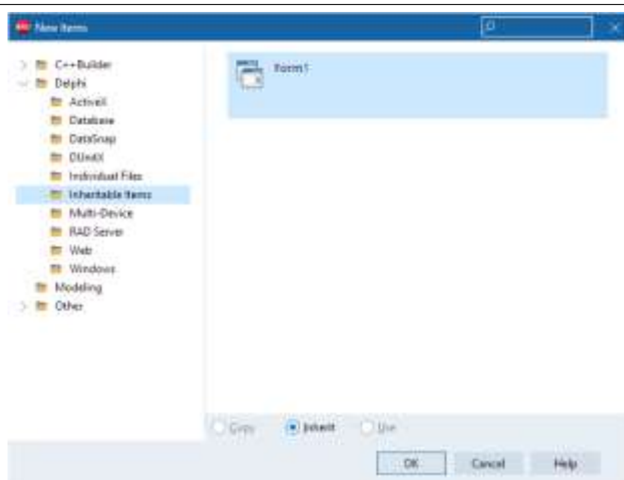


Рисунок 8.3: Диалоговое окно New Items (Новые элементы) позволяет создать унаследованную форму.

Новая форма имеет те же самые две кнопки. Вот первоначальное текстовое описание новой формы:

```
inherited Form2: TForm2
  Caption = 'Form2'
  ...
end
```

А вот ее первоначальное объявление класса, где видно, что базовый класс - это не обычная `форма TForm`, а реальная форма базового класса:

```
type
  TForm2 = class(TForm1)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Обратите внимание на наличие ключевого слова `inherited` в текстовом описании; также обратите внимание на то, что форма действительно имеет некоторые компоненты, хотя они и определены в базовом классе формы. Если вы измените подпись одной из кнопок и добавите новую, текстовое описание изменится соответствующим образом:

```
inherited Form2: TForm2
  Caption = 'Form2'
  ...
  inherited Button1: TButton
    Text = 'Hide Form'
  end
  object Button3: TButton
    ...
    Text = 'New Button'
    OnClick = Button3Click
  end
end
```

Перечисляются только свойства с другим значением, потому что остальные просто наследуются такими, какие они есть.

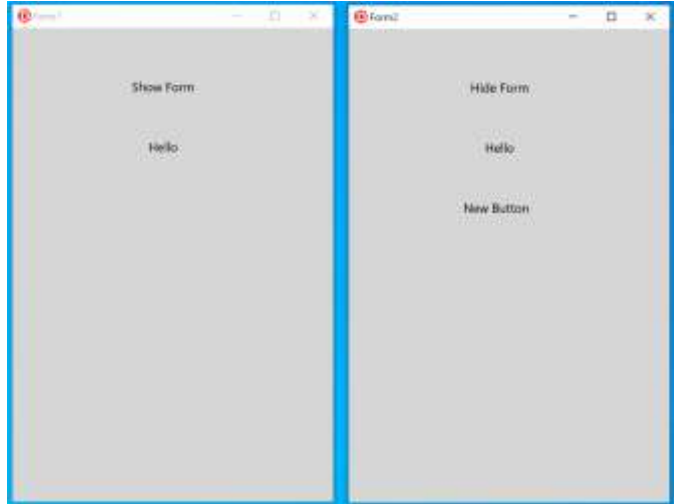


Рисунок 8.4: Две формы примера VirtualInheritTest во время выполнения

Каждая из кнопок первой формы имеет обработчик `OnClick`, с простым кодом. Первая кнопка показывает вторую форму, вызывающую ее метод `Show`; вторая кнопка - простое сообщение.

Что происходит в унаследованной форме? Сначала необходимо изменить поведение кнопки Show, чтобы реализовать ее в виде кнопки Hide. Это подразумевает, что обработчик события базового класса не будет выполняться (поэтому я прокомментировал наследуемый по умолчанию вызов). Для кнопки Hello, вместо нее, мы можем добавить второе сообщение к сообщению, отображаемому базовым классом, оставив `inherited` вызов:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  // inherited;
  Hide;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
  inherited;
  ShowMessage ('Hello from Form2');
end;
```

Помните, что в отличие от унаследованного метода, который может использовать ключевое `inherited` слово для вызова метода базового класса с тем же именем, в обработчике события ключевое слово `inherited` означает вызов соответствующего обработчика события базовой формы (независимо от имени метода обработчика события).

Конечно, вы также можете рассматривать каждый метод базовой формы как метод вашей формы, и называть их свободно. Этот пример позволяет вам исследовать некоторые особенности наследования визуальных форм, но, чтобы увидеть его истинную силу, вам нужно будет посмотреть на более сложные примеры из реального мира, чем в этой книге.

372- начало

09: Обработка Исключений

Прежде чем приступить к освещению других особенностей классов на языке Object Pascal, необходимо сосредоточиться на одной конкретной группе объектов, известных как *исключения* и используемых для обработки возникших ошибок.

Идея обработки исключений заключается в том, чтобы сделать программы более надежными, добавив возможность простой и унифицированной обработки программных или аппаратных ошибок (а также любых других видов ошибок). Программа может выдержать такие ошибки или аккуратно завершить свою работу, позволяя пользователю сохранять данные перед выходом из программы. Исключения позволяют отделить код обработки ошибок от обычного кода вместо того, чтобы смешивать их. В конечном итоге вы пишете более компактный и менее загроможденный работой по обслуживанию код, не связанный с фактической целью программирования.

Еще одним преимуществом является то, что исключения определяют унифицированный и универсальный механизм сообщения об ошибках, который также используется библиотеками компонентов. Во время работы система поднимает исключения, когда что-то идет не так. Если ваш код

написан корректно, он может распознать проблему и попытаться ее решить, в противном случае исключение передается в вызывающий его код, и так далее. В конечном счете, если ни одна часть вашего кода не обрабатывает исключение, система, как правило, обрабатывает его, выводя стандартное сообщение об ошибке и пытаясь продолжить программу. В необычном сценарии ваш код выполняется вне любого блока обработки исключений, поднятие исключения приведет к завершению работы программы. Весь механизм обработки исключений в Object Pascal основан на пяти отдельных ключевых словах:

try разграничить начало защищаемого блока кода

except разделяет конец защищаемого блока кода и вводит код обработки исключений

`on` маркировке отдельных операторов обработки исключений, привязанных к конкретным исключениям, каждый из которых имеет синтаксис on `exception-type` do `statement`.

finally используется для указания блоков кода, которые всегда должны выполняться, даже когда возникают исключения

raise - это оператор, используемый для запуска исключения и имеющий в качестве параметра объект исключения (в других языках программирования эта операция называется `throw`)

Ниже приведена простая сравнительная таблица исключений, обрабатывающих ключевые слова в Object Pascal с языками, основанными на синтаксисе исключений C++ (например, C# и Java):

`try`	`try`
`except on`	catch
`finally`	`finally`
`raise`	throw

Используя терминологию языка Си++, вы бросаете объект исключения и перехватываете его по типу. То же самое происходит и в Object Pascal, где вы передаете в операторе `raise` объект исключения и получаете его как параметр в операторах `except on`.

Блоки Try-Except

Начну с довольно простого примера (часть проекта приложения ExceptionsTest), который имеет общий блок обработки исключений:

```
function DividePlusOne (A, B: Integer): Integer;
begin
  try
    Result := A div B; // raises exception if B equals 0
    Inc (Result);
  except
    Result := 0;
  end;
  // more
end;
```

примечание Когда вы запускаете программу в отладчике Delphi, отладчик по умолчанию останавливает программу при возникновении исключения, даже если есть обработчик исключения. Конечно, это обычно то, чего вы хотите, потому что вы хотите знать, где произошло исключение, и вы можете видеть вызов обработчика пошагово. Если вы просто хотите дать программе запуститься, когда исключение корректно обработано, и посмотреть, что увидит пользователь, запустите программу с помощью команды "Выполнить без отладки" или отключите все (или некоторые) исключения в опциях отладчика.

"Сокрытие" исключения, как в коде выше, и установка результата в ноль на самом деле не имеет смысла в реальном приложении, но код предназначен для того, чтобы помочь вам понять основной механизм в простом сценарии. Это код обработчика событий, используемый для вызова функции:

```
var
  N: Integer;
begin
```

```
  N := DividePlusOne (10, Random(3));
  Show (N.ToString);
```

Как видите, программа использует случайно сгенерированное значение, так что при нажатии на кнопку Вы можете оказаться в действительной ситуации (2 раза из 3) или в недействительной. Таким образом, может быть два различных потока программы:

Если `B` не равен нулю, программа производит деление, выполняет инкремент, а затем пропускает блок до конца оператора except, выполняя следующий за ним (`//more`).

Если `B` равен нулю, то деление вызывает исключение, все следующие утверждения пропускаются (ну, в данном случае только одно) до первого вложенного блока `try-expect,` который выполняется вместо него. После блока исключения программа не возвращается к исходному оператору, а пропускает до тех пор, пока после блока исключения не выполнит первый после него оператор (`//more`).

Для описания этой модели исключения можно сказать, что она следует подходу невозобновления. В случае ошибки очень опасно пытаться обрабатывать состояние ошибки и возвращаться к тому заявлению, которое ее вызвало, так как статус программы на тот момент, вероятно, не определен. Исключения существенно изменяют поток выполнения, пропуская выполнение следующего оператора и откатывая стек до тех пор, пока не будет найден правильный код обработки ошибок.

Код, приведенный выше, был очень простым, без утверждения on. Когда вам нужно работать с несколькими типами исключений (или с несколькими типами классов исключений), или если вы хотите получить доступ к объекту исключения, переданному в блок, у вас должен быть один или несколько операторов `on`:

```
function DividePlusOneBis (A, B: Integer): Integer;
begin
```

```
  try
    Result := A div B; // error if B equals 0
    Result := Result + 1;
  except
    on E: EDivByZero do
    begin
      Result := 0;
      ShowMessage (E.Message);
    end;
  end;
end;
```

В заявлении об обработке исключений мы ловим EDivByZero исключение, которое определяется библиотекой времени исполнения. Существует ряд таких типов исключений, относящихся к проблемам времени выполнения (например, деление на ноль или неправильное динамическое приведение), к системным проблемам (например, ошибки вне памяти) или к ошибкам компонентов (например, неправильный индекс). Все эти классы исключений наследуют от базового класса `Exception`, который предлагает некоторые минимальные возможности, такие как свойство `Message`, которое я использовал в коде выше. Эти классы образуют реальную иерархию с некоторой логической структурой.

примечание Обратите внимание, что в то время, как типы в Object Pascal обычно обозначаются начальной буквой `T`, классы исключений делают *исключение* из правила и обычно начинаются с буквы `E`.

Иерархия исключений

Вот неполный список основных классов исключений, определенных в модуле System.SysUtils библиотеки времени исполнения (большинство других системных библиотек добавляют свои собственные типы исключений):

```
Exception
  EArgumentException
    EArgumentOutOfRangeException
    EArgumentNilException
  EPathTooLongException
```

```
ENotSupportedException
EDirectoryNotFoundException
EFileNotFoundException
EPathNotFoundException
EListError
EInvalidOpException
ENoConstructException
EAbort
EHeapException
  EOutOfMemory
  EInvalidPointer
EInOutError
EExternal
  EExternalException
  EIntError
    EDivByZero
    ERangeError
    EIntOverflow
  EMathError
    EInvalidOp
    EZeroDivide
    EOverflow
    EUnderflow
  EAccessViolation
  EPrivilege
  EControlC
  EQuit
EInvalidCast
EConvertError
ECodesetConversion
EVariantError
EPropReadOnly
EPropWriteOnly
EAssertionFailed
EAbstractError
EIntfCastError
EInvalidContainer
EInvalidInsert
EPackageError
ECFError
EOSError
ESafecallException
EMonitor
  EMonitorLockException
  ENoMonitorSupportException
EProgrammerNotFound
ENotImplemented
EObjectDisposed
EJNIException
```

примечание Не знаю как вы, но мне трудно представить точный сценарий использования самого странного класса исключений, забавное исключение EProgrammerNotFound!

Теперь, когда вы увидели иерархию основных исключений, я могу добавить еще немного информации к предыдущему описанию утверждений `-on` исключений. Эти утверждения оцениваются последовательно до тех пор, пока система не найдет класс исключения, соответствующий типу поднятого объекта исключения. Теперь используется соответствующее правило совместимости типов, которое мы рассмотрели в последней главе: объект исключения совместим с любым из базовых типов своего определенного типа (например, объект `TDog` был совместим с классом `TAnimal`).

Это означает, что вы можете иметь несколько типов обработчиков исключений, которые соответствуют исключению. Если вы хотите иметь возможность обрабатывать более гранулированные исключения (нижние классы иерархии) вместе с более общими в случае, если ни один из предыдущих типов не совпадает, вам нужно перечислить блоки обработчиков от более специфичных до более общих (или от дочернего класса исключения до его родительских классов). Также, если вы пишете обработчик для типа `Exception,` то это будет пункт catch-all, так что он должен быть последним из последовательности.

Вот фрагмент кода с двумя обработчиками в одном блоке:

```
function DividePlusOne (A, B: Integer): Integer;
begin
  try
    Result := A div B; // error if B equals 0
    Result := Result + 1;
  except
    on EDivByZero do
    begin
      Result := 0;
      MessageDlg ('Divide by zero error',
        mtError, [mbOK], 0);
    end;
    on E: Exception do
    begin
      Result := 0;
      MessageDlg (E.Message,
        mtError, [mbOK], 0);
```

```
    end;
  end; // end of except block
end;
```

В данном коде есть два различных обработчика исключений после одного и того же блока `проб.` Вы можете иметь любое количество этих обработчиков, которые оцениваются последовательно, как описано выше.

Имейте в виду, что использование обработчика для всех возможных исключений, как правило, не является хорошим выбором. Лучше оставить неизвестные исключения системе. Обработчик исключения по умолчанию обычно выводит сообщение об ошибке класса исключения в окно сообщений, а затем возобновляет нормальную работу программы.

совет На самом деле вы можете модифицировать обычный обработчик исключений, предоставив метод для события `Application.`OnException, например, протоколирование сообщения об исключении в файле, а не его отображение пользователю.

Поднятие (вызов) исключения

Большинство исключений, с которыми вы столкнетесь при программировании на Object Pascal, будут генерироваться системой, но вы также можете вызвать исключения в вашем собственном коде, когда обнаружите недействительные или противоречивые данные во время выполнения.

В большинстве случаев для пользовательского исключения вы определяете свой собственный тип исключения. Просто создайте новый подкласс класса исключения по умолчанию или один из его существующих подклассов, которые мы видели выше:

```
type
  EArrayFull = class (Exception);
```

В большинстве случаев нет необходимости добавлять методы или поля к новому классу исключений, и достаточно объявить пустой производный класс.

Сценарий для данного типа исключения будет представлять собой метод, который добавляет элементы в массив, что приводит к ошибке при заполнении массива. Это достигается созданием объекта исключения и передачей его ключевому слову `raise`:

```
if MyArray.IsFull then
  raise EArrayFull.Create ('Array full');
```

Метод `Create` (унаследованный от базового класса `Exception`) имеет строковый параметр для описания исключения для пользователя.

примечание Вам не нужно беспокоиться об уничтожении объекта, который вы создали для исключения, так как он будет автоматически удален механизмом обработки исключений.

Существует второй сценарий использования ключевого слова " `raise` ". Внутри блока исключений вы можете захотеть выполнить некоторые действия, но не поймать исключение, позволив ему обрабатываться во внешнем блоке исключения. В этом случае вы можете вызвать `raise` без параметров. Операция называется *повторным поднятием* исключения.

Исключения и Стек

Когда программа поднимает исключение и текущая подпрограмма не обрабатывает его, что происходит со стеком вызовов методов и функций? Программа начинает поиск обработчика среди функций, уже находящихся в стеке. Это означает, что программа выходит из существующих функций и не выполняет остальных операторов. Чтобы понять, как это работает, можно воспользоваться отладчиком или добавить

несколько простых выходных строк, чтобы быть информированным при выполнении определенного оператора исходного кода. В следующем проекте приложения, ExceptionFlow, я придерживался этого второго подхода.

Например, при нажатии кнопки `Raise1` в форме проекта приложения ExceptionFlow поднимается и не обрабатывается исключение, так что заключительная часть кода никогда не будет выполнена:

```
procedure TForm1.ButtonRaise1Click(Sender: TObject);
begin
  // unguarded call
  AddToArray (24);
  Show ('Program never gets here');
end;
```

Обратите внимание, что этот метод вызывает процедуру `AddToArray`, которая неизменно вызывает исключение. При обработке исключения поток начинается снова после обработчика, а не после кода, который поднимает исключение. Рассмотрим этот модифицированный метод:

```
procedure TForm1.ButtonRaise2Click(Sender: TObject);
begin
  try
    // this procedure raises an exception
    AddToArray (24);
    Show ('Program never gets here');
  except
  on EArrayFull do
      Show ('Handle the exception');
  end;
  Show ('ButtonRaise1Click call completed');
end;
```

Последний вызов `Show будет` выполнен сразу после второго, а первый всегда игнорируется. Я предлагаю запустить программу, изменить ее код и поэкспериментировать с ней, чтобы полностью разобраться в потоке программы при вызове исключения.

примечание Учитывая, что место в коде, где вы работаете с исключением, отличается от того, где было поднято исключение, было бы неплохо узнать, в каком методе на самом

деле было поднято исключение. Хотя есть способы получить след стека, когда исключение было поднято, и сделать эту информацию доступной в обработчике, это действительно продвинутая тема, которую я не планирую здесь обсуждать. В большинстве случаев разработчики Object Pascal полагаются на сторонние библиотеки и инструменты (такие как JclDebug из Jedi Component Library, madExcept или EurekaLog). Кроме того, вы должны сгенерировать и включить в свой код MAP-файл, созданный компилятором и содержащий список адресов памяти каждого метода и функции в вашем приложении.

Блок Finally

Есть четвертое ключевое слово для обработки исключений, которое я упомянул, но до сих пор не использовал, `finally`. Блок `finally` используется для выполнения некоторых действий (обычно операций по очистке), которые должны выполняться в любом случае. На самом деле, выражения в блоке `finally` обрабатываются независимо от того, происходит ли исключение или нет. Простой код, следующий за блоком `try`, напротив, выполняется только в том случае, если исключение не было поднято или если оно было поднято и обработано. Другими словами, код в блоке `finally` всегда выполняется после кода блока `try`, даже если было поднято и обработано исключение.

Рассмотрим метод (часть проекта ExceptFinally application), который выполняет некоторые трудоемкие операции и отображает в заголовке формы свой статус:

```
procedure TForm1.BtnWrongClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := 'Calculating';

  J := 0;
  // long (and wrong) computation...
  for I := 1000 downto 0 do
    J := J + J div I;
```

```
  Caption := 'Finished';
  Show ('Total: ' + J.ToString);
end;
```

Из-за ошибки в алгоритме (так как переменная `I` может достигать значения 0 и также используется в делении), программа сломается, но не сбросит подписи формы. Для этого и предназначен блок `try-finally`:

```
procedure TForm1.BtnTryFinallyClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := 'Calculating';
  J := 0;
  try
    // long (and wrong) computation...
    for I := 1000 downto 0 do
      J := J + J div I;
    Show ('Total: ' + J.ToString);
  finally
    Caption := 'Finished';
  end;
end;
```

Когда программа выполняет эту функцию, она всегда сбрасывает курсор, независимо от того, происходит ли исключение (любого рода) или нет. Недостатком данной версии функции является то, что она не обрабатывает исключение.

Finally и Except

Любопытно, что в языке Object Pascal за блоком try может следовать либо исключение, либо окончательное утверждение, но не оба одновременно. Учитывая, что Вы часто хотите иметь оба блока, типичным решением является использование двух вложенных блоков try, связывающих внутренний блок с оператором `finally`, а внешний - с оператором `except` или наоборот, как того требует ситуация. Вот код этой третьей кнопки проекта приложения ExceptFinally:

```
procedure TForm1.BtnTryTryClick(Sender: TObject);
```

```
var
  I, J: Integer;
begin
  Caption := 'Calculating';
  J := 0;
  try try
    // long (and wrong) computation...
    for I := 1000 downto 0 do
      J := J + J div I;
    Show ('Total: ' + J.ToString);
  except
    on E: EDivByZero do
    begin
      // re-raise the exception with a new message
      raise Exception.Create ('Error in Algorithm');
    end;
  end;
  finally
    Caption := 'Finished';
  end;
end;
```

Восстановление курсора с блоком Finally.

Обычным примером использования блоков `try-finally` является распределение и высвобождение ресурсов. Другой связанный случай – это временная конфигурация, которую необходимо сбросить после завершения операции, даже в том случае, если эта операция вызывает исключение.

Одним из примеров временных настроек, которые необходимо восстановить, является курсор "Песочные часы", отображаемый во время длительной операции и удаляемый в конце, восстанавливающий исходный активный курсор. Даже если код прост, всегда есть шанс получить исключение, поэтому всегда следует использовать блок `try-finally`.

В примере приложения RestoreCursor (VCL приложение, так как управление курсором в FireMonkey немного сложнее) я написал следующий код для сохранения текущего курсора,

временной установки песочных часов и восстановления исходного курсора в конце:

```
 var CurrCur := Screen.Cursor;
  Screen.Cursor := crHourGlass;
  try
    // some slow operation
    Sleep (5000);
  finally
    Screen.Cursor := CurrCur;
  end;
```

Восстановить курсор с помощью управляемой записи.

Чтобы защитить выделение ресурсов или определить временную конфигурацию для восстановления, вместо явного блока `try-finally` можно использовать управляемую запись, которая заставит компилятор добавить обязательный блок `finally`. Это приводит к написанию меньшего количества кода для защиты ресурса или восстановления конфигурации, даже если есть некоторые первоначальные усилия по определению записи.

Вот управляемая запись, представляющая то же самое поведение кода в предыдущем разделе, сохраняющая текущий курсор в поле в методе `Initialize` и сбрасывающая его в методе `Finalize`:

```
type
  THourCursor = record
  private
    FCurrCur: TCursor;
  public
    class operator Initialize (out ADest: THourCursor);
    class operator Finalize (var ADest: THourCursor);
  end;

class operator ThourCursor.Initialize (out ADest: THourCursor);
begin
  ADest.FCurrCur := Screen.Cursor;
  Screen.Cursor := crHourGlass;
```

```
end;

class operator ThourCursor.Finalize (var ADest: THourCursor);
begin
  Screen.Cursor := ADest.FCurrCur;
end;
```

После того, как вы определили эту управляемую запись

```
 var HC: THourglassCursor;
  // some slow operation
  Sleep (5000);
```

примечание Более подробные примеры защиты ресурсов с помощью управляемых записей вы можете найти в следующем посте блога Эрика ван Билсена: https://blog.grijjy.com/2020/08/03/automate-restorable-operations-with-custom-managed-records/. Это часть серии очень подробных блогов об управляемых записях.

Исключения в реальном мире

Исключениями являются отличным механизмом сообщения об ошибках и обработки ошибок в целом (то есть не в рамках одного фрагмента кода, а как часть более крупной архитектуры). Исключения в целом не должны подменять собой проверку состояния локальной ошибки (хотя некоторые разработчики используют их таким образом).

Например, если вы не уверены в имени файла, проверка существования файла перед его открытием обычно считается лучшим подходом, чем открытие файла в любом случае с использованием исключений для работы со сценарием, в котором файла нет. Однако, проверка наличия достаточного места на диске перед записью в файл — это тип проверки, который не имеет особого смысла делать везде, так как это крайне редкое условие.

Другими словами, программа должна проверять на наличие распространенных ошибок и оставлять необычные и неожиданные условия механизму обработки исключений. Конечно, граница между двумя сценариями часто размыта, и разные разработчики будут иметь разные способы суждения.

Где вы неизменно используете исключения, так это для того, чтобы позволить различным классам и модулям передавать друг другу условия ошибок. Возвращать коды ошибок крайне утомительно и склонно к ошибкам по сравнению с использованием исключений. Поднятие исключений чаще встречается в классе компонента или библиотеке, чем в обработчике событий. В итоге можно написать много кода, не поднимая и не обрабатывая исключения.

Что чрезвычайно важно и очень часто встречается в повседневном коде, так это использование блоков `finally` для защиты ресурсов в случае исключения. Всегда следует защищать блоки, которые ссылаются на внешние ресурсы с использованием утверждения `finally`, чтобы избежать утечки ресурсов в случае возникновения исключения. Каждый раз, когда вы открываете и закрываете, подключаете и отключаете, создаете и уничтожаете что-то в рамках одной функции или метода, требуется утверждение finally. В конечном счете, утверждение finally позволяет поддерживать стабильность программы даже в случае возникновения исключения, позволяя пользователю продолжить использование или (в случае более серьезных проблем) должным образом закрыть приложение.

Обработка глобальных

исключений

Если исключение, вызванное обработчиком события, останавливает стандартный поток выполнения, будет ли он также завершать программу, если обработчик исключения не найден? Это действительно так для консольного приложения или других структур кода специального назначения, в то время как большинство визуальных приложений (включая те, которые основаны на библиотеках VCL или FireMonkey) имеют глобальный цикл обработки сообщений, который обертывает каждое выполнение в блоке `try-except`, так что если в обработчике события поднимается исключение, то оно будет перехвачно.

примечание Заметьте, что если в коде запуска до активации цикла сообщений поднять исключение, то, как правило, исключения не попадают в ловушку библиотеки, и программа просто завершится с ошибкой. Это поведение можно частично смягчить, добавив пользовательский блок `try-except` в основной программе. Инициализация кода библиотеки будет происходить до выполнения основной программы и запуска пользовательского блока `try-except`.

В общем случае если исключение поднято во время выполнения, что произойдет, зависит от библиотеки, но, как правило, существует программный способ перехвата этих исключений с помощью глобальных обработчиков или способ отображения сообщения об ошибке. Хотя некоторые детали отличаются, это справедливо как для VCL, так и для FireMonkey. В предыдущих примерах вы видели простое сообщение об ошибке, отображаемое при поднятии исключения.

Если вы хотите изменить это поведение, вы можете обработать событие OnException глобального объекта Application. Хотя эта операция в большей степени относится к визуальной библиотеке и обработке событий приложения, она также

связана с обработкой исключений, поэтому здесь стоит рассмотреть ее.

Я взял предыдущий проект приложения, названный ErrorLog, и добавил новый метод в основную форму:

```
public
    procedure LogException (Sender: TObject; E: Exception);
```

В обработчик событий `OnCreate` я добавил код для подключения метода к глобальному событию `OnException`, после чего написал реальный код глобального обработчика:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnException := LogException;
end;

procedure TForm1.LogException(Sender: TObject; E: Exception);
begin
  Show('Exception ' + E.Message);
end;
```

примечание Вы узнаете подробности того, как можно назначить указатель метода на событие (как я делал это выше) в следующей главе.

С помощью нового метода в глобальном обработчике исключений программа пишет на выходе сообщение об ошибке, не останавливая приложение с сообщением об ошибке.

Исключения и конструкторы

В теме исключений есть и более продвинутый вопрос, а именно, что происходит, когда исключение поднимается внутри конструктора объекта. Не все программисты Object Pascal знают, что при таких обстоятельствах будет вызван деструктор этого объекта (если он есть).

Это важно знать, так как это подразумевает, что деструктор может быть вызван для частично инициализированного объекта. Само собой разумеется, что внутренние объекты существуют в деструкторе, так как они создаются в конструкторе, что может привести к возникновению опасных ситуаций в случае возникновения реальных ошибок (т.е. к возникновению еще одного исключения перед обработкой первого).

Это также подразумевает, что правильная последовательность для `try-finally` должна включать в себя создание объекта вне блока, так как он автоматически защищается компилятором. Таким образом, если конструктор выходит из строя, то нет необходимости `Free` объект. Поэтому стандартным стилем кодирования в Object Pascal является защита объекта путем записи:

```
AnObject := AClass.Create;
try
  // use the object...
finally
  AnObject.Free;
end;
```

примечание Нечто похожее происходит и с двумя специальными методами класса TObject, AfterDestruction и BeforeConstruction, псевдо-конструктором и псевдо-деструктором, введенными для совместимости с C++ (но редко используемыми в Object Pascal). Обратите внимание, что если в методе `AfterConstruction` возникает исключение, то вызывается метод `BeforeDestruction` (а также обычный деструктор).

Учитывая, что я часто сталкивался с ошибками при правильном размещении объекта в деструкторе, позвольте подробнее остановиться на актуальном демо-примере, показывающем проблему… вместе с реальным исправлениями. Допустим, у вас есть класс со строковым списком, и вы пишете следующий код для создания и уничтожения класса (часть проекта `ConstructorExcept`):

```
type
  TObjectWithList = class
```

```
  private
    FStringList: TStringList;
  public
    constructor Create (Value: Integer);
    destructor Destroy; override;
  end;

constructor TObjectWithList.Create(Value: Integer);
begin
  if Value < 0 then
    raise Exception.Create('Negative value not allowed');

  FStringList := TStringList.Create;
  FStringList.Add('one');
end;

destructor TObjectWithList.Destroy;
begin
  FStringList.Clear;
  FStringList.Free;
  inherited;
end;
```

На первый взгляд, код кажется правильным. Конструктор выделяет подобъект, а деструктор правильно освобождает его. Более того, код вызова написан таким образом, что если исключение поднимается после конструктора, то вызывается метод Free, но если исключение находится в конструкторе, то ничего не происходит:

```
var
  Obj: TObjectWithList;
begin
  Obj := TObjectWithList.Create (-10);
  try
    // do something
  finally
    Show ('Freeing object');
    Obj.Free;
  end;
```

Сработает ли это? Ни в коем случае! Хотя здесь задействован такой код, когда перед созданием списка строк в конструкторе поднимается исключение, система тут же вызывает деструктор, который пытается очистить несуществующий список, что приводит к нарушению доступа или подобной ошибке.

Почему это случилось? Опять же, если перевернуть последовательность в конструкторе (сначала создать список строк, а потом поднять исключение), все работает правильно, потому что деструктору действительно нужно освободить список строк. Но это не настоящее исправление, а способ обхода. Всегда нужно защищать код деструктора так, чтобы он никогда не предполагал, что конструктор был полностью выполнен. Это пример:

```
destructor TObjectWithList.Destroy;
begin
  if Assigned (FStringList) then
  begin
    FStringList.Clear;
    FreeAndNil(FStringList);
  end;
  inherited;
end;
```

Расширенные возможности исключений

Это один из разделов книги, который вы, возможно, захотите пропустить при первом прочтении, если вы недостаточно хорошо знаете язык. Вы можете перейти к следующей главе и вернуться к этому разделу в будущем.

В заключительной части главы я расскажу о некоторых более продвинутых темах, связанных с обработкой исключений. Я расскажу о вложенных исключениях (`RaiseOuterException`) и перехвате исключений класса (`RaisingException`). Эти возможности не были частью ранних версий языка Object Pascal, и значительно расширяют мощность системы.

Вложенные исключения и Механизм Внутренних Исключений

Что случится, если вы поднимете исключение внутри обработчика исключений? Традиционный ответ заключается в том, что новое исключение заменит существующее, поэтому общепринятой практикой является объединение, по крайней мере, сообщений об ошибках, написание подобного кода (без какой-либо реальной реакции, показ только выражений, связанных с исключениями):

```
procedure TFormExceptions.ClassicReraise;
begin
  try
    // do something...
    raise Exception.Create('Hello');
  except on E: Exception do
    // try some fix...
    raise Exception.Create('Another: ' + E.Message);
  end;
end;
```

Данный код является частью прикладного проекта AdvancedExcept. При вызове метода и работе с исключением вы увидите единственное исключение с комбинированным сообщением:

```
procedure TFormExceptions.BtnTraditionalClick(
  Sender: TObject);
begin
  try
    ClassicReraise;
  except
    on E: Exception do
      Show ('Message: ' + E.Message);
  end;
end;
```

Вывод (вполне очевидный):

```
Message: Another: Hello
```

Теперь в Object Pascal существует общесистемная поддержка вложенных исключений. В обработчике исключения можно создавать и поднимать новое исключение и при этом сохранять текущий объект исключения, подключая его к новому исключению. Для этого класс Exception имеет свойство InnerException, ссылающееся на предыдущее исключение, и свойство `BaseException`, позволяющее получить доступ к первому исключению серии, так как вложенность исключений может быть рекурсивной. Это элементы класса `Exception`, связанные с управлением вложенными исключениями:

```
type
  Exception = class(TObject)
  private
    FInnerException: Exception;
    FAcquireInnerException: Boolean;
  protected
    procedure SetInnerException;
  public
    function GetBaseException: Exception; virtual;
    property BaseException: Exception read GetBaseException;
    property InnerException: Exception read FInnerException;
    class procedure RaiseOuterException(E: Exception); static;
    class procedure ThrowOuterException(E: Exception); static;
  end;
```

примечание Статические методы классов — это особая форма методов классов. Эта языковая возможность будет объяснена в Главе 12.

С точки зрения пользователя, для поднятия исключения с сохранением существующего, следует вызвать метод класса RaiseOuterException (или идентичный `ThrowOuterException`, использующий C++-ориентированное именование). При работе с подобным исключением можно использовать новые свойства для доступа к дополнительной информации. Обратите внимание, что вы можете вызвать `RaiseOuterException` только в обработчике исключения, как сказано в документации, *основанной на исходном коде*:

> *Используйте данную функцию, чтобы поднять экземпляр исключения из обработчика исключения, и вы*

хотите "приобрести" активное исключение и привязать его к новому исключению и сохранить контекст. Это приведет к тому, что поле `FInnerException` *будет установлено с выполняющимся в данный момент исключением.*

Вы должны вызывать эту процедуру только из блока исключений, где ожидается, что это новое исключение будет обработано в другом месте.

Для фактического примера можно обратиться к прикладному проекту AdvancedExcept. В этом примере я добавил метод, который поднимает вложенное исключение по-новому (по сравнению с описанным ранее методом `ClassicReraise`):

```
procedure TFormExceptions.MethodWithNestedException;
begin
  try
    raise Exception.Create ('Hello');
  except
    Exception.RaiseOuterException (
      Exception.Create ('Another'));
  end;
end;
```

Теперь в обработчике этого внешнего исключения мы можем получить доступ к обоим объектам исключения (а также увидеть эффект вызова нового метода `ToString`):

```
 try
    MethodWithNestedException;
  except
    on E: Exception do
    begin
      Show ('Message: ' + E.Message);
      Show ('ToString: ' + E.ToString);
      if Assigned (E.BaseException) then
        Show ('BaseException Message: ' +
          E.BaseException.Message);
      if Assigned (E.InnerException) then
        Show ('InnerException Message: ' +
          E.InnerException.Message);
    end;
  end;
```

Вывод этого вызова следующий:

```
Message: Another
ToString: Another
Hello
BaseException Message: Hello
InnerException Message: Hello
```

Существуют два момента, которые следует заметить. Первый заключается в том, что в случае одного вложенного исключения свойство `BaseException` и свойство `InnerException` относятся к одному и тому же объекту исключения, к исходному. Второй заключается в том, что если сообщение нового исключения содержит только действительное сообщение, то при вызове ToString вы получаете доступ к комбинированным сообщениям всех вложенных исключений, разделенных `sLineBreak` (как видно из кода метода Exception.`ToString`).

Выбор использования разрыва строки в этом случае производит странный вывод, но как только вы об этом узнаете, вы можете отформатировать его так, как вам нравится, заменив разрывы строк символом по вашему выбору или назначив их свойству `Text` в списке строк.

В качестве еще одного примера, позвольте мне показать вам, что происходит при поднятии двух вложенных исключений. Это новый метод:

```
procedure TFormExceptions.MethodWithTwoNestedExceptions;
begin
  try
    raise Exception.Create ('Hello');
  except
    begin
      try
        Exception.RaiseOuterException (
          Exception.Create ('Another'));
      except
        Exception.RaiseOuterException (
          Exception.Create ('A third'));
      end;
    end;
  end;
end;
```

Это вызывается методом, идентичным тому, который мы видели ранее, и производит следующий вывод:

```
Message: A third
ToString: A third
Another
Hello
BaseException Message: Hello
InnerException Message: Another
Внутреннее сообщение: Другой
```

На этот раз свойство `BaseException` и свойство InnerException относятся к разным объектам, а вывод `ToString` охватывает три строки.

Перехват исключения

Еще одной усовершенствованной функцией, добавленной со временем в исходную систему обработки исключений языка Object Pascal, является метод:

```
procedure RaisingException(P: PExceptionRecord); virtual;
```

Согласно документации по исходному коду:

> *Эта виртуальная функция будет вызвана непосредственно перед тем, как будет поднято это исключение. В случае внешнего исключения, оно вызывается вскоре после создания объекта, так как условие "поднять" уже выполняется.*

Реализация функции в классе `Exception` управляет внутренним исключением (путем вызова внутреннего `SetInnerException`), что, вероятно, объясняет, почему она была введена в первую очередь, одновременно с внутренним механизмом исключения.

В любом случае, теперь, когда у нас есть эта функция, мы можем ею воспользоваться. Переопределив этот метод, на самом деле, мы имеем единственную функцию выполняемую после конструктора, которая неизменно вызывается,

независимо от того, какой конструктор использовался для создания исключения. Другими словами, вы можете избежать определения пользовательского конструктора для вашего класса исключения и позволить пользователям вызывать один из множества конструкторов базового класса `Exception`, и при этом иметь собственное поведение. В качестве примера можно записать в журнал любое исключение данного класса (или подкласса).

Вот пользовательский класс исключений (снова определен в проекте приложения AdvancedExcept), который переопределяет метод `RaisingException`:

```
type
  ECustomException = class (Exception)
  protected
    procedure RaisingException(
      P: PExceptionRecord); override;
  end;

procedure ECustomException.
  RaisingException(P: PExceptionRecord);
begin
  // log exception information
  FormExceptions.Show('Exception Addr: ' + IntToHex (
    Integer(P.ExceptionAddress), 8));
  FormExceptions.show('Exception Mess: ' + Message);

  // modify the message
  Message := Message + ' (filtered)';

  // standard processing
  inherited;
end;
```

Реализация этого метода заключается в протоколировании некоторой информации об исключении, изменении сообщения об исключении и последующем вызове стандартной обработки базовых классов (необходимой для работы механизма вложенных исключений). Метод вызывается после создания объекта исключения, но до того, как исключение будет поднято. Это можно заметить, так как вывод, выдаваемый вызовами `Show`, генерируется до того, как исключение будет

перехвачено отладчиком! Аналогично, если поставить точку останова в методе `RaisingException`, то отладчик остановится на ней до того, как будет перехвачено исключение.

Опять же, вложенные исключения и этот механизм перехвата не часто используются в коде приложений, так как они являются особенностями языка, в большей степени предназначенными для разработчиков библиотек и компонентов.

10: Свойства и события

В последних трех главах я рассказал об основах ООП в Object Pascal, объяснив многие понятия и показав, как особенности, доступные в большинстве объектно-ориентированных языков, реализуются в частности. С первых дней существования Delphi, Object Pascal язык был полностью объектно-ориентированным языком, но со специфическими особенностями, *вкусом*. Фактически, он также состоялся в качестве языка визуального инструмента разработки, основанного на компонентах.

Это связанные черты: поддержка этой модели разработки основана на некоторых основных особенностях языка, таких как свойства и события, первоначально введенные в Object Pascal раньше любого другого языка, а затем частично скопированные несколькими ООП-языками. Свойства, например, могут быть найдены на Java и C#, среди других языков, но они имеют прямое происхождение от Object Pascal... хотя лично я предпочитаю оригинальную реализацию, как я объясню вкратце ниже.

Способность Object Pascal поддерживать быструю разработку приложений (RAD) и визуальное программирование является причиной таких концепций, как свойства, спецификатор

доступа published, события, концепция компонента и некоторые другие идеи, рассмотренные в этой главе.

Определение свойств

Что такое свойство (property)? Свойства можно описать как идентификаторы, которые позволяют получить доступ и изменить статус объекта - то, что может вызвать выполнение скрытого кода. В Object Pascal свойства абстрагируют и скрывают доступ к данным через поля или методы, что делает их основным путем реализации инкапсуляции. Одним из способов описания свойств является "инкапсуляция *по максимуму*".

Технически, свойство — это идентификатор с типом данных, который сопоставляется с некоторыми реальными данными с помощью некоторого спецификатора `read` и `write`. В отличие от Java или C#, в Object Pascal спецификатор чтения и записи может быть как методом *геттера* или *сеттера*, так и непосредственно полем.

Например, вот определение свойства для объекта даты с помощью общего подхода (читать из поля, писать методом):

```
private
  FMonth: Integer;
  procedure SetMonth(Value: Integer);
public
  property Month: Integer read FMonth write SetMonth;
```

Чтобы получить доступ к значению свойства `Month`, данный код должен прочитать значение частного поля `FMonth`, а чтобы изменить значение, он вызывает метод `SetMonth`. Код для изменения значения (защита от отрицательных значений) может быть чем-то вроде:

```
procedure TDate.SetMonth (Value: Integer);
begin
  if Value <= 0 then
    FMonth := 1
  else
    FMonth := Value;
end;
```

примечание В случае неправильного ввода, как и в случае отрицательного числа месяца, обычно лучше показать ошибку (подняв исключение), чем настраивать значение за кулисами, но я оставляю код *как есть* ради простой вводной демонстрации.

Обратите внимание, что тип данных поля и свойства должны точно совпадать (в случае расхождения можно использовать простой метод преобразования); аналогично тип единственного параметра процедуры сеттера или возвращаемое значение функции геттера должны точно совпадать с типом свойства.

Возможны различные комбинации (например, можно использовать метод для чтения значения или непосредственно изменять поле в директиве write), но наиболее распространенным является использование метода для изменения значения свойства. Вот несколько альтернативных реализаций для одного и того же свойства:

```
property Month: Integer read GetMonth write SetMonth;
property Month: Integer read FMonth write FMonth;
```

примечание Когда вы пишете код, который обращается к свойству, важно понимать, что метод может быть вызван. Проблема в том, что некоторые из этих методов требуют некоторого времени на выполнение; они также могут вызывать ряд побочных эффектов, часто включая (медленное) перерисовывание элемента управления на экране. Хотя побочные эффекты свойств редко документируются, вы должны знать, что они существуют, особенно когда вы пытаетесь оптимизировать код.

Директиву `write` свойства также можно опустить, сделав его свойством, доступным только для чтения:

```
property Month: Integer read GetMonth;
```

Технически можно также пропустить директиву `read` и определить свойство *только для записи*, но это обычно не имеет большого смысла и делается очень редко.

Свойства сравнительно с другими языками программирования

Если вы сравните это с Java или C#, то в обоих языках свойства отображаются с помощью методов, но в первом из них есть неявное отображение (свойства, по сути, являются конвенцией), в то время как во втором есть явное отображение, как в Object Pascal, пусть даже только с помощью методов:

```
// properties in Java language

private int mMonth;

public int getMonth() { return mMonth; }

public void setMonth(int value) {
  if (value <= 0)
    mMonth = 1;
  else
    mMonth = value;
}

int s = date.getMonth ();
date.setMonth (s+1);


// properties in C# language

private int mMonth;

public int Month {
  get { return mMonth; }
  set {
    if (value <= 0)
      mMonth = 1;
    else
      mMonth = value;
  }
}

date.Month++;
```

Не то, чтобы я не буду подробно обсуждать относительные достоинства свойств в различных языках программирования, но, как я уже упоминал во введении к этой главе, я думаю, что явное определение свойств является полезной идеей, а также

то, что дальнейший уровень абстракции, получаемый при отображении свойств на поля без дополнительной нагрузки на метод, является очень хорошим дополнением. Поэтому я предпочитаю Object Pascal реализацию свойств по сравнению с другими языками.

Свойства — это очень надежный механизм ООП, очень хорошо продуманное применение идеи инкапсуляции. По сути, у вас есть имя, которое скрывает реализацию того, как получить доступ к информации класса (прямой доступ к данным или вызов метода).

На самом деле, используя свойства, вы получаете интерфейс, который вряд ли изменится. В то же время, если вы хотите разрешить пользователям доступ только к некоторым полям вашего класса, вы можете легко обернуть эти поля в свойства вместо того, чтобы делать их общедоступными. Вам не требуется кода для реализации (кодирование простых методов `Get` и `Set` ужасно скучно), и вы все равно можете изменить реализацию вашего класса. Даже если вы замените прямой доступ к данным на доступ, основанный на методе, вам вообще не придется менять исходный код, использующий эти свойства. Вам останется только перекомпилировать его. Подумайте об этом, как о концепции инкапсуляции, поднятой на максимальную мощность!

примечание Вы можете задаться вопросом, если свойство определено с прямым доступом к частной переменной, не устраняет ли это одно из преимуществ инкапсуляции? Пользователь не может быть защищен от любого изменения типа данных приватной переменной, в то время как они могут быть с геттерами и сеттерами. Однако, учитывая, что пользователь будет получать доступ к данным через свойство, разработчик класса может в любой момент изменить базовый тип данных и ввести геттеры и сеттеры, не затрагивая при этом код, использующий его. Поэтому я назвал это "инкапсуляцией по максимуму". С другой стороны, это показывает прагматичную сторону Object Pascal, в том, что она позволяет программисту выбирать любой более легкий способ (и быстрое выполнение кода), где это соответствует обстоятельствам, и плавный переход к "правильной ООП", когда это необходимо.

Однако есть одна оговорка в использовании свойств в Object Pascal. Обычно вы можете присвоить значение свойству или прочитать его, и вы можете свободно использовать свойства в выражениях. Однако, вы не можете передать свойство в качестве параметра ссылки на процедуру или метод. Это связано с тем, что свойство является не ячейкой памяти, а абстракцией, поэтому его нельзя использовать в качестве ссылочного параметра (`var`). В качестве примера, в отличие от C#, нельзя вызывать Inc для свойства.

примечание Сопутствующая функция, передающая свойства по ссылке, описывается далее в этой главе. Однако, это слабо используемая возможность, которая требует включения конкретной настройки компилятора, и уж точно не является основной.

Code Completion и свойства

Если добавление свойств в класс может показаться утомительной работой, то редактор IDE позволяет легко *автосоздавать код* свойства при написании начальной части объявления свойств (внутри класса), с помощью *завершения код*а как показано ниже:

```
type
  TMyClass = class
  public
    property Month: Integer;
  end;
```

Нажмите комбинацию клавиш Ctrl+Shift+C, пока курсор находится в объявлении свойства, и вы получите новое поле, добавленное в класс вместе с новым методом сеттера, с правильным отображением в определении свойства и полной реализацией метода сеттера, с основным кодом для изменения значения поля. Другими словами, код, приведенный выше, становится с помощью клавиатурной комбинации (или соответствующего пункта локального меню редактора) таким:

```
type
 TMyClass = class
  private
    FMonth: Integer;
    procedure SetMonth(const Value: Integer);
  public
    property Month: Integer read FMonth write SetMonth;
  end;

{ TMyClass }
procedure TMyClass.SetMonth(const Value: Integer);
begin
  FMonth := Value;
end;
```

Если вам нужен еще и метод геттера, замените `read` часть определения на `GetMonth`, например, как в `GetMonth`:

```
property Month: Integer read GetMonth write SetMonth;
```

Теперь снова нажмите Ctrl+Shift+C и функция тоже будет добавлена, но на этот раз без предопределенного кода для доступа к значению:

```
function TMyClass.GetMonth: Integer;
begin

end;
```

Добавление свойств в формы

Рассмотрим конкретный пример инкапсуляции с использованием свойств. Вместо того, чтобы строить пользовательский класс, на этот раз я модифицирую класс формы, который IDE генерирует для каждой создаваемой вами визуальной формы... а также воспользуюсь преимуществом Class Completion (Завершение класса).

Когда приложение имеет несколько форм, часто бывает удобно получить доступ к информации одной формы из другой. Может возникнуть соблазн добавить открытое поле, но это неизменно плохая идея. Каждый раз, когда вы хотите сделать некоторую

информацию из формы доступной для других форм, вы должны использовать свойство.

Просто запишите в декларацию класса формы имя и тип свойства:

```
property Clicks: Integer;
```

Затем нажмите Ctrl+Shift+C, чтобы активировать *Завершение Кода*. Вы увидите следующий эффект:

```
type
  TFormProp = class(TForm)
  private
    FClicks: Integer;
    procedure SetClicks(const Value: Integer);
  public
    property Clicks: Integer
      read FClicks write SetClicks;
  end;

implementation

procedure TForm1.SetClicks(const Value: Integer);
begin
  FClicks := Value;
end;
```

Нет нужды говорить, что это избавляет от необходимости печатать. Теперь, когда пользователь кликает на форму, вы можете увеличить счетчик кликов, написав следующую строку, как я сделал в событии `OnMouseDown` на форме проекта приложения FormProperties:

```
Clicks := Clicks + 1;
```

Вы можете задаться вопросом, как насчет прямого увеличения `FClicks`? Ну, в этом конкретном сценарии это может сработать, но вы также можете использовать метод `SetClicks` для обновления пользовательского интерфейса и отображения текущего значения. Если вы обойдете свойство и напрямую получите доступ к полю, дополнительный код в методе Setter, используемый для обновления пользовательского интерфейса, не будет выполнен, и отображение может выйти из синхронизации.

Другое преимущество этой инкапсуляции заключается в том, что из другой формы вы сможете ссылаться на количество кликов правильно абстрагированным образом. Фактически, свойства в классах формы могут быть использованы для доступа к пользовательским значениям, а также для инкапсуляции доступа к компонентам формы. Например, если у вас есть форма с меткой, используемой для отображения некоторой информации, и вы хотите изменить текст из вторичной формы, у вас может возникнуть соблазн написать:

```
Form1.StatusLabel.Text := 'new text';
```

Это обычная практика, но она не является хорошей, потому что не обеспечивает никакой инкапсуляции структуры формы или ее компонентов. Если у вас подобный код во многих местах в приложении, и вы позже решите изменить пользовательский интерфейс формы (заменив объект `StatusLabel` на другой элемент управления), вам придется исправлять код во многих местах.

Альтернативой является использование метода или, что еще лучше, свойства, чтобы скрыть конкретный элемент управления. Вы можете последовать указанным выше шагам, чтобы добавить свойство, как с методом чтения, так и с методом записи, или ввести их полностью, вроде такого:

```
property StatusText: string read GetStatusText write SetStatusText;
```

и снова нажмите комбинацию Ctrl+Shift+C, чтобы редактор добавил определение обоих методов для чтения и записи свойства:

```
function TFormProp.GetStatusText: string;
begin
  Result := LabelStatus.Text
end;

procedure TFormProp.SetStatusText(const Value: string);
begin
  LabelStatus.Text := Value;
end;
```

Обратите внимание, что в этом случае свойство сопоставляется не с локальным полем класса, а с полем подобъекта, меткой (в случае, если вы использовали автоматическую генерацию кода, не забудьте фактически удалить свойство `FStatusText`, которое редактор мог добавить от вашего имени).

В других формах программы можно просто обратиться к свойству `StatusText` формы, и если пользовательский интерфейс изменится, то будут затронуты только методы `Set` и `Get` свойства. Также вы можете сделать то же самое внутри оригинальной формы, сделав код двух свойств более независимым:

```
procedure TFormProp.SetClicks(const Value: Integer);
begin
  FClicks := Value;
  StatusText := FClicks.ToString + ' clicks';
end;
```

Добавление свойств в класс TDate

В главе 7 я построил класс `TDate`. Теперь мы можем расширить его с помощью свойств. Этот новый проект приложения, DateProperties, по сути, является расширением прикладного проекта ViewDate из главы 7. Вот новое объявление класса. В нем есть несколько новых методов (используемых для установки и получения значений свойств) и четыре свойства:

```
type
  TDate = class
  private
    FDate: TDateTime;
    function GetYear: Integer;
    function GetDay: Integer;
    function GetMonth: Integer;
    procedure SetDay (const Value: Integer);
    procedure SetMonth (const Value: Integer);
    procedure SetYear (const Value: Integer);
  public
    constructor Create; overload;
    constructor Create (Y, M, D: Integer); overload;
    procedure SetValue (Y, M, D: Integer); overload;
```

```
    procedure SetValue (NewDate: TDateTime); overload;
    function LeapYear: Boolean;
    procedure Increase (NumberOfDays: Integer = 1);
    procedure Decrease (NumberOfDays: Integer = 1);
    function GetText: string; virtual;
    property Day: Integer read GetDay write SetDay;
    property Month: Integer read GetMonth write SetMonth;
    property Year: Integer read GetYear write SetYear;
    property Text: string read GetText;
  end;
```

Свойства `Year`, `Day`, и `Month` читают и записывают свои значения соответствующими методами. Вот два из них, относящиеся к свойству `Month`:

```
function TDate.GetMonth: Integer;
var
  Y, M, D: Word;
begin
  DecodeDate (FDate, Y, M, D);
  Result := M;
end;

procedure TDate.SetMonth(const Value: Integer);
begin
  if (Value < 1) or (Value > 12) then
    raise EDateOutOfRange.Create ('Invalid month');
  SetValue (Year, Value, Day);
end;
```

Вызов `SetValue` выполняет фактическую кодировку даты, вызывая исключение в случае ошибки. Я определил пользовательский класс исключения, который поднимается каждый раз, когда значение выходит за пределы диапазона:

```
type
  EDateOutOfRange = class (Exception);
```

Четвертое свойство, `Text`, привязывается только к методу чтения. Эта функция объявлена виртуальной, так как заменяется подклассом `TNewDate`. Нет причин, по которым метод свойства `Get` или `Set` не должен использовать привязку к позднему времени (подробное объяснение этого свойства приведено в главе 8).

примечание Важно отметить в этом примере, что свойства не привязываются непосредственно к данным. Они просто вычисляются из информации, хранящейся в другом типе и с иной структурой, чем кажется на первый взгляд.

Обновив класс новыми свойствами, мы теперь можем обновить пример для использования свойств при необходимости. Например, мы можем использовать свойство `Text` напрямую, и мы можем использовать некоторые поля редактирования, чтобы дать пользователю возможность прочитать или записать значения трех основных свойств. Это происходит при нажатии кнопки `BtnRead`:

```
procedure TDateForm.BtnReadClick(Sender: TObject);
begin
  EditYear.Text := IntToStr (TheDay.Year);
  EditMonth.Text := IntToStr (TheDay.Month);
  EditDay.Text := IntToStr (TheDay.Day);
end;
```

Кнопка `BtnWrite` выполняет обратную операцию. Код можно написать одним из двух следующих способов:

```
// direct use of properties
TheDay.Year := StrToInt (EditYear.Text);
TheDay.Month := StrToInt (EditMonth.Text);
TheDay.Day := StrToInt (EditDay.Text);

// update all values at once
TheDay.SetValue (StrToInt (EditMonth.Text),
  StrToInt (EditDay.Text),
  StrToInt (EditYear.Text));
```

Разница между этими двумя подходами связана с тем, что происходит, когда вход не соответствует действительной дате. Когда мы устанавливаем каждое значение отдельно, программа может изменить год, а затем поднять исключение и пропустить выполнение остального кода, так что дата будет изменена только частично. Когда мы устанавливаем все значения сразу, либо они корректны и все установлены, либо одно из них недействительно, а объект даты сохраняет исходное значение.

Использование массива свойств

Свойства обычно позволяют получить доступ к одному значению, даже если это один из сложных типов данных. Object Pascal также определяет массивы свойств, или индексаторы, как они называются в C#. Массив свойств— это свойство с дополнительным параметром любого типа данных, который используется в качестве индекса или (более широко) селектора актуального значения.

Приведем пример определения массива свойств, использующего целочисленный индекс и ссылающегося на целое значение:

```
private
   function GetValue(I: Integer): Integer;
   procedure SetValue(I: Integer; const Value: Integer);
 public
   property Value [I: Integer]: Integer read GetValue write SetValue;
```

Массив свойств должен быть связан с методами чтения и записи, которые имеют дополнительный параметр, представляющий индекс... и вы по-прежнему можете использовать Code Completion для определения методов как для обычного свойства. Существует множество комбинаций значений и индексов и несколько классов в RTL делают массивы свойств очень полезными. Например, класс TStrings определяет 5 из них:

```
    property Names[Index: Integer]: string read GetName;
   property Objects[Index: Integer]: TObject
     read GetObject write PutObject;
   property Values[const Name: string]: string
     read GetValue write SetValue;
   property ValueFromIndex[Index: Integer]: string
     read GetValueFromIndex write SetValueFromIndex;
   property Strings[Index: Integer]: string
     read Get write Put; default;
```

В то время как большинство этих свойств используют индекс строки в качестве параметра в списке, другие используют

строку в качестве значения поиска или поиска (как свойство `Values` выше). Последнее из этих определений использует другую важную особенность: оно помечено ключевым словом «по умолчанию» (default). Это мощный помощник синтаксиса: имя свойства-массива может быть опущено, так что вы можете применить оператор квадратных скобок непосредственно к рассматриваемому объекту. Таким образом, если у вас есть объект `SList` этого типа `TStrings`, оба следующих оператора будут читать одно и то же значение:

```
SList.Strings[1]
SList[1]
```

Другими словами, свойства массива по умолчанию предлагают способ определения пользовательского оператора `[ ]` для любого объекта.

Установка значения свойств по ссылке

Это довольно продвинутая тема (и редко используемая функция), которую, вероятно, следует пропустить, если у вас еще нет опыта работы с Object Pascal. Но даже, если он есть, есть шанс, что вы никогда не слышали о такой возможности.

В то время компилятор Object Pascal был расширен для непосредственной поддержки COM-программирования в Windows, он получил возможность работать со свойствами "put by ref" (на COM жаргоне) или со свойствами, которые могут получать не значение, а ссылку.

примечание "Put by ref" - это имя, которое Крис Бенсен дал этой функции в своем посте в блоге, представляющем ее: http://chrisbensen.blogspot.com/2008/04/delphi-put-by-ref-properties.html (в то время Крис был инженером-разработчиком этого продукта).

Это достигается с помощью параметра *var* в методе сеттера. Учитывая, что это может привести к довольно неудобным сценариям, эта функция (хотя и является частью языка) рассматривается скорее, как исключение, чем как правило, поэтому по умолчанию она не активна. Другими словами, чтобы включить эту возможность, необходимо специально запросить ее с помощью директивы компилятора:

```
{ $VARPROPSETTER ON}
```

Без этой директивы следующий код не скомпилируется и выдаст ошибку *"E2282 Property Setters cannot take var parameters*":

```
type
  TMyIntegerClass = class
  private
    FNumber: Integer;
    function GetNumber: Integer;
    procedure SetNumber(var Value: Integer);
  public
    property Number: Integer
      read GetNumber write SetNumber;
  end;
```

Данный класс является частью проекта приложения VarProp. И что необычно, вы можете создавать побочные эффекты внутри сеттера свойств:

```
procedure TMyIntegerClass.SetNumber(var Value: Integer);
begin
  Inc (Value); // side effect
  FNumber := Value;
end;
```

Другой очень необычный эффект заключается в том, что свойству нельзя присвоить постоянную величину, только переменную (чего следует ожидать, как и при любом вызове с параметром, переданным по ссылке):

```
var
  Mic: TMyIntegerClass;
  N: Integer;
begin
  ...
  Mic.Number := 10; // Error: E2036 Variable required
  Mic.Number := N;
```

Хотя это и не функция, которую вы регулярно используете, но это довольно продвинутый способ работы со свойством, позволяющий инициализировать или изменять присвоенное ему значение. Это может привести к крайне странному коду, например:

```
N := 10;
Mic.Number := N;
Mic.Number := N;
Show(Mic.Number.ToString);
```

Два последовательных идентичных присвоения выглядят довольно странно, но они действительно производят побочный эффект, превращая фактическое число в 12. Наверное, это самый запутанный и бессмысленный способ получить такой результат!

Спецификатор доступа published

Наряду с директивами `public`, `protected` и `private` доступа (а также менее часто используемыми `strict private` и `strict protected`), язык Object Pascal имеет еще один очень специфический, называемый `published (опубликованный)`. Свойство `published` (или поле или метод) не только доступно во время выполнения, как публичное, но и генерирует расширенную информацию о времени выполнения (RTTI), которую можно запросить.

В скомпилированном языке скомпилированные символы обрабатываются компилятором и могут быть использованы отладчиком при тестировании приложения, но, как правило, не оставляют следов во время выполнения. Другими словами (по крайней мере, в ранние времена Object Pascal), если класс

имеет свойство под названием `Name`, вы можете использовать его в своем коде для взаимодействия с классом, но у вас нет возможности выяснить, имеет ли класс свойство, совпадающее с заданной строкой, *"Name"*.

примечание И язык Java, и язык C# - это скомпилированные языки, которые используют преимущества сложной виртуальной среды исполнения, и по этой причине они обладают обширной информацией о времени исполнения, обычно называемой reflection. Язык Object Pascal ввел рефлексию (также называемую расширенной RTTI) спустя несколько лет, так что в нем все еще есть как некоторые базовые RTTI, привязанные к опубликованному ключевому слову, рассмотренному в этой главе, так и более полную форму рефлексии, описанную в Главе 16.

Почему необходима дополнительная информация о классе? Это одна из основ компонентной модели и визуальной модели программирования, на которую опираются библиотеки Object Pascal. Часть этой информации используется во время проектирования в среде разработки, чтобы заполнить Object Inspector списком свойств, предоставляемых компонентом. Это не жестко закодированный список, он генерируется во время выполнения скомпилированного кода.

Другим примером, возможно, слишком сложным, чтобы вникнуть в него сейчас, является механизм потокового воспроизведения файлов FMX и DFM, а также любых сопровождающих их визуальных форм. Потоковая передача будет представлена только в Главе 18, так как она в большей степени является частью библиотеки времени исполнения, чем языка ядра.

Подводя итог концепции, важно регулярно использовать ключевое слово `published`, когда вы пишете компоненты для использования вами или другими людьми в среде разработки. Обычно `published` части компонента содержат только свойства, в то время как классы формы также используют опубликованные поля и методы, о чем я расскажу позже.

Свойства при проектировании

Ранее в этой главе мы видели, что свойства играют важную роль для инкапсуляции данных класса. Они также играют фундаментальную роль в создании визуальной модели развития. Фактически, можно написать класс компонента, сделать его доступным в среде разработки, создать объект, добавив его в форму или подобную поверхность проектирования, и взаимодействовать с его свойствами с помощью Object Inspector. Не все свойства могут быть использованы в данном сценарии, только те, которые помечены как опубликованные в классе компонента. Поэтому программисты Object Pascal делают различие между свойствами времени разработки и *только времени исполнения*.

Свойствами Design-time являются свойства, объявленные в опубликованном разделе декларации класса, и могут быть использованы во время проектирования в IDE, а также в коде. Любое другое свойство, объявленное в `секции public` класса, недоступно во время проектирования, а только в коде, и его часто называют *только временем исполнения*.

Другими словами, вы можете увидеть значение и изменить значения `published` свойств во время проектирования с помощью Object Inspector. Это инструмент, который предоставляет визуальная среда программирования для доступа к свойствам. Во время выполнения можно получить доступ к любому публичному или `published` свойству другого класса точно так же, прочитав или записав его значение в коде.

Не все классы обладают свойствами. Свойства присутствуют в компонентах и других подклассах класса **TPersistent**, так как свойства обычно могут быть переданы и сохранены в файл. Файл формы, на самом деле, является ничем иным, как набором опубликованных свойств компонентов на форме.

Более точно, для поддержки концепции опубликованной секции не нужно наследовать от `TPersistent`, но нужно скомпилировать класс с директивой компилятора **$M**. Каждый класс, скомпилированный с помощью этой директивы или полученный из класса, скомпилированного с ее помощью, поддерживает опубликованную секцию. Учитывая, что `TPersistent` компилируется с данной настройкой, любой класс, производный от этого, имеет такую поддержку.

примечание Следующие два раздела, посвященные видимости по умолчанию и автоматическому RTTI, содержат дополнительную информацию о влиянии директивы $M и опубликованного ключевого слова.

Published и формы

Когда IDE генерирует форму, она помещает определения своих компонентов и методов в начальную часть своего определения, перед ключевыми словами `public` и `private`. Эти поля и методы начальной части класса являются `published`. `Публикация` по умолчанию происходит в том случае, когда перед элементом класса компонента не добавляется специальное ключевое слово.

примечание Точнее, используется ключевое слово published по умолчанию только в том случае, если класс был скомпилирован с помощью директивы компилятора $M+ или происходит от класса, скомпилированного с помощью `$M+`. Поскольку эта директива используется в классе TPersistent, большинство классов библиотеки и все классы компонентов по умолчанию published. Однако некомпонентные классы (такие как `TStream` и `TList`) по умолчанию компилируются с `$M-` и по умолчанию public.

Вот пример:

```
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    BtnTest: TButton;
```

Методы, назначенные любому событию, должны быть `published`, а поля, соответствующие вашим компонентам в форме, должны `быть published` для автоматического связывания с объектами, описанными в файле формы и созданы вместе с формой. В Object Inspector (в списке компонентов формы или в списке доступных методов, отображаемом при выборе выпадающего списка для события) могут отображаться только те компоненты и методы, которые присутствуют в изначально `published` части объявления формы.

Почему компоненты класса должны быть объявлены с опубликованным полем, в то время как они могут быть приватными и лучше следовать правилам инкапсуляции ООП? Причина заключается в том, что эти компоненты создаются путем чтения их потокового представления, и после их создания их необходимо присвоить соответствующим полям формы.

Это делается с использованием RTTI, генерируемого для опубликованных полей (который изначально был единственным видом RTTI, доступным в Object Pascal, пока не была введена расширенная RTTI, описанная в Главе 16).

примечание Технически использование published полей для компонентов не является обязательным. Вы можете сделать свой код более ООП-разборчивым, сделав его приватным. Однако для этого требуется дополнительный код для выполнения. Я объясню это немного подробнее в последнем разделе этой главы, "RAD и ООП".

Автоматическая RTTI

Еще одним особенным поведением компилятора Object Pascal является то, что при добавлении опубликованного ключевого слова published в класс, который не наследуется от TPersistent, компилятор автоматически включит генерацию RTTI, автоматически добавляя поведение {$M+}.

Предположим, у имеется такой класс:

```
type
  TMyTestClass = class
  private
    FValue: Integer;
    procedure SetValue(const Value: Integer);
  published
    property Value: Integer read FValue write SetValue;
  end;
```

Компилятор показывает предупреждение типа:

```
[dcc32 Warning] AutoRTTIForm.pas(27): W1055 PUBLISHED caused RTTI ($M+)
to be added to type 'TMyTestClass'
```

Происходит то, что компилятор автоматически инжектирует директиву {$M+} в код, как видно из проекта AutoRTTI, в который входит приведенный выше код. В этой программе можно написать следующий код, который динамически обращается к свойству (используя старомодный модуль System.TypInfo):

```
uses
  TypInfo;

procedure TFormAutoRtti.BtnTetClick(Sender: TObject);
var
  Test1: TMyTestClass;
begin
  Test1 := TMyTestClass.Create;
  try
    Test1.Value := 22;
    Memo1.Lines.Add (GetPropValue (Test1, 'Value'));
  finally
    Test1.Free;
  end;
end;
```

примечание Хотя время от времени я буду использовать модуль `TypInfo` и такие функции, как GetPropValue, определенные в нем, реальная сила доступа к RTTI обеспечивается более современным модулем `RTTI` и его расширенной поддержкой reflection. Учитывая, что это довольно сложная тема, я посчитал важным посвятить ей отдельную главу, а также выделить два *варианта* RTTI, которые поддерживает Object pascal.

Программирование, ориентированное на события

В библиотеке, основанной на компонентах (но также и во многих других сценариях), написанный вами код представляет собой не просто плоскую последовательность действий, а, в основном, набор реакций. Под этим я подразумеваю, что вы определяете, как приложение должно "реагировать", когда что-то происходит. Это "что-то" может быть пользовательской операцией, такой как нажатие на кнопку, системная операция, изменение состояния датчика, некоторые данные становятся доступными через удаленное соединение, или что-то еще.

Эти внешние или внутренние триггеры действий обычно называются событиями. События изначально были отображением message-oriented операционных систем, таких как Windows, но прошли долгий путь по сравнению с этой первоначальной концепцией. В современных библиотеках, на самом деле, большинство событий запускается внутренне, когда вы устанавливаете свойства, вызываете методы или взаимодействуете с заданным компонентом (или косвенно с другим).

Как события и событийно-ориентированное программирование связаны с ООП? Эти два подхода отличаются друг от друга в отношении того, когда и как вы создаете новый унаследованный класс.

В чистом виде объектно-ориентированного программирования, всякий раз, когда объект имеет иное поведение (или другой

метод), чем другой, он должен принадлежать к другому классу. Мы видели это в нескольких демо-примерах.

Теперь давайте рассмотрим такой сценарий. Форма имеет четыре кнопки. Каждая кнопка требует различного поведения при ее нажатии. Таким образом, в терминах чистой ООП, у вас должно быть четыре различных подкласса кнопок, каждый из которых имеет свою версию метода "*клика*". Такой подход формально корректен, но для его написания и сопровождения потребовалось бы много дополнительного кода, что увеличило бы сложность.

Событийно-ориентированное программирование рассматривает похожий сценарий и предлагает разработчику добавить некоторое поведение к кнопочным объектам, которые относятся к одному классу. Поведение становится украшением или расширением статуса объекта, не требуя нового класса. Схема также называется делегированием, так как поведение объекта делегируется методу класса, отличного от собственного класса объекта.

События по-разному реализуются разными языками программирования, например:

Использование ссылок на методы (называемые указателями на методы, как в Object Pascal) или на объекты событий с внутренним методом (как происходит в C#)

Выделение кода событий в специализированный класс, реализующий интерфейс (как это обычно происходит в Java).

Использование замыканий, как это обычно происходит в JavaScript (подход Object Pascal также поддерживается анонимными методами, описанными в главе 15), хотя в JavaScript все методы являются замыканиями, поэтому различия между этими двумя понятиями немного размыты в этом языке.

Понятие событийного и событийно-ориентированного программирования получило широкое распространение и поддерживается многими различными языками программирования и библиотеками пользовательского интерфейса. Однако способ, которым Delphi реализует поддержку событий, достаточно уникален. В следующем разделе подробно описана технология, лежащая в ее основе.

Указатели на методы

В последней части главы 4 мы видели, что язык имеет концепцию указателей на функции. Это переменная, содержащая адрес области памяти функции, которую можно использовать для косвенного вызова функции. Указатель функции объявляется со специальной сигнатурой (в виде набора типов параметров и типа возврата, если таковой имеется).

Точно так же в языке есть понятие указателей на метод. Указатель метода — это ссылка на ячейку памяти метода, принадлежащую классу. Как и тип указателя функции, тип указателя метода имеет специфическую сигнатуру. Однако указатель метода несет в себе дополнительную информацию, то есть объект, к которому будет применяться метод (или объект, который будет использоваться в качестве параметра Self при вызове метода).

Другими словами, указатель метода — это ссылка на метод (по определенному адресу памяти) для одного конкретного объекта в памяти. При присваивании значения указателю на метод, необходимо обращаться к методу данного объекта, т.е. к методу конкретного экземпляра!

примечание Лучше понять реализацию указателя на метод можно, если посмотреть на определение структуры данных, часто используемой на низком уровне для выражения этой конструкции, которая называется TMethod. Эта запись имеет два поля `Code` и `Data`, представляющие соответственно адрес метода и объект, к которому он будет применен. В других подобных языках ссылка на код перехватывается классом делегата (C#) или методом интерфейса (Java).

Объявление типа указателя метода аналогично объявлению процедурного типа, за исключением того, что в конце объявления указаны ключевые слова `of object`:

```
type
  IntProceduralType = procedure (Num: Integer);
  TStringEventType = procedure (const S: string) of object;
```

Когда вы объявили указатель метода, как, например, выше, вы можете объявить переменную этого типа и присвоить ей совместимый метод любого объекта. Что такое совместимый метод? Тот, который имеет те же самые параметры, что и тот, который запрашивается указателем метода, например, один строковый параметр в приведенном выше примере.

примечание Ссылка на метод любого объекта может быть присвоена указателю метода при условии его *совместимости с типом указателя метода*.

Теперь, когда у вас есть тип указателя на метод, вы можете объявить переменную этого типа и присвоить ей совместимый метод:

```
type
  TEventTest = class
  public
     procedure ShowValue (const S: string);
     procedure UseMethod;
  end;

procedure TEventTest.ShowValue (const S: string);
begin
  Show (S);
end;

procedure TEventTest.UseMethod;
var
  StringEvent: TStringEventType;
begin
  StringEvent := ShowValue;
  StringEvent ('Hello');
```

```
end;
```

Пока этот простой код не совсем объясняет полезность событий, так как он ориентирован на концепцию низкоуровневых типов указателей метода. События основаны на этой технической реализации, но выходят за ее рамки, сохраняя указатель метода в одном объекте (скажем, кнопке) для обращения к методу другого объекта (скажем, к форме с обработчиком OnClick для кнопки). В большинстве случаев события также реализуются с использованием свойств.

примечание Хотя это встречается гораздо реже, в Object Pascal вы также можете использовать анонимные методы для определения обработчика события. Причина, по которой это менее распространено, вероятно, в том, что эта возможность была введена в язык сравнительно недавно, и на тот момент уже существовало много библиотек. Более того, это добавляет немного дополнительной сложности. Пример такого подхода можно найти в Гл. 15. Другим возможным расширением является определение нескольких обработчиков для одного события, например, поддержка C#, которая является не стандартной функцией, а той, которую вы могли бы реализовать самостоятельно.

Концепция делегирования

На первый взгляд, цель этой техники может быть неясной, но это один из краеугольных камней технологии компонентов Object Pascal. Секрет в слове "*делегирование*". Если кто-то создал объект, у которого есть указатели на методы, то вы вольны изменить поведение объекта, просто назначив указателям новые методы. Звучит знакомо? Должно.

Когда вы добавляете обработчик события OnClick для кнопки, среда разработки делает именно это. Кнопка имеет указатель на метод, названный `OnClick`, и вы можете прямо или косвенно назначить ей метод формы. Когда пользователь нажимает кнопку, этот метод выполняется, даже если вы определили его внутри другого класса (обычно в форме).

Далее следует фрагмент, который описывает код, фактически используемый в библиотеках Delphi для определения обработчика события компонента кнопки и связанного с ним метода формы:

```
type
  TNotifyEvent = procedure (Sender: TObject) of object;

  TMyButton = class
    OnClick: TNotifyEvent;
  end;

  TForm1 = class (TForm)
    procedure Button1Click (Sender: TObject);
    Button1: TMyButton;
  end;

var
  Form1: TForm1;
```

Теперь, внутри процедуры, вы можете написать

```
MyButton.OnClick := Form1.Button1Click;
```

Единственным реальным отличием этого фрагмента кода от кода библиотеки является то, что OnClick является именем свойства, а фактические данные, на которые он ссылается, называются `FOnClick`. Событие, которое появляется на странице События Инспектора объектов, на самом деле является не более чем свойством, являющимся указателем на метод. Это означает, например, что вы можете динамически изменять обработчик события, прикрепленный к компоненту во время проектирования, или даже создавать новый компонент во время выполнения и назначать ему обработчик события.

Проект приложения DynamicEvents демонстрирует оба сценария. Форма имеет кнопку со стандартным обработчиком событий `OnClick`, связанным с ней. Однако я добавил второй публичный метод к форме с той же самой сигнатурой (те же самые параметры):

```
   public
    procedure BtnTest2Click(Sender: TObject);
```

При нажатии кнопки, наряду с отображением сообщения, она переключает обработчик события на второй, изменяя будущее поведение операции щелчка:

```
procedure TForm1.BtnTestClick(Sender: TObject);
begin
  ShowMessage ('Test message');
  BtnTest.OnClick := BtnTest2Click;
end;

procedure TForm1.BtnTest2Click(Sender: TObject);
begin
  ShowMessage ('Test message, again');
end;
```

Теперь при первом нажатии кнопки выполняется первый (по умолчанию) обработчик события, в то время как в любой другой раз выполняется второй обработчик события.

примечание При вводе кода для назначения метода событию, инструмент завершение кода предложит вам доступное имя события и превратит его в реальный вызов функции, заключенный в круглые скобки в конце. Это неправильно. Необходимо присвоить событию сам метод, не вызывая его. Иначе компилятор попытается присвоить результат вызова метода (которого, как процедуры, не существует), что приведет к ошибке.

Вторая часть проекта демонстрирует полностью динамическое назначение событий. При нажатии на поверхность формы динамически создается новая кнопка с обработчиком события, который показывает надпись связанной с ней кнопки (объект `Sender`):

```
procedure TForm1.BtnNewClick(Sender: TObject);
begin
  ShowMessage ('You selected ' + (Sender as TButton).Text);
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Single);
var
  AButton: TButton;
begin
  AButton := TButton.Create(Self);
  AButton.Parent := Self;
  AButton.SetBounds(X, Y, 100, 40);
  Inc (FCounter);
  AButton.Text := 'Button' + IntToStr (FCounter);
  AButton.OnClick := BtnNewClick;
```

```
end;
```

С помощью этого кода каждая из динамически создаваемых кнопок будет реагировать на щелчок мыши, показывая сообщение, зависящее от кнопки, даже если используется один обработчик события, благодаря использованию параметра Sender события. Пример вывода показан на рисунке 10.1.

Рисунок 10.1: Сообщение, отображаемое динамически созданными кнопками в проекте приложения DynamicEvents.

События являются свойствами

Очень важной концепцией является то, что события в Object Pascal практически всегда реализуются в виде свойств типа указателя метода. Это означает, что для обработки события компонента необходимо присвоить метод соответствующему свойству события. С точки зрения кода это означает, что обработчику события можно присвоить метод объекта, используя код, подобный тому, что мы уже видели в предыдущем разделе:

```
Button1.OnClick := ButtonClickHandler;
```

Опять же, правило заключается в том, что тип указателя метода события должен совпадать с сигнатурой метода, который вы назначаете, иначе компилятор выдаст ошибку. Система определила несколько стандартных типов указателей на события метода, которые обычно используются, начиная с простого:

```
type
  TNotifyEvent = procedure (Sender: TObject) of object;
```

Обычно это тип обработчика события OnClick, поэтому это подразумевает, что вышеприведенный метод должен быть объявлен (внутри класса) как:

```
procedure ButtonClickHandler (Sender: TObject);
```

Если это звучит немного запутанно, подумайте о том, что происходит в среде разработки. Вы выбираете кнопку, скажем, `Button1`, дважды щелкаете на ней и на событии `OnClick`, перечисленном в объектном инспекторе среды разработки, и в модуль-контейнер добавляется новый пустой метод (скорее всего, форма):

```
procedure TForm1.Button1Click (Sender: TObject)
begin

end;
```

Вы заполняете код метода, и вуаля, все работает. Это происходит потому, что назначение метода обработчика события происходит за кулисами точно так же, как и со всеми остальными свойствами, которые вы устанавливаете во время проектирования у компонентов.

Из вышеприведенного описания можно понять, что между событием и назначенным ему методом нет однозначного соответствия. Совсем наоборот. Вы можете иметь несколько событий, которые имеют один и тот же обработчик событий, что объясняет причину часто используемого параметра `Sender`, который указывает, какой из объектов инициировал событие.

Например, если у вас один и тот же обработчик события `OnClick` для двух кнопок, значение `Sender будет` содержать ссылку на объект кнопки, которая была нажата.

примечание Один и тот же метод можно назначить различным событиям в коде, как показано выше, но также и во время проектирования. При выборе события в Object Inspector можно нажать кнопку со стрелкой справа от названия события, чтобы увидеть выпадающий список "совместимых" методов - список методов, имеющих один и тот же тип указателя на метод. Это позволяет легко выбрать один и тот же метод для одного и того же события из разных компонентов. В некоторых случаях вы также можете назначить один и тот же обработчик различным совместимым событиям одного и того же компонента.

Добавление события в класс TDate

Поскольку мы добавили некоторые свойства в класс `TDate`, теперь мы можем добавить событие. Событие будет очень простым. Оно будет называться `OnChange`, и может быть использовано для предупреждения пользователя компонента об изменении значения даты. Для определения события мы просто определяем соответствующее ему свойство и добавляем некоторые данные для хранения реального указателя метода, на который ссылается событие. Это новые определения, добавленные в класс в проекте приложения DateEvent:

```
type
  TDate = class
  private
    FOnChange: TNotifyEvent;
    ...
  protected
    procedure DoChange; dynamic;
    ...
  public
    property OnChange: TNotifyEvent
      read FOnChange write FOnChange;
    ...
  end;
```

Определение свойства на самом деле очень простое. Разработчик, использующий этот класс, может присвоить новое

значение свойству и, следовательно, частному полю `FOnChange`. Обычно это поле не присваивается при запуске программы: обработчики событий предназначены для пользователей компонента, а не для писателя компонента. Автор компонента, при нужде в некотором поведении, добавит его в методы компонента.

Другими словами, класс `TDate` просто принимает обработчик события и вызывает метод, хранящийся в поле `FOnChange`, при изменении значения даты. Конечно, вызов происходит только в том случае, если свойству события было что-то присвоено.

Метод `DoChange` (объявленный как динамический метод, как это принято в случае с методами работы с событиями) делает тест и вызов метода:

```
procedure TDate.DoChange;
begin
  if Assigned (FOnChange) then
    FOnChange (Self);
end;
```

примечание Как вы, возможно, помните из Гл. 8, динамический метод похож на виртуальный метод, но использует другую реализацию, которая уменьшает объем памяти до расходов, связанных с несколько более медленным вызовом.

Метод `DoChange`, в свою очередь, вызывается каждый раз при изменении одного из значений, как в следующем коде:

```
procedure TDate.SetValue (Y, M, D: Integer);
begin
  FDate := EncodeDate (Y, M, D);
  // Fire the event
  DoChange;
end;
```

Теперь, если мы посмотрим на программу, использующую данный класс, то сможем значительно упростить ее код. Сначала мы добавляем новый пользовательский метод в класс формы:

```
type
  TDateForm = class(TForm)
    ...
    procedure DateChange(Sender: TObject);
```

Код этого метода просто обновляет метку текущим значением свойства `Text` объекта `TDate`:

```
procedure TDateForm.DateChange;
begin
  LabelDate.Text := TheDay.Text;
end;
```

В методе `FormCreate` обработчику этого события назначается метод:

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
  TheDay := TDate.Init (7, 4, 1995);
  LabelDate.Text := TheDay.Text;
  // Assign the event handler for future changes
  TheDay.OnChange := DateChange;
end;
```

Ну, похоже, много работы. Я соврал, когда сказал, что обработчик событий избавит нас от кодирования? Нет. Теперь, после того, как мы добавили код, мы можем полностью забыть об обновлении метки, когда мы изменяем некоторые данные объекта. Вот, например, обработчик события `OnClick` одной из кнопок:

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);
begin
  TheDay.Increase;
end;
```

Такой же упрощенный код присутствует и во многих других обработчиках событий. После того, как мы установили обработчик событий, мы сможем полностью забыть, что нужно постоянно обновлять метку. Это устраняет существенный потенциальный источник ошибок в программе. Также обратите внимание, что нам пришлось написать некоторый код в самом начале, так как это не компонент, установленный в среде разработки, а просто класс. С компонентом достаточно выбрать обработчик события в Object Inspector и написать одну строку кода для обновления метки. Вот и все.

Это подводит нас к вопросу, насколько сложно написать новый компонент в Delphi? На самом деле все настолько просто, что я покажу вам, как это сделать в следующем разделе.

примечание Это было лишь краткое введение в роль свойств и событий, а также в написание компонентов. Базовое понимание этих особенностей важно для каждого разработчика Delphi. В этой книге мы не будем углубляться в детали написания пользовательских компонентов.

Создание компонента TDate

Теперь, когда мы понимаем свойства и события, следующий шаг - увидеть, что такое компонент. Мы вкратце рассмотрим эту тему, превратив наш класс `TDate` в компонент. Сначала мы должны унаследовать наш класс от класса TComponent, а не от класса по умолчанию `TObject`. Вот код:

```
type
  TDate = class (TComponent)
    ...
  public
    constructor Create (AOwner: TComponent); overload; override;
    constructor Create (Y, M, D: Integer); reintroduce; overload;
```

Как видите, вторым шагом было добавление в класс нового конструктора, перекрывающего конструктор по умолчанию для компонентов, чтобы обеспечить подходящее начальное значение. Так как существует перегруженная версия, необходимо также использовать директиву reintroduce для нее, чтобы избежать предупреждения от компилятора. Код нового конструктора просто устанавливает дату на текущую после вызова конструктора базового класса:

```
constructor TDate.Create (AOwner: TComponent);
var
  Y, D, M: Word;
begin
  inherited Create (AOwner);
  // today...
```

```
  FDate := Date;
```

После этого нам необходимо добавить в модуль, определяющий наш класс (модуль `Dates` проекта приложения DateComp) процедуру `Register`. (Убедитесь, что этот идентификатор начинается с заглавного `буквы R`, иначе он не будет распознан). Это необходимо для добавления компонента в IDE. Просто объявите процедуру, которая не требует никаких параметров, в `интерфейсной` части модуля `interface`, а затем напишите этот код в разделе `implementation`:

```
procedure Register;
begin
  RegisterComponents ('Sample', [TDate]);
end;
```

Этот код добавляет новый компонент на страницу *Пример* Палитры Инструментов, создавая страницу при необходимости.

Последний шаг - установка компонента. Для этого необходимо создать пакет, который является специальным видом проекта приложения для хостинга компонентов. Все, что нужно сделать:

Выберите пункт File | New | Other в меню IDE, открыв диалоговое окно New Items (Новые элементы).

Выберите Package (Пакет)

Сохранить пакет с именем (возможно, в той же папке, где находится юнит с действительным кодом компонента).

В только что созданном проекте пакета, в панели Project Manager, щелкните правой кнопкой мыши на узле Contains, чтобы добавить в проект новый модуль, и выберите юнит с классом компонента `TDate`.

Теперь в менеджере проектов можно щелкнуть правой кнопкой мыши на узле пакета и не только выполнить команду Build, но и выбрать пункт меню Install, чтобы установить компонент в среду разработки

Если вы начнете с кода, который поставляется с книгой, все, что вам нужно сделать, это последний шаг последовательности выше: Откройте проект DatePackage из папки DateComponent, скомпилируйте и установите его.

Если вы теперь создаете новый проект и переходите к Палитре инструментов, вы должны увидеть новый компонент в разделе *Sample*. Просто начните вводить его название для поиска. Он будет показан с помощью иконки по умолчанию для компонентов. В этот момент вы можете поместить компонент на форму и начать манипулировать его свойствами в Object Inspector, как показано на рисунке 10.2. Вы также можете управлять событием `OnChange` гораздо проще, чем в предыдущем примере.

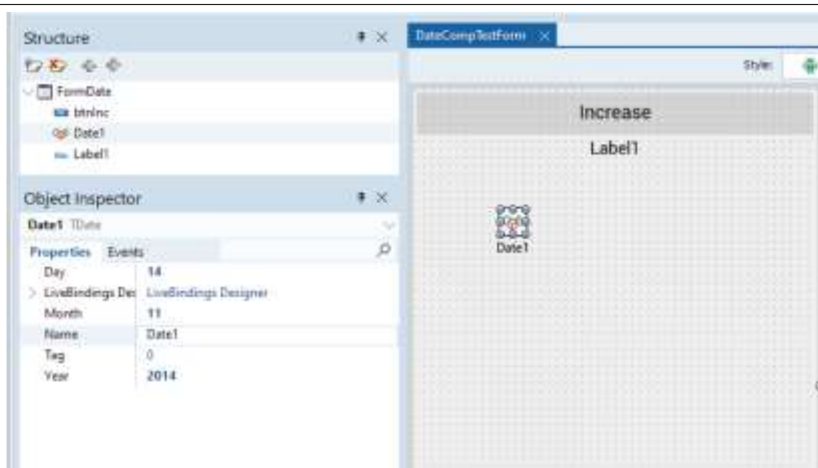


Рисунок 10.2: Свойства нашего нового компонента TDate в Object Inspector

Кроме того, чтобы собрать для пробы свое собственное приложение, использующее этот компонент (что я действительно советую вам сделать), теперь вы можете открыть проект приложения DateComponent, который является обновленной версией компонента, который мы построили шаг за шагом в течение последних нескольких разделов этой главы. Это, по сути, упрощенная версия проекта приложения

DateEvent, потому что теперь обработчик событий доступен непосредственно в Object Inspector.

примечание Если вы откроете проект приложения DateCompTest перед компиляцией и установкой конкретного пакета компонентов (проект приложения DatePackage), IDE не распознает компонент, так как откроет форму и выдаст сообщение об ошибке. Вы не сможете скомпилировать программу или заставить ее работать должным образом, пока не установите новый компонент.

Реализация поддержки перечислений в классе

В Главе 3 мы видели, как можно использовать цикл for-in в качестве альтернативы классическому циклу `for`. В этом разделе я описал, как можно использовать цикл `for-in` для массивов, строк, наборов и некоторых других системных типов данных. Такой цикл можно применять к любому классу до тех пор, пока он определяет поддержку перечисления. В то время как наиболее очевидным примером будут классы, содержащие списки элементов, с технической точки зрения эта возможность полностью открыта.

Для поддержки перечисления элементов класса в Object Pascal есть два правила, которым необходимо следовать: добавить вызов метода GetEnumerator, который возвращает класс (фактический класс перечисления); определить этот класс перечисления, добавив метод `Next` и свойство Current, первое из которых предназначено для навигации по элементам, а второе - для возврата актуального элемента. После этого (а в реальном примере я покажу через секунду) компилятор может разрешить цикл `for-in`, работающий с нашим классом, причем отдельные

элементы должны быть того же типа, что и `Current` свойство класса enumerator.

Хотя это и не является строго необходимым, может оказаться хорошей идеей реализовать класс поддержки перечисления как вложенный тип (языковая особенность, описанная в Гл. 7), потому что действительно нет смысла использовать специфический тип, используемый для перечисления сам по себе.

Следующий класс, являющийся частью прикладного проекта NumbersEnumerator, хранит ряд чисел (разновидность абстрактной коллекции) и позволяет проводить по ним итерации. Это становится возможным благодаря определению перечислителя enumerator, объявленного как вложенный тип и возвращаемого функцией `GetEnumerator`:

```
type
  TNumbersRange = class
  public
    type
      TNumbersRangeEnum = class
      private
        NPos: Integer;
        FRange: TNumbersRange;
      public
        constructor Create (aRange: TNumbersRange);
        function MoveNext: Boolean;
        function GetCurrent: Integer;
        property Current: Integer read GetCurrent;
      end;

  private
    FNStart: Integer;
    FNEnd: Integer;
  public
    function GetEnumerator: TNumbersRangeEnum;
    procedure Set_NEnd(const Value: Integer);
    procedure Set_NStart(const Value: Integer);

    property NStart: Integer read FNStart write Set_NStart;
    property NEnd: Integer read FNEnd write Set_NEnd;
  end;
```

Метод `GetEnumerator` создает объект вложенного типа, который хранит информацию о состоянии для итерации над данными.

Обратите внимание, как конструктор перечисления сохраняет ссылку на фактический объект, который он перечисляет (объект, который передается в качестве параметра с помощью `self`) и устанавливает начальную позицию в самое начало:

```
function TNumbersRange.GetEnumerator: TNumbersRangeEnum;
begin
  Result := TNumbersRangeEnum.Create (self);
end;

constructor TNumbersRange.TNumbersRangeEnum.
  Create(ARange: TNumbersRange);
begin
  inherited Create;
  FRange := ARange;
  NPos := FRange.NStart - 1;
end;
```

примечание Почему конструктор устанавливает начальное значение на первое значение минус 1, а не на первое, как ожидалось? Получается, что сгенерированный компилятором код для for в цикле соответствует созданию перечисления и выполнению кода*, в то время как Next используют Current*. Тестирование выполняется до получения первого значения, так как список может не иметь значения. Это означает, что `Next` вызывается перед использованием первого элемента. Вместо того, чтобы реализовывать это с более сложной логикой, я просто установил начальное значение на *единицу перед первым*, поэтому при первом вызове Next перечислитель располагается на первом значении.

Наконец, методы перечисления позволяют получить доступ к данным и предоставляют возможность перейти к следующему значению в списке (или к следующему элементу в пределах диапазона):

```
function TNumbersRange.TNumbersRangeEnum.GetCurrent: Integer;
begin
  Result := NPos;
end;

function TNumbersRange.TNumbersRangeEnum.MoveNext: Boolean;
begin
  Inc (NPos);
  Result := NPos <= FRange.NEnd;
end;
```

Как видно из кода выше, метод `Next` служит двум разным целям, переходя к следующему элементу списка и проверяя,

достиг ли перечислитель конца, в этом случае метод возвращает false.

После всей этой работы, теперь можно использовать цикл `for...in` для итерации по значениям объекта диапазона:

```
var
  ARange: TNumbersRange;
  I: Integer;
begin
  ARange := TNumbersRange.Create;
  ARange.nStart := 10;
  ARange.nEnd := 23;

  for I in ARange do
    Show (IntToStr (I));
```

Вывод - это просто список значений*, перечисленных в* диапазоне от 10 до 23 включительно:

```
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

примечание Существует много случаев, когда RTL и VCL библиотеки определяют счетчики, например, каждый TComponent может перечислить компонент, которым он владеет. Чего не хватает, так это перечисления дочерних элементов управления. В Гл. 12, в разделе "Добавление перечисления с помощью помощника класса", мы посмотрим, как его можно создать. Причина, по которой этого примера здесь нет, заключается в том, что сначала нам нужно обсудить помощников классов.

15 советов по RAD и ООП

В этой главе я рассказал о свойствах, событиях и ключевом слове published, которые составляют основные особенности языка, связанные с быстрой разработкой приложений (RAD), визуальной разработкой или событийно-ориентированным программированием (три термина относятся к одной и той же концептуальной модели). Несмотря на то, что это очень мощная модель, она опирается на солидную архитектуру ООП. Иногда, подход RAD может подтолкнуть разработчиков забыть о хороших практиках ООП. В то же время, возвращаясь к написанию чистого кода, игнорирование подхода RAD часто может оказаться контрпродуктивным. В этом последнем разделе главы я перечислил несколько советов и предложений по наведению мостов между этими двумя подходами. Другой способ описания - рассмотреть его в разделе "ООП за пределами RAD".

примечание Материал этого заключительного раздела главы был первоначально опубликован в 17 выпуске "The Delphi Magazine" (июль 1999) под названием "20 правил для ООП в Дельфах". Сейчас я обрезал несколько правил и переформулировал другие, но суть осталась.

Совет 1: Форма - это класс

Программисты часто относятся к формам как к объектам, в то время как на самом деле они являются классами. Разница заключается в том, что вы можете иметь несколько объектов формы, основанных на одном классе формы.

Путаница заключается в том, что IDE создает глобальную переменную по умолчанию и (в зависимости от ваших настроек) также может создать объект формы при запуске для

каждого класса формы, который вы определите в своем проекте. Это, конечно, удобно для новичков, но обычно это плохая привычка для любого нетривиального приложения.

Конечно, очень важно дать осмысленное название каждой форме (и ее классу) и каждому юниту. К счастью, эти два имени должны быть разными, но вы можете использовать соглашение, чтобы последовательно сопоставить их (например, `AboutForm` и `About.pas`).

По мере того, как вы будете проходить следующие шаги, вы увидите практический эффект этой концепции "форма — это класс".

Совет 2: Название Компоненты

Очень важно использовать описательные имена для компонентов. Наиболее распространенной нотацией является использование нескольких строчных начальных букв для типа класса, за которыми следует роль компонента, как в `BtnAdd` или `EditName`. На самом деле существует много похожих нотаций, следуя этому стилю, и нет причин говорить, что любая из них лучше, это зависит от Вашего личного вкуса.

Совет 3: Имя События

Не менее важно дать правильные имена методам обработки событий. При правильном названии компонентов, например, по умолчанию `Button1Click` становится `BtnAddClick`. Хотя мы можем догадаться, что делает метод по имени кнопки, я думаю, что лучше использовать имя, описывающее действие метода, а не когда он срабатывает. Например, событие OnClick кнопки `BtnAdd` можно назвать `AddToList`, если это то, что делает метод.

Это делает код более читабельным, особенно при вызове обработчика событий из другого метода класса, и помогает разработчикам прикреплять один и тот же метод к нескольким событиям или к различным компонентам, хотя я должен сказать, что использование Actions является предпочтительным выбором для прикрепления одного и того же события к нескольким элементам пользовательского интерфейса в любых нетривиальных программах.

примечание Действия и компонент ActionList являются очень хорошими архитектурными особенностями библиотек пользовательского интерфейса VCL и FireMonkey, предлагая концептуальное разделение пользовательских операций (и их состояния) от элементов управления пользовательским интерфейсом, с которыми они ассоциируются. Активируя элемент управления, выполняет действие. Но если логически отключить действие, элементы пользовательского интерфейса также отключаются. Эта тема выходит за рамки данной книги, но стоит изучить, если вы используете любой из этих фреймворков.

Совет 4: Используйте методы формы

Если формы являются классами, то их код собирается в методах. Кроме обработчиков событий, которые играют особую роль, но все же могут быть вызваны как другие методы, часто бывает полезно добавить пользовательские методы для классов форм. Можно добавлять методы, выполняющие действия и имеющие доступ к статусу формы. Гораздо лучше добавлять к форме публичный метод, чем позволять другим формам работать непосредственно с ее компонентами.

Совет 5: Добавить конструкторы

формы

Вторичная форма, созданная во время исполнения, может предоставлять другие специфические конструкторы, кроме конструктора по умолчанию (наследуемая форма класса TComponent). Я предлагаю перегрузить метод `Create`, добавив необходимые параметры инициализации, как в следующем фрагменте кода:

```
public
  constructor Create (Text: string); reintroduce; overload;

constructor TFormDialog.Create(Text: string);
begin
  inherited Create (Application);
  Edit1.Text := Text;
end;
```

Совет 6: Избегайте глобальных переменных

Следует избегать глобальных переменных (т.е. переменных, объявленных в интерфейсной части модуля). Вот несколько предложений, которые помогут вам в этом. Если вам нужно дополнительное хранилище данных для формы, добавьте к ней некоторые приватные поля. В этом случае каждый экземпляр формы будет иметь свою копию данных.

Вы можете использовать переменные модуля (объявленные в части реализации модуля) для данных, совместно используемых несколькими экземплярами класса формы, но гораздо лучше использовать данные класса (объяснено в Главе 12).

Совет 7: Никогда не используйте

Form1 в методах TForm1.

Никогда не следует ссылаться на конкретный объект в методе класса этого объекта. Другими словами, никогда не ссылайтесь на `Form1` в методе класса `TForm1`. Если вам нужно обратиться к текущему объекту, используйте ключевое слово self. Помните, что в большинстве случаев в этом нет необходимости, так как вы можете обращаться непосредственно к методам и данным текущего объекта. Если вы не будете следовать этому правилу, у вас возникнут реальные проблемы при создании нескольких экземпляров формы.

Совет 8: Редко используйте Form1 в других формах.

Даже в коде других форм старайтесь избегать прямых ссылок на глобальные объекты, такие как `Form1`. Намного лучше объявить локальные переменные или приватные поля, чтобы ссылаться на другие формы. Например, основная форма программы может иметь приватное поле, ссылающееся на диалоговое окно. Очевидно, что это правило становится необходимым, если вы планируете создать несколько экземпляров вторичной формы. Вы можете хранить список в динамическом массиве основной формы, или просто использовать массив `Forms` глобального объекта Screen для ссылки на любую форму, существующую в настоящее время в приложении.

Совет 9: Удалить Глобальную

переменную Form1

На самом деле, я предлагаю удалить объект глобальной формы, который автоматически добавляется IDE в проект при добавлении в него новой формы, например, `Form1`. Это возможно только в том случае, если вы отключите автоматическое создание этой формы, от чего я предлагаю избавиться в любом случае. Кроме того, вы можете удалить соответствующую строку, использованную для создания формы, из исходного кода проекта.

Нет необходимости говорить, что, если форма не создается автоматически, вам понадобится какой-нибудь код в вашем приложении, чтобы создать ее, и, возможно, какая-нибудь другая переменная, чтобы на нее ссылаться.

Я думаю, что удаление объекта глобальной формы очень полезно для новичков в Object Pascal, которые больше не будут путаться между классом и глобальным объектом. На самом деле, после удаления глобального объекта любая ссылка на него приведет к ошибке.

Совет 10: Добавить свойства формы

Как я уже говорил в разделе "Добавление свойств к формам" в этой главе, когда Вам нужны данные для формы, добавьте приватное поле. Если Вам необходимо получить доступ к этим данным из других классов, то добавьте свойства к форме. При таком подходе Вы сможете изменить код формы и ее данные (включая пользовательский интерфейс) без необходимости изменения кода других форм или классов. Вы также должны использовать свойства или методы для инициализации

вторичной формы или диалогового окна, а также для чтения его конечного состояния. Инициализация также может быть выполнена с помощью конструктора, как я уже описывал.

Совет 11: Показать свойства компонентов

Когда вам необходимо получить доступ к статусу другой формы, вы не должны обращаться непосредственно к ее компонентам. Это привяжет код других форм или классов к пользовательскому интерфейсу, который является одной из частей приложения, подверженной наибольшим изменениям. Вместо этого, объявите свойство формы, привязанное к свойству компонента: это делается с помощью метода `Get`, который читает статус компонента, и метода `Set`, который его записывает. Предположим, теперь вы измените пользовательский интерфейс, заменив компонент на другой. Всё, что вам нужно сделать, это исправить методы `Get` и `Set`, связанные со свойством, вам не придётся проверять и изменять исходный код всех форм и классов, которые могут ссылаться на этот компонент.

Совет 12: Используйте массив свойств, когда это необходимо

Если вам необходимо обрабатывать ряд значений в форме, вы можете объявить свойство как массив. В случае, если это важная информация для формы, вы можете сделать его также индексированным свойством по умолчанию, так что вы сможете получить прямой доступ к ее значению, записав `SpecialForm[3]`. Это уже рассматривалось для более общего

случая, чем формы в разделе "Использование индексированных свойств".

Совет 13: Запуск операций в свойствах

Помните, что одним из преимуществ использования свойств вместо доступа к глобальным данным является возможность вызова методов и выполнения любой операции при записи (или чтении) значения свойства. Например, можно рисовать непосредственно на поверхности формы, устанавливать значения нескольких свойств, вызывать специальные методы, изменять состояние сразу нескольких компонентов или запускать событие, если оно доступно.

Другой связанный пример - использование геттеров свойств для реализации отложенного создания. Вместо того, чтобы создавать подобъект в конструкторе класса, можно создать его в первый раз, когда он потребуется, написав код типа:

```
private
  FBitmap: TBitmap;
public
  property Bitmap: TBitmap read GetBitmap;

function TBitmap.GetBitmap: TBitmap;
begin
  if not Assigned (FBitmap) then
    FBitmap := ... // create it and initialize it
  Result := FBitmap;
end;
```

Совет 14: Спрячьте компоненты

Слишком часто я слышу жалобы пуристов ООП, потому что формы включают список компонентов в опубликованном разделе, подход, который не соответствует принципу

инкапсуляции. На самом деле они указывают на важный вопрос, но большинство из них, похоже, не знает, что решение находится под рукой, не переписывая библиотеки и не меняя язык. Ссылки на компоненты, которые добавляются в форму, могут быть перенесены в частную часть декларации формы, так что они не будут доступны для других форм. Таким образом, вы можете в принудительном порядке использовать свойства, привязанные к компонентам (см. раздел выше), чтобы получить доступ к их статусу.

Если IDE размещает все компоненты в опубликованном разделе, то это связано со способом привязки этих полей к компонентам, созданным из потокового файла (DFM или FMX). При установке имени компонента VCL автоматически прикрепляет объект компонента к его ссылке в форме. Это возможно только в том случае, если ссылка опубликована, поскольку для выполнения операции потоковая система использует традиционный RTTI и некоторые методы `TObject`.

Так что, при перемещении ссылок на компонент из опубликованной в приватную секцию вы теряете это автоматическое поведение. Чтобы исправить проблему, просто сделайте это вручную, добавив следующий код для каждого компонента в обработчик события `OnCreate` формы:

```
Edit1 := FindComponent('Edit1') as TEdit;
```

Вторая операция, которую необходимо выполнить, это зарегистрировать классы компонентов в системе, чтобы их RTTI информация была включена в скомпилированную программу и сделана доступной системе. Это нужно сделать только один раз для каждого класса компонента, и только в том случае, если вы переместите все ссылки на компоненты этого типа в приватную секцию. Вы можете добавить этот вызов, даже если не уверены, что он нужен для вашего приложения, учитывая, что дополнительный вызов метода RegisterClasses

для того же самого класса не имеет никакого эффекта. Вызов RegisterClasses обычно добавляется в секцию инициализации модуля, в котором находится форма:

```
RegisterClasses([TEdit]);
```

Совет 15: Используйте мастер создания формы ООП

Повторение двух вышеуказанных операций для каждого компонента каждой формы, безусловно, скучно и трудоемко. Чтобы избежать этой чрезмерной нагрузки, я написал простого мастера, который в маленьком окне генерирует строки кода для добавления в программу. Для каждой формы необходимо сделать простое копирование и вставку операций, так как мастер автоматически не помещает исходный код в секцию инициализации модуля.

Как получить мастера? Вы можете найти его в разделе "Мастера инициализации Cantools " на сайте:

```
https://github.com/marcocantu/cantools
```

Советы: Заключение

Это лишь небольшая коллекция советов и предложений по более сбалансированной модели разработки RAD и ООП. Конечно, в этой теме есть гораздо больше вопросов, но это выходит далеко за рамки этой книги, которая в первую очередь посвящена самому языку, а не лучшим практикам для архитектур приложений.

примечание Есть несколько книг, посвященных архитектуре приложений с Delphi, но ничем не лучше серии томов, написанных Ником Ходжесом, включая “Coding in Delphi” ("Кодирование в Delphi"), “More Coding in Delphi” ("Больше кодирования в Delphi") и “Dependency Injection in Delphi” ("Впрыскивание зависимостей в Delphi"). Более подробную информацию об этих книгах вы можете найти по адресу
`http://www.codingindelphi.com/`

11: Интерфейсы

В отличие от того, как происходит в C++ и некоторых других языках, модель наследования Object Pascal не поддерживает множественное наследование. Это означает, что каждый класс может иметь только один базовый класс.

Полезность множественного наследования является предметом обсуждения среди экспертов ООП. Отсутствие этой конструкции в Object Pascal можно считать недостатком, так как у вас нет мощности C++, но также и преимущества, так как вы получаете более простой язык и избегаете некоторых проблем, связанных с множественным наследованием. Одним из способов компенсировать отсутствие множественного наследования в Object Pascal является использование интерфейсов, которые позволяют определить класс, реализующий несколько абстракций одновременно.

примечание Большинство современных объектно-ориентированных языков программирования не поддерживают множественное наследование, а используют интерфейсы, в том числе Java и C#. Интерфейсы обеспечивают гибкость и мощь декларирования поддержки множественных интерфейсов, реализованных на классе, избегая при этом проблем наследования множественных реализаций. Поддержка множественного наследования остается в основном ограниченной для языка Си++. Некоторые динамические объектно-ориентированные языки поддерживают mix-ins, другой и более простой способ достижения чего-то похожего на множественное наследование.

Вместо того, чтобы открывать дебаты, я просто предположу, что полезно рассматривать один объект с нескольких "точек зрения". Но прежде, чем я построю пример, чтобы объяснить

этот принцип, мы должны представить роль интерфейсов в Object Pascal и выяснить, как они работают.

С более общей точки зрения, интерфейсы поддерживают несколько иную объектно-ориентированную модель программирования, чем классы. Объекты, реализующие интерфейсы, подвержены полиморфизму для каждого из поддерживаемых ими интерфейсов. Действительно, модель, основанная на интерфейсах, является мощной. Интерфейсы поддерживают инкапсуляцию и обеспечивают более свободную связь между классами, чем наследование.

примечание Методы, рассмотренные в этой главе, и общая поддержка интерфейсов были первоначально добавлены в язык Object Pascal как способ поддержки и реализации архитектуры Windows COM (Component Object Model). Позже эта возможность была расширена, чтобы быть полностью пригодной для использования вне этого сценария, но некоторые элементы COM, такие как идентификация интерфейса через ID и поддержка подсчета ссылок, все еще остаются в текущей реализации интерфейсов Object Pascal, что делает их немного отличными от большинства других языков.

Использование интерфейсов

Помимо декларирования абстрактных классов (классов с абстрактными методами), в Object Pascal можно написать и *полностью абстрактный класс, т.е.* класс, в котором используются только виртуальные абстрактные методы. Это осуществляется с использованием определенного ключевого слова, `интерфейса`. По этой причине мы называем эти типы данных интерфейсами.

Технически интерфейс не является классом, хотя и может походить на него. В то время как класс может иметь экземпляр,

интерфейс не может. Интерфейс может быть реализован одним или несколькими классами, так что экземпляры этих классов в конечном итоге *поддерживают* или *реализуют* интерфейс.

В Object Pascal интерфейсы имеют несколько отличительных особенностей:

У переменных типа интерфейса имеются счетчики ссылок, в отличие от переменных типа класса, использующих вид автоматического управления памятью

Класс может наследовать только от одного базового класса, но он может реализовать несколько интерфейсов

Подобно тому, как все классы наследуют от TObject, все интерфейсы идут от IInterface, образуя отдельную, ортогональную иерархию.

По соглашению имена интерфейсов начинаются с буквы I, а не с буквы T, используемой большинством других типов данных.

примечание Первоначально базовый тип интерфейса в Object Pascal назывался `IUnknown`, так как именно этого требует COM. Интерфейс IUnknown был позже переименован в `IInterface`, чтобы подчеркнуть тот факт, что интерфейс в Object Pascal можно использовать даже вне области COM и на операционных системах, где COM не существует. В любом случае, реальное поведение `IInterface` все равно идентично предыдущему интерфейсу `IUnknown`.

Объявление интерфейса

Рассмотрев основные концепции, давайте перейдем к какому-нибудь реальному коду, который должен помочь вам понять, как работают интерфейсы в Object Pascal. На практике интерфейс имеет определение, напоминающее определение класса. Это определение имеет список методов, но эти методы никак не реализованы, как это происходит с абстрактным методом в обычном классе.

Ниже приводится определение интерфейса:

```
type
  ICanFly = interface
    function Fly: string;
  end;
```

Учитывая, что каждый интерфейс прямо или косвенно наследует от базового типа интерфейса, это соответствует записи:

```
type
  ICanFly = interface (IInterface)
    function Fly: string;
  end;
```

Через некоторое время я покажу вам, что подразумевает наследование от IInterface и что оно приносит на стол. На данный момент достаточно сказать, что у `IInterface` есть несколько базовых методов (опять же, похожих на `TObject`).

Есть последний лакомый кусочек, связанный с объявлениями интерфейса. Для интерфейсов часть проверки типов выполняется динамически, и система требует от каждого интерфейса иметь уникальный идентификатор, или GUID, который можно получить в редакторе Delphi, нажав Ctrl+Shift+G. Это полный код интерфейса:

```
type
  ICanFly = interface
    ['{D7233EF2-B2DA-444A-9B49-09657417ADB7}']
    function Fly: string;
  end;
```

Этот интерфейс и его реализация (описана ниже) доступны в прикладном проекте Intf101.

примечание Хотя вы можете скомпилировать и использовать интерфейс без указания GUID, вы, как правило, захотите сгенерировать GUID, потому что он необходим для выполнения запроса к интерфейсу или динамического `as` типовой запрос, используя этот тип интерфейса. Весь смысл интерфейсов заключается в том, чтобы использовать преимущества значительно расширенного типа гибко во время выполнения, что зависит от того, какие интерфейсы имеют GUID.

Реализация интерфейса

Любой класс может реализовать один или несколько интерфейсов, перечисляя их после базового класса, от которого он наследует, и предоставляя реализацию для каждого из методов каждого из интерфейсов:

```
type
  TAirplane = class (..., ICanFly)
    function Fly: string;
  end;

function TAirplane.Fly: string;
begin
  // actual code
end;
```

Когда класс реализует интерфейс, он *обязан* обеспечить реализацию всех методов интерфейса с одной и той же сигнатурой, поэтому в этом случае класс `TAirplane` должен реализовать метод `Fly` как функцию, возвращающую строку. Учитывая, что интерфейс также наследуется от базового интерфейса (IInterface), класс, реализующий интерфейс, должен неизменно предоставлять все методы интерфейса и всех своих базовых интерфейсов.

Поэтому достаточно распространена реализация интерфейсов в классах, унаследованных от базового класса, который уже реализует методы базового интерфейса IInterface. Библиотека исполнения Object Pascal уже предоставляет несколько базовых классов для реализации базового поведения. Самым простым из них является класс TInterfacedObject, поэтому приведенный выше код может стать таковым:

```
type
  TAirplane = class (TInterfacedObject, ICanFly)
    function Fly: string;
  end;
```

примечание При реализации интерфейса можно использовать как статический, так и виртуальный метод. Если вы планируете переопределить методы в наследуемом классе, то использование виртуальных методов имеет смысл. Однако существует

альтернативный подход, который заключается в том, чтобы указать, что базовый класс также наследует от того же самого интерфейса, и переопределяет метод. Я, как правило, предпочитаю объявлять метод, реализующий методы интерфейса, виртуальными методами, когда это необходимо.

Теперь, когда мы определили интерфейс и класс, реализующий его, мы можем создать объект этого класса. Мы можем относиться к этому как к обычному классу, написав его:

```
var
  Airplane1: TAirplane;
begin
  Airplane1 := TAirplane.Create;
  try
    Airplane1.Fly;
  finally
    Airplane1.Free;
  end;
end;
```

В данном случае мы игнорируем тот факт, что класс реализует интерфейс. Разница в том, что теперь мы также можем объявить переменную типа интерфейс. Использование переменной типа интерфейса *автоматически* включает в себя модель ссылочной памяти, поэтому мы можем пропустить блок `try-finally`:

```
var
  Flyer1: ICanFly;
begin
  Flyer1 := TAirplane.Create;
  Flyer1.Fly;
end;
```

Есть несколько релевантных соображений о первой строке этого, казалось бы, простого фрагмента кода, тоже части проекта приложения Intf101. Во-первых, как только вы присваиваете объект переменной интерфейса, во время выполнения автоматически проверяется, реализует ли объект этот интерфейс, используя специальную версию оператора `as`. Эту операцию можно явно выразить, написав ту же самую строку кода, как показано ниже:

```
Flyer1 := TAirplane.Create as ICanFly;
```

Во-вторых, независимо от того, используем ли мы прямое присваивание или оператор `as`, во время выполнения выполняется одно дополнительное действие: вызывается метод _AddRef объекта, увеличивая его счетчик ссылок. Это делается путем вызова метода, который наш объект наследует от базового класса TInterfacedObject.

В то же время, как только переменная `Flyer1` выходит за пределы области видимости (т.е. при выполнении оператора `end`), во время выполнения Delphi вызывается метод _Release, который уменьшает счетчик ссылок, проверяет, равен ли счетчик ссылок нулю, и в этом случае уничтожает объект. Поэтому в приведенном выше коде нет необходимости вручную освобождать созданный нами объект и не нужно писать блок `try-finally`.

примечание Хотя в приведенном выше исходном коде нет `try-finally` block, компилятор автоматически добавит в метод неявный `try-finally` block с неявным вызовом `_Release`. Во многих случаях это происходит в Object Pascal: практически каждый раз, когда метод имеет один или несколько управляемых типов (например, строки, интерфейс или динамические массивы), компилятор автоматически и неявно добавляет try-finally блок.

Интерфейсы и подсчет ссылок

Как мы видели в коде выше, объекты Object Pascal, на которые ссылаются переменные интерфейса, являются ссылочными (если только переменная типа интерфейса не помечена как слабая или небезопасная, как будет объяснено ниже). Мы также видели, что они могут быть автоматически уничтожены, когда на них больше не ссылается ни одна переменная интерфейса.

Важно отметить, что хотя в компиляторе и присутствует некая магия компиляции (скрытые вызовы _AddRefи _Release),

фактический механизм подсчета ссылок зависит от конкретной реализации, предоставляемой разработчиком или библиотекой времени исполнения. В последнем примере подсчет ссылок действительно происходит из-за кода в методах класса TInterfacedObject (здесь приведен несколько упрощенный вариант):

```
function TInterfacedObject._AddRef: Integer;
begin
  Result := AtomicIncrement(FRefCount);
end;

function TInterfacedObject._Release: Integer;
begin
  Result := AtomicDecrement(FRefCount);
  if Result = 0 then
  begin
    Destroy;
  end;
end;
```

Теперь рассмотрим другой базовый класс, реализующий `IInterface`, который также находится в RTL (в модуле Generics.Defaults), TSingletonImplementation. Этот странно названный класс, по сути, отключает реальный механизм подсчета ссылок:

```
function TSingletonImplementation._AddRef: Integer;
begin
  Result := -1;
end;

function TSingletonImplementation._Release: Integer;
begin
  Result := -1;
end;
```

примечание Класс TSingletonImplementation — дезориентирующее название, так как он не имеет никакого отношения к шаблону singleton. Мы рассмотрим пример этого шаблона в следующей главе.

Хотя `TSingletonImplementation` обычно не используется, есть еще один класс, который реализует интерфейсы и отключает механизм подсчета ссылок просто потому, что у него есть своя собственная модель управления памятью, и это класс TComponent.

Если вы хотите иметь пользовательский компонент, реализующий интерфейс, вам не нужно беспокоиться о подсчете ссылок и управлении памятью. Пример пользовательского компонента, реализующего интерфейс, мы увидим в разделе "Реализация шаблонов с интерфейсами" в конце этой главы.

Ошибки при смешивании способов применения ссылок

При использовании объектов, как правило, следует обращаться к ним либо только с помощью объектных переменных, либо только с переменными интерфейса. Смешивание этих двух подходов нарушает схему подсчета ссылок, предоставляемую Object Pascal, и может привести к ошибкам в памяти, которые крайне сложно отследить. На практике, если вы решили использовать интерфейсы, то, скорее всего, следует использовать только переменные, основанные на интерфейсах.

Вот один пример из многих возможных подобных случаев. Допустим, у вас есть интерфейс, класс, реализующий его, и глобальная процедура, принимающая интерфейс за параметр:

```
type
  IMyInterface = interface
    ['{F7BEADFD-ED10-4048-BB0C-5B232CF3F272}']
    procedure Show;
  end;

  TMyIntfObject = class (TInterfacedObject, IMyInterface)
  public
    procedure Show;
  end;

procedure ShowThat (AnIntf: IMyInterface);
begin
  AnIntf.Show;
end;
```

Код выглядит довольно тривиально и на 100% корректен. Что может быть неправильным, так это то, как вы называете процедуру (этот код является частью проекта приложения IntfError):

```
procedure TForm1.BtnMixClick(Sender: TObject);
var
  AnObj: TMyIntfObject;
begin
  AnObj := TMyIntfObject.Create;
  try
    ShowThat (AnObj);
  finally
    AnObj.Free;
  end;
end;
```

В этом коде происходит то, что я передаю обычный объект функции, ожидающей интерфейс. Учитывая, что объект действительно поддерживает интерфейс, компилятор не имеет проблем с вызовом. Проблема в том, как управляется память.

Изначально, объект имеет счетчик ссылок равный нулю, так как отсутствует интерфейс, ссылающийся на него. При входе в процедуру `ShowThat` счетчик ссылок увеличивается до 1. Это нормально, и вызов происходит. Теперь при выходе из процедуры счетчик ссылок уменьшается и становится равным нулю, поэтому объект уничтожается. Другими словами, `anObj` будет уничтожен при передаче его в процедуру, что на самом деле довольно неудобно. Если вы запустите этот код, то он выйдет из строя с ошибкой памяти.

Решений может быть множество. Можно искусственно увеличивать счетчик ссылок и использовать другие трюки низкого уровня. Но реальное решение заключается в том, чтобы не смешивать интерфейсы и ссылки на объекты, а использовать только интерфейсы для ссылки на объект (этот код снова взят из проекта приложения IntfError):

```
procedure TForm1.BtnIntfOnlyClick(Sender: TObject);
var
  AnIntf: IMyInterface;
```

```
begin
  AnIntf := TMyIntfObject.Create;
  ShowThat (AnIntf);
end;
```

В данном конкретном случае решение под рукой, но во многих других обстоятельствах очень сложно придумать правильный код. Опять же, правило заключается в том, чтобы не смешивать ссылки разных типов… но продолжайте читать следующий раздел о некоторых недавних альтернативных подходах.

Слабые и небезопасные ссылки на интерфейс.

Начиная с Delphi 10.1 Berlin, язык Object Pascal предлагает некоторые улучшения в управлении ссылками на интерфейс. Язык на самом деле предлагает различные типы ссылок:

Регулярные ссылки увеличивают и уменьшают счетчик ссылок объекта при назначении и освобождении, в конечном счете освобождая объект при достижении нулевого значения счетчика ссылок

Слабые ссылки (отмеченные модификатором `[weak]`) не увеличивают счетчик ссылок объекта, на который они ссылаются. Эти ссылки полностью управляются автоматически, поэтому при уничтожении объекта, на который они ссылаются, они автоматически устанавливаются на `nil`.

Небезопасные ссылки (помеченные модификатором `[unsafe]`) не увеличивают счетчик ссылок объекта, на который они ссылаются и которым не управляют - что-то мало чем отличающееся от базового указателя.

примечание Использование слабых и небезопасных ссылок изначально было введено в рамках поддержки управления памятью ARC для мобильных платформ. Поскольку ARC сейчас постепенно сворачивается, эта функция остается доступной только для ссылок на интерфейс.

В распространенных сценариях, в которых счетчик ссылок активен, у вас может быть код, подобный приведенному ниже, который полагается на счетчик ссылок, чтобы распорядиться временным объектом:

```
procedure TForm3.Button2Click(Sender: TObject);
var
  OneIntf: ISimpleInterface;
begin
  OneIntf := TObjectOne.Create;
  OneIntf.DoSomething;
end;
```

Что, если объект имеет стандартную реализацию счетчика ссылок, и вы хотите создать ссылку на интерфейс, которая будет держаться вне общего количества ссылок? Теперь этого можно добиться, добавив атрибут `[unsafe]` в объявление переменной интерфейса, изменив приведенный выше код на:

```
procedure TForm3.Button2Click(Sender: TObject);
var
  [unsafe] OneIntf: ISimpleInterface;
begin
  OneIntf := TObjectOne.Create;
  OneIntf.DoSomething;
end;
```

Не думаю, что это хорошая идея, так как приведенный выше код приведет к утечке памяти. При отключении подсчета ссылок, когда переменная выходит за пределы области видимости, ничего не происходит. Есть несколько сценариев, в которых это выгодно, так как можно использовать интерфейсы и не запускать лишнюю ссылку. Другими словами, небезопасная ссылка рассматривается так же, как и... указатель, без дополнительной поддержки компилятора.

Теперь, прежде чем рассматривать использование небезопасного атрибута для наличия ссылки без увеличения счетчика, учтите, что в большинстве случаев есть более хороший вариант, то есть использование слабых ссылок. Слабые ссылки также позволяют избежать увеличения количества ссылок, но они управляются. Это означает, что

система отслеживает слабые ссылки, и в случае, если реальный объект будет удален, то слабая ссылка будет равна `nil`. При небезопасной ссылке, напротив, невозможно узнать статус целевого объекта (сценарий, называемый висячей ссылкой).

В каких случаях полезно использовать слабые ссылки? Классический сценарий — это случай двух объектов с перекрестными ссылками. В таком случае, на самом деле, объект искусственно раздувает счетчик ссылок других объектов, и они, по сути, останутся в памяти навсегда (при счетчике ссылок, установленном в 1), даже когда они станут недоступны.

В качестве примера рассмотрим следующий интерфейс, принимающий ссылку на другой интерфейс в то же время, и класс, реализующий его с внутренней ссылкой:

```
type
  ISimpleInterface = interface
    procedure DoSomething;
    procedure AddObjectRef (Simple: ISimpleInterface);
  end;

  TObjectOne = class (TInterfacedObject, ISimpleInterface)
  private
    AnotherObj: ISimpleInterface;
  public
    procedure DoSomething;
    procedure AddObjectRef (Simple: ISimpleInterface);
  end;
```

Если вы создадите два объекта и сделаете перекрестные ссылки на них, то в конечном итоге произойдет утечка памяти:

```
var
  One, Two: ISimpleInterface;
begin
  One := TObjectOne.Create;
  Two := TObjectOne.Create;
  One.AddObjectRef (Two);
  Two.AddObjectRef (One);
```

Решение, доступное в Delphi, заключается в том, чтобы пометить приватное поле `AnotherObj` как слабую ссылку на интерфейс:

```
private
  [weak] AnotherObj: ISimpleInterface;
```

При таком изменении счетчик ссылок не изменяется при передаче объекта в качестве параметра при вызове `AddObjectRef`, он остается на 1, и возвращается к нулю, когда переменные выходят за пределы области видимости, освобождая объекты из памяти.

Сейчас есть много других случаев, когда эта функция становится удобной, и есть некоторая реальная сложность в базовой реализации. Это отличная функция, но для ее полного освоения требуется некоторое усилие. Кроме того, она имеет некоторую стоимость исполнения, так как слабые ссылки управляются (в то время как небезопасные - нет).

Дополнительную информацию о слабых ссылках на интерфейсы и о том, как они работают, можно найти в разделе "Управление памятью и интерфейсы:" в главе 13 "Объекты и память".

Расширенные технологии интерфейсов

Для дальнейшего углубления в возможности интерфейсов, прежде чем мы рассмотрим реальные сценарии использования, важно ознакомиться с некоторыми из их более продвинутых технических возможностей, например, как реализовать несколько интерфейсов или как реализовать метод интерфейса с другим именем метода (в случае конфликта имен).

Другой важной особенностью интерфейса являются свойства. Чтобы продемонстрировать все эти более продвинутые

возможности, связанные с интерфейсами, я написал проект приложения IntfDemo.

Свойства у интерфейса

Код в этом разделе основан на двух различных интерфейсах, `IWalker` и `IJumper`, оба из которых определяют несколько методов и свойство. Свойство интерфейса — это просто имя, привязанное к методу `read` и `write`. В отличие от класса, вы не можете сопоставить свойство интерфейса с полем просто потому, что интерфейс не может иметь полей.

Вот определения интерфейсов:

```
IWalker = interface
  ['{0876F200-AAD3-11D2-8551-CCA30C584521}']
  function Walk: string;
  function Run: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;

  property Position: Integer
    read GetPos write SetPos;
end;

IJumper = interface
  ['{0876F201-AAD3-11D2-8551-CCA30C584521}']
  function Jump: string;
  function Walk: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;

  property Position: Integer
    read GetPos write SetPos;
end;
```

Когда вы реализуете интерфейс со свойством, все, что вам нужно реализовать — это реальные методы доступа, так как это свойство прозрачно и недоступно в самом классе:

```
TRunner = class (TInterfacedObject, IWalker)
private
  FPos: Integer;
public
  function Walk: string;
  function Run: string;
```

```
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
end;
```

Код реализации не является сложным (его можно найти в проекте приложения IntfDemo), с методами, вычисляющими новую позицию и отображающими то, что выполняется:

```
function TRunner.Run: string;
begin
  Inc (FPos, 2);
  Result := FPos.ToString + ': Run';
end;
```

Демонстрационный код, использующий интерфейс `IWalker` и его реализацию `TRunner`, таков:

```
var
  Intf: IWalker;
begin
  Intf := TRunner.Create;
  Intf.Position := 0;
  Show (Intf.Walk);
  Show (Intf.Run);
  Show (Intf.Run);
end;
```

The output should not be surprising:

```
1: Walk
3: Run
5: Run
```

Делегирование интерфейса

Аналогичным образом, я могу определить простой класс, реализующий интерфейс `IJumper`:

```
TJumperImpl = class (TAggregatedObject, IJumper)
private
  FPos: Integer;
public
  function Jump: string;
  function Walk: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
end;
```

Данная реализация отличается от предыдущей использованием конкретного базового класса, `TAggregatedObject`. Это специальный класс для определения объектов, используемых внутри интерфейса, с синтаксисом, который я сейчас покажу.

примечание Класс `TAggregatedObject` является еще одной реализацией IInterface, определенной в модуле `System`. По сравнению с TInterfacedObject он имеет отличия в реализации подсчета ссылок (в основном, делегирование всего подсчета ссылок *контейнеру* или *контроллеру*) и в реализации запроса интерфейса в случае, если контейнер поддерживает несколько интерфейсов.

Я собираюсь использовать его по-другому. В следующем классе, `TMyJumper`, я не хочу повторять реализацию интерфейса `IJumper` с похожими методами. Вместо этого я хочу *делегировать* реализацию этого интерфейса классу, уже реализующему его. Это невозможно сделать с помощью наследования (у нас не может быть двух базовых классов); вместо этого можно использовать специфические возможности языка - делегирование интерфейса. Следующий класс реализует интерфейс, ссылаясь на объект реализации со свойством, а не реализуя реальные методы интерфейса:

```
TMyJumper = class (TInterfacedObject, IJumper)
private
  FJumpImpl: TJumperImpl;
public
  constructor Create;
  destructor Destroy; override;
  property Jumper: TJumperImpl
    read FJumpImpl implements IJumper;
end;
```

Эта декларация указывает, что интерфейс `IJumper` реализован для класса `TMyJumper` полем `FJumpImpl`. Это поле, конечно, должно на самом деле реализовывать все методы интерфейса. Чтобы это работало, вам нужно создать соответствующий объект для поля при создании объекта `TMyJumper` (параметр конструктора требуется базовому классу `TAggregatedObject`):

```
constructor TMyJumper.Create;
```

```
begin
  FJumpImpl := TJumperImpl.Create (self);
end;
```

В классе также есть деструктор для освобождения внутреннего объекта, на который ссылается обычное поле, а не интерфейс (так как подсчет ссылок в этом сценарии не сработает).

Этот пример прост, но в целом все усложняется по мере того, как вы начинаете модифицировать некоторые методы или добавлять другие методы, которые все еще работают с данными внутреннего объекта `FJumpImpl`. Общая концепция здесь заключается в том, что вы можете повторно использовать реализацию интерфейса в нескольких классах.

Код, который косвенно использует этот интерфейс, идентичен стандартному коду, который можно было бы написать:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Intf: IJumper;
begin
  Intf := TMyJumper.Create;
  Intf.Position := 0;
  Show (Intf.Walk);
  Show (Intf.Jump);
  Show (Intf.Walk);
end;
```

Несколько интерфейсов и псевдонимы методов

Еще одной очень важной особенностью интерфейсов является возможность для класса реализовать более одного. Об этом свидетельствует следующий класс `TAthlete`, реализующий интерфейсы `IWalker` и `IJumper`:

```
TAthlete = class (TInterfacedObject, IWalker, IJumper)
private
  FJumpImpl: TJumperImpl;
public
  constructor Create;
  destructor Destroy; override;
```

```
  function Run: string; virtual;
  function Walk1: string; virtual;
  function IWalker.Walk = Walk1;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;

  property Jumper: TJumperImpl
    read FJumpImpl implements IJumper;
end;
```

Один из интерфейсов реализован напрямую, а другой делегирован внутреннему объекту `FJumpImpl`, как я делал в предыдущем примере.

Теперь у нас проблема. Оба интерфейса, которые мы хотим реализовать, имеют метод `Walk`, с одной и той же сигнатурой параметров, так как же мы реализуем оба из них в нашем классе? Как язык поддерживает столкновение имен методов в случае множества интерфейсов? Решение заключается в том, чтобы дать методу другое имя и сопоставить его с конкретным методом интерфейса, используя его в качестве префикса, с оператором:

```
function IWalker.Walk = Walk1;
```

Это объявление указывает на то, что класс реализует метод `Walk` интерфейса `IWalker` с помощью метода `Walk1` (а не с одноименным методом). Наконец, при реализации всех методов этого класса необходимо обратиться к свойству `Position` внутреннего объекта `FJumpImpl`.

Объявив новую реализацию для свойства `Position`, мы получим две позиции для одного спортсмена, довольно странная ситуация. Вот пара примеров:

```
function TAthlete.GetPos: Integer;
begin
  Result := FJumpImpl.Position;
end;

function TAthlete.Run:string;
begin
  FJumpImpl.Position := FJumpImpl.Position + 2;
  Result := IntToStr (FJumpImpl.Position) + ': Run';
end;
```

Как создать интерфейс к этому объекту `TAthlete` и ссылаться на обе операции в интерфейсах `IWalker` и `IJumper`? Ну, мы не можем сделать это точно, потому что нет базового интерфейса, который мы могли бы использовать. Интерфейсы позволяют более динамическую проверку типов и приведение типов, однако, что мы можем сделать, так это преобразовать интерфейс в другой, до тех пор, пока объект, на который мы ссылаемся, поддерживает оба интерфейса, что компилятор сможет узнать только во время выполнения. вот код для такого сценария:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  Intf: IWalker;
begin
  Intf := TAthlete.Create;
  Intf.Position := 0;
  Show (Intf.Walk);
  Show (Intf.Run);
  Show ((Intf as IJumper).Jump);
end;
```

Конечно, мы могли бы выбрать любой из двух интерфейсов и преобразовать его в другой. Использование оператора as — это способ сделать преобразование во время выполнения, но есть и другие варианты, когда вы имеете дело с интерфейсами, как мы увидим в следующем разделе.

Полиморфизм интерфейсов

В предыдущем разделе мы видели, как можно определить несколько интерфейсов и иметь класс, реализующий два из них. Конечно, это можно расширить до любого числа. Вы также можете создать иерархию интерфейсов, так как интерфейс может наследовать от существующего интерфейса:

```
ITripleJumper = interface (IJumper)
  ['{0876F202-AAD3-11D2-8551-CCA30C584521}']
  function TripleJump: string;
end;
```

Этот новый тип интерфейса имеет все методы (и свойства) своего базового типа интерфейса и добавляет новый метод. Конечно, правила совместимости интерфейсов во многом совпадают с правилами классов.

Однако в этом разделе я хочу сфокусироваться на несколько иной теме - полиморфизме на основе интерфейса. Учитывая общий базовый объект класса, мы можем вызвать виртуальный метод и быть уверенными, что будет вызвана правильная версия. То же самое может произойти и с интерфейсами. Но с интерфейсами мы можем выйти на шаг дальше и часто иметь динамический код, который запрашивает интерфейс. Учитывая, что отношение объект-интерфейс может быть довольно сложным (объект может реализовать несколько интерфейсов и косвенно реализовать также и их базовые интерфейсы), важно иметь более ясное представление о том, что возможно в этом сценарии.

В качестве отправной точки предположим, что у нас есть общая ссылка на `IInterface`. Как мы узнаем, поддерживает ли он определенный интерфейс? На самом деле существует несколько техник, которые несколько отличаются от своих классовых аналогов:

Для тестирования (и, возможно, для следующего конвертирования as) используйте утверждение "is". Это можно использовать для проверки, поддерживает ли объект интерфейс, но не в том случае, если объект, на который ссылаются с интерфейсом, также поддерживает другой (т.е. вы не можете применить `is` к интерфейсам). Заметьте, что в любом случае требуется преобразование `as`: прямое приведение к типу интерфейса почти всегда приводит к ошибке.

Вызовите глобальную функцию Support, используя одну из ее многочисленных перегруженных версий. Вы можете передать в эту функцию объект или интерфейс для тестирования, целевой

интерфейс (используя GUID или имя типа), а также вы можете передать переменную интерфейса, где будет храниться фактический интерфейс, если функция даст положительный ответ.

Прямой вызов метода QueryInterface базового интерфейса IInterface, который является чуть более низким уровнем, всегда требует переменную типа интерфейса в качестве дополнительного результата и использует численное значение `HRESULT`, а не булевский результат.

Вот фрагмент, взятый из того же самого прикладного проекта IntfDemo, показывающий, как можно применить последние две техники к общей переменной `IInterface`:

`IInterface` variable:

```
procedure TForm1.Button4Click(Sender: TObject);
var
  Intf: IInterface;
  WalkIntf: IWalker;
begin
  Intf := TAthlete.Create;
  if Supports (Intf, IWalker, WalkIntf) then
    Show (WalkIntf.Walk);

  if Intf.QueryInterface (IWalker, WalkIntf) = S_OK then
    Show (WalkIntf.Walk);
end;
```

Я полностью рекомендую использовать одну из перегруженных версий функции `Supports`, по сравнению с вызовом `QueryInterface`. В конце концов, `Supportorts` вызовет ее, но предлагает более простые и высокоуровневые опции.

Другой случай, в котором вы захотите использовать полиморфизм с интерфейсами, это когда у вас есть массив типов интерфейсов более высокого уровня (а также, возможно, массив объектов, некоторые из которых могут поддерживать данный интерфейс).

Получение объектов из ссылок на

интерфейс

Во многих версиях Object Pascal, когда вы назначали объект переменной интерфейса, не было возможности получить доступ к исходному объекту. Иногда разработчики добавляли метод `GetObject` к своим интерфейсам для выполнения операции, но это было довольно странным дизайном.

На современном языке вы можете преобразовать ссылки интерфейса назад к оригинальному объекту, которому они были назначены. Есть три отдельные операции, которые вы можете использовать:

Вы можете написать тест `is` для проверки того, что объект данного типа действительно может быть извлечен из ссылки на интерфейс:

```
IntfVar is TMyObject
```

Вы можете написать кастинг `as` для выполнения приведения типов, поднимая исключение в случае ошибки:

```
IntfVar as TMyObject.
```

Можно написать жесткое приведение типа, чтобы выполнить то же самое преобразование, вернув `nil` указатель в случае ошибки:

```
TMyObject(IntfVar)
```

примечание В любом случае, операция приведения типа работает только в том случае, если интерфейс изначально получен от объекта Object Pascal, а не от COM-сервера. Также обратите внимание, что можно приводить не только к точному классу исходного объекта, но и к одному из его базовых классов (следуя правилам совместимости типов).

В качестве примера рассмотрим следующий простой интерфейс и класс реализации (часть прикладного проекта ObjFromIntf):

```
type
  ITestIntf = interface (IInterface)
    ['{2A77A244-DC85-46BE-B98E-A9392EF2A7A7}']
    procedure DoSomething;
  end;

  TTestImpl = class (TInterfacedObject, ITestIntf)
  public
    procedure DoSomething;
```

```
    procedure DoSomethingElse; // not in the interface
    destructor Destroy; override;
  end;
```

С помощью этих описаний теперь можно определить переменную интерфейса, присвоить ей объект, а также использовать ее для вызова метода, не находящегося в интерфейсе, с помощью нового преобразования типа:

```
var
  Intf: ITestIntf;
begin
  Intf := TTestImpl.Create;
  Intf.DoSomething;
  (Intf as TTestImpl).DoSomethingElse;
```

Также можно написать код следующим образом, используя тест и прямое приведение, и всегда можно выполнить кастинг к базовому классу фактического класса объекта:

```
var
  Intf: ITestIntf;
  Original: TObject;
begin
  Intf := TTestImpl.Create;
  Intf.DoSomething;
  if Intf is TObject then
    Original := TObject (Intf);
  (Original as TtestImpl).DoSomethingElse;
```

Учитывая, что прямой кастинг в случае неудачи возвращает `nil`, можно также написать код следующим образом (без проверки):

```
 Original := TObject (intf);
  if Assigned (Original) then
    (Original as TTestImpl).DoSomethingElse;
```

Обратите внимание, что присваивание объекта, извлеченного из интерфейса, переменной приводит к проблемам подсчета ссылок: когда интерфейс установлен в `nil` или выходит за пределы области видимости, объект на самом деле удаляется, а переменная, на которую он ссылается, становится недействительной. Код, выделяющий проблему, вы найдете в обработчике события `BtnRefCountIssueClick` в примере.

Реализация паттерна адаптера с интерфейсами

В качестве реального примера использования интерфейсов я добавил в эту главу раздел, посвященный шаблону (pattern) адаптера. Короче говоря, шаблон адаптера используется для преобразования интерфейса одного класса в интерфейс, ожидаемый пользователем класса. Это позволяет использовать существующий класс в рамках фреймворка, требующий определенного интерфейса.

Шаблон может быть реализован путем создания новой иерархии классов, которая делает отображение, или путем расширения существующих классов таким образом, чтобы они предоставляли новый интерфейс. Это может быть сделано как путем множественного наследования (на поддерживающих его языках), так и с помощью интерфейсов. В этом последнем случае, который я собираюсь использовать здесь, новый наследуемый класс будет реализовывать данный интерфейс и отображать своему методу его существующее поведение.

В специфическом сценарии, адаптер предоставляет общий интерфейс для запроса значений нескольких компонентов, которые имеют несовместимые интерфейсы (как это часто бывает в библиотеках UI). Это интерфейс, называемый `ITextAndValue`, потому что он позволяет получить доступ к статусу компонента путем получения либо текстового, либо числового описания:

```
type
  ITextAndValue = interface
    '[51018CF1-OD3C-488E-81B0-0470B09013EB]'
    procedure SetText(const Value: string);
    procedure SetValue(const Value: Integer);
    function GetText: string;
    function GetValue: Integer;
```

```
    property Text: string read GetText write SetText;
    property Value: Integer read GetValue write SetValue;
end;
```

Следующим шагом является создание нового подкласса для каждого из компонентов, которые мы хотим иметь возможность использовать с интерфейсом. Например, мы можем написать:

```
type
  TApadterLabel = class(TLabel, ITextAndValue)
  protected
    procedure SetText(const Value: string);
    procedure SetValue(const Value: Integer);
    function GetText: string;
    function GetValue: Integer;
  end;
```

Реализация этих четырех методов достаточно проста, так как они могут быть привязаны к свойству `Text`, выполняющему приведение типа в случае, если значение (или текст) является числом. Однако теперь, когда у вас есть новый компонент, вам придется установить его (как мы упоминали в предыдущей главе) и заменить существующие компоненты в ваших формах на этот новый. Повторение одного и того же процесса для каждого из компонентов, которые вы хотите *адаптировать*, будет очень трудоемким.

Гораздо более простой альтернативой было бы использование *идиомы* класса-посредника interposer (т.е. определение класса с таким же именем базового класса, но в другом юните). Это будет правильно распознано компилятором и системой потокового исполнения, так что во время выполнения вы получите объект нового конкретного класса. Единственное отличие состоит в том, что во время проектирования вы будете видеть и взаимодействовать с экземплярами базового класса компонента.

примечание Впервые о классах-посредниках было упомянуто и дано такое название много лет назад в журнале " The Delphi Magazine". Они, конечно, немного хакерские, но

иногда и удобные. Я рассматриваю посреднические классы, то есть классы с одним и тем же именем базового класса, но определённые в другом модуле, скорее, идиомой Object Pascal. Обратите внимание, что для работы этого механизма очень важно, чтобы юнит с классом-посредником был указан в операторе `uses` после модуля с обычным классом, который он должен заменить. Другими словами, символы, определенные в последних модулях в операторе uses, заменяют идентичный символ, определенный в ранее включенных модулах. Конечно, вы всегда можете отличить символы, прикрепив их к имени модуля, но это действительно лишит смысла всю идею этого приема, использующего правила глобального разрешения имен.

Для определения класса-посредника обычно создают новый модуль с классом, имеющим то же имя, что и существующий базовый класс. Чтобы обратиться к базовому классу, необходимо предварить его именем модуля (иначе компилятор посчитает его рекурсивным определением):

```
type
  TLabel = class(StdCtrls.TLabel, ITextAndValue)
  protected
    procedure SetText(const Value: string);
    procedure SetValue(const Value: Integer);
    function GetText: string;
    function GetValue: Integer;
  end;
```

В этом случае вам не придется устанавливать компоненты или прикасаться к существующим программам, а только добавить к ним в конце списка uses. В обоих случаях (но в демо-приложении, которое я написал, используются посреднические классы) можно запросить компоненты формы для данного интерфейса адаптера и, например, написать код для установки всех *значений* в 50, что повлияет на различные свойства разных компонентов:

```
var
  Intf: ITextAndValue;
  I: integer;
begin
  for I := 0 to ComponentCount - 1 do
    if Supports (Components [I], ITextAndValue, Intf) then
      Intf.Value := 50;
end;
```

В конкретном примере этот код будет влиять на `Value` индикатора выполнения или числового поля, а также на `Text` метки или поля редактирования. Он также полностью проигнорирует пару других компонентов, для которых я не определил интерфейс адаптера. Хотя это всего лишь очень специфический случай, если вы изучите другие шаблоны проектирования, вы легко обнаружите, что многие из них могут быть лучше реализованы, используя преимущества сверх-гибкости интерфейсов по сравнению с классами в Object Pascal (как в Java и C#, просто чтобы назвать еще пару популярных языков, которые широко используют интерфейсы).

480- начало

12: Работа с классами

В последних главах вы видели основы объектной стороны языка Object Pascal: классы, объекты, методы, конструкторы, наследование, поздняя привязка, интерфейсы и многое другое. Теперь нам нужно двигаться на шаг дальше, рассматривая некоторые более продвинутые и довольно специфические особенности языка, связанные с управлением классами. Начав с ссылок на классы до помощников классов, в этой главе мы рассмотрим многие особенности, которые не встречаются в других ООП-языках или, по крайней мере, реализованы значительно иначе.

В центре внимания находятся классы, а также манипулирование классами во время выполнения - тема, которую мы дополнительно рассмотрим в Главе 16, когда будем рассматривать отражение и атрибуты.

Методы и данные классов

Когда вы описываете класс в Object Pascal и большинстве других ООП-языков, вы определяете структуру данных объектов (или экземпляров) класса и операции, которые вы можете выполнять над таким объектом. Однако существует также возможность определить данные, совместно используемые всеми объектами класса, и методы, которые могут быть вызваны для класса независимо от любого реального объекта, созданного из него.

Чтобы объявить метод класса в Object Pascal, достаточно добавить ключевое слово `class` перед ним, как показано здесь для процедур и для функций:

```
type
  TMyClass = class
    class function ClassMeanValue: Integer;
```

Имея объект `MyObject` класса `TMyClass`, вы можете вызвать метод, применив его как к объекту, так и к классу в целом:

```
var
  MyObject: TMyClass;
begin
  ...
  I := TMyClass.ClassMeanValue;
  J := MyObject.ClassMeanValue;
```

Данный синтаксис подразумевает, что вызов метода класса возможен еще до того, как будет создан объект класса. Существуют случаи классов, сделанных только из методов класса, с неявной идеей, что вы никогда не создадите объекты этих классов (например, вы можете реализовать это, объявив конструктор `Create` private).

примечание Использование методов классов вообще и классов, сделанных только из методов классов в частности, более распространено в ООП-языках, которые не допускают использования глобальных функций. Объект Pascal по-прежнему позволяет объявить *старомодные* глобальные функции, но за последние годы системные

библиотеки и код, написанный разработчиками, все больше и больше смещаются в сторону последовательного использования методов классов. Преимущество использования методов классов состоит в том, что они становятся логически связанными с классом, который выступает в роли своеобразного пространства имен для группы связанных функций.

Данные классов

Данные класса - это данные, совместно используемые всеми объектами класса, предлагающие глобальное хранилище, но доступ к ним зависит от класса (включая ограничения доступа). Как объявить данные класса? Просто определив новую секцию описания класса, отмеченную комбинацией ключевых слов class var:

```
type
  TMyData = class
  private
    class var
      CommonCount: Integer;
  public
    class function GetCommon: Integer;
```

В секции `class var` вводится блок из одной или нескольких деклараций. Вы можете использовать секцию `var` (что является новым способом использования этого ключевого слова) для объявления других полей *экземпляров* в той же секции (`private` ниже):

```
type
  TMyData = class
  private
    class var
      CommonCount: Integer;
    var
      MoreObjectData: string;
  public
    class function GetCommon: Integer;
```

Помимо декларирования данных класса, можно также определить свойства класса, как мы увидим в следующем разделе.

Виртуальные методы класса и скрытый параметр self

В то время как сама концепция методов класса популярна среди языков программирования, реализация Object Pascal имеет несколько особенностей. Во-первых, методы класса имеют неявный (или скрытый) параметр `self`, как и метод instance. Однако этот скрытый параметр `self` является ссылкой на сам класс, а не на экземпляр класса.

На первый взгляд, тот факт, что метод класса имеет скрытый параметр, относящийся к самому классу, может показаться совершенно бесполезным. Ведь компилятор все-таки знает класс метода. Однако есть особенность языка, которая объясняет это: в отличие от большинства других языков, в Object Pascal методы класса могут быть виртуальными. В производном классе, как и в обычном методе, можно переопределить метод базового типа.

примечание Поддержка виртуального метода класса связана с поддержкой виртуальных конструкторов (которые являются своего рода специальными методами класса). Обе эти возможности не встречаются во многих скомпилированных и сильно типизированных ООП языках.

Статические методы классов

Для обеспечения совместимости платформ в язык были введены статические методы классов. Различия между обычными и статическими методами классов заключаются в том, что статические методы классов не имеют ссылок на собственный класс (нет параметра self, указывающего на сам класс) и не могут быть виртуальными.

Приведем простой пример с закомментированными некоторыми некорректными утверждениями, взятыми из проекта приложения ClassStatic:

```
type
  TBase = class
  private
    Tmp: Integer;
  public
    class procedure One;
    class procedure Two; static;
    ...
  end;

class procedure TBase.One;
begin
  // Error: Instance member 'Tmp' inaccessible here
  // Show (Tmp);
  Show ('one');
  Show (self.ClassName);
end;

class procedure TBase.Two;
begin
  Show ('two');
  // error: Undeclared identifier: 'self'
  // Show (self.ClassName);
  Show (ClassName);
  Two;
end;
```

В обоих случаях можно напрямую вызывать эти методы класса или вызывать их через объект:

```
  TBase.One;
  TBase.Two;

  Base := TBase.Create;
  Base.One;
  Base.Two;
```

Есть две интересные особенности, которые делают статические методы класса полезными в Object Pascal. Первая заключается в том, что они могут быть использованы для определения свойств класса, как описано в следующем разделе. Вторая заключается в том, что статические методы класса полностью совместимы с языком Си, как это объясняется ниже.

Статические методы классов и обратные вызовы Windows API

Тот факт, что они не имеют скрытых параметров self, подразумевает, что статические методы классов могут передаваться в операционную систему (например, в Windows) в качестве функций обратного вызова. На практике можно объявить статический метод класса с условным вызовом `stdcall` и использовать его в качестве прямого обратного вызова Windows API, как это было сделано для метода `TimerCallBack` прикладного проекта StaticCallBack:

```
type
  TFormCallBack = class(TForm)
    ListBox1: TListBox;
    procedure FormCreate(Sender: TObject);
  private
    class var
      NTimerCount: Integer;
  public
    class procedure TimerCallBack (hwnd: THandle;
      uMsg, idEvent, dwTime: Cardinal); static; stdcall;
  end;
```

Данные класса используются обратным вызовом в качестве счетчика. Обработчик OnCreate вызывает API SetTimer, передавая адрес статической процедуры класса:

```
procedure TFormCallBack.FormCreate(Sender: TObject);
var
  Callback: TFNTimerProc;
begin
  NTimerCount := 0;
  Callback := TFNTimerProc(@TFormCallBack.TimerCallBack);
  SetTimer(Handle, TIMERID, 1000, Callback);
end;
```

примечание Параметр в `TFNTimeProc` является указателем на метод, поэтому имени статического метода класса должен предшествовать @ или передаваться с помощью функции `Addr`. Это связано с тем, что нам нужно получить адрес метода, а не выполнять его.

Теперь фактическая функция обратного вызова увеличивает таймер и обновляет форму, ссылаясь на нее с помощью

соответствующей глобальной переменной, так как метод класса не может ссылаться на форму как на self:

```
class procedure TFormCallBack.TimerCallBack(
  hwnd: THandle; uMsg, idEvent, dwTime: Cardinal);
begin
  try
    Inc (NTimerCount);
    FormCallBack.ListBox1.Items.Add (
      IntToStr (NTimerCount) + ' at ' + TimeToStr(Now));
  except on E: Exception do
    Application.HandleException(nil);
  end;
end;
```

Блок `try-except` здесь для того, чтобы избежать любых исключений, отправляемых обратно в Windows... правило, которое вы должны последовательно использовать для функций обратного вызова или DLL.

Свойства класса

Одной из причин использования статических методов класса является реализация свойств класса. Что такое свойство класса? Как и стандартное свойство, это символ, прикрепленный к механизмам чтения и записи. В отличие от стандартного свойства, оно относится к классу и должно быть реализовано либо с помощью данных класса, либо с помощью статических методов класса. Класс `TBase` (опять же из проекта приложения ClassStatic) имеет два свойства класса, определенных двумя разными способами:

```
type
  TBase = class
  private
    class var
      FMyName: string;
  public
    class function GetMyName: string; static;
    class procedure SetMyName (Value: string); static;
    class property MyName: string read GetMyName write SetMyName;
    class property DirectName: string read FMyName write FMyName;
  end;
```

Класс со счетчиком инстанций

Данные и методы класса могут использоваться для хранения информации о классе в целом. Примером такой информации может служить количество экземпляров класса, которые были созданы до сих пор... за вычетом уже уничтоженных.

Проект приложения CountObj показывает этот сценарий. Программа не очень полезна, так как я предпочитал сосредоточиться только на конкретной проблеме и ее решении. Другими словами, целевой объект имеет очень простой класс, в котором просто хранится числовое значение:

```
type
  TCountedObj = class (TObject)
  private
    FValue: Integer;
  private class var
    FTotal: Integer;
    FCurrent: Integer;
  public
    constructor Create;
    destructor Destroy; override;
    property Value: Integer read FValue write FValue;
  public
    class function GetTotal: Integer;
    class function GetCurrent: Integer;
  end;
```

Каждый раз, когда создается объект, программа инкрементирует оба счетчика после вызова конструктора базового класса (если таковой имеется). Каждый раз при уничтожении объекта счетчик текущих объектов уменьшается:

```
constructor TCountedObj.Create (AOwner: TComponent);
begin
  inherited Create;
  Inc (FTotal);
  Inc (FCurrent);
end;

destructor TCountedObj.Destroy;
begin
  Dec (FCurrent);
  inherited Destroy;
end;
```

Доступ к информации о классе возможен без обращения к конкретному объекту. На самом деле, вполне возможно, что в определенный момент времени в памяти нет объектов:

```
class function TCountedObj.GetTotal: Integer;
begin
  Result := FTotal;
end;
```

Текущее состояние можно отобразить с помощью кода, например:

```
Label1.Text := TCountedObj.GetCurrent.ToString + '/' +
  TCountedObj.GetTotal.ToString;
```

В демо-примере это выполняется в таймере, который обновляет метку, так что нет необходимости обращаться к какому-либо конкретному экземпляру объекта, и он не запускается непосредственно ручной операцией. Кнопки в проекте приложения вместо этого просто создают и освобождают некоторые объекты... или оставляют их в памяти (на самом деле программа имеет некоторые потенциальные утечки памяти).

Конструкторы (и деструкторы) классов

Конструкторы классов предлагают способ инициализации данных, которые относятся к классу, и играют роль инициализаторов класса, так как в итоге они действительно ничего не *конструируют*. Конструктор класса не имеет ничего общего с конструктором стандартного экземпляра: это всего лишь код, используемый для инициализации самого класса один раз, перед использованием класса. Например, конструктор класса может установить начальные значения для

данных класса, загрузить конфигурацию или файлы поддержки класса и так далее.

В Object Pascal конструктор класса является альтернативой коду инициализации модуля. Если оба существуют (в модуле), то сначала будет выполнен конструктор класса, а затем код инициализации модуля. Напротив, можно определить деструктор класса, который будет выполнен после кода инициализации.

Существенное отличие, однако, заключается в том, что если код инициализации модуля всегда выполняется, если модуль скомпилирован в программе, то конструктор класса и деструктор связываются только в том случае, если класс фактически используется. Это означает, что использование конструктора класса более *дружественно к компоновке*, чем использование кода инициализации.

примечание Другими словами, с помощью конструкторов и деструкторов классов, если тип не связан, то код инициализации не является частью программы и не выполняется; в традиционном случае верно обратное, код инициализации может даже привести к тому, что компоновщик внесет часть кода класса, даже если на самом деле он никогда больше нигде не используется. На практике это было введено вместе с библиотекой поддержки жестов - довольно большим количеством кода, который не компилируется в исполняемый файл, если он не используется.

При кодировании можно написать следующее (см. прикладной проект ClassCtor):

```
type
  TTestClass = class
  public
  class var
    StartTime: TDateTime;
    EndTime: TDateTime;
  public
    class constructor Create;
    class destructor Destroy;
  end;
```

Класс имеет два поля данных класса, инициализированных конструктором класса и модифицированных деструктором

класса, в то время как секции инициализации и завершения модуля используют эти поля данных:

```
class constructor TTestClass.Create;
begin
  StartTime := Now;
end;

class destructor TTestClass.Destroy;
begin
  EndTime := Now;
end;

initialization
  ShowMessage (TimeToStr (TTestClass.StartTime));

finalization
  ShowMessage (TimeToStr (TTestClass.EndTime));
```

Последовательность запуска работает так, как и ожидалось, при этом данные класса уже доступны, так как вы показываете информацию. При закрытии вместо этого выполняется вызов `ShowMessage` до того, как значение будет присвоено деструктором класса.

Обратите внимание, что вы можете дать конструктору и деструктору класса любое имя, хотя `Create` и `Destroy` будут очень хорошими настройками по умолчанию. Однако, вы не можете определить несколько конструкторов или деструкторов класса. Если вы попробуете, компилятор выдаст следующее сообщение об ошибке:

```
[DCC Error] ClassCtorMainForm.pas(34): E2359 Multiple class constructors
in class TTestClass: Create and Foo
```

Конструкторы классов в RTL

Есть несколько RTL классов, которые уже используют эту возможность языка, например, класс Exception, который определяет как конструктор класса (с кодом ниже), так и деструктор класса:

```
class constructor Exception.Create;
```

```
begin
  InitExceptions;
end;
```

Процедура `InitExceptions` ранее вызывалась в разделе инициализации модуля System.SysUtils.

В целом, я считаю, что использование конструкторов и деструкторов классов лучше, чем использование инициализации и завершения модулей. В большинстве случаев это только синтаксический сахар, так что, возможно, вы не захотите возвращаться назад и изменять существующий код. Однако, если вы столкнетесь с риском инициализации структур данных, которые вы врядли будете использовать (потому что ни один класс этого типа никогда не будет создан), переход к конструкторам классов даст определённое преимущество. Очевидно, что это более часто встречается в общей библиотеке, из которой вы не используете все возможности, чем в коде приложения.

примечание Очень специфический случай использования конструкторов классов - в случае общих классов. Я расскажу об этом в главе, посвященной дженерикам.

Реализация паттерна Singleton

Есть классы, для которых имеет смысл создать один и только один экземпляр. Шаблон Singleton (еще один очень распространенный шаблон проектирования) требует этого, а также предполагает наличие *глобальной точки доступа* для данного объекта.

Шаблон Singleton может быть реализован различными способами, но классический подход заключается в вызове функции, используемой для доступа к единственному экземпляру в точности как `Instance`. Во многих случаях реализация также следует правилу ленивой инициализации,

так что этот глобальный экземпляр создается не при запуске программы, а только при первом обращении.

В реализации ниже я использовал не только данные класса, методы класса, но и деструкторы класса для окончательной очистки. Вот соответствующий код:

```
type
  TSingleton = class(TObject)
  public
    class function Instance: TSingleton;
  private
    class var TheInstance: TSingleton;
    class destructor Destroy;
  end;

class function TSingleton.Instance: TSingleton;
begin
  if TheInstance = nil then
    TheInstance := TSingleton.Create;
  Result := TheInstance;
end;

class destructor TSingleton.Destroy;
begin
  FreeAndNil (TheInstance);
end;
```

Вы можете получить единственный экземпляр класса (независимо от того, был он уже создан или нет), написав:

```
var
  ASingle: TSingleton;
begin
  ASingle := TSingleton.Instance;
```

Кроме того, можно скрыть обычный конструктор класса, объявив его приватным, так что будет очень сложно создать объект класса, не следуя шаблону.

Ссылки на классы

Рассмотрев несколько тем, связанных с методами, мы теперь можем перейти к теме *ссылок на классы* и далее расширить

наш пример динамического создания компонентов. Первый момент, который следует помнить, это то, что ссылка на класс — это не класс, это не объект, и это не ссылка на объект, это просто ссылка на тип класса.

Тип ссылки класса определяет тип переменной класса. Звучит запутанно? Несколько строк кода могут сделать это более понятным. Предположим, вы определили класс `TMyClass`. Теперь можно определить новый тип класса, связанный с этим классом:

```
type
  TMyClassRef = class of TMyClass;
```

Теперь можно объявлять переменные обоих типов. Первая переменная относится к объекту, вторая - к классу:

```
var
  AClassRef: TMyClassRef;
  AnObject: TMyClass;
begin
  AClassRef := TMyClass;
  AnObject := TMyClass.Create;
```

Вы можете задаться вопросом, для чего используются ссылки на классы. В общем случае, ссылки на классы позволяют манипулировать типом данных класса во время выполнения. Ссылку на класс можно использовать в любом выражении, где использование типа данных легально. На самом деле таких выражений не так уж и много, и интересны лишь немногие случаи. Самый простой случай — это создание объекта. Мы можем переписать две приведенные выше строки следующим образом:

```
AClassRef := TMyClass;
AnObject := AClassRef.Create;
```

В этот раз я применил конструктор `Create` к ссылке на класс, а не к реальному классу; я использовал ссылку на класс для создания объекта этого класса.

примечание Ссылки на классы связаны с понятием *метакласса*, доступного на других ООП языках. В объекте Pascal, однако, ссылка на класс сама по себе не является классом, а только определенным типом, определяющим ссылку на данные класса. Поэтому аналогия с метаклассами (классами, описывающими другие классы) немного вводит в заблуждение. На самом деле, `TMetaclass` является термином, тоже используемым в C++Builder.

Когда у вас есть ссылка на класс, вы можете применить к нему методы класса, относящиеся к соответствующему классу. Таким образом, если бы в `TMyClass` был метод класса под названием `Foo`, вы бы смогли написать и то, и другое:

```
TMyClass.Foo
AClassRef.Foo
```

Это было бы не очень полезно, если бы ссылки на классы не поддерживали то же самое правило совместимости с типами, которое применяется к типам классов. Когда вы объявляете переменную со ссылкой на класс, такую как `MyClassRef` выше, вы можете присвоить ей этот конкретный класс и любой подкласс. Таким образом, если `MyNewClass` является подклассом моего класса, то вы также можете написать

```
AClassRef := MyNewClass;
```

Теперь, чтобы понять, почему это действительно может быть интересно, нужно помнить, что методы класса, которые можно вызвать для ссылки на класс, могут быть виртуальными, поэтому конкретный подкласс может их переопределить. Используя ссылки на классы и виртуальные методы классов, можно реализовать на уровне методов класса форму полиморфизма, которую поддерживают немногие (если таковые имеются) другие статические языки ООП. Также учтите, что каждый класс наследует от TObject, поэтому вы можете применить к каждому классу ссылку на некоторые методы `TObject`, включая InstanceSize, ClassName, ParentClass и InheritsFrom. Я расскажу об этих и других методах класса `TObject` в Гл. 17.

Ссылки на класс в RTL

Модуль `System` блок и другие основные модули RTL объявляют множество ссылок на классы, в том числе и следующие:

```
TClass = class of TObject;
ExceptClass = class of Exception;
TComponentClass = class of TComponent;
TControlClass = class of TControl;
TFormClass = class of TForm;
```

В частности, тип ссылки на класс TClass может быть использован для хранения ссылки на любой класс, который вы пишете в Object Pascal, потому что каждый класс в конечном счете является производным от `TObject`. Вместо этого ссылка на класс `TFormClass` используется в исходном коде проекта Object Pascal по умолчанию на основе FireMonkey или VCL. Метод `CreateForm` объекта `Application` обеих библиотек, фактически, требует в качестве параметра создания класса формы:

```
Application.CreateForm(TForm1, Form1);
```

Первый параметр - ссылка на класс, второй - переменная, которая будет получать ссылку на созданный экземпляр объекта.

Создание компонентов с использованием ссылок на классы

Каково *практическое* использование ссылок на классы в Object Pascal? Возможность манипулировать типом данных во время выполнения является фундаментальным элементом среды. Когда вы добавляете новый компонент в форму, выбирая его из Палитры компонентов, вы выбираете тип данных и создаете объект этого типа данных. (На самом деле, это то, что среда разработки делает для вас за кулисами).

Чтобы дать вам лучшее представление о том, как работают ссылки на классы, я создал проект приложения под названием ClassRef. Форма, отображаемая в этом примере, достаточно проста. Она имеет три радиокнопки, расположенные внутри панели в верхней части формы. Когда вы выберете одну из этих радиокнопок и нажмете на форму, вы сможете создавать новые компоненты трех типов, обозначенных метками кнопок: радиокнопки, обычные кнопки и поля редактирования. Для корректной работы программы необходимо изменить названия трех компонентов. Форма также должна иметь поле ссылки на класс:

```
private
  FControlType: TControlClass;
  FControlNo: Integer;
```

Первое поле хранит новый тип данных каждый раз, когда пользователь нажимает одну из трех радиокнопок, изменяя ее статус. Вот один из трех методов:

```
procedure TForm1.RadioButtonRadioChange(Sender: TObject);
begin
  FControlType := TRadioButton;
end;
```

Две другие радиокнопки имеют обработчики событий OnChange, похожие на эту, присваивающие значение TEdit или TButton полю `FControlType`. Аналогичное назначение присутствует и в обработчике события `OnCreate` формы, используемого в качестве метода инициализации. Интересная часть кода выполняется, когда пользователь нажимает на элемент управления Layout, который покрывает большую часть поверхности формы. Я выбрал событие OnMouseDown формы для удержания позиции щелчка мыши:

```
procedure TForm1.Layout1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Single);
var
  NewCtrl: TControl;
  NewName: String;
begin
  // create the control
```

```
  NewCtrl := FControlType.Create (Self);

  // hide it temporarily, to avoid flickering
  NewCtrl.Visible := False;

  // set parent and position
  NewCtrl.Parent := Layout1;
  NewCtrl.Position.X := X;
  NewCtrl.Position.Y := Y;

  // compute the unique name (and text)
  Inc (FControlNo);
  NewName := FControlType.ClassName + FControlNo.ToString;
  Delete (NewName, 1, 1);
  NewCtrl.Name := NewName;

  // now show it
  NewCtrl.Visible := True;
end;
```

Первая строка кода для этого метода - главная. В ней создается новый объект типа данных класса, хранящийся в поле `FControlType`. Это достигается простым применением конструктора `Create` к ссылке на класс. Теперь вы можете установить значение свойства Parent, задать положение нового компонента, дать ему имя (которое автоматически используется также как `Text`), и сделать его видимым.

Обратите внимание, в частности, на код, используемый для построения имени; для имитации стандартного соглашения об именовании Object Pascal я взял имя класса с выражением `FControlType.ClassName`, используя метод класса `TObject`. Затем я добавил число в конце имени и удалил начальную букву строки. Для первой радиокнопки основной строкой является `TRadioButton`, плюс *1* в конце, и минус *T* в начале имени класса - RadioButton1. Выглядит знакомо?

Рисунок 12.1: Пример вывода приложения ClassRef, под Window

Пример вывода этой программы показан на рисунке 12.1. Обратите внимание, что именование не совсем то же самое, что используется IDE, которая использует отдельный счетчик для каждого типа элемента управления. Вместо этого программа использует один счетчик для всех компонентов. Если вы поместите радиокнопку, кнопку и окно редактирования в виде приложения ClassRef, то их именами будут *RadioButton1, Button2* и *Edit3*, как показано на изображении (хотя у поля Edit нет видимого описания ее имени).

Кроме того, учтите, что после создания общего компонента, вы можете получить доступ к его свойствам очень динамичным способом, используя рефлексию - тема, подробно описанная в Главе 16. В той же главе мы увидим другие способы обращения к информации о типах и классах, помимо ссылок на классы.

Помощники классов и

записей

Как мы видели в Главе 8, концепция наследования между классами — это способ расширения класса, предоставляющий новые возможности, ни в коей мере не затрагивая оригинальную реализацию. Это реализация так называемого принципа open-closed: тип данных полностью определен (закрыт), но все же модифицируемый (открытый).

Хотя наследование типов является чрезвычайно мощным механизмом, существуют сценарии, в которых оно не идеально. Дело в том, что при работе с существующими и сложными библиотеками, возможно, вам захочется расширить тип данных, не наследуя от него новый. Это особенно верно, когда объекты создаются каким-то автоматическим способом, и замена их создания может быть чрезвычайно сложной.

Довольно очевидным сценарием для разработчиков Object Pascal является использование компонентов. Если вы хотите добавить метод в класс компонента, чтобы обеспечить некоторое новое поведение, вы действительно можете использовать наследование, но это подразумевает: создание нового производного типа, создание пакета для его установки, замену всех существующих компонентов в формах и других поверхностях проектирования на новый тип компонента (операция, которая влияет как на определение формы, так и на файл с исходным кодом).

Альтернативный подход заключается в использовании помощников (хелперов) класса или записи. Эти специальные типы данных могут расширить существующий тип за счет новых методов. Даже если у них есть несколько ограничений, хелперы классов позволяют работать со сценарием, подобным тому, который я только что описал, просто добавляя новые

методы к существующему компоненту, без необходимости изменять реальный тип компонента.

примечание На самом деле мы уже видели альтернативный подход к расширению класса библиотеки без полной замены его ссылок, используя наследование и одноименный класс, идиому межкомпозитного класса. Я рассказал об этой идиоме в заключительной части последней главы. Хелперы классов предлагают более чистую модель, однако они не могут быть использованы для замены виртуальных методов или реализации дополнительного интерфейса, как я делал это в применении последней главы.

Помощники класса

Помощник класса – это способ добавления методов и свойств к классу, который вы *не* имеете *права* изменять (как класс библиотеки). Использование помощника класса для расширения класса в вашем собственном коде действительно необычно, так как в этом случае вам, как правило, достаточно просто перейти к изменению собственно класса.

Что нельзя сделать в помощнике класса, так это добавить данные экземпляра, учитывая, что данные должны жить в реальных объектах, и они определяются их оригинальным классом, или коснуться виртуальных методов, опять же определенных в физической структуре оригинального класса. Другими словами, вспомогательный класс может добавлять или заменять только невиртуальные методы существующего класса. Таким образом, вы сможете применить новый метод к объекту исходного класса, даже если этот класс понятия не имеет о существовании метода.

Если это непонятно, а скорее всего нет, то давайте рассмотрим пример (взятый из прикладного проекта ClassHelperDemo — это всего лишь демонстрация того, что не следует делать, используйте помощники классов для дополнения своего собственного класса):

```
type
  TMyObject = class
  protected
    Value: Integer;
    Text: string;
  public
    procedure Increase;
  end;

  TMyObjectHelper = class helper for TMyObject
  public
    procedure Show;
  end;
```

Предыдущий код объявляет класс и помощника для этого класса. Это означает, что для объекта типа `TMyObject` можно вызывать как методы исходного класса, так и методы помощника класса:

```
  Obj := TMyObject.Create;
  Obj.Text := 'foo';
  Obj.Show;
```

Методы хелпер-класса становятся частью класса и могут, как и любой другой метод, использовать `Self` для обращения к текущему объекту (из класса это помогает, так как хелперы класса не инстанцируются), как демонстрирует данный код:

```
procedure TMyObjectHelper.Show;
begin
  Show (Text + ' ' + IntToStr (Value) + ' -- ' +
    ClassName + ' -- ' + ToString);
end;
```

Наконец, обратите внимание, что метод вспомогательного класса может переопределить исходный метод. В код я добавил метод `Show` как в класс, так и в вспомогательный класс, но вызывается только вспомогательный метод!

Конечно, нет смысла объявлять класс и расширение к тому же самому классу, используя синтаксис помощника класса в том же самом модуле или даже в той же программе. Я сделал это на демонстрационном примере только для того, чтобы было легче понять техническую сторону этого синтаксиса.
Вспомогательные классы *не должны* использоваться в качестве

общей языковой конструкции для разработки архитектур приложений, но в основном они предназначены для расширения библиотек классов, для которых у вас нет исходного кода, или которые вы не хотите изменять, чтобы избежать конфликтов в будущем.

Есть еще несколько правил, которые применяются к помощникам класса. Методы помощника класса:

могут иметь иные спецификаторы доступа, чем оригинальный метод в классе

могут быть методы класса или экземпляра, переменные класса и свойства

могут быть виртуальными методами, которые могут быть переопределены в производном классе (хотя я нахожу это немного неудобным с практической точки зрения)

могут внедрить дополнительные конструкторы

могут добавлять вложенные константы к определению типа

Единственная особенность, которой им не хватает по конструкции — это данные об экземплярах. Также обратите внимание, что помощники классов включаются по мере того, как они становятся видимыми в области видимости. Нужно добавить оператор `uses`, ссылающийся на модуль, который объявляет хелпер класса, чтобы видеть его методы, а не просто включать его один раз в процесс компиляции.

примечание Некоторое время в компиляторе Delphi возникла ошибка, в результате которой хелперы классов получили доступ к приватным полям класса, в котором они помогали, вне зависимости от того, в каком модуле класс был объявлен. Этот "взлом" в основном нарушил правила объектно-ориентированного программирования, инкапсуляции. Для реализации семантики видимости помощники классов и записей в последних версиях компиляторов Object Pascal (начиная с Delphi 10.1 Berlin) не могут получить доступ к приватным членам классов или записям, которые они расширяют. Это действительно привело к тому, что существующий код перестал работать, тот, который использовал этот

взлом (который никогда не был предназначен для использования в качестве функции языка).

Помощник класса для списка Listbox

Практическое использование помощников классов заключается в предоставлении дополнительных методов для классов библиотеки. Причина в том, что вы не хотите менять эти классы напрямую (даже если у вас есть исходный код, вы не хотите редактировать исходные тексты основной библиотеки) или наследовать от них (так как это заставит вас заменить компоненты в формах во время проектирования).

В качестве примера рассмотрим простой случай: вам нужен простой способ получить текст текущего выделения окна списка. Вместо того, чтобы написать классический код:

```
ListBox1.Items [ListBox1.ItemIndex]
```

можно определить помощник класса следующим образом (взятый из проекта ControlHelper):

```
type
  TListboxHelper = class helper for TListBox
    function ItemIndexValue: string;
  end;

function TListboxHelper.ItemIndexValue: string;
begin
  Result := '';
  if ItemIndex >= 0 then
    Result := Items [ItemIndex];
end;
```

Теперь вы можете обратиться к выбранному пункту списка как:

```
Show (ListBox1.ItemIndexValue);
```

Это очень простой случай, но он показывает идею очень выгодной в практическом плане.

Помощники классов и

наследование

Наиболее существенным ограничением помощников является то, что для каждого класса в данный момент времени может быть только один помощник. Если компилятор столкнется с двумя вспомогательными классами, то второй заменит первый. Цепочки помощников классов отсутствуют, т.е. есть помощник класса, который в дальнейшем расширяет класс, уже расширенный помощником другого класса.

Частичное решение этой проблемы заключается в том, что вы можете ввести помощника класса для класса и добавить следующего помощника класса для унаследованного класса... но вы не можете напрямую унаследовать помощника класса от другого помощника класса. Я не призываю к погружению в сложные структуры помощников классов, потому что они действительно могут превратить ваш код в какой-то очень запутанный код.

Примером может служить запись TGUID, структура данных Windows, которую можно использовать на разных платформах в Object Pascal, и которая имеет помощника, добавляющего несколько общих возможностей:

```
type
  TGuidHelper = record helper for TGUID
    class function Create(const B: TBytes): TGUID; overload; static;
    class function Create(const S: string): TGUID; overload; static;
    // ... more Create overloads omitted
    class function NewGuid: TGUID; static;
    function ToByteArray: TBytes;
    function ToString: string;
  end;
```

Вы, возможно, заметили, что `TGuidHelper является` помощником в записи, а не помощником в классе. Да, в записях могут быть помощники так же, как и в классах.

Добавление перечисления элементов управления с помощью помощника класса

Любой компонент Delphi в библиотеках автоматически определяет счетчик, который вы можете использовать для обработки каждого из дочерних (owned) компонентов. Например, в методе формы вы можете перечислить компоненты, принадлежащие форме, написав:

```
for var AComp in self do
    ... // use AComp
```

Другой распространенной операцией является навигация по дочерним элементам управления, которые включают только визуальные компоненты (исключая невизуальные компоненты, такие как TMainMenu), для которых имеют форма - прямой «родитель» (исключая элементы управления, размещенные на дочернем элементе управления, такие как кнопка на панели). Метод, который мы можем использовать для написания более простого кода для циклического перемещения по дочерним элементам управления, заключается в добавлении нового перечисления в класс TWinControl, написав следующий помощник класса:

```
type
  TControlsEnumHelper = class helper for TWinControl
  type
    TControlsEnum = class
    private
      NPos: Integer;
      FControl: TWinControl;
    public
      constructor Create (aControl: TWinControl);
      function MoveNext: Boolean;
      function GetCurrent: TControl;
      property Current: TControl read GetCurrent;
    end;
  public
    function GetEnumerator: TControlsEnum;
  end;
```

примечание Причина, по которой помощник для `TWinControl`, а не для `TControl`, заключается в том, что только элементы управления с оконным хендлером могут быть родителями других элементов управления. В основном это исключает графические элементы управления.

Это полный код помощника, включающий его единственный метод и вложенный класс `TControlsEnum`:

```
{ TControlsEnumHelper }

function TControlsEnumHelper.GetEnumerator: TControlsEnum;
begin
  Result := TControlsEnum.Create (self);
end;

{ TControlsEnumHelper.TControlsEnum }

constructor TcontrolsEnumHelper.TcontrolsEnum.Create(
  aControl: TWinControl);
begin
  FControl := aControl;
  NPos := -1;
end;

function TControlsEnumHelper.TControlsEnum.GetCurrent: TControl;
begin
  Result := FControl.Controls[nPos];
end;

function TControlsEnumHelper.TControlsEnum.MoveNext: Boolean;
begin
  Inc (NPos);
  Result := NPos < FControl.ControlCount;
end;
```

Теперь, если мы создадим такую форму, как показано на рисунке 12.2, мы сможем протестировать перечисление в различных сценариях. Первый случай — это то, для чего мы специально написали код, то есть перечислили дочерние элементы управления формой:

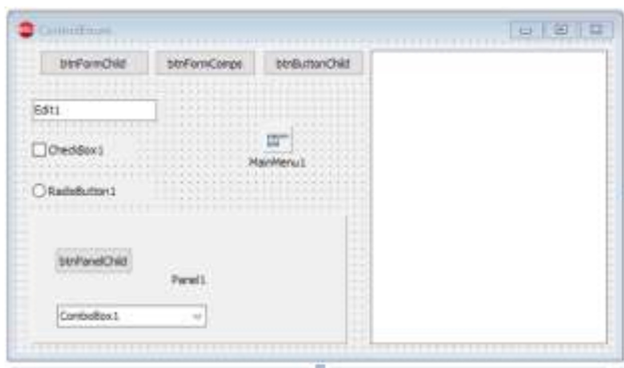


Рисунок 12.2: Форма, используемая для тестирования помощника по перечислению элементов управления во время проектирования в Delphi IDE

```
procedure TControlEnumForm.BtnFormChildClick(Sender: TObject);
begin
  Memo1.Lines.Add ('Form Child');
  for var ACtrl in self do
    Memo1.Lines.Add (ACtrl.Name);
  Memo1.Lines.Add ('');
end;
```

Вот вывод операции в поле memo, в котором перечислены дочерние элементы управления формы, но не другие компоненты или элементы управления, родителями которых является панель:

```
Form Child
Memo1
BtnFormChild
Edit1
CheckBox1
RadioButton1
Panel1
BtnFormComps
BtnButtonChild
```

Полный список будет показан, если мы перечислим все компоненты. Однако, у нас есть проблема с использованием кода, который я показывал в начале этого раздела, так как мы переопределили метод `GetNumerator` новой версией (в хелпере класса) и по этой причине не можем получить прямой доступ к базовому перечислителю `TComponent`. Хелпер определен для `TWinControl`, поэтому мы можем использовать трюк. Если мы

приведем наш объект к `TComponent`, то код вызовет стандартный, предопределенный перечислитель:

```
procedure TControlEnumForm.BtnFormCompsClick(Sender: TObject);
begin
  Memo1.Lines.Add ('Form Components');
  for var AComp in TComponent(self) do
    Memo1.Lines.Add (AComp.Name);
  Memo1.Lines.Add ('');
end;
```

Вот вывод, в котором перечислено больше компонентов, чем в предыдущем списке:

```
Form Components
Memo1
BtnFormChild
Edit1
CheckBox1
RadioButton1
Panel1
BtnPanelChild
ComboBox1
BtnFormComps
BtnButtonChild
MainMenu1
```

В проекте приложения ControlsEnum я также добавил код для перечисления дочерних элементов управления панели и одной из кнопок (в основном для проверки корректной работы перечислителя, когда список пуст).

Помощники записей для внутренних типов

Еще одним расширением концепций помощника записи является возможность добавления методов к родным (или *присущим компилятору*) типам данных. Хотя используется тот же синтаксис "record helper", он применяется не к записям, а к обычным типам данных.

примечание Помощники записей в настоящее время используются для дополнения и добавления похожих на методы операций к родным типам данных, но это может

> измениться в будущем. Сегодняшняя библиотека времени исполнения определяет несколько нативных помощников, которые могут исчезнуть в будущем, сохраняя то, как вы пишете код, использующий эти помощники... но нарушая совместимость в коде, который их определяет. Поэтому не стоит злоупотреблять этой возможностью, даже если она, конечно, очень красива и удобна.

Как на практике работают помощники внутреннего типа? Рассмотрим следующее описание помощника для типа `Integer`:

```
type
  TIntHelper = record helper for Integer
    function AsString: string;
  end;
```

Теперь, учитывая `Integer` переменную `N`, можно писать:

```
N.AsString;
```

Как определить этот псевдометод и как он может ссылаться на значение переменной? Путем растягивания значения ключевого слова `self` для обращения к значению, к которому применяется функция:

```
function TIntHelper.AsString: string;
begin
  Result := IntToStr (self);
end;
```

Обратите внимание, что вы можете применять методы также и к константам, как в:

```
Caption := 400000.AsString;
```

Однако, вы не можете сделать то же самое для значения small, так как компилятор интерпретирует константы меньшего возможного типа. Поэтому, если вы хотите получить значение 4 в виде строки, вы должны использовать вторую форму:

```
 Caption := 4.AsString; // nope!
  Caption := Integer(4).AsString; // ok
```

Или вы можете сделать первое утверждение для компиляции, определив другого помощника:

```
type
  TByteHelper = record helper for Byte...
```

Как мы уже видели в Гл. 2, на самом деле нет необходимости писать код выше для таких типов, как `Integer` и `Byte`, так как библиотека времени исполнения определяет довольно полный список помощников классов для большинства основных типов данных, включая следующие, которые определены в модуле System.SysUtils:

```
TStringHelper = record helper for string
TSingleHelper = record helper for Single
TDoubleHelper = record helper for Double
TExtendedHelper = record helper for Extended
TByteHelper = record helper for Byte
TShortIntHelper = record helper for ShortInt
TSmallIntHelper = record helper for SmallInt
TWordHelper = record helper for Word
TCardinalHelper = record helper for Cardinal
TIntegerHelper = record helper for Integer
TInt64Helper = record helper for Int64
TUInt64Helper = record helper for UInt64
TNativeIntHelper = record helper for NativeInt
TNativeUIntHelper = record helper for NativeUInt
TBooleanHelper = record helper for Boolean
TByteBoolHelper = record helper for ByteBool
TWordBoolHelper = record helper for WordBool
TLongBoolHelper = record helper for LongBool
TWordBoolHelper = record helper for WordBool
```

Есть несколько других внутренних помощников типа, которые в настоящее время определены в других модулях, например:

```
// System.Character:
TCharHelper = record helper for Char
// System.Classes:
TUInt32Helper = record helper for UInt32
```

Учитывая, что я рассказал об использовании этих помощников на многих примерах в начальной части книги, нет необходимости повторять их здесь. В этот раздел добавлено описание того, как можно определить intrinsic type helper.

Помощники для псевдонимов

типов

Как мы видели, невозможно определить двух помощников для одного типа, не говоря уже о внутреннем типе. Так как же добавить дополнительную прямую операцию к нативному типу, например, Integer? Четкого решения нет, но есть некоторые возможные обходные пути (если не скопировать исходный код помощника внутреннего класса и не дублировать его дополнительным методом).

Решение, которое мне нравится, это определение псевдонима. Псевдоним типа рассматривается компилятором как совершенно новый тип, поэтому у него может быть свой помощник без замены помощника оригинального типа. Теперь, когда типы разделены, к одной и той же переменной все равно нельзя применить методы обоих помощников класса, но один из наборов будет соответствовать типу. Поясню это в терминах кода. Предположим, вы создадите псевдоним типа:

```
type
  MyInt = type Integer;
```

Теперь вы можете определить помощника для этого типа:

```
type
  TMyIntHelper = record helper for MyInt
    function AsString: string;
  end;

function MyIntHelper.AsString: string;
begin
  Result := IntToStr (self);
end;
```

Если вы объявите переменную этого нового типа, вы можете вызвать помощник конкретного метода, но все равно вызываете методы помощника целочисленного типа с помощью преобразования:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  mi: MyInt;
begin
```

```
  mi := 10;
  Show (mi.asString);
  // Show (mi.toString); // this doesn't work
  Show (Integer(mi).ToString)
end;
```

Этот код находится в прикладном проекте TypeAliasHelper для того, чтобы вы могли попробовать другие вариаци

13: Объекты и память

Эта глава посвящена очень специфической и довольно важной теме - управлению памятью на языке Object Pascal. Язык и его среда исполнения предлагают довольно уникальное решение, которое находится между ручным управлением памятью в стиле C++ и сборкой мусора на Java или C#.

Причина такого промежуточного подхода заключается в том, что он помогает избежать большинства проблем, связанных с ручным управлением памятью (но явно не всех), без ограничений и проблем, вызванных сбором мусора, от дополнительного выделения памяти до недетерминированной утилизации.

примечание У меня нет особого намерения углубляться в проблемы стратегий GC (сбор мусора) и как они реализуются на различных платформах. Это скорее

исследовательская тема. Актуальным является то, что на ограниченных устройствах, таких как мобильные, GC, кажется, далеки от идеала, но некоторые из тех же вопросов относятся к каждой платформе. Тенденция игнорирования потребления памяти Windows-приложений принесла нам небольшие утилиты по 1 ГБ каждая.

Однако в Object Pascal все немного усложняет тот факт, что переменная использует память в зависимости от типа данных, при этом некоторые типы используют подсчет ссылок, а некоторые - более традиционный подход. Модель собственности, основанная на компонентах, и несколько других опций делают управление памятью сложной темой. В этой главе мы рассмотрим ее, начиная с некоторых основ управления памятью в современных языках программирования и концепций, лежащих в основе объектной модели обращения.

примечание В течение многих лет компиляторы Delphi для мобильных устройств предлагали другую модель памяти, называемую ARC, или автоматический подсчет ссылок. Поддерживаемый компанией Apple на своих языках, ARC добавляет поддержку компилятора для отслеживания и подсчета ссылок на объект, уничтожая его, когда он больше не нужен, счетчик ссылок уменьшается до нуля). Это очень похоже на то, что происходит со ссылками на интерфейс в Delphi на всех платформах. Начиная с Delphi 10.4 поддержка ARC была удалена из языка для всех платформ.

Глобальные данные, стек и куча

Память, используемая любым приложением Object Pascal на любой платформе, может быть разделена на две области: код и данные. В область памяти приложения загружаются порции исполняемого файла программы, ее ресурсов (битовых карт и файлов описания форм), а также библиотек, используемых программой. Эти блоки памяти доступны только для чтения и

(на некоторых платформах типа Windows) могут совместно использоваться несколькими процессами.

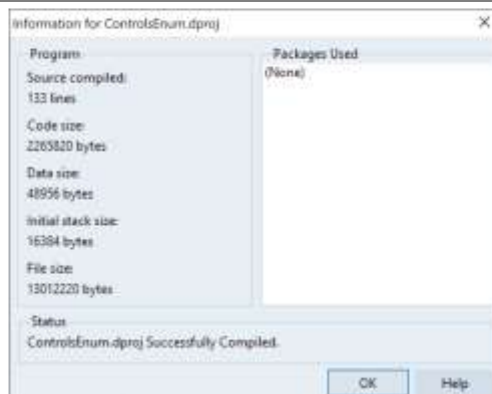
Интереснее посмотреть на область данных. Данные программы Object Pascal (как и данных программ, написанных на большинстве других языков) хранятся в трех четко различимых областях: глобальной памяти, стеке и куче.

Глобальная память

Когда компилятор Object Pascal генерирует исполняемый файл, он определяет пространство, необходимое для хранения переменных, которые существуют в течение всего времени жизни программы. Глобальные переменные, объявленные в интерфейсе или в части реализации модулей, попадают в эту категорию. Обратите внимание, что если глобальная переменная имеет тип класса (а также строковый или динамический массив), то в глобальной памяти хранится только 4- или 8-байтовая ссылка на объект.

Размер глобальной памяти можно определить с помощью пункта меню Project | Information после компиляции программы. Конкретное поле, на которое вы хотите посмотреть - *Размер данных*. На рисунке 13.1 показано использование почти 50K глобальных данных (48 956 байт), которые включают в себя глобальные данные как вашей программы, так и библиотек, которые вы используете.

Рисунок 13.1: Информация о скомпилированной программе



Иногда глобальную память называют статической, так как после загрузки программы переменные остаются в исходном месте и память никогда не освобождается.

Стек

Стек — это динамическая область памяти, которая выделяется и разворачивается в соответствии с ордером LIFO: Last In, First Out. Это означает, что последний выделенный объект памяти будет удален первым. Представление стековой памяти показано на рисунке 13.2.

Стековая память обычно используется подпрограммами (вызовами процедур, функций и методов) для передачи параметров и их возвратных значений, а также для локальных переменных, которые вы объявляете внутри функции или метода. После завершения вызова подпрограммы область памяти стека освобождается. В любом случае, помните, что при использовании стандартного для Object Pascal соглашения о вызове по умолчанию, параметры передаются в регистрах CPU, а не в стеке, когда это возможно.

Обратите также внимание, что стековая память, как правило, не инициализируется и не очищается, чтобы сэкономить время. Поэтому, если объявить, скажем, целое число как локальную переменную и просто прочитать его значение, то можно получить практически все. Все локальные переменные должны быть инициализированы перед их использованием.

Размер стека, как правило, фиксируется и определяется процессом компиляции. Этот параметр можно задать на странице компоновщика опций проекта. Тем не менее, по умолчанию, как правило, все в порядке. Если вы получаете сообщение об ошибке "переполнение стека", то, скорее всего, это происходит потому, что у вас есть функция, рекурсивно вызывающая себя бесконечно, а не потому, что пространство стека слишком ограничено. Начальный размер стека — это еще одна информация, предоставленная в информационном диалоге проекта |.

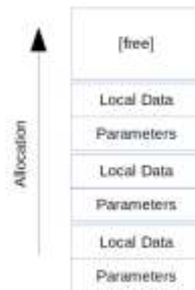


Рисунок 13.2: Представление области стековой памяти

Куча

Куча — это область, в которой распределение и разделение памяти происходит в случайном порядке. Это означает, что при последовательном выделении трех блоков памяти они могут быть в дальнейшем уничтожены в любом порядке. Менеджер

кучи заботится обо всех деталях, поэтому вы просто запрашиваете новую память с помощью низкоуровневой функции GetMem или вызываете конструктор для создания объекта, и система возвращает вам новый блок памяти (возможно, повторно используя уже отброшенные блоки памяти). Объект Pascal использует кучу памяти для выделения памяти каждого объекта, текста строк, динамических массивов и большинства других структур данных.

Так как она динамическая, то куча — это область памяти, в которой у программ, как правило, больше всего проблем:

- Каждый раз, когда создается объект, он должен быть уничтожен. Неспособность сделать это - сценарий, называемый "утечкой памяти", который не причинит слишком много вреда, если его не повторить снова и снова, пока вся куча памяти не будет заполнена.
- Каждый раз, когда уничтожается объект, необходимо убедиться, что он больше не используется, и что программа не пытается уничтожить его второй раз.

То же самое верно и для любой другой динамически создаваемой структуры данных, но языковая среда исполнения заботится о строках и динамических массивах в основном автоматически, так что вам почти никогда не придётся беспокоиться о них.

Объектная модель

Как мы видели в главе 7, объекты в языке реализованы в виде ссылок. Переменная типа класса — это просто указатель на область памяти на куче, где живут данные объекта. На самом деле есть немного дополнительной информации, например,

ссылка на класс, способ доступа к таблице виртуальных методов объекта, но это выходит за рамки данной главы (я вкратце представлю ее в разделе "Является ли этот указатель ссылкой на объект?" в главе 13).

Мы также видели, как назначение объекта другому делает только копию ссылки, так что у вас будет две ссылки на один объект в памяти. Чтобы иметь два полностью отдельных объекта, необходимо создать второй и скопировать в него данные первого объекта (операция недоступна автоматически, так как детали ее реализации могут варьироваться в зависимости от реальной структуры данных).

В терминах кодирования, если вы пишете следующий код, то вы создаете не новый объект, а новую ссылку на тот же самый объект в памяти:

```
var
  Button2: TButton;
begin
  Button2 := Button1;
```

Другими словами, в памяти есть только один объект, и обе переменные `Button1` и `Button2` ссылаются на него, как показано на рисунке 13.3.

Рисунок 13.3: Копирование ссылок на объекты

Объекты как параметры

Что-то подобное происходит при передаче объекта в качестве параметра в функцию или метод. В общем, вы просто копируете ссылку на один и тот же объект, а внутри метода или

функции вы можете выполнять операции с этим объектом и модифицировать его, независимо от того, передается ли параметр в качестве `const` параметра.

Например, написав эту процедуру и вызвав ее следующим образом, вы измените подпись `Button1`, или `AButton`, если угодно:

```
procedure ChangeCaption (AButton: TButton; Text: string);
begin
  AButton.Text := Text;
end;

// call...
ChangeCaption (Button1, 'Hello')
```

Что, если вместо этого нужно создать новый объект? В основном, вам нужно будет создать его, а затем скопировать каждое из соответствующих свойств. Некоторые классы, в частности, большинство классов, полученных из TPersistent, а не из `TComponent`, определяют метод Assign для копирования данных объекта. Например, можно написать

```
ListBox1.Items.Assign (Memo1.Lines);
```

Даже если вы назначите эти свойства напрямую, Object Pascal выполнит подобный код за вас. Фактически, метод `SetItems`, связанный со свойством items компонента listbox, вызывает метод `Assign` класса TStringList, представляющий собой фактические элементы listbox.

Итак, давайте попробуем вспомнить, что делают различные модификаторы при передаче параметров, при применении к объектам:

- Если нет **модификатора**, то можно выполнить любую операцию с объектом и переменной, на которую он ссылается. Вы можете модифицировать исходный объект, но если вы присваиваете новый объект параметру, то этот новый объект не будет иметь никакого отношения к исходному объекту и переменной, на которую он ссылается.

- Если есть **модификатор** const, то можно изменять значения и вызывать методы объекта, но нельзя назначать новый объект параметру. Обратите внимание, что при передаче объекта как const нет выигрыша в быстродействии.
- Если есть **модификатор** `var`, вы можете изменить что угодно в объекте, а также заменить исходный объект на новый в вызывающей локации, как это происходит с другими параметрами `var`. Ограничение состоит в том, что нужно передать ссылку на переменную (а не общее выражение) и тип ссылки должен точно совпадать с типом параметра.
- Наконец, есть довольно неизвестный вариант передачи объекта в качестве параметра, называемый **константной ссылкой** и записываемый как `[ref] const`. Когда параметр передается как константная ссылка, он ведет себя аналогично передаче по ссылке (`var`), но позволяет более гибко подходить к типу передаваемого параметра, не требуя точного совпадения по типу (так как передача объекта подкласса допускается).

Советы по управлению памятью

Управление памятью в Object Pascal подчиняется двум простым правилам: Вы должны уничтожить каждый объект и блок памяти, который вы создаете и выделяете, и вы должны уничтожить каждый объект и освободить каждый блок только один раз. Object Pascal поддерживает три типа управления памятью для динамических элементов (т.е. элементов, не находящихся в стеке и глобальной области памяти), подробно описанных в этой оставшейся части данного раздела:

- Каждый раз, когда вы создаете объект, вы также должны освободить его. Если вы этого не сделаете, память, используемая этим объектом, не будет освобождена для других объектов до тех пор, пока программа не завершит свою работу.
- При создании компонента можно указать компонент-хозяин (owner), передав его конструктору компонента. Компонент-хозяин (часто форма или модуль данных) становится ответственным за уничтожение всех объектов, которыми он владеет. Другими словами, когда вы освобождаете форму или модуль данных, он освобождает все компоненты, которыми он владеет. Таким образом, если вы создаёте компонент и даёте ему владельца, вам не придётся беспокоиться об его уничтожении.
- При выделении памяти для строк, динамических массивов и объектов, на которые ссылаются переменные интерфейса (как обсуждалось в Гл. 11), Object Pascal автоматически освобождает память, когда ссылка выходит за пределы области видимости. Вам не нужно освобождать строку: когда она становится недоступной, ее память освобождается.

Уничтожение объектов, которые вы создаете

В самом простом сценарии, на компиляторах рабочего стола вам придется создавать и уничтожать временные объекты. Любой не временный объект должен иметь владельца, быть частью коллекции, или должен быть доступен через какую-то структуру данных, которая в свое время станет ответственной за его уничтожение.

Код, используемый для создания и уничтожения временного объекта, обычно инкапсулируется в `try-finally` блок, так что

объект уничтожается, даже если что-то пойдет не так при его использовании:

```
MyObj := TMyClass.Create;
try
  MyObj.DoSomething;
finally
  MyObj.Free;
end;
```

Другой распространенный сценарий - это объект, используемый другим объектом, который становится его владельцем:

```
constructor TMyOwner.Create;
begin
  FSubObj := TSubObject.Create;
end;

destructor TMyOnwer.Destroy;
begin
  FSubObj.Free;
end;
```

Существуют некоторые общие, более сложные сценарии, в случае, если субъект не будет создан до тех пор, пока в нем нет необходимости (ленивая инициализация) или он может быть уничтожен перед владельцем, в случае, если он больше не нужен. Для реализации ленивой инициализации вы создаете субъект не в конструкторе объекта-обладателя, а когда он нужен:

```
function TMyOwner.GetSubObject: TSubObject
begin
  if not Assigned (FSubObj) then
    FSubObj := TSubObject.Create;
  Result := FSubObj;
end;

destructor TMyOnwer.Destroy;
begin
  FSubObj.Free;
end;
```

Обратите внимание, что вам не нужно проверять, назначен ли объект перед освобождением, потому что это именно то, что делает Free, как мы увидим в следующем разделе.

Уничтожение объектов только один раз

Другая проблема заключается в том, что если дважды вызвать деструктор объекта, то получится ошибка. Деструктор – это метод, который освобождает память объекта. Можно написать код для деструктора, обычно переопределяя деструктор по умолчанию `Destroy`, чтобы позволить объекту выполнить некоторый код до того, как он будет уничтожен.

Destroy - виртуальный деструктор класса TObject. Большинство классов, требующих пользовательского кода очистки при уничтожении объектов, переопределяют этот виртуальный метод. Причина, по которой никогда не следует определять новый деструктор, заключается в том, что объекты, как правило, уничтожаются путем вызова метода Free, и этот метод вызывает для вас виртуальный деструктор Destroy (возможно, переопределенную версию).

Как я только что упомянул, `Free` — это просто метод класса `TObject`, унаследованный всеми другими классами. Метод `Free` в основном проверяет, не нулевой ли текущий объект (Self) перед вызовом виртуального деструктора `Destroy`.

примечание Вы можете поинтересоваться, почему вы можете безопасно вызвать `Free`, если ссылка на объект `nil`, но вы не можете вызвать `Destroy`. Причина в том, что `Free` - известный метод на заданном участке памяти, в то время как виртуальная функция `Destroy` определяется во время выполнения, глядя на тип объекта, является очень опасной операцией, если объект больше не существует.

Вот псевдокод для `Free`:

```
procedure TObject.Free;
begin
  if Self <> nil then
    Destroy;
end;
```

Далее мы можем обратить внимание на функцию `Assigned`. Когда мы передаем указатель в эту функцию, она просто проверяет, равен ли указатель `nil`. Поэтому следующие два утверждения эквивалентны, по крайней мере, в большинстве случаев:

```
if Assigned (MyObj) then
  ...
if MyObj <> nil then
  ...
```

Обратите внимание, что эти утверждения проверяют только то, не является ли указатель `nil`; они не проверяют, является ли указатель действительным. Если написать следующий код

```
MyObj.Free;
if MyObj <> nil then
  MyObj.DoSomething;
```

тест даст True, и вы получите ошибку на строке с вызовом метода объекта. Важно понимать, что вызов `Free` не устанавливает ссылку объекта на `nil`.

Автоматическая установка объекта на `nil` невозможна. У вас может быть несколько ссылок на один и тот же объект, и Object Pascal не отслеживает их. В то же время, внутри метода (например, `Free`) мы можем работать с объектом, но мы ничего не знаем об объекте ссылки - адресе памяти переменной, которую мы использовали для вызова метода.

Другими словами, внутри метода `Free` или любого другого метода класса мы знаем адрес памяти объекта (`Self`), но мы не знаем расположение памяти переменной, ссылающейся на объект, например, `MyObj`. Поэтому метод `Free` не может повлиять на переменную `MyObj`.

Однако при вызове внешней функции, передающей объект в качестве параметра по ссылке, это позволяет этой функции изменять исходное значение параметра. Готовая к

использованию функция для этого случая - процедура FreeAndNil. Вот текущий код `FreeAndNil`:

```
procedure FreeAndNil(const [ref] Obj: TObject); inline;
var
  Temp: TObject;
begin
  Temp := Obj;
  TObject(Pointer(@Obj)^) := nil;
  Temp.Free;
end;
```

Раньше параметр был просто указателем, но недостатком было то, что в процедуру FreeAndNil можно было передать и сырой указатель, и ссылки на интерфейс, и другие несовместимые структуры данных. Это часто приводило к повреждению памяти и затруднению поиска ошибок. Начиная с Delphi 10.4 код был модифицирован, как показано выше, с использованием параметра `const` ref типа `TObject`, ограничивающего параметры объектами.

примечание Довольно много экспертов Delphi утверждают, что `FreeAndNil` никогда не следует использовать, так как видимость переменной, ссылающейся на объект, должна совпадать с его временем жизни. Если объект принадлежит другому и free находится в деструкторе, нет необходимости устанавливать ссылку на nil, так как он является частью объекта, который вы больше не будете использовать. Аналогично, локальная переменная с попыткой окончательно заблокировать освобождение, не должна устанавливать значение `nil`, так как она вот-вот выйдет из-под контроля.

В качестве примечания, кроме метода `Free`, в `TObject` также есть метод DisposeOf, который остаётся от поддержки ARC, которая была у языка в течение нескольких лет. В настоящее время метод `DisposeOf` просто вызывает `Free`.

Подводя итоги использования этих операций по очистке памяти, приведем пару рекомендаций:

- Всегда вызывайте `Free`, чтобы уничтожить объекты вместо того, чтобы вызывать деструктор `Destroy`.

- Используйте `FreeAndNil`, или установите `nil` для ссылок на объекты после вызова `Free`, если только ссылка не выходит из области видимости сразу после этого.

Управление памятью и интерфейсы

В главе 11 я представил ключевые элементы управления памятью для интерфейсов, которые в отличие от объектов управляются и отсчитываются по ссылкам. Как я уже упоминал, ссылки на интерфейс увеличивают счетчик ссылок на объект, на который делается ссылка, но вы можете объявить ссылку на интерфейс слабой, чтобы отключить подсчет ссылок (но все равно заставить компилятор управлять ссылкой за вас), или вы можете использовать небезопасный модификатор, чтобы полностью отключить поддержку компилятора для конкретной ссылки. В этом разделе мы немного углубимся в эту область, показав дополнительный пример того, что было представлено в Гл. 11.

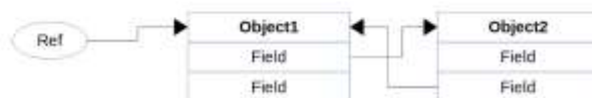
Подробнее о Слабых (Weak) Ссылках

Одна из проблем с моделями подсчета ссылок, которые Delphi использует для интерфейсов, заключается в том, что если два объекта ссылаются друг на друга, то они образуют круговую ссылку, и их счетчик ссылок практически никогда не достигнет нуля. Слабые ссылки предлагают механизм *разрыва* этих

кругов, позволяющий определить ссылку, которая не увеличивает счетчик ссылок.

Предположим, что два интерфейса ссылаются друг на друга, используя одно из своих полей, а внешняя переменная относится к первому. Счетчик ссылок первого объекта будет равен 2 (внешняя переменная и поле второго объекта): в то время как счетчик ссылок второго объекта равен 1 (поле первого объекта). На рисунке 13.4 показан этот сценарий.

Рисунок 13.4: Ссылки между объектами могут образовывать циклы, на что указывают слабые ссылки.



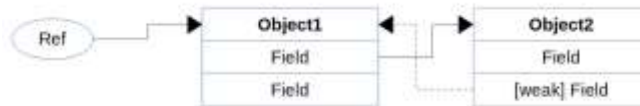
Теперь, когда внешняя переменная выходит за пределы области видимости, счетчик ссылок двух объектов остается 1, и они останутся в памяти навсегда. Чтобы решить такую ситуацию, необходимо разорвать круговые ссылки, что далеко не просто, учитывая, что вы не знаете, когда выполнить эту операцию (она должна выполняться, когда последняя внешняя ссылка выходит за пределы области видимости, факт, о котором объекты не знают). Решением этой ситуации, а также многих подобных сценариев, является использование слабой ссылки. Как уже упоминалось, слабая ссылка - это ссылка на объект, количество ссылок на который не увеличивается. Технически, слабая ссылка определяется путем применения к ней атрибута `[weak]`.

примечание Атрибуты — это продвинутая функция языка Object Pascal, описанная в Главе 16. Достаточно сказать, что они являются способом добавления некоторой информации о времени выполнения символа, чтобы внешний код мог определить, как с ним работать.

В предыдущем сценарии, если ссылка со второго объекта обратно на первый является слабой ссылкой (смотри рисунок

13.5) по мере того, как внешняя переменная выходит за пределы области видимости, оба объекта будут уничтожены.

Рисунок 13.5: Цикл ссылок на Рисунке 13.4, разбит на слабую ссылку (пунктирная линия).



Давайте посмотрим на эту простую ситуацию в коде. Прежде всего, пример приложения `ArcExperiments` декларирует два интерфейса, один из которых ссылается на другой:

```
type
  IMySimpleInterface = interface
    ['{B6AB548A-55A1-4D0F-A2C5-726733C33708}']
    procedure DoSomething(bRaise: Boolean = False);
    function RefCount: Integer;
  end;

  IMyComplexInterface = interface
    ['{5E8F7B29-3270-44FC-B0FC-A69498DA4C20}']
    function GetSimple: IMySimpleInterface;
    function RefCount: Integer;
  end;
```

Код программы определяет два различных класса, реализующих каждый из интерфейсов. Обратите внимание, что перекрестные ссылки (`FOwnedBy` и `FSimple` основаны на интерфейсах, а один из них определен как слабый):

```
type
  TMySimpleClass = class (TInterfacedObject, IMySimpleInterface)
  private
    [Weak] FOwnedBy: IMyComplexInterface;
  public
    constructor Create(Owner: IMyComplexInterface);
    destructor Destroy (); override;
    procedure DoSomething(bRaise: Boolean = False);
    function RefCount: Integer;
  end;

  TMyComplexClass = class (TInterfacedObject, IMyComplexInterface)
  private
    FSimple: IMySimpleInterface;
  public
    constructor Create();
    destructor Destroy (); override;
```

```
    function GetSimple: IMySimpleInterface;
    function RefCount: Integer;
  end;
```

Здесь конструктор класса "complex" создает объект другого класса:

```
constructor TMyComplexClass.Create;
begin
  inherited Create;
  FSimple := TMySimpleClass.Create (self);
end;
```

Помните, что поле `FOwnedBy` является слабой ссылкой, поэтому оно не увеличивает счетчик ссылок на объект, на который ссылается, в данном случае на текущий объект (`себя`). Учитывая такую структуру кода, можно писать:

```
class procedure TMyComplexClass.CreateOnly;
var
  MyComplex: IMyComplexInterface;
begin
  MyComplex := TMyComplexClass.Create;
  MyComplex.FSimple.DoSomething;
end;
```

Это не приведет к утечке памяти, пока используется слабая ссылка. Например, в коде вроде:

```
var
  MyComplex: IMyComplexInterface;
begin
  MyComplex := TMyComplexClass.Create;
  Log ('Complex = ' + MyComplex.RefCount.ToString);
  MyComplex.GetSimple.DoSomething (False);
```

Учитывая, что каждый конструктор и деструктор записывает в журнал свое исполнение, вы получите журнал типа:

```
Complex class created
Simple class created
Complex = 1
Simple class doing something
Complex class destroyed
Simple class destroyed
```

Если удалить слабый атрибут в коде, то появится утечка памяти, а также (при выполнении кода выше) значение для счетчика ссылок 2, а не 1.

Слабые ссылки управляются

Очень важным элементом является управление слабыми ссылками. Другими словами, система хранит в памяти список слабых ссылок, а при уничтожении объекта проверяет, есть ли на него слабая ссылка, и устанавливает его в `nil`. Это означает, что слабые ссылки имеют стоимость исполнения.

Положительным моментом в управлении слабыми ссылками, по сравнению с традиционными, является то, что вы можете проверить, является ли ссылка на интерфейс все еще действительной или нет (это означает, что объект, на который она ссылается, был уничтожен). Когда вы используете слабую ссылку, вы всегда должны проверить, назначена ли она, прежде чем использовать ее.

В примере приложения `ArcExperiments` форма имеет приватное поле типа `IMySimpleInterface`, объявленное как слабая ссылка:

```
  private
    [weak] MySimple: IMySimpleInterface;
```

Существует также кнопка, назначающая ссылку на это поле, и другая, использующая его после того, как вы удостоверились, что оно все еще является действительным:

```
procedure TForm3.BtnGetWeakClick(Sender: TObject);
var
  MyComplex: IMyComplexInterface;
begin
  MyComplex := TMyComplexClass.Create;
  MyComplex.GetSimple.DoSomething (False);
  MySimple := MyComplex.GetSimple;
end;

procedure TForm3.BtnUseWeakClick(Sender: TObject);
begin
  if Assigned (MySimple) then
    MySimple.DoSomething(False)
  else
    Log ('Nil weak reference');
end;
```

Если только вы не модифицируете код, то проверка `if Assigned` будет неудачной, потому что обработчик первого события кнопки создает и немедленно освобождает объекты, так что слабая ссылка становится *недействительной*. Но, если им управлять, компилятор помогает отслеживать его реальный статус (в отличие от ссылки на объект).

Атрибут Unsafe

Существуют некоторые очень специфические обстоятельства (например, во время создания экземпляра), при которых функция может вернуть объект с количеством ссылок, равным нулю. В этом случае, чтобы компилятор сразу же не удалил объект (до того, как у него появится возможность присвоения переменной, что увеличит счетчик ссылок до 1), необходимо пометить объект как "*опасный*".

Подразумевается, что его счетчик ссылок должен быть временно проигнорирован, чтобы сделать код "безопасным". Такое поведение достигается с помощью нового специфического атрибута `[Unsafe]`, который необходим только в очень специфических обстоятельствах. Вот синтаксис:

```
var
  [Unsafe] Intf1: IInterface;

[Result: Unsafe] function GetIntf: IInterface;
```

Использование данного атрибута может иметь смысл при реализации конструкционных шаблонов, например, шаблон «фабрика», в библиотеках общего назначения.

примечание Для поддержки устаревшей модели памяти ARC модуль System использовал директиву `unsafe`, так как не мог использовать атрибут до его определения (позже в том же самом блоке). Этот атрибут не предполагается использовать ни в одном коде вне этого модуля, и он больше не используется (это видно в директиве `$IFDEF`).

Отслеживание и проверка памяти

В этой главе мы рассмотрели основы управления памятью в Object Pascal. В большинстве случаев достаточно простого применения выделенных здесь правил, чтобы сохранить стабильность ваших программ, избежать чрезмерного использования памяти и, по сути, забыть об управлении памятью. Далее в этой главе мы расскажем о некоторых наиболее эффективных способах написания надежных приложений.

В этом разделе я концентрируюсь на техниках, которые можно использовать для отслеживания использования памяти, мониторинга аномальных ситуаций и поиска утечек памяти. Это важные знания для разработчика, даже если они не являются строго частью языка, а скорее поддержкой во время выполнения. Кроме того, реализация менеджера памяти зависит от целевой платформы и операционной системы, и вы даже можете подключить пользовательский менеджер памяти в приложении Object Pascal (не то, чтобы это было обычным делом).

Обратите внимание, что все обсуждения, связанные с отслеживанием состояния памяти, менеджерами памяти, обнаружением утечек, относятся только к *памяти кучи*. Стеком и глобальной памятью управляют по-разному, и у вас, по сути, нет возможности вмешаться, но это области памяти, которые редко вызывают какие-либо проблемы.

Статус памяти

Как насчет отслеживания состояния памяти кучи? RTL предлагает пару удобных функций, GetMemoryManagerState и GetMemoryMap. В то время как состояние менеджера памяти является показателем количества выделенных блоков различного размера, карта кучи достаточно хороша, так как она отображает состояние памяти приложения на системном уровне. Реальное состояние каждого следующего блока памяти можно проверить, написав код типа:

```
for I := Low(aMemoryMap) to High(aMemoryMap) do
begin
  case AMemoryMap[I] of
    csUnallocated: ...
    csAllocated: ...
    csReserved: ...
    csSysAllocated: ...
    csSysReserved: ...
  end;
end;
```

Данный код используется в проекте приложения ShowMemory для создания графического представления состояния памяти приложения.

FastMM4

На платформе Windows текущий менеджер памяти Object Pascal называется FastMM4 и был разработан как проект с открытым исходным кодом в основном Пьером Ла Ришем. FastMM4 оптимизирует выделение памяти, ускоряя ее и освобождая больше оперативной памяти для последующего использования. FastMM4 способен выполнять обширные проверки памяти на эффективность очистки памяти, на неправильное использование удаленных объектов, в том числе интерфейс на основе доступа к этим данным, на перезапись

памяти и на переполнение буфера. Она также может обеспечить некоторую обратную связь по оставшимся объектам, помогая отслеживать утечки памяти.

Фактически некоторые из более продвинутых возможностей FastMM4 доступны только в полной версии библиотеки (рассматривается в разделе "Переполнения буфера в полном FastMM4"), а не в версии, входящей в стандартную RTL. Поэтому, если вы хотите иметь возможности полной версии, вам необходимо скачать ее полный исходный код:

`https://github.com/pleriche/FastMM4`

примечание Появилась новая версия этой библиотеки под названием FastMM5, которая была специально оптимизирована для многопоточных приложений и может работать намного лучше на больших многоядерных системах. Новая версия библиотеки доступна с лицензией GPL (для проектов с открытым исходным кодом) или с платной коммерческой лицензией (полностью оплачиваемой) для любого другого. Более подробная информация о проекте доступна по адресу https://github.com/pleriche/FastMM5.

Отслеживание утечек и другие глобальные настройки

RTL-версия FastMM4 может быть настроена с помощью глобальных настроек в модуле `System`. Обратите внимание, что пока соответствующие глобальные декларации находятся в модуле `System`, реальный менеджер памяти реализован в файле исходного кода `getmem.inc` RTL. Опять же, по умолчанию это активно только для Windows-приложений, а не для других операционных систем, которые используют свою платформу и нативный менеджер памяти.

Самой простой в использовании настройкой является глобальная переменная ReportMemoryLeaksOnShutdown, которая позволяет легко отслеживать утечку памяти. Ее необходимо включить в начале выполнения программы, а по

окончании работы программы она сообщит, есть ли утечка памяти в Вашем коде (или в любой из библиотек, которые Вы используете).

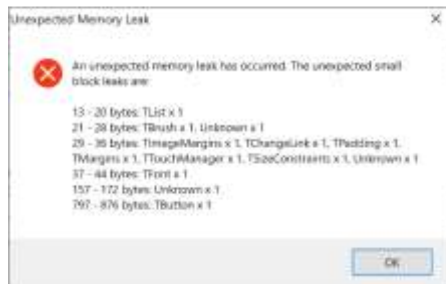
примечание Более продвинутые настройки менеджера памяти включают глобальную переменную NeverSleepOnMMThreadContention для многопоточных распределений; функции GetMinimumBlockAlignment и SetMinimumBlockAlignment, которые могут ускорить некоторую работу SSE за счет использования большего объема памяти; возможность зарегистрировать ожидаемую утечку памяти, вызвав глобальную процедуру RegisterExpectedMemoryLeak.

Для демонстрации стандартных отчетов об утечке памяти и регистрации я написал простой проект приложения LeakTest. В нем есть кнопка с таким обработчиком `OnClick`:

```
var
  P: Pointer;
begin
  GetMem (P, 100); // leak!
end;
```

Этот код выделяет 100 байт, которые будут утеряны. Если вы запустите программу `LeakTest` во время работы IDE и один раз нажмете первую кнопку, то при закрытии программы вы получите сообщение, похожее на то, что показано в верхней части рисунка 13.6.

Рисунок 13.6: Утечка памяти, о которой сообщается менеджером памяти в Windows при завершении работы приложения LeakTest



Другая "утечка" программы вызвана созданием и оставлением в памяти TButton, но так как этот объект включает в себя множество подэлементов, то отчет об утечке становится более сложным, как тот, который представлен в нижней части рисунка 13.6. Тем не менее, мы имеем некоторую ограниченную информацию о самой утечке.

Программа также выделяет часть памяти для глобального указателя, которая никогда не будет освобождена, но при регистрации этой потенциальной утечки, как и ожидалось, о ней не будет сообщено:

```
procedure TFormLeakTest.FormCreate(Sender: TObject);
begin
  GetMem (GlobalPointer, 200);
  RegisterExpectedMemoryLeak(GlobalPointer);
end;
```

Опять же, этот базовый отчет об утечках доступен по умолчанию только на Windows-платформе, где FastMM4 фактически используется по умолчанию.

Перерасход буфера в Full FastMM4

> *Это довольно продвинутая тема, и она специфична для Windows-платформы, поэтому я бы порекомендовал читать этот раздел только самым опытным разработчикам.*

Если вы хотите иметь больше контроля над отчетами об утечках (например, активировать файловый лог), точно настроить стратегию распределения и использовать проверки памяти, предоставляемые FastMM4, вам нужно скачать полную версию. Она состоит из файла `FastMM4.pas` и конфигурационного файла `FastMM4Options.inc`.

Именно этот последний файл необходимо отредактировать для точной настройки параметров, просто комментируя и не комментируя большое количество директив. По общему правилу, это делается путем помещения периода перед оператором $DEFINE, превращая его в простой комментарий, как в первой из этих двух строк, взятых из включаемого файла:

```
{. $DEFINE Align16Bytes} //комментарий
{$DEFINE UseCustomFixedSizeMoveRoutines} // активная настройка
```

Для этой демонстрации я включил следующие соответствующие настройки, представленные здесь, чтобы дать вам представление о доступных определениях:

```
{$DEFINE FullDebugMode}
{$DEFINE LogErrorsToFile}
{$DEFINE EnableMemoryLeakReporting}
{$DefINE HideExpectedLeaksRegisteredByPointer}
{$DEFINE RequireDebuggerPresenceForLeakReporting}
```

Тестовая программа (в папке `FastMMCode`, которая также включает в себя полный исходный код используемой мной версии FastMM4, для вашего удобства) активирует пользовательскую версию менеджера памяти в файле исходного кода проекта, установив его в качестве первого блока:

```
program FastMMCode;

uses
  FastMM4 in 'FastMM4.pas',
  Forms,
  FastMMForm in 'FastMMForm.pas'; {Form1}
```

Вам также понадобится локальная копия файла `FastMM_FullDebugMode.dll`, чтобы пример заработал. Эта демонстрационная программа приводит к выходу за границы буфера за счет получения большего количества текста, которое может поместиться в локальный буфер, так как `Length(Caption)` больше, чем 5 символов:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  pch1: PChar;
begin
  GetMem (pch1, 5);
  GetWindowText(Handle, pch1, Length(Caption));
  ShowMessage (pch1);
  FreeMem (pch1);
end;
```

Менеджер памяти выделяет дополнительные байты в начале и в конце каждого блока памяти со специальными значениями и проверяет эти значения при освобождении каждого блока памяти. Поэтому вы получаете ошибку при вызове FreeMem. При нажатии кнопки (в отладчике) вы увидите очень длинное сообщение об ошибке, которое также заносится в файл:

```
FastMMCode_MemoryManager_EventLog.txt
```

Это вывод ошибки переполнения, с трассами стека во время выделения и свободных операций, плюс текущая трасса стека и дамп памяти (частично здесь):

```
FastMM has detected an error during a FreeMem operation. The block
footer has been corrupted.

The block size is: 5

Stack trace of when this block was allocated (return addresses):
40305E [System][System.@GetMem]
44091A [Controls][Controls.TControl.Click]
44431B [Controls][Controls.TWinControl.WndProc]
42D959 [StdCtrls][StdCtrls.TButtonControl.WndProc]
44446C [Controls][Controls.DoControlMsg]
```

```
44431B [Controls][Controls.TWinControl.WndProc]
45498A [Forms][Forms.TCustomForm.WndProc]
443A43 [Controls][Controls.TWinControl.MainWndProc]
41F31A [Classes][Classes.StdWndProc]
76281A10 [GetMessageW]

The block is currently used for an object of class: Unknown

The allocation number is: 381

Stack trace of when the block was previously freed (return addresses):
40307A [System][System.@FreeMem]
42DB8A [StdCtrls][StdCtrls.TButton.CreateWnd]
443863 [Controls][Controls.TWinControl.UpdateShowing]
44392B [Controls][Controls.TWinControl.UpdateControlState]
44431B [Controls][Controls.TWinControl.WndProc]
45498A [Forms][Forms.TCustomForm.WndProc]
44009F [Controls][Controls.TControl.Perform]
43ECDF [Controls][Controls.TControl.SetVisible]
45F770
76743833 [BaseThreadInitThunk]

The current stack trace leading to this error (return addresses):
40307A [System][System.@FreeMem]
44091A [Controls][Controls.TControl.Click]
44431B [Controls][Controls.TWinControl.WndProc]
42D959 [StdCtrls][StdCtrls.TButtonControl.WndProc]
44446C [Controls][Controls.DoControlMsg]
44431B [Controls][Controls.TWinControl.WndProc]
45498A [Forms][Forms.TCustomForm.WndProc]
443A43 [Controls][Controls.TWinControl.MainWndProc]
41F31A [Classes][Classes.StdWndProc]
76281A10 [GetMessageW]

Current memory dump of 256 bytes starting at pointer address 133DEF8:
46 61 73 74 4D 4D 43 6F 64 [... omitted...]
```

Не то, чтобы все было чрезвычайно очевидно, но это должно дать достаточно информации, чтобы вы могли начать поиск ошибки. Заметьте, что без этих настроек в менеджере памяти вы, по сути, не увидите никакой ошибки, и программа продолжает работать… хотя вы можете столкнуться со случайными ошибками в случае, если переполнение буфера повлияет на область памяти, в которой хранится что-то другое. В этот момент вы можете получить некоторые странные и очень трудно отслеживаемые ошибки.

В качестве примера я однажды видел частичную перезапись исходной части данных объекта, где хранится ссылка на класс.

Благодаря этому повреждению памяти класс стал неопределенным и каждый вызов одной из его виртуальных функций приводил к аварийному завершению… чего-то очень сложного в связи с операцией записи в память в совершенно другой области программы.

Управление памятью на платформах, отличных от Windows

Учитывая то, как работает управление памятью в компиляторах Object Pascal, стоит рассмотреть некоторые варианты, чтобы убедиться, что все под контролем. Прежде чем продолжить, важно заметить, что на не-Windows платформах Delphi не использует менеджер памяти FastMM4, поэтому установка глобального флага `ReportMemoryLeaksOnShutdown` для проверки утечек памяти при закрытии программы бесполезна. Есть еще одна причина, которая заключается в том, что обычно нет способа закрыть приложение на мобильном устройстве, учитывая, что приложения остаются в памяти до тех пор, пока принудительно не будут удалены пользователем или операционной системой.

На платформах MacOS, iOS и Android Object Pascal RTL напрямую вызывает функции `malloc` и `free` родной библиотеки `libc`. Один из способов контролировать использование памяти на этой платформе - полагаться на внешние инструменты платформы. Например, на iOS (и MacOS) вы можете использовать инструмент Apple's Instruments, который представляет собой полную систему мониторинга всех аспектов ваших приложений, работающих на физическом устройстве.

Отслеживание выделения памяти класса

Наконец, существует подход Object Pascal, ориентированный на отслеживание для конкретного класса, а не на управление памятью в целом. Выделение памяти для объекта, по сути, происходит путем вызова виртуального метода класса NewInstance, а очистка производится с помощью виртуального метода FreeInstance. Это виртуальные методы, которые можно переопределить в определенном классе для настройки конкретной стратегии выделения памяти.

Преимущество заключается в том, что это можно сделать независимо от конструкторов (так как их может быть несколько) и деструктора, четко отделяя код отслеживания памяти от стандартного кода инициализации и доработки объектов.

Хотя это довольно экстремальный случай (вероятно, это стоит делать только для некоторых больших структур памяти), вы можете переопределить эти методы для подсчета количества объектов данного класса, которые создаются и уничтожаются, вычислить количество активных экземпляров, а в конце проверить, что подсчет идет до нуля, как и ожидалось.

Написание надежных приложений

В этой главе и во многих предыдущих главах этого раздела я рассказал о довольно большом количестве методик,

сосредоточенных на написании надежных приложений и правильном управлении распределением и перераспределением памяти.

В этой заключительной части главы, посвященной управлению памятью, я решил упомянуть несколько немного более продвинутых тем, которые дополняют рассмотренные ранее. Даже если использование блоков try-finally и вызов деструкторов уже были рассмотрены, выделенные здесь сценарии немного сложнее и включают в себя использование нескольких функций языка вместе.

Это не совсем продвинутый раздел, но все Object Pascal разработчики должны действительно освоить его, чтобы иметь возможность писать надежные приложения. Только последний подраздел, посвященный указателям и ссылкам на объекты, определенно более продвинутый в области применения, так как он углубляется во внутреннюю структуру внутренней памяти объекта и ссылку на класс.

Конструкторы, деструкторы и исключения

Конструкторы и деструкторы часто могут быть источником проблем в приложениях. Виртуальные конструкторы должны обязательно *сначала* вызывать конструктор своего базового класса. Деструкторы, как правило, должны вызывать свои унаследованные классы *последними*.

примечание Чтобы следовать хорошей практике кодирования, вы должны обычно добавлять вызов конструктора базового класса в каждый конструктор вашего кода Object Pascal, даже если это не является обязательным и дополнительный вызов может быть бесполезен (как при вызове `TObject.Create`).

В этом разделе я хочу обратить особое внимание на то, что происходит, когда конструктор терпит неудачу, например, в классическом сценарии:

```
MyObj := TMyClass.Create;
try
  MyObj.DoSomething;
finally
  MyObj.Free;
end;
```

Если объект создан и присвоен переменной `MyObj`, то блок finally позаботится об его уничтожении. Но если при вызове Create возникает исключение, то блок `try-finally` не вызывается (*и это правильно!*). Когда конструктор вызывает исключение, соответствующий код деструктора *автоматически выполняется* на том, что может быть частично инициализированным объектом. Если конструктор создает, например, два подобъекта, то они должны быть уничтожены, вызывая соответствующий деструктор. Однако это может привести к потенциальным неприятностям, если в деструкторе предполагается, что объект был полностью инициализирован...

Это не просто понять в теории, поэтому давайте посмотрим на практическую демонстрацию в коде. Проект приложения SafeCode содержит класс с конструктором и деструктор, который будет в целом корректен... если только конструктор сам не выйдет из строя:

```
type
  TUnsafeDesctructor = class
  private
    FList: TList;
  public
    constructor Create (PositiveNumber: Integer);
    destructor Destroy; override;
  end;

constructor TUnsafeDesctructor.Create(PositiveNumber: Integer);
begin
  inherited Create;

  if PositiveNumber <= 0 then
    raise Exception.Create ('Not a positive number');
```

```
  FList := TList.Create;
end;

destructor TUnsafeDesctructor.Destroy;
begin
  FList.Clear;
  FList.Free;
  inherited;
end;
```

Проблема в том, что в случаях, когда объект создан полностью, деструктор работает корректно, но если он выполняется, когда поле `FList` все еще установлено на `ноль`, то при вызове Clear возникнет исключение "Нарушение доступа".

Безопасный способ написания одного и того же кода следующий:

```
destructor TUnsafeDesctructor.Destroy;
begin
  if assigned (FList) then
    FList.Clear;
  FList.Free;
  inherited;
end;
```

И мораль истории, опять же, никогда не принимать как должное в деструкторе то, что соответствующий конструктор полностью инициализировал объект. Это предположение можно сделать для любого другого метода, но не для деструктора.

Вложенные блоки Finally

Блоки Finally — это, пожалуй, самая важная и распространенная техника, позволяющая сделать ваши программы безопасными. Я не думаю, что это продвинутая тема, но используете ли вы finally везде или нет? И правильно ли вы используете ее в пограничных случаях, таких как вложенные операции, или Вы комбинируете несколько

утверждений finally в одном блоке finally? Вот далеко не идеальный пример кода:

```
procedure TForm1.BtnTryFClick(Sender: TObject);
var
  A1, A2: TAClass;
begin
  A1 := TAClass.Create;
  A2 := TAClass.Create;
  try
    A1.Whatever := 'one';
    A2.Whatever := 'two';
  finally
    A2.Free;
    A1.Free;
  end;
end;
```

Это более безопасная версия того же самого кода (снова извлеченного из проекта приложения SafeCode):

```
procedure TForm1.BtnTryFClick(Sender: TObject);
var
  A1, A2: TAClass;
begin
  A1 := TAClass.Create;
  try
    A2 := TAClass.Create;
    try
      A1.Whatever := 'one';
      A2.Whatever := 'two';
    finally
      A2.Free;
    end;
  finally
     A1.Free;
  end;
end;
```

Динамическая проверка типа

Динамическое преобразование между типами вообще и типами классов в частности является еще одним возможным источником подводных камней. Особенно, если вы не используете `is` и `as` операторов и просто делаете *жесткие* приведения. Каждое прямое приведение типов, на самом деле,

является потенциальным источником ошибок (если только оно не следует за проверкой is).

Преобразование типов из объектов в указатели, в и из ссылок на класс, из объектов в интерфейсы, в строки и из строк потенциально очень опасны, но в некоторых особых обстоятельствах их трудно избежать. Например, вы можете захотеть сохранить ссылку на объект в свойстве Tag компонента. Другой случай, когда вы сохраняете объекты в списке указателей, используя старомодный список TList (а не безопасный для типов общий список, о котором пойдет речь в следующей главе). Это довольно глупый пример:

```
procedure TForm1.BtnCastClick(Sender: TObject);
var
  List: TList;
begin
  List := TList.Create;
  try
    List.Add(Pointer(Sender));
    List.Add(Pointer (23422));
    // direct cast
    TButton(List[0]).Caption := 'ouch';
    TButton(List[1]).Caption := 'ouch';
  finally
    List.Free;
  end;
end;
```

Запуск этого кода, как правило, приводит к нарушению доступа.

примечание Я писал *как правило,* потому что, когда вы получаете случайный доступ к памяти, вы никогда не знаете действительного эффекта. Иногда программы просто перезаписывают память, не вызывая немедленной ошибки... но позже вам будет трудно понять, почему повреждены некоторые другие данные.

По возможности следует избегать подобных ситуаций, но, если у вас нет альтернативы, что вы можете сделать, чтобы исправить этот код? Естественным подходом было бы использовать либо безопасное приведение as, либо проверку типа is, как в следующих фрагментах:

```
 // as cast
(TObject(List[0]) as TButton).Caption := 'ouch';
(TObject(List[1]) as TButton).Caption := 'ouch';

// is cast
if TObject(List[0]) is TButton then
  TButton(List[0]).Caption := 'ouch';
if TObject(List[1]) is TButton then
  TButton(List[1]).Caption := 'ouch';
```

Однако, это *не* решение, вы будете продолжать получать нарушения доступа. Проблема в том, что `is` или `as` в конечном итоге вызывает `TObject.InheritsFrom`, сложная операция для выполнения с числами!

Решение? Реальное решение заключается в том, чтобы избежать подобных ситуаций в первую очередь (этот тип кода, честно говоря, имеет мало смысла), используя TObjectList или какую-либо другую безопасную технику (опять же, смотрите следующую главу, посвященную общим классам-контейнерам). Если вам действительно нравятся низкоуровневые хаки и вам нравится *играть* с указателями, вы можете попробовать выяснить, действительно ли заданное "*числовое значение*" является ссылкой на объект или нет. Однако это не тривиальная операция. В ней есть и интересная сторона, которую я беру в качестве оправдания данной демонстрации, чтобы объяснить вам внутреннюю структуру объекта... и ссылку на класс.

Является ли этот указатель ссылкой на объект?

Этот раздел объясняет внутреннюю структуру объектов и ссылки на классы, и выходит далеко за рамки обсуждения в большей части этой книги. Тем не

менее, он может дать более опытным читателям некоторые интересные идеи, поэтому я решил оставить этот материал, который я написал в прошлом для расширенного доклада по управлению памятью. Обратите также внимание, что конкретная реализация, описанная ниже, специфична для Windows, с точки зрения проверки памяти.

Бывают случаи, когда указатели находятся рядом (указатель - это всего лишь числовое значение, относящееся к физическому месту в памяти некоторых данных). На самом деле эти указатели могут быть ссылками на объекты, и вы обычно знаете когда, и используете их как таковые. Но каждый раз, когда вы выполняете низкоуровневое приведение типов, вы реально в шаге от того, чтобы испортить целую программу. Существуют приёмы, позволяющие сделать управление указателями такого типа немного безопаснее, даже если они не гарантируют их на 100%.

Отправной точкой, которую вы, возможно, захотите рассмотреть, прежде чем работать с указателем, является то, является ли он на самом деле законным указателем или нет. Функция Assigned только проверяет, не является ли указатель `nil`, что в данном случае не помогает. Однако малоизвестная функция FindHInstance Object Pascal RTL (в модуле `System`, доступном на платформе Windows) возвращает базовый адрес блока кучи, включая объект, переданный в качестве параметра, или ноль, если указатель ссылается на некорректную страницу (что предотвращает довольно редкие, но крайне трудно отслеживаемые ошибки страницы памяти). Если взять число почти случайным образом, то, скорее всего, оно не будет ссылаться на действительную страницу памяти.

Это хорошая отправная точка, но мы можем сделать лучше, так как это не поможет, если значение является строковой ссылкой

или любым другим допустимым указателем, а не объектной ссылкой. Теперь как узнать, является ли указатель на самом деле ссылкой на объект? Я придумал следующий эмпирический тест. Первые 4 байта объекта - это указатель на его класс. Если рассматривать внутреннюю структуру данных ссылки на класс, то она имеет в своей позиции vmtSelfPtr указатель на себя. Примерно это показано на рисунке 13.7.

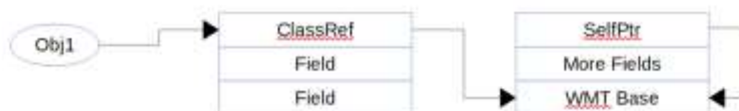


Рисунок 13.7: Приблизительное представление внутренней структуры объектов и ссылок на классы.

Другими словами, разыменовав значение в ячейке памяти `vmtSelfPtr` байт от ссылочного указателя класса (это отрицательное смещение, ниже в памяти), вы должны снова получить тот же ссылочный указатель класса. Более того, во внутренней структуре данных ссылки класса можно прочитать информацию о размере экземпляра (на позиции vmtInstanceSize) и посмотреть, есть ли там *разумное* число. Вот реальный код:

```
function IsPointerToObject (Address: Pointer): Boolean;
var
  ClassPointer, VmtPointer: PChar;
  Instsize: Integer;
begin
  Result := False;
  if (FindHInstance (Address) > 0) then
  begin
    VmtPointer := PChar(Address^);
    ClassPointer := VmtPointer + vmtSelfPtr;
    if Assigned (VmtPointer) and
      (FindHInstance (VmtPointer) > 0) then
    begin
      Instsize := (PInteger(
        Vmtpointer + vmtInstanceSize))^;
      // check self pointer and "reasonable" instance size
      if (Pointer(Pointer(ClassPointer)^) =
           Pointer(VmtPointer)) and
          (Instsize > 0) and (Instsize < 10000) then
        Result := True;
```

```
    end;
  end;
end;
```

Имея под рукой эту функцию, в предыдущем проекте приложения SafeCode мы можем добавить проверку указателя на проект перед выполнением безопасного кастинга:

```
    if IsPointerToObject (List[0]) then
      (TObject(list[0]) as TButton).Caption := 'ouch';
    if IsPointerToObject (List[1]) then
      (TObject(list[1]) as TButton).Caption := 'ouch';
```

Эту же идею можно применить и непосредственно к ссылкам на классы, в том числе и для реализации безопасных передач. Опять же, лучше всего попытаться избежать подобных проблем в первую очередь, написав более безопасный и чистый код, но в случае, если вы не можете этого избежать, эта функция может пригодиться. В любом случае, этот раздел должен был немного объяснить внутреннюю часть этих системных структур данных.

Часть III: Расширенные функции

Теперь, когда мы углубились в основы языка и в парадигму объектно-ориентированного программирования, пришло время открыть для себя некоторые из последних и более продвинутых возможностей языка Object Pascal. Дженерики, анонимные методы и рефлексия открывают возможности для разработки кода с использованием новых парадигм, которые существенно расширяют возможности объектно-ориентированного программирования.

Некоторые из этих более продвинутых возможностей языка, на самом деле, позволяют разработчикам принять новые способы

написания кода, предлагая еще больше типов и абстракций кода и позволяя более динамично подходить к кодированию, используя рефлексию в полной мере.

В последней части раздела мы рассмотрим эти особенности языка, предложив обзор элементов библиотеки времени исполнения, которые являются настолько ядром модели разработки Object Pascal, что различие между языком и библиотекой достаточно размыто. Мы рассмотрим, например, класс `TObject`, который, как мы видели ранее, является базовым классом всех классов, которые вы пишете: далеко до заметной роли, которая должна быть ограничена деталями реализации библиотеки.

Главы части III

Глава 14: Дженерики

Глава 15: Анонимные методы

Глава 16: Отражение и атрибуты

Глава 17: Класс TObject

Глава 18: Библиотека времени выполнения (RTL)

14: Дженерики

Строгая (или сильная) проверка типа, предоставляемая Object Pascal, полезна для улучшения корректности кода – тема, на которую я много обращал внимание в этой книге. Сильная проверка типов, однако, также может быть помехой, так как вы, возможно, захотите написать процедуру или класс, который может действовать одинаково на разных типах данных. Этот вопрос решается за счет возможности языка Object Pascal, называемого *generics,* также доступного в похожих языках, таких как C# и Java.

Понятие общих или шаблонных классов на самом деле происходит от языка Си++. Вот, что я написал в 1994 году в книге о C++:

> *Можно объявить класс без указания типа одного или нескольких членов данных: эта операция может быть отложена до тех пор, пока объект этого класса не будет фактически объявлен. Аналогично можно объявить функцию без указания типа одного или нескольких ее параметров до вызова функции.*

примечание Текст извлечен из книги "Объектно-ориентированное программирование Borland C++ 4.0", которую я написал со Стивом Тендоном в начале 90-х годов.

В этой главе мы рассмотрим эту тему, начиная с основ, но также рассмотрим некоторые продвинутые сценарии

использования, и даже укажем, как дженерики могут быть применены к стандартному визуальному программированию.

Generic пары ключей-значений

В качестве первого примера generic класса я реализовал структуру данных пар ключ-значение. Первый фрагмент кода, приведенный ниже, показывает структуру данных, написанную традиционным образом, с объектом, используемым для хранения значения:

```
type
  TKeyValue = class
  private
    FKey: string;
    FValue: TObject;
    procedure SetKey(const Value: string);
    procedure SetValue(const Value: TObject);
  public
    property Key: string read FKey write SetKey;
    property Value: TObject read FValue write SetValue;
  end;
```

Для использования данного класса можно создать объект, задать его ключ и значение, а также использовать его, как и в следующих фрагментах различных методов основной формы прикладного проекта KeyValueClassic:

```
// FormCreate
Kv := TKeyValue.Create;

// Button1Click
Kv.Key := 'mykey';
Kv.Value := Sender;

// Button2Click
Kv.Value := self; // the form

// Button3Click
ShowMessage('[' + Kv.Key +',' + Kv.Value.ClassName + ']');
```

Что, если вам нужен аналогичный класс, содержащий целое, а не объект? Ну, либо вы делаете очень неестественное (и опасное) приведение типа, либо вы создаете новый и отдельный класс для хранения строкового ключа с числовым значением. Хотя копирование и вставка оригинального класса может показаться решением, в конечном итоге вы получаете две копии для очень похожего куска кода, вы идете против общепринятых принципов программирования… и вам придется обновлять новыми возможностями или исправлять одни и те же ошибки два, три или двадцать раз.

Дженерики позволяют использовать гораздо более широкое определение значения, записывая один общий класс. Как только вы конкретизировали generic класс "ключ-значение", он становится специфическим классом, привязанным к данному типу данных. Таким образом, вы все равно получаете два, три, или двадцать классов, скомпилированных в вашем приложении, но у вас есть единое определение исходного кода для всех них, одинаково отвечающее на проверку правильности строкового типа и без накладных расходов на выполнение. Но я забегаю вперед: давайте начнем с синтаксиса, используемого для определения generic класса:

```
type
  TKeyValue<T> = class
  private
    FKey: string;
    FValue: T;
    procedure SetKey(const Value: string);
    procedure SetValue(const Value: T);
  public
    property Key: string read FKey write SetKey;
    property Value: T read FValue write SetValue;
  end;
```

В определении этого класса имеется один неопределенный тип, на который указывает заполнитель `T`, заключенный в угловые скобки. Символ `T` часто используется условно, но в компиляторе можно использовать любой символ.

Использование символа `T` обычно делает код более читабельным, когда generic класс использует только один параметрический тип; в случае, если классу необходимо несколько параметрических типов, то обычно их называют в соответствии с их реальной ролью, а не в соответствии с последовательностью букв (`T`, `U`, `V`), как это происходило в C++ в ранние годы.

примечание "T" является стандартным именем, или заполнителем, для типа generic с тех пор, как в начале 1990-х годов язык C++ ввел *шаблоны*. В зависимости от авторов, "T" означает либо "Type", либо "Template type".

Generic класс `TKeyValue<T>` в качестве типа одного из двух своих полей, значения свойства и параметра метода setter использует неопределенный тип. Методы определяются как обычно, но обратите внимание, что независимо от того, что они имеют отношение к универсальному типу, их определение содержит полное имя класса, включая универсальный тип:

```
procedure TKeyValue<T>.SetKey(const Value: string);
begin
  FKey := Value;
end;

procedure TKeyValue<T>.SetValue(const Value: T);
begin
  FValue := Value;
end;
```

Чтобы использовать класс, вместо этого, вы должны полностью его квалифицировать, предоставив фактический тип родового типа. Например, теперь вы можете объявить key-value object, содержащий кнопки в качестве значений, написав так:

```
var
  Kv: TKeyValue<TButton>;
```

Полное имя требуется и при создании экземпляра, поскольку это действительное имя типа (в то время как общее, ничем не подкрепленное имя типа похоже на механизм построения типа).

Использование определенного типа значения пары "ключ-значение" делает код намного более надежным, так как теперь к паре "ключ-значение" можно добавлять только объекты `TButton` (или производные объекты) и использовать различные методы извлеченного объекта. Вот несколько фрагментов из основной формы прикладного проекта KeyValueGeneric:

```
// FormCreate
Kv := TKeyValue<TButton>.Create;

// Button1Click
Kv.Key := 'mykey';
Kv.Value := Sender as TButton;

// Button2Click
Kv.Value := Sender as TButton; // was "self"

// Button3Click
ShowMessage ('[' + Kv.Key + ',' + Kv.Value.Name + ']');
```

При назначении generic объекта в предыдущей версии кода мы могли добавить либо кнопку, либо форму, теперь мы можем использовать только кнопку, правило, введенное компилятором. Аналогично, вместо общего `kv.Value.ClassName` в выводе мы можем использовать `Name` компонента или любое другое свойство класса `TButton`.

Конечно, мы также можем имитировать первоначальный вариант программы, объявив пару ключ-значение с типом объекта, например:

```
var
  Kvo: TKeyValue<TObject>;
```

В данной версии generic класса пары ключ-значение мы можем добавить любой объект в качестве значения. Однако, мы не сможем многое сделать с извлечёнными объектами, если не приведём их к более определённому типу. Чтобы найти хороший баланс, вам может понадобиться что-нибудь между конкретными кнопками и любым объектом, запрашивающим значение в виде компонента:

```
var
```

```
  Kvc: TKeyValue<TComponent>;
```

Соответствующие фрагменты кода можно увидеть в том же проекте приложения KeyValueGeneric. Наконец, мы также можем создать экземпляр generic класса пары ключ-значение, в котором будут храниться не значения объектов, а обычные целые числа:

```
var
  Kvi: TKeyValue<Integer>;
begin
  Kvi := TKeyValue<Integer>.Create;
  try
    Kvi.Key := 'object';
    kvi.Value := 100;
    Kvi.Value := Left;
    ShowMessage ('[' + Kvi.Key + ',' +
      IntToStr (Kvi.Value) + ']');
  finally
    Kvi.Free;
  end;
```

Встроенные переменные и выведение типа у дженериков

Когда вы объявляете переменную generic типа, объявление может быть довольно длинным. При создании объекта такого типа необходимо повторить то же самое объявление. Это так, если только вы не воспользуетесь объявлениями встроенных переменных и их способностью делать выводы о типе переменной. Последний фрагмент кода, приведенный выше, можно записать как:

```
begin
  var Kvi := TKeyValue<Integer>.Create;
  try
    ...
```

В этом коде нет необходимости повторять полное типовое объявление дважды. Это особенно удобно при использовании контейнеров, как мы увидим позже.

Правила типов для дженериков

Когда вы объявляете экземпляр generic типа, этот тип получает определенную версию, которая принудительно внедряется компилятором во все последующие операции. Так, если у вас есть общий класс типа:

```
type
  TSimpleGeneric<T> = class
    Value: T;
  end;
```

поскольку вы объявляете конкретный объект с заданным типом, вы не можете присвоить полю `Value` значение другого типа. Для следующих двух объектов, некоторые из приведенных ниже присваиваний (часть прикладного проекта TypeCompRules) некорректны:

```
var
  Sg1: TSimpleGeneric<string>;
  Sg2: TSimpleGeneric<Integer>;
begin
  Sg1 := TSimpleGeneric<string>.Create;
  Sg2 := TSimpleGeneric<Integer>.Create;

  Sg1.Value := 'foo';
  Sg1.Value := 10; // Error
  // E2010 Incompatible types: 'string' and 'Integer'

  Sg2.Value := 'foo';  // Error
  // E2010 Incompatible types: 'Integer' and 'string'
  Sg2.Value := 10;
```

Как только вы определяете конкретный тип в generic объявлении, это компилятор накладывает ограничения, как и следовало ожидать от языка со строгой типизацией, такого как Object Pascal. Проверка типа также применяется для generic объектов в целом. Так как вы указываете generic параметр для объекта, вы не можете присвоить ему похожий generic тип, основанный на другом и несовместимом экземпляре типа. Если пока это кажется запутанным, пример должен помочь прояснить:

```
Sg1 := TSimpleGeneric<Integer>.Create; // Error
```

```
// E2010 Incompatible types:
// 'TSimpleGeneric<System.string>'
// and 'TSimpleGeneric<System.Integer>'
```

Как мы увидим в разделе "Правила совместимости Generic типов", в данном конкретном случае правило совместимости типов происходит по структуре, а не по имени типа. Вы не можете присвоить другой и несовместимый тип после того, как он был объявлен.

Дженерики в Object Pascal

В предыдущем примере мы видели, как можно определить и использовать generic класс в Object Pascal. Я решил представить эту возможность на примере, прежде чем углубляться в технические подробности, которые достаточно сложны и в то же время очень важны. После рассмотрения дженериков с точки зрения языка мы вернемся к большему количеству примеров, включая использование и определение общих контейнерных классов, одно из основных применений этой техники в языке.

Мы видели, что когда вы определяете класс, вы можете добавить дополнительный "параметр" в угловых скобках, чтобы удержать место типа, который будет предоставлен позже:

```
type
  TMyClass<T> = class
    ...
  end;
```

Тип generic может использоваться как тип поля (как я делал это в предыдущем примере), как тип свойства, как тип параметра или возвращаемого значения функции, и многое другое. Обратите внимание, что использование типа для локального поля (или массива) не является обязательным, так

как бывают случаи, когда родовой тип используется только как результат, как параметр, или не используется в объявлении класса, а только в определении некоторых его методов.

Такая форма расширенного или общего объявления типа доступна не только для классов, но и для записей (которые, как я уже говорил в Гл. 5, могут также иметь методы, свойства и перегруженные операторы). Generic класс может также иметь несколько параметризованных типов, как в следующем случае, когда вы можете указать входной параметр и возвращаемое значение другого типа для метода:

```
type
  TPWGeneric<TInput,TReturn> = class
  public
    function AnyFunction (Value: TInput): TReturn;
  end;
```

Реализация дженериков в Object Pascal, как и в других статических языках, не основана на поддержке исполнения. Она обрабатывается компилятором и компоновщиком, практически ничего не оставляя механизму исполнения. В отличие от вызовов виртуальных функций, которые привязываются во время выполнения, общие методы класса генерируются один раз для каждого генерируемого Вами типа и генерируются во время компиляции! Мы увидим возможные недостатки такого подхода, но с положительной стороны он подразумевает, что общие классы не менее эффективны, чем простые классы, или даже более эффективны, так как необходимость в проверке во время выполнения уменьшается. Однако, прежде чем мы рассмотрим некоторые внутренние моменты, позвольте мне сосредоточиться на некоторых очень важных правилах, которые нарушают традиционные правила совместимости типов языка Pascal.

Правила совместимости Generic

типов

В традиционном Pascal и в Object Pascal правила совместимости типов основаны на эквивалентности имен типов. Другими словами, две переменные совместимы по типам только в том случае, если имя их типа совпадает, независимо от фактической структуры данных, к которым они относятся.

Это классический пример несовместимости типов со статическими массивами (часть прикладного проекта TypeCompRules):

```
type
  TArrayOf10 = array [1..10] of Integer;

procedure TForm30.Button1Click(Sender: TObject);
var
  Array1: TArrayOf10;
  Array2: TArrayOf10
  Array3, Array4: array [1..10] of Integer;
begin
  Array1 := Array2;
  Array2 := Array3; // Error
  // E2010 Incompatible types: 'TArrayOf10' and 'Array'

  Array3 := Array4;
  Array4 := Array1; // Error
  // E2010 Incompatible types: 'Array' and 'TArrayOf10'
end;
```

Как видно из приведенного кода, все четыре массива структурно идентичны. Однако компилятор позволит присваивать только те массивы, которые совместимы с типами, либо потому, что их тип имеет одно и то же явное имя (например, `TArrayOf10`), либо потому, что они имеют одно и то же неявное (или сгенерированное компилятором имя типа, как два массива, объявленные в одном выражении).

Это правило совместимости типов имеет очень ограниченные исключения, например, те, которые относятся к производным классам. Другое исключение из правила, и значительное, это совместимость типов для generic типов, которая, вероятно,

также используется компилятором внутренне, чтобы определить, когда *генерировать* новый тип из generic, со всеми его методами.

Новое правило гласит, что generic типы совместимы, когда они разделяют одно и то же определение generic класса и тип экземпляра, независимо от имени типа, связанного с этим определением. Другими словами, полное имя экземпляра общего типа является комбинацией generic типа и типа экземпляра.

В следующем примере все четыре переменные совместимы по типу:

```
type
  TGenericArray<T> = class
    AnArray: array [1..10] of T;
  end;

  TIntGenericArray = TGenericArray<Integer>;

procedure TForm30.Button2Click(Sender: TObject);
var
  Array1: TIntGenericArray;
  Array2: TIntGenericArray;
  Array3, Array4: TGenericArray<Integer>;
begin
  Array1 := TIntGenericArray.Create;
  Array2 := Array1;
  Array3 := Array2;
  Array4 := Array3;
  Array1 := Array4;
end;
```

Generic методы для стандартных классов

В то время как использование дженериковых типов для определения классов, вероятно, является наиболее распространенным сценарием, generic типы могут также использоваться в обычных классах. Другими словами, обычный

класс может иметь generic метод. В этом случае, при создании экземпляра класса, а также при вызове метода, вы не указываете конкретный тип для generic заполнителя. Приведем пример класса с общим методом из прикладного проекта GenericMethod:

```
type
  TGenericFunction = class
  public
    function WithParam <T> (T1: T): string;
  end;
```

примечание Когда я впервые написал этот код, вероятно, с воспоминаниями о моих днях на Си++, я написал параметр как `(t: T)`. Излишне говорить, что в случае с таким нечувствительным языком, как Object Pascal, это не самая лучшая идея. На самом деле компилятор пропустит его, но будет выдавать ошибки каждый раз, когда вы ссылаетесь на generic тип `T`.

Внутри аналогичного метода класса мало что можно сделать (по крайней мере, если вы не используете ограничения, рассмотренные позже в этой главе), поэтому я написал код, использующий специальные функции generic типа (опять же рассмотренные позже) и специальную функцию для приведения типа к строке, что здесь обсуждать не уместно:

```
function TGenericFunction.WithParam<T>(T1: T): string;
begin
  Result := GetTypeName (TypeInfo (T));
end;
```

Как видно, в этом методе в качестве параметра даже не используется фактическое значение, переданное в качестве параметра, а только захватывается информация какого-то типа. Опять же, незнание типа `t1` вообще усложняет его использование в коде.

Различные версии этой "глобальной общей функции" можно вызвать так:

```
var
  Gf: TGenericFunction;
begin
  Gf := TGenericFunction.Create;
  try
    Show (Gf.WithParam<string>('foo'));
```

```
    Show (Gf.WithParam<Integer> (122));
    Show (Gf.WithParam('hello'));
    Show (Gf.WithParam (122));
    Show (Gf.WithParam(Button1));
    Show (Gf.WithParam<TObject>(Button1));
  finally
    Gf.Free;
  end;
```

Все вышеперечисленные вызовы корректны, так как параметрический тип может быть неявным в этих вызовах. Обратите внимание, что отображается тип дженерика (как заданный или выведенный), а не фактический тип параметра, что объясняет этот вывод:

```
string
Integer
string
ShortInt
TButton
TObject
```

Если вызвать метод без указания типа между угловыми скобками, то фактический тип будет выведен из типа параметра. Если метод вызывается с типом и параметром, то тип параметра должен соответствовать общему объявлению типа. Таким образом, три строки, приведенные ниже, не скомпилируются:

```
    Show (Gf.WithParam<Integer>('foo'));
    Show (Gf.WithParam<string> (122));
    Show (Gf.WithParam<TButton>(self));
```

Инициализация типа Generic

Обратите внимание, что это довольно продвинутый раздел, посвященный некоторым внутренним компонентам дженериков и их потенциальной оптимизации. Хорошо для второго прочтения, но не в том случае, если вы смотрите на дженерики в первый раз.

За исключением некоторых оптимизаций, каждый раз при инициализации generic типа, как в методе, так и в классе, компилятор генерирует новый тип. Этот новый тип не разделяет код с разными экземплярами одного и того же generic типа (или разными версиями одного и того же метода).

Рассмотрим пример (который является частью проекта приложения GenericCodeGen). В программе есть generic класс, определяемый как:

```
type
  TSampleClass <T> = class
  private
    data: T;
  public
    procedure One;
    function ReadT: T;
    procedure SetT (value: T);
  end;
```

Эти три метода реализованы следующим образом (обратите внимание, что метод `One` абсолютно не зависит от generic типа):

```
procedure TSampleClass<T>.One;
begin
  Form30.Show ('OneT');
end;

function TSampleClass<T>.ReadT: T;
begin
  Result := data;
end;

procedure TSampleClass<T>.SetT(value: T);
begin
  data := value;
end;
```

Теперь основная программа в основном использует универсальный тип для вычисления адреса в памяти своих методов после генерации экземпляра (компилятором). Это код

```
procedure TForm30.Button1Click(Sender: TObject);
var
  T1: TSampleClass<Integer>;
  T2: TSampleClass<string>;
begin
  T1 := TSampleClass<Integer>.Create;
  T1.SetT (10);
```

```
  T1.One;

  T2 := TSampleClass<string>.Create;
  T2.SetT ('hello');
  T2.One;

  Show ('T1.SetT: ' +
    IntToHex (PInteger(@TSampleClass<Integer>.SetT)^, 8));
  Show ('T2.SetT: ' +
    IntToHex (PInteger(@TSampleClass<string>.SetT)^, 8));

  Show ('T1.One: ' +
    IntToHex (PInteger(@TSampleClass<Integer>.One)^, 8));
  Show ('T2.One: ' +
    IntToHex (PInteger(@TSampleClass<string>.One)^, 8));
end;
```

В результате получается нечто подобное (фактические значения будут меняться):

```
T1.SetT: C3045089
T2.SetT: 51EC8B55
T1.One: 4657F0BA
T2.One: 46581CBA
```

Как я и предполагал, метод `SetT` не только получает в памяти разные версии, генерируемые компилятором для каждого используемого типа данных, но и метод `One` несмотря на то, что все они идентичны.

Более того, если Вы заново объявите идентичный generic тип, то получите новый набор функций реализации. Аналогично, один и тот же экземпляр generic типа, используемый в разных юнитах, заставляет компилятор генерировать один и тот же код снова и снова, что может привести к значительному раздуванию кода. По этой причине, если у вас есть общий класс со множеством методов, которые не зависят от generic типа, рекомендуется определить базовый не-generic класс с этими общими методами и унаследованный generic класс - с generic методами: таким образом, методы базового класса компилируются и включаются в исполняемый файл только один раз.

примечание В настоящее время ведется работа над компилятором, компоновщиком и низкоуровневой RTL, чтобы уменьшить увеличение размеров, вызываемое дженериками в сценариях, подобных описанным в этом разделе. См., например, соображения по этому поводу на сайте http://delphisorcery.blogspot.it/2014/10/new-language-feature-in-xe7.html.

Функции типа Generic

Самая большая проблема с определениями типов generic, которые мы видели до сих пор, заключается в том, что очень мало можно сделать с элементами класса generic типа. Есть две техники, которые вы можете использовать, чтобы преодолеть это ограничение. Первая – это использование нескольких специальных функций библиотеки времени исполнения, которые специально поддерживают generic типы; вторая (и гораздо более мощная) – это определение generic классов с ограничениями на типы, которые вы можете использовать.

Я сосредоточусь на первой технике в этом разделе и на ограничениях в следующем разделе. Как я уже упоминал, есть несколько RTL-функций, которые работают с параметрическим типом (`T`) определения типа generic:

- Default `(T)` - это фактически новая функция, введенная вместе с дженериками, которая возвращает пустое или "нулевое значение" или нулевое значение для текущего типа; это может быть ноль, пустая строка, `nil` и т.д.; нулевая инициализированная память имеет то же самое значение глобальной переменной того же типа (в отличие от локальных переменных, на самом деле, глобальные переменные инициализируются компилятором до "нуля");
- TypeInfo `(T)` возвращает указатель на информацию о выполнении для текущей версии generic типа; вы найдете намного больше информации о типах в Гл. 16;

- SizeOf `(T)` возвращает размер памяти типа в байтах (в случае ссылочного типа строка или объект будет размером ссылки, то есть 4 байта для 32-битного компилятора и 8 байт для 64-битного компилятора).
- IsManagedType`(T)` указывает, управляется ли тип в памяти, как это происходит для строк и динамических массивов
- HasWeakRef`(T)` привязан к компиляторам с поддержкой ARC и указывает, имеет ли целевой тип слабые ссылки, требующие специальной поддержки управления памятью.
- GetTypeKind`(T)` - это ярлык для доступа к виду типа из информации о типе; это немного более высокое определение типа, чем определение, возвращаемое `TypeInfo`.

примечание Все эти методы возвращают компилятору константы, а не вызывают актуальные функции во время выполнения. Важность этого заключается не в том, что эти операции выполняются очень быстро, а в том, что это позволяет компилятору и компоновщику оптимизировать сгенерированный код, удаляя неиспользуемые ветки. Если у вас есть оператор case или if, основанный на возвращаемом значении одной из этих функций, то компилятор может выяснить, что для данного типа будет выполняться только одна из ветвей, удалив неиспользуемый код. Когда один и тот же generic метод компилируется для другого типа, он может закончиться использованием другой ветви, но опять же компилятор может вычислить заранее и оптимизировать размер метода.

В проекте приложения GenericTypeFunc имеется общий класс, показывающий в действии три функции общего типа:

```
type
  TSampleClass <T> = class
  private
    FData: T;
  public
    procedure Zero;
    function GetDataSize: Integer;
    function GetDataName: string;
  end;

function TSampleClass<T>.GetDataSize: Integer;
begin
  Result := SizeOf (T);
end;

function TSampleClass<T>.GetDataName: string;
```

```
begin
  Result := GetTypeName (TypeInfo (T));
end;

procedure TSampleClass<T>.Zero;
begin
  FData := Default (T);
end;
```

В методе `GetDataName` я использовал функцию `GetTypeName` (модуля System.TypeInfo) вместо прямого доступа к структуре данных, так как она выполняет правильное преобразование из закодированного значения строки, содержащей имя типа.

Учитывая вышеприведенное объявление, можно скомпилировать следующий тестовый код, который повторяется три раза на трех различных экземплярах общего типа. Я пропустил повторный код, но покажу утверждения, используемые для доступа к полю `data`, так как они меняются в зависимости от фактического типа:

```
var
  T1: TSampleClass<Integer>;
  T2: TSampleClass<string>;
  T3: TSampleClass<double>;
begin
  T1 := TSampleClass<Integer>.Create;
  T1.Zero;
  Show ('TSampleClass<Integer>');
  Show ('data: ' + IntToStr (T1.FData));
  Show ('type: ' + T1.GetDataName);
  Show ('size: ' + IntToStr (T1.GetDataSize));

  T2 := TSampleClass<string>.Create;
  ...
  Show ('data: ' + T2.FData);

  T3 := TSampleClass<double>.Create;
  ...
  Show ('data: ' + FloatToStr (T3.FData));
```

Запуск этого кода (из прикладного проекта GenericTypeFunc) дает следующий результат:

```
TSampleClass<Integer>
data: 0
type: Integer
size: 4
TSampleClass<string>
```

```
data:
type: string
size: 4
TSampleClass<double>
data: 0
type: Double
size: 8
```

Обратите внимание, что функции generic типа можно использовать и для конкретных типов, вне контекста generic классов. Например, вы можете писать:

```
var
  I: Integer;
  s: string;
begin
  I := Default (Integer);
  Show ('Default Integer': + IntToStr (I));

  s := Default (string);
  Show ('Default String': + s);

  Show ('TypeInfo String': +
    GetTypeName (TypeInfo (string));
```

что дает тривиальный выход:

```
Default Integer: 0
Default String:
TypeInfo String: string
```

примечание Вы не можете применить вызов `TypeInfo` к переменной, как `TypeInfo(s)` в коде выше, а только к типу данных.

Конструкторы классов для Generic классов

Очень интересный случай возникает, когда вы определяете конструктор класса для generic класса. Фактически, один такой конструктор генерируется компилятором и вызывается для каждого экземпляра generic класса, т.е. для каждого реального типа, определенного с помощью общего шаблона. Это довольно интересно, так как было бы очень сложно выполнить код

инициализации для каждого фактического экземпляра generic класса, который Вы собираетесь создать в своей программе без конструктора класса.

В качестве примера рассмотрим generic класс с некоторыми данными класса. Вы получите экземпляр данных этого класса для каждого общего экземпляра класса. Если вам нужно присвоить начальное значение данным этого класса, вы не можете использовать код инициализации модуля, так как в модуле, определяющем generic класс, вы не знаете, какие именно классы вам понадобятся.

Ниже приведен пример generic класса с конструктором классов, который используется для инициализации поля `DataSize` класса, взятого из прикладного проекта GenericClassCtor:

```
type
  TGenericWithClassCtor <T> = class
  private
    FData: T;
    procedure SetData(const Value: T);
  public
    class constructor Create;
    property Data: T read FData write SetData;
    class var
      DataSize: Integer;
  end;
```

Ниже код общего конструктора класса generic, который использует внутренний список строк (подробнее о реализации см. полный исходный код) для отслеживания того, какие конструкторы классов на самом деле вызываются:

```
class constructor TGenericWithClassCtor<T>.Create;
begin
  DataSize := SizeOf (T);
  ListSequence.Add(ClassName);
end;
```

Демонстрационная программа создает и использует пару экземпляров generic класса, а также объявляет тип данных для третьего, который удаляется компоновщиком:

```
var
  GenInt: TGenericWithClassCtor <SmallInt>;
```

```
  GenStr: TGenericWithClassCtor <string>;
type
  TGenDouble = TGenericWithClassCtor <Double>;
```

Если вы попросите программу показать содержимое списка строк `ListSequence`, вы увидите только те типы, которые были фактически инициализированы:

```
TGenericWithClassCtor<Система.SmallInt>
TGenericWithClassCtor<Система.string>
```

Однако, если вы создаёте generic экземпляры, основанные на одном и том же типе данных в разных модулях, компоновщик может работать не так, как ожидалось, и у вас будет несколько вызовов одного и того же generic конструктора классов (или, точнее, двух generic конструкторов классов для одного и того же типа).

примечание Нелегко решить подобную проблему. Чтобы избежать повторной инициализации, можно проверить, не был ли уже выполнен конструктор класса. В целом, однако, эта проблема является частью более полного ограничения generic классов и неспособности компоновщиков их оптимизировать.

Я добавил процедуру под названием `Useless` (`"Бесполезно")` во вторичный модуль этого примера, которая, когда она не будет закомментирована, высветит проблему, с такой последовательностью инициализации в виде:

```
TGenericWithClassCtor<Система.string>
TGenericWithClassCtor<Система.SmallInt>
TGenericWithClassCtor<Система.string>
```

Ограничения Generic

Как мы видели, очень мало что можно сделать в методах вашего обобщенного класса над обобщенным значением типа. Вы можете передать его (то есть присвоить его) и выполнить

ограниченные операции, разрешенные функциями общего типа, о которых я только что рассказал.

Для того чтобы иметь возможность выполнять некоторые фактические операции класса generic типа, как правило, необходимо наложить на него ограничение. Например, если вы ограничиваете generic тип классом, компилятор позволит вам вызывать на нем все методы `TObject`. Вы также можете дополнительно ограничить класс, чтобы он был частью заданной иерархии, или чтобы он реализовывал определенный интерфейс, делая возможным вызов метода класса или интерфейса на экземпляре generic типа.

Ограничения класса

Самое простое ограничение, которое вы можете принять - это ограничение класса. Для его использования можно объявить тип generic как:

```
type
  TSampleClass <T: class> = class
```

Указав ограничение класса, вы указываете, что в качестве generic типов можно использовать только типы объектов. С помощью следующего объявления (взятого из прикладного проекта ClassConstraint):

```
type
  TSampleClass <T: class> = class
  private
    FData: T;
  public
    procedure One;
    function ReadT: T;
    procedure SetT (T1: T);
  end;
```

вы можете создать первые два экземпляра, но не третий:

```
  sample1: TSampleClass<TButton>;
  sample2: TSampleClass<TStrings>;
  sample3: TSampleClass<Integer>; // Error
```

Ошибка компилятора, вызванная этим последним объявлением:

```
E2511 Type parameter 'T' must be a class type
```

В чем преимущество указания на это ограничение? В generic методах класса теперь можно вызывать любые методы `TObject`, в том числе и виртуальные! Это метод `One` общего класса `TSampleClass`:

```
procedure TSampleClass<T>.One;
begin
  if Assigned (FData) then
  begin
    Form30.Show ('ClassName: ' + FData.ClassName);
    Form30.Show ('Size: ' + IntToStr (FData.InstanceSize));
    Form30.Show ('ToString: ' + FData.ToString);
  end;
end;
```

примечание Два комментария. Первый заключается в том, что InstanceSize возвращает фактический размер объекта, в отличие от общей функции SizeOf, которую мы использовали ранее, возвращающей размер ссылочного типа. Во-вторых, обратите внимание на использование метода `ToString` класса `TObject`.

Вы можете поиграть с программой, чтобы увидеть ее реальный эффект, так как она определяет и использует несколько экземпляров generic типа, как в следующем фрагменте кода:

```
var
  Sample1: TSampleClass<TButton>;
begin
  Sample1 := TSampleClass<TButton>.Create;
  try
    Sample1.SetT (Sender as TButton);
    Sample1.One;
  finally
    Sample1.Free;
  end;
```

Обратите внимание, что объявив класс с настраиваемым методом `ToString`, эта версия будет вызываться, когда объект данных имеет определенный тип, независимо от фактического типа, предоставленного для generic типа. Другими словами, если у вас есть такой же потомок `TButton`:

```
type
  TMyButton = class (TButton)
```

```
  public
    function ToString: string; override;
  end;
```

Вы можете передать этот объект в качестве значения `TSampleClass<TButton>` или определить конкретный экземпляр общего типа, и в обоих случаях вызов `One` заканчивается выполнением определенной версии ToString:

```
var
  Sample1: TSampleClass<TButton>;
  Sample2: TSampleClass<TMyButton>;
  Mb: TMyButton;
begin
  ...
  Sample1.SetT (Mb);
  Sample1.One;
  Sample2.SetT (Mb);
  Sample2.One;
```

Подобно ограничению класса, вы можете иметь ограничение на запись, объявленное как:

```
type
  TSampleRec <T: record> = class
```

Однако очень мало общего у разных записей (нет общего предка), поэтому такое определение несколько ограничено.

Конкретные ограничения классов

Если ваш generic класс должен работать с определенным подмножеством классов (определенной иерархией), вы можете прибегнуть к указанию ограничения, основанного на данном базовом классе. Например, если вы объявите:

```
type
  TCompClass <T: TComponent> = class
```

экземпляры этого общего класса могут быть применены только к классам компонентов, то есть к любому классу-потомку `TComponent`. Это позволит вам иметь очень специфический generic тип (да, это звучит странно, но на самом деле это так), и

компилятор позволит вам использовать все методы класса `TComponent` при работе с общим типом.

Если это кажется чрезвычайно мощным, подумайте дважды. Если подумать о том, чего можно добиться с помощью правил наследования и совместимости типов, то можно решить ту же проблему, используя традиционные объектно-ориентированные техники, вместо того, чтобы использовать общие классы. Я не говорю, что конкретное ограничение класса никогда не приносит пользы, но оно, конечно, не такое мощное, как ограничение класса более высокого уровня или (что мне кажется очень интересным) ограничение, основанное на интерфейсе.

Ограничения, основанные на интерфейсе

Вместо того, чтобы ограничивать общий generic определенным классом, обычно более гибко принимать в качестве параметров типа только классы, реализующие заданный интерфейс. Это позволяет вызывать интерфейс на экземплярах generic типа. Такое использование интерфейсных ограничений для дженериков также очень распространено в языке C#. Позвольте мне начать с показа примера (из проекта приложения IntfConstraint). Сначала нам нужно объявить интерфейс:

```
type
  IGetValue = interface
    ['{60700EC4-2CDA-4CD1-A1A2-07973D9D2444}']
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    property Value: Integer read GetValue write SetValue;
  end;
```

Далее мы можем определить класс, реализующий его:

```
type
```

```
  TGetValue = class (TSingletonImplementation, IGetValue)
  private
    FValue: Integer;
  public
    constructor Create (Value: Integer = 0);
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
```

Все становится интереснее с определением generic класса, ограниченного типами, реализующими данный интерфейс:

```
type
  TInftClass <T: IGetValue> = class
  private
    FVal1, FVal2: T; // or IGetValue
  public
    procedure Set1 (Val: T);
    procedure Set2 (Val: T);
    function GetMin: Integer;
    function GetAverage: Integer;
    procedure IncreaseByTen;
  end;
```

Заметьте, что в коде generic методов данного класса можно написать, например:

```
function TInftClass<T>.GetMin: Integer;
begin
  Result := Min (FVal1.GetValue, FVal2.GetValue);
end;

procedure TInftClass<T>.IncreaseByTen;
begin
  FVal1.SetValue (FVal1.GetValue + 10);
  FVal2.Value := FVal2.Value + 10;
end;
```

Со всеми этими определениями, теперь мы можем использовать generic класс следующим образом:

```
procedure TFormIntfConstraint.BtnValueClick(
  Sender: TObject);
var
  IClass: TInftClass<TGetValue>;
begin
  IClass := TInftClass<TGetValue>.Create;
  try
    IClass.Set1 (TGetValue.Create (5));
    IClass.Set2 (TGetValue.Create (25));
    Show ('Average: ' + IntToStr (IClass.GetAverage));
    IClass.IncreaseByTen;
    Show ('Min: ' + IntToStr (IClass.GetMin));
  finally
```

```
    IClass.val1.Free;
    IClass.val2.Free;
    IClass.Free;
  end;
end;
```

Чтобы показать гибкость этого generic класса, я создал еще одну совершенно другую реализацию для интерфейса:

```
type
  TButtonValue = class (TButton, IGetValue)
  public
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    class function MakeTButtonValue (Owner: TComponent;
      Parent: TWinControl): TButtonValue;
  end;

function TButtonValue.GetValue: Integer;
begin
  Result := Left;
end;

procedure TButtonValue.SetValue(Value: Integer);
begin
  Left := Value;
end;
```

Функция класса (не перечисленная в книге) создает кнопку внутри `Parent` компонента в случайном положении и используется в следующем примере кода:

`Parent` control in a random position and is used in the following sample code:

```
procedure TFormIntfConstraint.BtnValueButtonClick(
  Sender: TObject);
var
  IClass: TInftClass<TButtonValue>;
begin
  IClass := TInftClass<TButtonValue>.Create;
  try
    IClass.Set1 (TButtonValue.MakeTButtonValue (
      self, ScrollBox1));
    IClass.Set2 (TButtonValue.MakeTButtonValue (
      self, ScrollBox1));
    Show ('Average: ' + IntToStr (IClass.GetAverage));
    Show ('Min: ' + IntToStr (IClass.GetMin));
    IClass.IncreaseByTen;
    Show ('New Average: ' + IntToStr (IClass.GetAverage));
  finally
    IClass.Free;
```

```
  end;
end;
```

Ссылки на интерфейс vs. Generic ограничения по интерфейсу

В последнем примере я определил generic класс, который работает с любым объектом, реализующим заданный интерфейс. Я мог бы получить аналогичный эффект, создав стандартный (не дженерик) класс на основе ссылок на интерфейс. На самом деле, я мог бы определить такой же класс (опять же, часть проекта приложения IntfConstraint):

```
вводить
type
  TPlainInftClass = class
  private
    FVal1, FVal2: IGetValue;
  public
    procedure Set1 (Val: IGetValue);
    procedure Set2 (Val: IGetValue);
    function GetMin: Integer;
    function GetAverage: Integer;
    procedure IncreaseByTen;
  end;
```

Чем отличаются эти два подхода? Первое отличие состоит в том, что в вышестоящем классе вы можете передать в методы setter два объекта разных типов при условии, что оба их класса реализуют данный интерфейс, в то время как в варианте с generic вы можете передать только объекты данного типа (в любой данный экземпляр generic класса). Таким образом, версия generic является более *консервативной* и строгой с точки зрения проверки типов.

Ключевое отличие, на мой взгляд, заключается в том, что использование интерфейсной версии означает действие механизма подсчета ссылок Object Pascal, в то время как при использовании стандартной версии класс имеет дело с

обычными объектами данного типа и подсчет ссылок не осуществляется.

Более того, версия generic может иметь множество ограничений, таких как ограничение конструктора, и позволяет вам использовать различные общие функции (например, запрос фактического типа общего типа), то, что вы не можете сделать при использовании интерфейса. (Когда вы работаете с интерфейсом, на самом деле, у вас нет возможности ссылаться на базовые методы `TObject`).

Другими словами, использование generic класса с ограничениями по интерфейсу дает возможность пользоваться преимуществами интерфейсов без их неудобств. Тем не менее, стоит отметить, что в большинстве случаев эти два подхода были бы эквивалентны, а в других - решение на основе интерфейсов было бы более гибким.

Ограничение конструктор по умолчанию

Существует еще одно возможное ограничение generic типа, называемое конструктором по умолчанию или безпараметрическим конструктором. Если необходимо вызвать конструктор по умолчанию для создания нового объекта generic типа (например, для заполнения списка), то можно использовать это ограничение. Теоретически (и согласно документации) компилятор должен позволять использовать его только для тех типов, которые имеют конструктор по умолчанию. На практике, если конструктор по умолчанию не существует, компилятор пропустит его и вызовет конструктор `TObject` по умолчанию.

Generic класс с ограничением конструктора можно записать следующим образом (этот класс извлекается прикладным проектом IntfConstraint):

```
type
  TConstrClass <T: class, constructor> = class
  private
    FVal: T;
  public
    constructor Create;
    function Get: T;
  end;
```

примечание Также можно указать ограничение конструктора без ограничения класса, так как первое, вероятно, подразумевает второе. Наличие обоих делает код более читабельным.

Учитывая это объявление, вы можете использовать конструктор для создания общего внутреннего объекта, не зная заранее его фактического типа, и написать:

```
constructor TConstrClass<T>.Create;
begin
  FVal := T.Create;
end;
```

Как мы можем использовать этот generic класс и каковы реальные правила? В следующем примере я определил два класса, один с конструктором по умолчанию (без параметров), второй с одним конструктором, имеющим один параметр:

```
type
  TSimpleConst = class
  public
    FValue: Integer;
    constructor Create;  // set Value to 10
  end;

  TParamConst = class
  public
    FValue: Integer;
    constructor Create (I: Integer); // set Value to I
  end;
```

Как я уже говорил, теоретически можно использовать только первый класс, а на практике можно использовать и то, и другое:

```
var
  ConstructObj: TConstrClass<TSimpleConst>;
```

```
  ParamConstObj: TConstrClass<TParamConst>;
begin
  ConstructObj := TConstrClass<TSimpleConst>.Create;
  Show ('Value 1: ' + IntToStr (ConstructObj.Get.FValue));

  ParamCostObj := TConstrClass<TParamConst>.Create;
  Show ('Value 2: ' + IntToStr (ParamConstObj.Get.FValue));
```

Этот код выводит:

```
Value 1: 10
Value 2: 0
```

На самом деле, второй объект никогда не инициализируется. Если вы выполните трассировку кода в отладчике, то увидите вызов `TObject.Create` (что я считаю неправильным). Обратите внимание, что если вы попробуете вызвать напрямую:

```
  with TParamConst.Create do
```

компилятор (корректно) поднимет ошибку:

```
[DCC Error] E2035 Not enough actual parameters
[Ошибка DCC] E2035 Недостаточно фактических параметров.
```

примечание Даже если прямой вызов `TParamConst.Create` не удастся выполнить во время компиляции (как объяснено здесь), аналогичный вызов с использованием ссылки на класс или любой другой формы индирекции будет успешным, что, вероятно, объясняет поведение эффекта ограничения конструктора.

Сводка ограничений и их комбинация

Поскольку существует так много различных ограничений, которые вы можете наложить на тип дженерика, позвольте мне дать краткое описание здесь, в терминах кода:

```
type
  TSampleClass <T: class> = class
  TSampleRec <T: record> = class
  TCompClass <T: TButton> = class
  TInftClass <T: IGetValue> = class
  TConstrClass <T: constructor> = class
```

То, что вы, возможно, не сразу поймете после изучения ограничений (безусловно, и мне потребовалось некоторое

время, чтобы привыкнуть к ним), так это то, что вы можете комбинировать их. Например, вы можете определить generic класс, ограниченный суб-иерархией и требующий также заданного интерфейса, например:

```
type
  TInftComp <T: TComponent, IGetValue> = class
```

Не все комбинации имеют смысл: например, вы не можете указать и класс, и запись, в то время как использование ограничения класса в сочетании с конкретным ограничением класса было бы излишним. Наконец, заметьте, что нет ничего похожего на ограничение метода, чего можно достичь с помощью ограничения по интерфейсу с одним методом (гораздо более сложного в выражении).

Готовые Generic контейнеры

С первых дней существования шаблонов в языке Си++ одним из наиболее очевидных применений шаблонных классов стало определение контейнеров или списков шаблонов, вплоть до того, что в языке Си++ была определена Стандартная библиотека шаблонов (или STL).

Когда вы определяете список объектов, например, собственный TObjectList Object Pascal, у вас есть список, который потенциально может содержать объекты любого типа. Используя наследование или композицию, вы действительно можете определить пользовательские контейнеры для определенного типа, но это утомительный (и потенциально склонный к ошибкам) подход.

Компиляторы Object Pascal поставляются с небольшим набором универсальных контейнерных классов, которые можно найти в модуле Generics.Collections. Все четыре основных контейнерных класса реализованы независимым образом (среди этих классов нет наследования), все они реализованы аналогичным образом (с использованием динамического массива) и все они привязаны к соответствующему контейнерному классу без джененриков старого модуля `Contnrs`:

```
type
  TList<T> = class
  TQueue<T> = class
  TStack<T> = class
  TDictionary<TKey,TValue> = class
  TObjectList<T: class>  = class(TList<T>)
  TObjectQueue<T: class> = class(TQueue<T>)
  TObjectStack<T: class> = class(TStack<T>)
  TObjectDictionary<TKey,TValue> = class(TDictionary<TKey,TValue>)
```

Логическое различие между этими классами должно быть вполне очевидным, учитывая их названия. Хороший способ их проверки - это выяснить, сколько изменений нужно внести в существующий код, использующий класс-контейнер без дженериков.

примечание Программа использует всего несколько методов, так что это не очень хороший тест на совместимость интерфейса между generic и не-дженерик списками, но я решил взять существующую программу, а не создавать ее. Другая причина для демонстрации этого примера состоит в том, что у вас могут быть также существующие программы, которые не используют generic классы коллекции, и вам будет предложено улучшить их, воспользовавшись этой особенностью языка.

Использование TList<T>

Программа, называемая ListDemoMd2005, имеет модуль, определяющий класс `TDate`, и основную форму, используемую для обращения к списку дат `TList`. В качестве отправной точки я добавил в uses ссылку на Generics.Collections, а затем изменил объявление основного поля формы на:

```
  private
    FListDate: TList <TDate>;
```

Конечно, обработчик событий `OnCreate` основной формы, который на деле создает список, также необходимо обновить:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FListDate := TList<TDate>.Create;
end;
```

Теперь мы можем попробовать скомпилировать остальной код как есть. Программа имеет ожидаемую ошибку, пытаясь добавить объект TButton в список. Соответствующий код, который использовался для компиляции, теперь не работает:

```
procedure TForm1.ButtonWrongClick(Sender: TObject);
begin
  // add a button to the list
  FListDate.Add (Sender);  // Error:
    // E2010 Incompatible types: 'TDate' and 'TObject'
```

```
end;
```

Новый список дат является более надежным с точки зрения проверки типа, чем оригинальный общий список указателей. После удаления этой строки, программа компилируется и работает. Тем не менее, он может быть улучшен.

Вот оригинальный код, используемый для отображения всех дат списка в элемент управления ListBox:

```
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add (
      (TObject(FListDate [I]) as TDate).Text);
```

Обратите внимание на приведенный тип, так как программа использовала список указателей (`TList`), а не список объектов (TObjectList). Мы можем легко улучшить программу, написав:

```
  for I := 0 to FListDate.Count - 1 do
    Listbox1.Items.Add (FListDate [I].Text);
```

Другое улучшение этого фрагмента может быть достигнуто за счет использования перечисления (что полностью поддерживается готовыми generic списками), вместо простого цикла `for`:

```
var
  ADate: TDate;
begin
  for ADate in FListDate do
  begin
    Listbox1.Items.Add (ADate.Text);
  end;
```

Наконец, программу можно улучшить, используя generic `TObjectList`, владеющий объектами `TDate`, но это тема для следующего раздела.

Как я упоминал ранее, generic класс `TList<T>` имеет высокую степень совместимости. В нем есть все классические методы, такие как `Add`, `Insert`, `Remove` и `IndexOf`. Свойства `Capacity` и `Count` также есть. Как не странно, `Items` становится `Item`, но будучи

свойством по умолчанию (доступ к которому осуществляется с помощью квадратных скобок без имени свойства), вы все равно редко явно на него ссылаетесь.

Сортировка TList<T>

Интересно понять, как работает сортировка `TList<T>` (моя цель здесь - добавить поддержку сортировки в проект приложения ListDemoMd2005). Метод `Sort` определяется как:

```
procedure Sort; overload;
procedure Sort(const AComparer: IComparer<T>); overload;
```

где интерфейс IComparer<T> объявлен в модуле `Generics.Defaults.` При вызове первой версии программы будет использоваться функция сравнения по умолчанию, инициализированный конструктором по умолчанию `TList<T>.` В нашем случае это будет бесполезно.

Вместо этого нам нужно определить правильную реализацию интерфейса `IComparer<T>.` Для совместимости типов нам нужно определить реализацию, которая работает на конкретном классе `TDate`.

Для этого есть несколько способов, включая использование анонимных методов (рассматривается в следующем разделе, хотя эта тема будет представлена в следующей главе). Интересная техника, также потому, что она дает мне возможность показать несколько вариантов использования дженериков, состоит в том, чтобы воспользоваться *структурным* классом, который является частью модуля Generics.Defaults и называется TComparer.

примечание Я называю этот класс *структурным*, потому что он помогает определить структуру кода, его архитектуру, но не добавляет много с точки зрения реальной реализации. Хотя, возможно, есть и лучшее название для обозначения такого класса.

Класс определяется как абстрактная и generic реализация интерфейса:

```
type
  TComparer<T> = class(TInterfacedObject, IComparer<T>)
  public
    class function Default: IComparer<T>;
    class function Construct(
      const Comparison: TComparison<T>): IComparer<T>;
    function Compare(
      const Left, Right: T): Integer; virtual; abstract;
  end;
```

Что нам нужно сделать, так это инстанцировать этот generic класс для конкретного типа данных (`TDate`, в примере), а также наследовать конкретный класс, реализующий метод `Compare` для конкретного типа. Две операции можно выполнить сразу, используя идиому кодирования, которая может занять некоторое время для переваривания:

```
type
  TDateComparer = class (TComparer<TDate>)
    function Compare(
      const Left, Right: TDate): Integer; override;
  end;
```

Если вы думаете, что этот код выглядит очень необычно, то вы не одиноки. Новый класс наследует от конкретного экземпляра generic класса, что можно выразить двумя отдельными шагами как:

```
type
  TAnyDateComparer = TComparer<TDate>;
  TMyDateComparer = class (TAnyDateComparer)
    function Compare(
      const Left, Right: TDate): Integer; override;
  end;
```

примечание Наличие двух отдельных деклараций может помочь уменьшить генерируемый код при повторном использовании базового типа `TAnyDateComparer` в одном и том же модуле.

Реальную реализацию функции `Compare` можно найти в исходных текстах, т.к. это не ключевой момент, на который я хочу обратить внимание. Но имейте в виду, что даже если

отсортировать список, то его метод `IndexOf` не воспользуется им (в отличие от класса `TStringList`).

Сортировка анонимным методом

Сортировочный код, представленный в предыдущем разделе, выглядит достаточно сложным, и это действительно так. Было бы намного проще и чище передать функцию сортировки непосредственно в метод `Sort`. В прошлом это обычно достигалось передачей указателя функции. В Object Pascal это можно сделать, передав анонимный метод (своего рода указатель на метод, с несколькими дополнительными функциями, подробно рассмотренными в следующей главе).

примечание Я предлагаю вам взглянуть на этот раздел, даже если вы мало что знаете об анонимных методах, а затем прочитать его еще раз после того, как вы перейдете к следующей главе.

Параметр IComparer<T> метода `Sort` класса `TList<T>`, фактически, может быть использован путем вызова метода `Construct` TComparer<T>, передавая анонимный метод в качестве параметра, определенного как:

```
type
  TComparison<T> = reference to function(
    const Left, Right: T): Integer;
```

На практике можно записать несовместимую с типом функцию и передать ее в качестве параметра:

```
function DoCompare (const Left, Right: TDate): Integer;
var
  LDate, RDate: TDateTime;
begin
  LDate := EncodeDate(Left.Year, Left.Month, Left.Day);
  RDate := EncodeDate(Right.Year, Right.Month, Right.Day);
  if LDate = RDate then
    Result := 0
  else if LDate < RDate then
    Result := -1
  else
    Result := 1;
```

```
end;

procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  FListDate.Sort (TComparer<TDate>.Construct (DoCompare));
end;
```

примечание Метод DoCompare, описанный выше, работает как анонимный метод, даже если у него есть имя. Однако, позже мы увидим в фрагменте кода, что этого не требуется. Имейте терпение до следующей главы для получения более подробной информации об этой конструкции языка Object Pascal. Заметьте также, что с записью `TDate` я мог бы определить операторы меньше и больше, чем, делая этот код проще, но даже с классом я мог бы поместить код сравнения в метод класса.

Если это выглядит достаточно традиционным, то можно было бы избежать объявления отдельной функции и передать ее (ее исходный код) в качестве параметра методу `Construct` следующим образом:

```
procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  ListDate.Sort (TComparer<TDate>.Construct (
    function (const Left, Right: TDate): Integer
    var
      LDate, RDate: TDateTime;
    begin
      LDate := EncodeDate(Left.Year, Left.Month, Left.Day);
      RDate := EncodeDate(Right.Year, Right.Month, Right.Day);
      if LDate = RDate then
        Result := 0
      else if LDate < RDate then
        Result := -1
      else
        Result := 1;
    end));
end;
```

Этот пример должен был подогреть ваш аппетит к тому, чтобы узнать больше об анонимных методах! Конечно, последнюю версию намного проще написать, чем оригинальное сравнение, описанное в предыдущем разделе, хотя для многих разработчиков Object Pascal наличие производного класса может выглядеть чище и проще для понимания (унаследованная версия лучше разделяет логику, облегчая

потенциальное повторное использование кода, но во многих случаях вы все равно не воспользуетесь им).

Контейнеры для объектов

Помимо generic классов, рассмотренных в начале данного раздела, существуют также четыре унаследованных generic класса, которые являются производными от базовых классов, определенных в модуле Generics.Collections, имитирующих существующие классы модуля `Contnrs`:

```
type
  TObjectList<T: class> = class(TList<T>)
  TObjectQueue<T: class> = class(TQueue<T>)
  TObjectStack<T: class> = class(TStack<T>)
```

По сравнению с их базовыми классами, есть два ключевых отличия. Первое заключается в том, что эти generic типы могут использоваться только для объектов; второе заключается в том, что они определяют настраиваемый метод Notification, который в случае удаления объекта из списка (помимо опционального вызова обработчика событий `OnNotify`), `освобождает` объект.

Другими словами, класс `TObjectList<T>` ведет себя как его обычный аналог, когда установлено свойство `OwnsObjects`. Если вам интересно, почему это больше не является опцией, учтите, что `TList<T>` теперь `можно` использовать непосредственно для работы с типами объектов, в отличие от его не генерического аналога.

Существует также четвертый класс, опять же, называемый `TObjectDictionary<TKey, TValue>`, который определяется по-другому, так как он может владеть ключевым объектом, объектами-значениями, или обоими из них. Смотрите набор `TDictionaryOwnerships` и конструктор класса для более подробной информации.

Использование Generic словаря

Из всех предопределенных классов generic контейнеров, вероятно, стоит подробнее изучить generic словарь, `TObjectDictionary<TKey, Tvalue>`.

примечание Словарь в данном случае означает коллекцию элементов, каждый из которых имеет (уникальное) ключевое значение, относящееся к нему. (Он также известен как ассоциативный массив.) В классическом словаре слова выступают в качестве ключей для их определения, но в терминах программирования ключ не обязательно должен быть строкой (даже если это довольно частый случай).

Другие классы не менее важны, но, похоже, их легче использовать и понимать. В качестве примера использования словаря я написал приложение, которое извлекает данные из таблицы базы данных, создает объект для каждой записи и использует составной индекс с идентификатором клиента и описанием в качестве ключа. Причина такого разделения заключается в том, что подобная архитектура может быть легко использована для создания прокси, в котором ключ занимает место облегченной версии реального объекта*, загруженного* из базы данных.

Это два класса, которые используются в прикладном проекте CustomerDictionary для ключа и фактического значения. Первый имеет только два соответствующих поля соответствующей таблицы БД, а второй - полную структуру данных (я опустил приватные поля, методы геттера и сеттера):

```
type
  TCustomerKey = class
  private
    ...
  published
    property CustNo: Double
      read FCustNo write SetCustNo;
    property Company: string
      read FCompany write SetCompany;
  end;

  TCustomer = class
  private
```

```
    ..
    procedure Init;
    procedure EnforceInit;
  public
    constructor Create (aCustKey: TCustomerKey);
    property CustKey: TCustomerKey
      read FCustKey write SetCustKey;
  published
    property CustNo: Double
      read GetCustNo write SetCustNo;
    property Company: string
      read GetCompany write SetCompany;
    property Addr1: string
      read GetAddr1 write SetAddr1;
    property City: string
      read GetCity write SetCity;
    property State: string
      read GetState write SetState;
    property Zip: string
      read GetZip write SetZip;
    property Country: string
      read GetCountry write SetCountry;
    property Phone: string
      read GetPhone write SetPhone;
    property FAX: string
      read GetFAX write SetFAX;
    property Contact: string
      read GetContact write SetContact;
  class var
    RefDataSet: TDataSet;
  end;
```

В то время как первый класс очень прост (каждый объект инициализируется при создании), класс `TCustomer` использует ленивую модель инициализации (или *прокси*) и сохраняет ссылки на исходную базу данных, разделяемую (`класс var`) всеми объектами. При создании объекта ему присваивается ссылка на соответствующий `TCustomerKey`, а поле данных класса ссылается на исходный набор данных. В каждом методе-геттере класс перед возвращением данных проверяет, действительно ли был инициализирован объект, как в следующем случае:

```
function TCustomer.GetCompany: string;
begin
  EnforceInit;
  Result := FCompany;
end;
```

Метод `EnforceInit` проверяет локальный флаг, в конце концов вызывая `Init` для загрузки данных из базы данных в объект in-memory:

```
procedure TCustomer.EnforceInit;
begin
  if not FInitDone then
    Init;
end;

procedure TCustomer.Init;
begin
  RefDataSet.Locate('CustNo', CustKey.CustNo, []);

  // could also load each published field via RTTI
  FCustNo := RefDataSet.FieldByName ('CustNo').AsFloat;
  FCompany := RefDataSet.FieldByName ('Company').AsString;
  FCountry := RefDataSet.FieldByName ('Country').AsString;
  ...
  FInitDone := True;
end;
```

Учитывая эти два класса, я добавил в приложение словарь специального назначения. Этот специальный словарь наследует от общего класса, инстанцированного соответствующими типами, и добавляет к нему специальный метод:

```
type
  TCustomerDictionary = class (
    TObjectDictionary <TCustomerKey, TCustomer>)
  public
    procedure LoadFromDataSet (Dataset: TDataSet);
  end;
```

Метод загрузки заполняет словарь, копируя данные в память только для ключевых объектов:

```
procedure TCustomerDictionary.LoadFromDataSet(Dataset: TDataSet);
var
  CustKey: TCustomerKey;
begin
  TCustomer.RefDataSet := dataset;
  Dataset.First;
  while not Dataset.EOF do
  begin
    CustKey := TCustomerKey.Create;
    CustKey.CustNo := Dataset ['CustNo'];
    CustKey.Company := Dataset ['Company'];
    self.Add(custKey, TCustomer.Create (CustKey));
```

```
    Dataset.Next;
  end;
end;
```

Демонстрационная программа имеет основную форму и модуль данных, в котором размещен компонент ClientDataSet. Основная форма имеет элемент управления ListView, который заполняется при нажатии пользователем единственной кнопки.

примечание Возможно, вы захотите заменить компонент ClientDataSet на реальный набор данных, значительно расширив пример с точки зрения полезности, так как можно выполнить один запрос на ключи и отдельный запрос на актуальные данные каждого отдельного объекта `TCustomer`. У меня есть похожий код, но его добавление здесь отвлекло бы нас слишком сильно от цели примера, который экспериментирует с generic классом словаря.

После загрузки данных в словарь метод `BtnPopulateClick` использует перечислитель по ключам словаря:

```
procedure TFormCustomerDictionary.BtnPopulateClick(
  Sender: TObject);
var
  Custkey: TCustomerKey;
  ListItem: TListItem;
begin
  DataModule1.ClientDataSet1.Active := True;
  CustDict.LoadFromDataSet(DataModule1.ClientDataSet1);
  for Custkey in CustDict.Keys do
  begin
    ListItem := ListView1.Items.Add;
    ListItem.Caption := Custkey.Company;
    ListItem.SubItems.Add(FloatTOStr (Custkey.CustNo));
    ListItem.Data := Custkey;
  end;
end;
```

Это заполняет первые две колонки элемента управления ListView данными, доступными в ключевых объектах. Всякий раз, когда пользователь выбирает элемент элемента управления ListView, программа заполняет третью колонку:

```
procedure TFormCustomerDictionary.ListView1SelectItem(
  Sender: TObject; Item: TListItem; Selected: Boolean);
var
  ACustomer: TCustomer;
begin
  ACustomer := CustDict.Items [Item.data];
```

```
  Item.SubItems.Add(
    IfThen (
      ACustomer.State <> '',
      ACustomer.State + ', ' + ACustomer.Country,
    ACustomer.Country));
end;
```

Метод, описанный выше, получает объект, привязанный к заданному ключу, и использует его данные. При первом использовании конкретного объекта, метод доступа к свойствам скрыто запускает загрузку всех данных для объекта `TCustomer`.

Словари или списки строк?

На протяжении многих лет многие разработчики Object Pascal, в том числе и я, злоупотребляли классом TStringList. Его можно использовать не только для составления простого списка строк и списка пар имя/значение, но и для создания списка объектов, связанных со строками, и поиска этих объектов. С внедрением дженериков, гораздо лучше использовать их вместо того, чтобы использовать любимый инструмент в качестве швейцарского ножа.

Конкретные и сфокусированные контейнерные классы являются гораздо лучшим вариантом. Например, generic TDictionary со строковым ключом и значением-объектом будет, как правило, лучше, чем список строк по двум параметрам: более чистый и безопасный код, так как в нем будет меньше приведений типов, и более быстрое выполнение, учитывая, что в словарях используются хэш-таблицы.

Для демонстрации этих различий я написал довольно простое приложение под названием StringListVsDictionary. Его основная форма содержит два одинаковых списка, объявленных как:

```
  private
```

```
    FList: TStringList;
    FDict: TDictionary<string,TMyObject>;
```

Два списка заполнены случайными, но идентичными записями с циклом, выполняющим этот код:

```
    FList.AddObject (AName, AnObject);
    FDict.Add (AName, AnObject);
```

Две кнопки извлекают каждый элемент списка и выполняют поиск по имени для каждого из них. Оба метода сканируют список строк на наличие значений, но первый находит объекты в списке строк, а второй использует словарь. Обратите внимание, что в первом случае для возврата заданного типа нужен оператор as, в то время как словарь уже привязан к этому классу. Вот основной цикл этих двух методов:

```
 TheTotal := 0;
  for I := 0 to sList.Count -1  do
  begin
    aName := FList[I];
    // now search for it
    anIndex := FList.IndexOf (AName);
    // get the object
    AnObject := FList.Objects [AnIndex] as TMyObject;
    Inc (TheTotal, AnObject.Value);
  end;

  TheTotal := 0;
  for I := 0 to FList.Count -1  do
  begin
    AName := FList[I];
    // get the object
    AnObject := FDict.Items [AName];
    Inc (TheTotal, AnObject.Value);
  end;
```

Я не хочу получать доступ к строкам по порядку, но выяснить, сколько времени занимает поиск в списке отсортированных строк (который выполняет двоичный поиск) по сравнению с хэшированными ключами словаря. Неудивительно, что словарь быстрее, вот цифры в миллисекундах для теста:

```
Total: 99493811
StringList: 2839
Total: 99493811
Dictionary: 686
```

Результат тот же, учитывая, что начальные значения были идентичны, но время совершенно другое, при этом словарю потребовалось только около *четверти времени* для миллиона записей.

Generic интерфейсы

В разделе "Сортировка TList<T>" вы могли заметить довольно странное использование предопределенного интерфейса, который имел generic объявление. Стоит подробно рассмотреть эту технику, так как она открывает значительные возможности.

Первый технический элемент, на который следует обратить внимание, это то, что определение generic интерфейса совершенно законно, как я сделал в проекте приложения GenericInterface:

```
type
  IGetValue<T> = interface
    function GetValue: T;
    procedure SetValue (Value: T);
  end;
```

примечание Это общая версия интерфейса `IGetValue` прикладного проекта IntfContraints, рассмотренная в предыдущем разделе "Ограничения интерфейсов" данной главы. В этом случае интерфейс имел Integer значение, теперь он имеет generic значение.

Обратите внимание, что в отличие от стандартного интерфейса, в случае generic интерфейса вам не нужно указывать GUID, который будет использоваться в качестве Interface ID (или IID). Компилятор сгенерирует IID для каждого экземпляра общего интерфейса, даже если он неявно объявлен. На самом деле, Вам не нужно создавать конкретный экземпляр generic интерфейса для его реализации, но Вы можете определить generic класс, реализующий generic интерфейс:

```
type
```

```
  TGetValue<T> = class (TInterfacedObject, IGetValue<T>)
  private
    FValue: T;
  public
    constructor Create (Value: T);
    destructor Destroy; override;
    function GetValue: T;
    procedure SetValue (Value: T);
  end;
```

В то время как конструктор присваивает объекту начальное значение, единственная цель деструктора - занести в журнал информацию о том, что объект был уничтожен. Мы можем создать экземпляр этого общего класса (таким образом, создав за кулисами специфический экземпляр типа интерфейса), написав его:

```
procedure TFormGenericInterface.BtnValueClick(
  Sender: TObject);
var
  AVal: TGetValue<string>;
begin
  AVal := TGetValue<string>.Create (Caption);
  try
    Show ('TGetValue value: ' + AVal.GetValue);
  finally
    AVal.Free;
  end;
end;
```

Альтернативный подход, как мы видели в прошлом для проекта IntfConstraint, заключается в использовании переменной интерфейса соответствующего типа, что делает определение конкретного типа интерфейса явным (и неявным, как в предыдущем фрагменте кода):

```
procedure TFormGenericInterface.BtnIValueClick(
  Sender: TObject);
var
  AVal: IGetValue<string>;
begin
  AVal := TGetValue<string>.Create (Caption);
  Show ('IGetValue value: ' + AVal.GetValue);
  // freed automatically, as it is reference counted
end;
```

Конечно, мы также можем определить конкретный класс, реализующий generic интерфейс, как в следующем сценарии (из прикладного проекта GenericInterface):

```
type
  TButtonValue = class (TButton, IGetValue<Integer>)
  public
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    class function MakeTButtonValue (Owner: TComponent;
      Parent: TWinControl): TButtonValue;
  end;
```

Обратите внимание, что в то время как generic класс `TGetValue<T>` реализует generic интерфейс `IGetValue<T>`, cпецифический класс `TButtonValue` реализует специфический интерфейс `IGetValue<Integer>`. В частности, как и в предыдущем примере, интерфейс возвращается в свойство `Left` элемента управления:

```
function TButtonValue.GetValue: Integer;
begin
  Result := Left;
end;
```

В приведенном выше классе функция класса `MakeTButtonValue` является готовым методом создания объекта класса. Этот метод используется третьей кнопкой основного вида, как показано ниже:

```
procedure TFormGenericInterface.BtnValueButtonClick(
  Sender: TObject);
var
  IVal: IGetValue<Integer>;
begin
  IVal := TButtonValue.MakeTButtonValue (self, ScrollBox1);
  Show ('Button value: ' + IntToStr (IVal.GetValue));
end;
```

Хотя это и не имеет никакого отношения к generic классам, здесь представлена реализация функции класса `MakeTButtonValue`:

```
class function TButtonValue.MakeTButtonValue(
  Owner: TComponent; Parent: TWinControl): TButtonValue;
begin
  Result := TButtonValue.Create(Owner);
  Result.Parent := Parent;
  Result.SetBounds(Random (Parent.Width),
    Random (Parent.Height), Result.Width, Result.Height);
  Result.Text := 'Btnv';
```

```
end;
```

Предопределенные Generic интерфейсы

Теперь, когда мы изучили, как определить generic интерфейсы и совместить их с использованием generic и специфических классов, мы можем вернуться для второго взгляда на модуль Generics.Defaults. Этот блок определяет два общих сравнительных интерфейса:

- IComparer<T> имеет метод `Compare`.
- IEqualityComparer<T> имеет методы Equals и `GetHashCode`.

Эти классы реализованы некоторыми generic и специфическими классами, перечисленными ниже (без подробной информации о реализации):

```
type
  TComparer<T> = class(TInterfacedObject, IComparer<T>)
  TEqualityComparer<T> = class(
    TInterfacedObject, IEqualityComparer<T>)
  TCustomComparer<T> = class(TSingletonImplementation,
    IComparer<T>, IEqualityComparer<T>)
  TStringComparer = class(TCustomComparer<string>)
```

В приведенном выше листинге видно, что базовым классом, используемым в generic реализациях интерфейсов, является либо классический класс `TInterfacedObject`, со счетчиком ссылок, либо новый класс TSingletonImplementation. Это странно названный класс, который обеспечивает базовую реализацию `IInterface` без подсчета ссылок.

примечание Термин singleton ("одиночка") обычно используется для определения класса, из которого можно создать только один экземпляр, а не один без учета ссылок. Я считаю это довольно неправильным.

Как мы уже видели в разделе "Сортировка TList<T>" ранее в этой главе, эти классы сравнения используются generic

контейнерами. Однако, чтобы усложнить ситуацию, модуль `Generics.Default` довольно сильно полагается на анонимные методы, так что вы, вероятно, должны посмотреть на него только после прочтения следующей главы.

Умные указатели в Object Pascal

При обращении к дженерикам может сложиться неправильное первое впечатление, что эта языковая конструкция в основном используется для коллекций. Хотя это самый простой случай использования общих классов, и очень часто это первый пример в книгах и документах, дженерики полезны далеко за пределами коллекционных (или контейнерных) классов. В последнем примере этой главы я покажу вам *не коллекционный* обобщенный тип, то есть определение умного указателя.

Если вы пришли из мира Object Pascal, вы, возможно, не слышали об умных указателях, идея, которая пришла из языка C++. В C++ вы можете иметь указатели на объекты, для которых вы должны управлять памятью напрямую и вручную, и локальные переменные объектов, которые управляются автоматически, но имеют много других ограничений (включая отсутствие полиморфизма). Идея умного указателя заключается в использовании локально управляемого объекта, чтобы позаботиться о времени жизни указателя на реальный объект, который вы хотите использовать. Если это звучит слишком сложно, то я надеюсь, что версия на Object Pascal (и её код) поможет прояснить это.

примечание Термин полиморфизм в ООП-языках используется для обозначения ситуации, когда к переменной базового класса присваивается объект производного класса и вызывается один из виртуальных методов базового класса, потенциально заканчивающийся вызовом версии виртуального метода конкретного подкласса.

Использование записей для умных указателей

В Object Pascal объекты управляются по ссылке, в то время как записи имеют время жизни, привязанное к методу, в котором они объявлены. По окончании работы метода область памяти для записи очищается. Таким образом, мы можем использовать запись для управления временем жизни объекта Object Pascal.

Однако до версии Delphi 10.4 в Object Pascal-записях не было возможности выполнить пользовательский код во время уничтожения, и эта функция была введена с управляемыми записями. Механизм в старом стиле вместо этого заключался в использовании в записях поля интерфейса, так как это поле интерфейса управляется, а объект, используемый для реализации интерфейса, имеет уменьшенное количество ссылок.

Другое соображение - хотим ли мы использовать стандартную запись или общую. Со стандартной записью, имеющей поле типа TObject, вы можете удалять этот объект, когда это необходимо, так что этого достаточно в общих чертах. Однако, имея generic версию, вы можете получить два преимущества:

- умный generic указатель может вернуть ссылку на объект, который он содержит, так что вам не нужно держать обе ссылки
- умный generic указатель может автоматически создать объект-контейнер, используя конструктор без параметров.

Здесь я расскажу только о двух примерах умных указателей, реализованных с использованием generic записей, даже если это добавляет немного дополнительной сложности. Отправной точкой будет generic запись с ограничением на объекты, например:

```
type
  TSmartPointer<T: class> = record
  strict private
    FValue: T;
    function GetValue: T;
  public
    constructor Create(AValue: T);
    property Value: T read GetValue;
  end;
```

Методы `Create` и `GetValue` записи просто присваивают и считывают значение. Сценарий использования - следующий фрагмент кода, который создает объект, создает умный указатель, обертывающий его, а также позволяет использовать умный указатель для обращения к встроенному объекту и вызова его методов (см. последнюю строку кода ниже):

```
var
  SL: TStringList;
begin
  SL := TStringList.Create;
  var SmartP: TsmartPointer<TStringList>.Create (SL);
  SL.Add('foo');
  SmartP.Value.Add ('bar');
```

Как вы могли догадаться, этот код вызывает утечку памяти точно так же, как и без умного указателя! На самом деле запись уничтожается по мере того, как она выходит из-под контроля, но не освобождает внутренний объект.

Реализация умных указателей с управляемой generic записью

Хотя наиболее актуальной операцией умной записи указателя является ее finalization, в коде ниже, как вы видите, я добавляю

еще и оператор инициализации для установки ссылки на объект в `nil`. В идеале, мы бы предотвратили любую операцию присваивания (так как наличие нескольких ссылок на внутренние объекты потребовало бы настройки довольно сложного механизма подсчета ссылок), но учитывая, что это невозможно, я добавил оператор и реализовал его для поднятия исключения в случае его срабатывания.

Это полный код generic управляемой записи:

```
type
  TSmartPointer<T: class, constructor> = record
  strict private
    FValue: T;
    function GetValue: T;
  public
    class operator Initialize(out ARec: TSmartPointer <T>);
    class operator Finalize(var ARec: TSmartPointer <T>);
    class operator Assign(var ADest: TSmartPointer <T>;
      const [ref] ASrc: TSmartPointer <T>);
    constructor Create (AValue: T);
    property Value: T read GetValue;
  end;
```

Обратите внимание, что кроме ограничения класса generic запись имеет еще и ограничение конструктора, так как я хочу иметь возможность создавать объекты с generic типом данных. Это происходит в том случае, если вызывается метод `GetValue`, а поле еще не инициализировано. Вот полный код всех методов:

```
constructor TSmartPointer<T>.Create(AValue: T);
begin
  FValue := AValue;
end;

class operator TsmartPointer<T>.Initialize(
  out ARec: TSmartPointer <T>);
begin
  ARec.FValue := nil;
end;

class operator TsmartPointer<T>.Finalize(
  var ARec: TSmartPointer<T>);
begin
  ARec.FValue.Free;
end;

class operator TsmartPointer<T>.Assign(
```

```
  var ADest: TSmartPointer <T>;
  const [ref] ASrc: TSmartPointer <T>);
begin
  raise Exception.Create(
    'Cannot copy or assign a TSmartPointer<T>');
end;

function TSmartPointer<T>.GetValue: T;
begin
  if not Assigned(FValue) then
    FValue := T.Create;
  Result := FValue;
end;
```

Данный код является частью проекта SmartPointersMR, который включает в себя также пример использования умного указателя. Первый очень напоминает пример кода, который мы рассмотрели несколько страниц назад:

```
procedure TFormSmartPointers.BtnSmartClick(Sender: TObject);
var
  SL: TStringList;
begin
  SL := TStringList.Create;
  var SmartP := TSmartPointer<TStringList>.Create (SL);
  SL.Add('foo');
  SmartP.Value.Add('bar');
  Log ('Count: ' + SL.Count.ToString);
end;
```

Однако, учитывая, что универсальный умный указатель имеет поддержку автоматического построения объекта указанного типа, вы также можете обойтись без явного указания переменной, ссылающейся на список строк, и кода для ее создания:

```
procedure TFormSmartPointers.BtnSmartShortClick(Sender: TObject);
var
  SmartP: TSmartPointer<TStringList>;
begin
  SmartP.Value.Add('foo');
  SmartP.Value.Add('bar');
  Log ('Count: ' + SmartP.Value.Count.ToString);
end;
```

В программе можно проверить, что все объекты действительно уничтожены и нет утечки памяти, установив в коде инициализации глобальный `ReportMemoryLeaksOnShutdown` в значение `True`. В качестве теста счетчика в программе имеется

кнопка, вызывающая утечку, которая перехватывается при завершении работы программы.

Реализация умного указателя с Generic записью и интерфейсом

Как я уже упоминал, до того, как Delphi 10.4 сделала доступными управляемые записи, возможным способом реализации интеллектуальных указателей было использование интерфейса, учитывая, что запись автоматически освобождает объект, на который ссылается поле интерфейса. Хотя этот подход сейчас менее интересен, он все же предлагает пару дополнительных возможностей, таких как неявные операторы преобразования. Учитывая, что это остается интересным, сложным примером, я решил оставить его, со слегка сокращенным описанием (это проект SmartPointers в исходнике).

Для реализации интеллектуального указателя с интерфейсом можно написать внутренний, поддерживающий класс, привязанный к интерфейсу, и использовать механизм подсчета ссылок интерфейса, чтобы определить, когда освободить объект. Внутренний класс выглядит следующим образом:

```
type
  TFreeTheValue = class (TInterfacedObject)
  private
    FObjectToFree: TObject;
  public
    constructor Create(AnObjectToFree: TObject);
    destructor Destroy; override;
  end;

constructor TFreeTheValue.Create(
  AnObjectToFree: TObject);
begin
  FObjectToFree := AnObjectToFree;
end;
```

```
destructor TFreeTheValue.Destroy;
begin
  FObjectToFree.Free;
  inherited;
end;
```

Я объявил это вложенным типом общего умного указателя. Все, что нам нужно сделать в generic типе смарт-указателя, чтобы включить эту возможность, это добавить ссылку на интерфейс и инициализировать его с помощью объекта `TFreeTheValue`, ссылающегося на содержащийся в нем объект:

```
type
  TSmartPointer<T: class> = record
  strict private
    FValue: T;
    FFreeTheValue: IInterface;
    function GetValue: T;
  public
    constructor Create(AValue: T); overload;
    property Value: T read GetValue;
  end;
```

Псевдо-конструктор:

```
constructor TSmartPointer<T>.Create(AValue: T);
begin
  FValue := AValue;
  FFreeTheValue := TFreeTheValue.Create(FValue);
end;
```

Теперь, имея такой код, мы можем написать следующий код в программе, не вызывая утечки памяти (опять же код похож на тот, что я привел изначально и использовал в версии с управляемой записью):

```
procedure TFormSmartPointers.BtnSmartClick(
  Sender: TObject);
var
  SL: TStringList;
  SmartP: TSmartPointer<TStringList>;
begin
  SL := TStringList.Create;
  SmartP.Create (SL);
  SL.Add('foo');
  Show ('Count: ' + IntToStr (SL .Count));
end;
```

В конце метода SmartP-запись утилизируется, что приводит к уничтожению ее внутреннего сопряженного объекта, освобождая объект `TStringList`.

примечание Код работает, даже если поднято исключение. На самом деле, при использовании управляемого типа компилятором везде добавляются неявные блоки `try-finally`, как в данном случае запись с полем интерфейса.

Добавление неявного преобразования

С помощью решения для управляемых записей нам нужно с некоторой долей осторожности избегать операций по копированию записей, так как это потребует добавления механизма ручного подсчета ссылок и усложнения структуры. Однако, учитывая, что это встроено в интерфейсное решение, мы можем использовать эту модель для добавления операторов преобразования, что может упростить инициализацию и создание структуры данных. В частности, я добавлю оператор преобразования Implicit для *присвоения* целевому объекту умного указателя:

```
class operator TSmartPointer<T>.
  Implicit(AValue: T): TSmartPointer<T>;
begin
  Result := TSmartPointer<T>.Create(AValue);
end;
```

С помощью этого кода (и используя поле `Value`) мы можем теперь написать более компактную версию кода, например:

```
var
  SmartP: TSmartPointer<TStringList>;
begin
  SmartP := TStringList.Create;
  SmartP.Value.Add('foo');
  Show ('Count: ' + IntToStr (SmartP.Value.Count));
```

В качестве альтернативы можно использовать переменную типа `TStringList` и использовать более сложный конструктор для

инициализации умной записи указателя даже без явной ссылки на нее:

```
var
  SL: TStringList;
begin
  SL := TSmartPointer<TStringList>.
    Create(TStringList.Create).Value;
  SL.Add('foo');
  Show ('Count: ' + IntToStr (SL .Count));
```

Пойдя по этому пути, мы также можем определить противоположное преобразование, и использовать приведенную нотацию, а не свойство `Value`:

```
class operator TSmartPointer<T>.
  Implicit(AValue: T): TSmartPointer<T>;
begin
  Result := TSmartPointer<T>.Create(AValue);
end;

var
  SmartP: TSmartPointer<TStringList>;
begin
  SmartP := TStringList.Create;
  TStringList(SmartP).Add('foo2');
```

Также вы можете заметить, что я всегда использовал псевдо-конструктор в приведенном выше коде, но в записи это не нужно. Все, что нам нужно - это способ инициализации внутреннего объекта, возможно, вызов его конструктора, при первом же его использовании.

Мы не можем выполнить проверку `Assigned на` внутренний объект, так как записи (в отличие от классов) не инициализируются до нуля. Однако мы можем выполнить этот тест на переменной интерфейса, которая инициализируется.

Сравнение решений на основе

интеллектуальных указателей

Версия управляемой записи умного указателя проще и достаточно эффективна, однако версия на основе интерфейса предлагает преимущество операторов преобразования. Оба они имеют свои достоинства, хотя лично я склонен отдавать предпочтение версии управляемой записи.

Для более четкого анализа и более сложных решений (вне рамок этой книги) я могу порекомендовать следующий пост в блоге Эрика ван Билсена:

```
https://blog.grijjy.com/2020/08/12/custom-managed-records-for-smart-pointers/
```

Ковариантные возвратные типы с дженериками

В общем случае в Object Pascal (и большинстве других статических объектно-ориентированных языков) метод может вернуть объект класса, но вы не можете переопределить его в производном классе, чтобы вернуть объект производного класса. Это довольно распространенная практика, называемая "Covariant Return Type" и явно поддерживаемая некоторыми языками, такими как C++.

Животных, собак и кошек.

В терминах кодирования, если `TDog` наследует от `TAnimal`, я бы хотел иметь методы:

```
function TAnimal.Get (AName: string): TAnimal;
```

```
function TDog.Get (AName: string): TDog;
```

Однако в Object Pascal нельзя использовать виртуальные функции с другим возвращаемым значением, а также нельзя перегружать тип возврата, но только при использовании разных параметров. Позвольте мне показать вам полный код простой демонстрации. Вот три класса:

```
type
  TAnimal = class
  private
    FName: string;
    procedure SetName(const Value: string);
  public
    property Name: string read FName write SetName;
  public
    class function Get (const AName: string): TAnimal;
    function ToString: string; override;
  end;

  TDog = class (TAnimal)

  end;

  TCat = class (TAnimal)

  end;
```

Реализация двух методов достаточно проста, если заметить, что функция класса на самом деле используется для создания новых объектов, внутреннего вызова конструктора. Причина, по которой я не хочу создавать конструктор напрямую, заключается в том, что это более общая техника, в которой метод класса может создавать объекты других классов (или иерархий классов). Вот код:

```
class function TAnimal.Get(const AName: string): TAnimal;
begin
  Result := Create;
  Result.FName := AName;
end;

function TAnimal.ToString: string;
begin
  Result := 'This ' + Copy (ClassName, 2, MaxInt) +
    ' is called ' + FName;
end;
```

Теперь мы можем использовать класс, написав следующий код, который мне ужасно не нравится, так как мы должны привести результат к нужному типу:

```
var
var
  ACat: TCat;
begin
  ACat := TCat.Get('Matisse') as TCat;
  Memo1.Lines.Add (ACat.ToString);
  ACat.Free;
```

Опять же, мне бы хотелось иметь возможность присвоить значение, возвращаемое `TCat.Get`, ссылке на класс `TCat` без явного приведения. Как это сделать?

Метод с Generic результатом

Оказывается, дженерики могут помочь нам решить проблему. Не типы generic (что является наиболее часто используемой формой дженериков). Но generic методы для обычных типов, рассмотренные ранее в этой главе. Что я могу добавить в класс `TAnimal`, так это метод с generic параметром *типа*, например:

```
  class function GetAs<T: class> (const AName: string): T;
```

Этот метод требует параметра типа generic, который должен быть классом (или типом экземпляра) и возвращает объект этого типа. Пример реализации приведен здесь:

```
class function TAnimal.GetAs<T>(const AName: string): T;
var
  Res: TAnimal;
begin
  Res := Get (aName);
  if res.inheritsFrom (T) then
    Result := T(Res)
  else
    Result := nil;
end;
```

Теперь мы можем создать экземпляр и использовать его, опустив преобразование *as*, хотя нам все равно придется передать тип в качестве параметра:

```
var
  ADog: TDog;
begin
  ADog := TDog.GetAs<TDog>('Pluto');
  Memo1.Lines.Add (ADog.ToString);
  ADog.Free;
```

Возвращение производного объекта другого класса

Когда вы возвращаете объект того же класса, вы можете заменить этот код на правильное использование конструкторов. Но использование дженериков для получения ковариантных типов возврата на самом деле более гибкое. Фактически, мы можем использовать его для возврата объектов другого класса, или иерархии классов:

```
type
  TAnimalShop = class
    class function GetAs<T: TAnimal, constructor> (
      const AName: string): T;
  end;
```

примечание Такой класс, используемый для создания объектов другого класса (или нескольких классов, в зависимости от параметров), обычно называется "*фабрикой классов*".

Теперь мы можем использовать специфическое ограничение класса (что-то невозможное в самом классе) и нам нужно указать ограничение конструктора, чтобы иметь возможность создать объект данного класса из generic метода:

```
class function TAnimalShop.GetAs<T>(const AName: string): T;
var
  Res: TAnimal;
begin
  Res := T.Create;
  Res.Name := AName;
  if res.inheritsFrom (T) then
    Result := T(Res)
```

```
  else
    Result := nil;
end;
```

Обратите внимание, что теперь в вызове нам не нужно повторять тип класса дважды:

```
ADog := TanimalShop.GetAs<TDog>('Pluto');
```

15: Анонимные методы

Язык Object Pascal включает в себя как типы процедур (то есть, типы, объявляющие указатели на процедуры и функции), так и указатели на методы (то есть, типы, объявляющие указатели на методы).

примечание В случае, если вам нужна дополнительная информация, процедурные типы были рассмотрены в Главе 4, в то время как события и типы указателей на методы были описаны в Главе 10.

Хотя вы можете редко использовать их напрямую, это ключевые особенности Object Pascal, с которыми работает каждый разработчик. Фактически, типы указателей на методы являются основой для обработчиков событий в компонентах и визуальных элементах управления: каждый раз, когда вы объявляете обработчик события, даже простую `Button1Click`, вы фактически объявляете метод, который будет связан с событием (в данном случае, с событием `OnClick`) с помощью указателя на метод.

Анонимные методы расширяют эту возможность, позволяя передавать фактический код метода в качестве параметра, а не имя метода, определенное в другом месте. Однако это не единственное различие. Что отличает анонимные методы от

других, это то, как они управляют временем жизни локальных переменных.

Вышеуказанное определение совпадает с функцией, называемой "closures" во многих других языках, например, в JavaScript. Если анонимные методы Object Pascal на самом деле являются замыканиями, то как получилось, что язык ссылается на них, используя другой термин? Причина заключается в том, что оба термина используются разными языками программирования и что компилятор Си++, выпускаемый Embarcadero, использует термин closures для того, что Object Pascal называет обработчиками событий.

Анонимные методы существуют в разных формах и с разными названиями в течение многих лет в довольно большом количестве языков программирования, в частности, динамических языков. У меня был большой опыт работы с замыканиями в JavaScript, особенно с библиотекой jQuery и AJAX вызовами. Соответствующая функция в C# называется анонимным делегатом.

Но здесь я не хочу выделять слишком много времени на сравнение замыканий и связанных с ними техник на различных языках программирования, а хочу подробно описать, как они работают в Object Pascal.

примечание С очень глобальной точки зрения, дженерики позволяют параметризовать код для типа, анонимные методы позволяют параметризовать код для метода.

Синтаксис и семантика

анонимных методов

Анонимный метод в Object Pascal представляет собой механизм *создания значения метода в контексте выражения*. Довольно загадочное, но довольно точное определение, если учесть, что оно подчеркивает ключевое отличие от указателей на метод, контекст выражения. Однако, прежде чем перейти к этому, позвольте мне начать с очень простого примера кода (включённого в проект AnonymFirst вместе с большинством других фрагментов кода в этом разделе).

Это объявление анонимного типа метода, которое вам нужно предоставить, учитывая, что Object Pascal является сильно типизированным языком:

```
type
  TIntProc = reference to procedure (N: Integer);
```

Это отличается от типа ссылки на метод только ключевыми словами, используемыми для объявления:

```
type
  TIntMethod = procedure (N: Integer) of object;
```

Переменная Анонимного метода

После того, как у вас есть тип анонимного метода, вы можете в простейших случаях объявить переменную этого типа, назначить анонимный метод, совместимый с типом, и вызвать метод через переменную:

```
procedure TFormAnonymFirst.BtnSimpleVarClick(Sender: TObject);
var
  AnIntProc: TIntProc;
begin
  AnIntProc :=
    procedure (N: Integer)
    begin
      Memo1.Lines.Add (IntToStr (N));
    end;
```

```
  AnIntProc (22);
end;
```

Обратите внимание на синтаксис, используемый для присвоения локальной переменной `AnIntProc` фактической процедуры с внутренним кодом.

Параметр анонимного метода

В качестве более интересного примера (с еще более удивительным синтаксисом) можно передать функции анонимный метод в качестве параметра. Предположим, что у вас есть функция, принимающая параметр анонимного метода:

```
procedure CallTwice (Value: Integer; AnIntProc: TIntProc);
begin
  AnIntProc (Value);
  Inc (Value);
  AnIntProc (Value);
end;
```

Функция вызывает метод, переданный в качестве параметра дважды с двумя последовательными целыми значениями, одно из которых передается в качестве параметра, и следующим. Вы вызываете функцию, передавая ей действительный анонимный метод, с прямым внутренним кодом, который выглядит удивительно:

```
procedure TFormAnonymFirst.BtnProcParamClick(Sender: TObject);
begin
  CallTwice (48,
    procedure (N: Integer)
    begin
      Show (IntToHex (N, 4));
    end);
  CallTwice (100,
    procedure (N: Integer)
    begin
      Show (FloatToStr(Sqrt(N)));
    end);
end;
```

С синтаксической точки зрения, процедура, передается в качестве параметра в круглых скобках и не завершается точкой

с запятой. Реальный эффект кода заключается в вызове `IntToHex` с 48 и 49 и `FloatToStr` с квадратным корнем 100 и 101, что приводит к следующему выводу:

```
0030
0031
10
10.0498756211209
```

Использование локальных переменных

Мы могли бы достичь того же эффекта, используя указатели метода, хотя и с другим и менее читаемым синтаксисом. Анонимный метод явно отличается тем, что он может ссылаться на локальные переменные вызывающего метода. Рассмотрим следующий код:

```
procedure TFormAnonymFirst.BtnLocalValClick(Sender: TObject);
var
  ANumber: Integer;
begin
  ANumber := 0;
  CallTwice (10,
    procedure (N: Integer)
    begin
      Inc (ANumber, N);
    end);
  Show (IntToStr (ANumber));
end;
```

Здесь метод, еще переданный в процедуру `CallTwice`, использует не только локальный параметр `N`, но и локальную переменную из контекста, из которого она была вызвана, `ANumber`. Какой эффект? Два вызова анонимного метода изменяют локальную переменную, добавляя к ней параметр 10 в первый раз и 11 во второй. Окончательное значение `ANumber` будет 21.

Продление срока службы локальных переменных

Предыдущий пример показывает интересный эффект, но с последовательностью вложенных вызовов функций тот факт, что вы можете использовать локальную переменную, не удивляет. Сила анонимных методов, однако, заключается в том, что они могут использовать локальную переменную, а также продлевать ее время жизни до тех пор, пока она не понадобится. Пример покажет это лучше, чем длительное объяснение.

примечание Чуть более подробно: технически, анонимные методы копируют переменные и параметры, которые они используют, в кучу, когда они создаются, и сохраняют их до тех пор, пока конкретный экземпляр анонимного метода не будет создан.

В класс формы `TFormAnonymFirst` прикладного проекта AnonymFirst я добавил (используя class completion) свойство типа указателя на анонимный метод (на самом деле, тот же самый тип указателя на анонимный метод, который я использовал во всем коде проекта):

```
 private
    FAnonMeth: TIntProc;
    procedure SetAnonMeth(const Value: TIntProc);
  public
    property AnonMeth: TIntProc
      read FAnonMeth write SetAnonMeth;
```

Потом я добавил еще две кнопки в форму программы. Первая сохраняет свойство анонимным методом, использующим локальную переменную (примерно, как в предыдущем методе BtnLocalValClick):

```
procedure TFormAnonymFirst.BtnStoreClick(Sender: TObject);
var
  ANumber: Integer;
begin
  ANumber := 3;
  AnonMeth :=
    procedure (N: Integer)
```

```
    begin
      Inc (ANumber, N);
      Show (IntToStr (ANumber));
    end;
end;
```

При выполнении этого метода анонимный метод не выполняется, а только сохраняется. Локальная переменная `ANumber` инициализируется в значение 3, не изменяется, выходит за пределы локальной области видимости (по мере завершения работы метода) и удаляется. По крайней мере, этого можно ожидать от стандартного кода Object Pascal.

Вторая кнопка, которую я добавил в форму для этого шага, вызывает анонимный метод, хранящийся в свойстве `AnonMeth`:

```
procedure TFormAnonymFirst.BtnCallClick(Sender: TObject);
begin
  if Assigned (AnonMeth) then
  begin
    CallTwice (2, AnonMeth);
  end;
end;
```

При выполнении этого кода вызывается анонимный метод, использующий локальную переменную `ANumber` метода, который больше не находится в стеке. Однако, поскольку анонимные методы *захватывают* контекст своего выполнения, переменная все еще находится там и может быть использована до тех пор, пока данный экземпляр анонимного метода (т.е. ссылка на метод) существует.

В качестве дополнительного доказательства сделайте следующее. Нажмите кнопку `Store` один раз, кнопку `Call` два раза, и вы увидите, что используется одна и та же захваченная переменная:

```
5
8
10
13
```

примечание Причина такой последовательности в том, что значение начинается с 3, каждый вызов `CallTwice` передавал свой параметр анонимным методам первый раз (т.е. 2), а затем второй раз после его инкремента (т.е. второй раз после 3).

Теперь снова нажмите `Store` и снова нажмите `Call`. Что происходит, почему сбрасывается значение локальной переменной? Присваивая новый анонимный экземпляр метода, старый анонимный метод удаляется (вместе с его собственным контекстом выполнения), а новый контекст выполнения перехватывается, включая новый экземпляр локальной переменной. Полная последовательность *Store - Call - Call - Store – Call* дает на выходе:

```
5
8
10
13
5
8
```

Именно последствия такого поведения, похожего на то, что делают некоторые другие языки, делают анонимные методы чрезвычайно мощной функцией языка, которую можно использовать для реализации чего-то, что просто не было возможно в прошлом.

Анонимные методы: за кулисами

Если функция захвата переменных является одной из наиболее актуальных для анонимных методов, есть еще несколько техник, на которые стоит обратить внимание, прежде чем мы сконцентрируемся на некоторых реальных примерах. *Тем не менее, если вы новичок в анонимных методах, вы можете*

пропустить этот довольно продвинутый раздел и вернуться к нему во время второго чтения.

(Потенциально) пропущенная скобка

Обратите внимание, что в приведенном выше коде я использовал символ `AnonMeth`, чтобы сослаться на анонимный метод, а не вызывать его. Для того, чтобы вызвать его, я должен был набрать:

```
AnonMeth (2)
```

Разница понятна, мне нужно передать соответствующий параметр для вызова метода. Немного больше путаницы с безпараметрическими анонимными методами. Если вы объявите:

```
type
  TAnyProc = reference to procedure;
var
  AnyProc: TAnyProc;
```

За обращением к `AnyProc` должны следовать пустые круглые скобки, иначе компилятор посчитает, что вы пытаетесь получить метод (его адрес), а не вызвать его:

```
AnyProc ();
```

Нечто подобное происходит при вызове функции, которая возвращает анонимный метод, как в следующем случае, взятом из обычного прикладного проекта AnonymFirst:

```
function GetShowMethod: TIntProc;
var
  X: Integer;
begin
  X := Random (100);
  ShowMessage ('New x is ' + IntToStr (X));
  Result :=
    procedure (N: Integer)
    begin
```

```
      X := X + N;
      ShowMessage (IntToStr (X));
    end;
end;
```

Теперь вопрос в том, как это вызвать? Если вы просто вызовете

```
 GetShowMethod;
```

Это скомпилируется и запустится, но все, что он делает, это вызывает код назначения анонимного метода, выбрасывая анонимный метод, возвращаемый функцией.

Как вызвать реальный анонимный метод, передавая ему параметр? Одним из вариантов является использование временной анонимной переменной метода:

```
var
  Ip: TIntProc;
begin
  Ip := GetShowMethod();
  Ip (3);
```

Обратите внимание в данном случае на скобки после вызова `GetShowMethod`. Если вы их опустите (стандартная практика Pascal), то получите следующую ошибку:

```
E2010 Incompatible types: 'TIntProc' and 'Procedure'
```

Без скобок компилятор думает, что нужно назначить саму функцию `GetShowMethod`, а не ее результат указателю на Ip-метод. Тем не менее, использование временной переменной в данном случае может быть не лучшим вариантом, так как делает код неестественно сложным. Простой вызов

```
  GetShowMethod(3);
```

не будет компилироваться, так как вы не можете передать параметр методу. К первому вызову необходимо добавить пустую скобку, а к результирующему анонимному методу - параметр Integer. Как ни странно, но можно написать:

```
 GetShowMethod()(3);
```

Реализация анонимных методов

Что происходит за кулисами при реализации анонимных методов? Реальный код, генерируемый компилятором для анонимных методов, основан на интерфейсах, с единственным (скрытым) методом вызова, называемым `Invoke`, плюс обычная поддержка подсчета ссылок (это полезно для определения времени жизни анонимных методов и контекста, который они захватывают).

Описание внутренних деталей, вероятно, очень сложно и имеет ограниченную ценность. Достаточно сказать, что реализация очень эффективна, с точки зрения скорости, и требует около 500 дополнительных байт для каждого анонимного метода.

Другими словами, ссылка на метод в Object Pascal реализуется с помощью *специального* однометодного интерфейса, при этом метод, сгенерированный компилятором, имеет ту же сигнатуру, что и ссылка на метод, который он реализует. Интерфейс использует подсчет ссылок для их автоматического удаления.

примечание Хотя практически интерфейс, используемый для анонимного метода, выглядит как любой другой интерфейс, компилятор различает эти *специальные* интерфейсы, поэтому их нельзя смешивать в коде.

Кроме этого скрытого интерфейса, для каждого вызова анонимного метода компилятор создает скрытый объект, который имеет реализацию метода и данные, необходимые для *захвата* контекста вызова. Таким образом, для каждого вызова метода создается новый набор перехватываемых переменных.

Готовые к использованию

ссылочные типы

Каждый раз при использовании анонимного метода в качестве параметра необходимо задавать соответствующий тип данных ссылочного указателя. Чтобы избежать распространения локальных типов, Object Pascal предоставляет ряд готовых к использованию типов ссылочных указателей в модуле System.SysUtils. Как видно из приведенного ниже фрагмента кода, в большинстве этих определений типов используются параметризованные типы, так что при одном типовом объявлении для каждого возможного типа данных существует свой тип ссылочного указателя:

```
type
  TProc = reference to procedure;
  TProc<T> = reference to procedure (Arg1: T);
  TProc<T1,T2> = reference to procedure (
    Arg1: T1; Arg2: T2);
  TProc<T1,T2,T3> = reference to procedure (
    Arg1: T1; Arg2: T2; Arg3: T3);
  TProc<T1,T2,T3,T4> = reference to procedure (
    Arg1: T1; Arg2: T2; Arg3: T3; Arg4: T4);
```

Используя эти декларации, вы можете определить процедуры, которые принимают параметры анонимного метода, как показано ниже:

```
procedure UseCode (Proc: TProc);
function DoThis (Proc: TProc): string;
function DoThat (ProcInt: TProc<Integer>): string;
```

В первом и втором случае вы передаете анонимный метод без параметров, в третьем - метод с одним параметром Integer:

```
UseCode (
  procedure
  begin
    ...
  end);
StrRes := DoThat (
  procedure (I: Integer)
  begin
    ...
  end);
```

Аналогично `System.SysUtils` определяет набор анонимных типов методов с возвращаемым значением типа generic:

```
type
  TFunc<TResult> = reference to function: TResult;
  TFunc<T,TResult> = reference to function (
    Arg1: T): TResult;
  TFunc<T1,T2,TResult> = reference to function (
    Arg1: T1; Arg2: T2): TResult;
  TFunc<T1,T2,T3,TResult> = reference to function (
    Arg1: T1; Arg2: T2; Arg3: T3): TResult;
  TFunc<T1,T2,T3,T4,TResult> = reference to function (
    Arg1: T1; Arg2: T2; Arg3: T3; Arg4: T4): TResult;
  TPredicate<T> = reference to function (
    Arg1: T): Boolean;
```

Эти определения очень широкие, так как можно использовать бесчисленное множество комбинаций типов данных для использования до четырех параметров и разных типов результата. Последнее определение очень похоже на второе, но соответствует конкретному случаю, который очень часто встречается, функции, принимающей общий параметр и возвращающей булевский результат.

Анонимные методы в реальном мире

На первый взгляд, нелегко в полной мере понять силу анонимных методов и сценарии, в которых можно получить выигрыш от их использования. Поэтому вместо того, чтобы приводить более запутанные примеры, описывающие язык, я решил сосредоточиться на тех из них, которые оказывают практическое воздействие и дают отправные точки для дальнейшего изучения.

Анонимные обработчики событий

Одной из отличительных особенностей Object Pascal является реализация обработчиков событий с использованием указателей на методы. Анонимные методы могут быть использованы для прикрепления нового поведения к событию без объявления отдельного метода и захвата контекста выполнения метода. Это позволяет избежать необходимости добавлять дополнительные поля в форму для передачи параметров от одного метода к другому.

В качестве примера (проект приложения AnonButton) я добавил *анонимное* событие *click* к кнопке, объявив правильный тип указателя метода и добавив новый обработчик события в пользовательский класс кнопки (определяемый с помощью класса-интерпостера):

```
type
  TAnonNotif = reference to procedure (Sender: TObject);

  // interposer class
  TButton = class (FMX.StdCtrls.TButton)
  private
    FAnonClick: TAnonNotif;
    procedure SetAnonClick(const Value: TAnonNotif);
  public
    procedure Click; override;
  public
    property AnonClick: TAnonNotif
      read FAnonClick write SetAnonClick;
  end;
```

примечание *interposer* Класс - это производный класс, имеющий то же имя, что и его базовый класс. Наличие двух классов с одним и тем же именем возможно, так как два класса находятся в разных юнитах, поэтому их полное имя (*unitname.classname*) различно. Объявление класса-интерпостера может быть удобным, так как вы можете просто поместить элемент управления Button на форму и прикрепить к нему дополнительное поведение, не устанавливая новый компонент в IDE и заменяя элементы управления на форме на элемент управления нового типа. Единственный трюк, который вам нужно запомнить, это то, что если определение класса-интерполятора находится в отдельном юните (а не в юните формы, как в этом простом примере), то этот юнит должен быть перечислен в операторе uses после юнита, определяющего базовый класс.

Код этого класса достаточно прост, так как метод Setter сохраняет новый указатель, а метод `Click` вызывает его перед выполнением стандартной обработки (т.е. вызовом обработчика события OnClick, если он доступен):

```
procedure TButton.SetAnonClick(const Value: TAnonNotif);
begin
  FAnonClick := Value;
end;

procedure TButton.Click;
begin
  if Assigned (FAnonClick) then
    FAnonClick (self)
  inherited;
end;
```

Как вы можете использовать этот новый обработчик событий? В принципе, вы можете назначить ему анонимный метод:

```
procedure TFormAnonButton.BtnAssignClick(Sender: TObject);
begin
  BtnInvoke.AnonClick :=
    procedure (Sender: TObject)
    begin
      Show ((Sender as TButton).Text);
    end;
end;
```

Сейчас это выглядит довольно бессмысленно, так как такого же эффекта можно легко добиться с помощью стандартного метода обработчика событий. Вместо этого начинает действовать следующий метод, так как анонимный метод перехватывает ссылку на компонент, назначивший обработчик события, со ссылкой на параметр Sender.

Это можно сделать после временного присвоения локальной переменной, так как параметр `Sender` анонимного метода скрывает параметр `Sender` метода `BtnKeepRefClick`:

```
procedure TFormAnonButton.BtnKeepRefClick(Sender: TObject);
var
  ACompRef: TComponent;
begin
  ACompRef := Sender as TComponent;
  BtnInvoke.AnonClick :=
    procedure (Sender: TObject)
    begin
```

```
      Show ((Sender as TButton).Text +
        ' assigned by ' + ACompRef.Name);
    end;
end;
```

При нажатии кнопки `BtnInvoke` вы увидите ее текст вместе с именем компонента, назначившего обработчик анонимного метода.

Анонимные методы работы со временем

Разработчики часто добавляют код работы со временем к существующим подпрограммам для сравнения их относительной скорости. Предположим, что у вас есть два фрагмента кода и вы хотите сравнить их скорость, выполнив их несколько миллионов раз. Для этого вы можете написать следующее (взятое из проекта приложения LargeString из Главы 6):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Str1, Str2: string;
  I: Integer;
  T1: TStopwatch;
begin
  Str1 := 'Marco ';
  Str2 := 'Cantu ';

  T1 := TStopwatch.StartNew;
  for I := 1 to MaxLoop do
    Str1 := Str1 + Str2;

  T1.Stop;
  Show('Length: ' + Str1.Length.ToString);
  Show('Concatenation: ' + T1.ElapsedMilliseconds.ToString);
end;
```

Второй метод имеет аналогичный код, но использует класс `TStringBuilder`, а не простое конкатенирование. Теперь мы можем воспользоваться анонимными методами для создания временного шаблона и передать конкретный код в качестве

параметра, как это было сделано в обновленной версии кода, в проекте приложения AnonLargeStrings.

Вместо того, чтобы повторять тайминговый код снова и снова, можно написать функцию с тайминговым кодом, которая будет вызывать фрагмент кода без параметров анонимным методом:

```
function TimeCode (NLoops: Integer; Proc: TProc): string;
var
  T1: TStopwatch;
  I: Integer;
begin
  T1 := TStopwatch.StartNew;
  for I := 1 to NLoops do
    Proc;
  T1.Stop;
  Result := T1.ElapsedMilliseconds.toString;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  Str1, Str2: string;
begin
  Str1 := 'Marco ';
  Str2 := 'Cantu ';
  Show ('Concatenation: ' +
    TimeCode (MaxLoop,
      procedure ()
      begin
        Str1 := Str1 + Str2;
      end));
  Show('Length: ' + Str1.Length.ToString);
end;
```

Обратите внимание, что если вы выполните стандартную версию и версию, основанную на анонимных методах, вы получите немного другой результат, то анонимная версия метода будет медленнее примерно на 10%. Причина в том, что вместо непосредственного выполнения локального кода программа должна сделать виртуальный вызов реализации анонимного метода. Так как эта разница последовательна, тестовый код в любом случае имеет полный смысл.

Однако, если вам нужно выжать производительность из вашего кода, использование анонимных методов не будет таким быстрым, как непосредственное написание кода, с

использованием прямой функции. Использование невиртуального указателя на метод, вероятно, будет где-то между ними с точки зрения производительности.

Синхронизация потоков

В многопоточных приложениях, требующих обновления пользовательского интерфейса, доступ к свойствам визуальных компонентов (или в объектах памяти), работающих в основном потоке, невозможен без механизма синхронизации. Библиотеки визуальных компонентов Delphi по своей природе не являются потокобезопасными (как в большинстве библиотек пользовательского интерфейса). Доступ двух потоков к объекту одновременно может нарушить его состояние.

Классическое решение, предлагаемое классом TThread в Object Pascal, заключается в вызове специального метода Synchronize, передавая в качестве параметра ссылку на другой метод, который должен быть безопасно выполнен. Этот второй метод не может иметь параметров, поэтому общепринятой практикой является добавление в класс потока дополнительных полей для передачи информации от одного метода к другому.

В качестве практического примера в книге Mastering Delphi 2005 я написал приложение WebFind (программа, которая выполняет поиск в Google по HTTP и извлекает результирующие ссылки из HTML страницы), со следующим классом потоков:

```
type
  TFindWebThread = class(TThread)
  protected
    FAddr, FText, FStatus: string;
    procedure Execute; override;
    procedure AddToList;
    procedure ShowStatus;
    procedure GrabHtml;
    procedure HtmlToList;
```

```
    procedure HttpWork (Sender: TObject;
      AWorkMode: TWorkMode; AWorkCount: Int64);
  public
    FStrUrl: string;
    FStrRead: string;
  end;
```

Три защищенных строковых поля и некоторые дополнительные методы были введены для поддержки синхронизации с пользовательским интерфейсом. Например, обработчик события `HttpWork` подключился к событию `OnReceiveData` внутреннего компонента `THTTPClient` (компонент клиентской библиотеки HTTP Delphi), который имел следующий код, называемый методом `ShowStatus`:

```
procedure TFindWebThread.HttpWork(const Sender: TObject;
  AcontentLength, AReadCount: Int64; var AAbort: Boolean);
begin
  FStatus := 'Received ' + IntToStr (AReadCount) + ' for ' + FStrUrl;
  Synchronize (ShowStatus);
end;

procedure TFindWebThread.ShowStatus;
begin
  Form1.StatusBar1.SimpleText := FStatus;
end;
```

Метод `Synchronize` Object Pascal RTL имеет два различных перегруженных определения:

```
type
  TThreadMethod = procedure of object;
  TThreadProcedure = reference to procedure;

  TThread = class
    ...
    procedure Synchronize(AMethod: TThreadMethod); overload;
    procedure Synchronize(AThreadProc: TThreadProcedure); overload;
```

По этой причине мы можем удалить текстовое поле `FStatus` и функцию `ShowStatus`, а также переписать обработчик событий `HttpWork`, используя новую версию `Synchronize` и анонимный метод:

```
procedure TFindWebThreadAnon.HttpWork(const Sender: TObject;
  AcontentLength, AReadCount: Int64; var AAbort: Boolean);
begin
  Synchronize (
    procedure
```

```
      begin
        Form1.StatusBar1.SimpleText :=
          'Received ' + IntToStr (AReadCount) + ' for ' + FStrUrl;
      end);
end;
```

Используя один и тот же подход во всем коде класса, класс нитей становится следующим (оба класса нитей можно найти в версии прикладного проекта WebFind, поставляемой вместе с исходным кодом данной книги):

```
type
  TFindWebThreadAnon = class(TThread)
  protected
    procedure Execute; override;
    procedure GrabHtml;
    procedure HtmlToList;
    procedure HttpWork (const Sender: TObject; AContentLength: Int64;
      AReadCount: Int64; var AAbort: Boolean);
  public
    FStrUrl: string;
    FStrRead: string;
  end;
```

Использование анонимных методов упрощает код, необходимый для синхронизации потоков, так как можно избежать временных полей.

примечание Анонимные методы имеют много отношений с потоками, потому что поток используется для выполнения некоторого кода, а анонимный метод представляет собой код. Поэтому их использование поддерживается не только в классе `TThread`, но и в библиотеке параллельного программирования (в `TParallel.For` и для определения `TTask`). Многопоточность выходит за рамки данной главы, поэтому я не буду добавлять больше примеров в этом направлении. Тем не менее, в следующем примере я буду использовать еще один поток, так как это чаще всего является требованием при выполнении HTTP вызова.

AJAX в Object Pascal

Последний пример в этом разделе, демонстрация приложения AnonAjax, является одним из моих любимых примеров анонимных методов (пусть даже немного экстремальных). Причина в том, что я научился использовать замыкание (или

анонимные методы) в JavaScript, во время написания AJAX-приложений с библиотекой jQuery несколько лет назад.

примечание Сокращение AJAX означает Asynchronous JavaScript XML, так как изначально это был формат, используемый при вызове веб-сервисов из браузера. По мере того, как эта технология становилась все более популярной и широко распространенной, а веб-сервисы перешли на архитектуру REST и формат JSON, термин AJAX немного сдал, в пользу REST. Я все равно решил оставить это старое название для примера, учитывая, что оно объясняет назначение приложения, а также историю, стоящую за ним. Подробнее вы можете прочитать по адресу: `https://en.wikipedia.org/wiki/Ajax_(программирование)`.

Глобальная функция `AjaxCall` порождает поток, передавая ему анонимный метод для выполнения по завершению. Функция представляет собой просто обертку вокруг конструктора потока:

```
type
  TAjaxCallback = reference to procedure (
    ResponseContent: TStringStream);

procedure AjaxCall (const StrUrl: string;
  AjaxCallback: TAjaxCallback);
begin
  TAjaxThread.Create (StrUrl, AjaxCallback);
end;
```

Весь код находится в классе `TAjaxThread`, классе потока с внутренним компонентом HTTP-клиента (из HTTP-библиотеки Delphi Client), используемого для асинхронного доступа к заданному URL:

```
вводить
type
  TAjaxThread = class (TThread)
  private
    FHttp: THTTPClient;
    FURL: string;
    FAjaxCallback: TAjaxCallback;
  protected
    procedure Execute; override;
  public
    constructor Create (const StrUrl: string;
      AjaxCallback: TAjaxCallback);
    destructor Destroy; override;
  end;
```

Конструктор производит некоторую инициализацию, копируя свои параметры в соответствующие локальные поля класса потока и создавая объект `FHttp`. Реальное ядро класса находится в его методе `Execute`, который выполняет HTTP-запрос, сохраняя результат в потоке, который затем сбрасывается и передается в функцию обратного вызова - анонимный метод:

```
procedure TAjaxThread.Execute;
var
  AResponseContent: TStringStream;
begin
  AResponseContent := TStringStream.Create;
  try
    FHttp.Get (FURL, AResponseContent);
    AResponseContent.Position := 0;
    FAjaxCallback (AResponseContent);
  finally
    AResponseContent.Free;
  end;
end;
```

В качестве примера его использования можно указать, что в прикладном проекте AnonAjax есть кнопка для копирования содержимого веб-страницы в элемент управления Memo (после добавления запрашиваемого URL):

```
procedure TFormAnonAjax.BtnReadClick(Sender: TObject);
begin
  AjaxCall (EdUrl.Text,
    procedure (AResponseContent: TStringStream)
    begin
      Memo1.Lines.Text := AResponseContent.DataString;
      Memo1.Lines.Insert (0, 'From URL: ' + EdUrl.Text);
    end);
end;
```

После завершения HTTP-запроса вы можете выполнить любую обработку. В качестве примера можно привести извлечение ссылок из HTML-файла (таким образом, чтобы это было похоже на приложение WebFind, рассмотренное ранее). Опять же, чтобы сделать эту функцию гибкой, в качестве параметра используется анонимный метод для каждой ссылки:

```
type
  TLinkCallback = reference to procedure (const StrLink: string);

procedure ExtractLinks (StrData: string; ProcLink: TLinkCallback);
```

```
var
  strAddr: string;
  NBegin, NEnd: Integer;
begin
  StrData := LowerCase (StrData);
  NBegin := 1;
  repeat
    nBegin := PosEx ('href="http', StrData, nBegin);
    if NBegin <> 0 then
    begin
      // find the end of the HTTP reference
      NBegin := NBegin + 6;
      NEnd := PosEx ('"', StrData, NBegin);
      StrAddr := Copy (StrData, NBegin, NEnd - NBegin);
      // move on
      NBegin := NEnd + 1;
      // execute anon method
      ProcLink (StrAddr)
    end;
  until NBegin = 0;
end;
```

Если применить эту функцию к результату вызова AJAX и предоставить дальнейший метод обработки, то в конечном итоге вы получите два вложенных анонимных вызова метода, как во второй кнопке прикладного проекта AnonAjax:

```
procedure TFormAnonAjax.BtnLinksClick(Sender: TObject);
begin
  AjaxCall (EdUrl.Text,
    procedure (AResponseContent: TStringStream)
    begin
      ExtractLinks(AResponseContent.DataString,
        procedure (const AUrl: string)
        begin
          Memo1.Lines.Add (AUrl + ' in ' + EdUrl.Text);
        end);
    end);
end;
```

В этом случае элемент управления Memo получит набор ссылок, а не HTML возвращаемой страницы. Вариант вышеописанной процедуры извлечения ссылок будет представлять собой процедуру извлечения картинки. Функция `ExtractImages` захватывает источник (`src`) img-тегов возвращаемого HTML-файла и вызывает другой анонимный метод, совместимый с `TLinkCallback` (подробнее о функции см. в исходном коде).

Теперь можно представить себе открытие HTML-страницы (с помощью функции `AjaxCall`), извлечение ссылок на изображение и повторное использование `AjaxCall` для захвата реальных изображений. Это означает использование тройного вложенного замыкания, в структуре кодирования, которую некоторые Object Pascal программисты могут найти очень необычной, но без сомнения, очень мощной и выразительной:

```
procedure TFormAnonAjax.BtnImagesClick(Sender: TObject);
var
  NHit: Integer;
begin
  NHit := 0;
  AjaxCall (EdUrl.Text,
    procedure (AResponseContent: TStringStream)
    begin
      ExtractImages(AResponseContent.DataString,
        procedure (const AUrl: string)
        begin
          Inc (NHit);
          Memo1.Lines.Add (IntToStr (NHit) + '.' +
            AUrl + ' in ' + EdUrl.Text);
          if nHit = 1 then // load the first
          begin
            var RealURL := IfThen (AURL[1]='/',
              EdUrl.Text + AURL, AURL); // expand URL
            AjaxCall (RealUrl,
              procedure (AResponseContent: TStringStream)
              begin
                Image1.Picture.Graphic.
                  LoadFromStream (AResponseContent);
              end);
          end;
        end);
    end);
end;
```

примечание Этот фрагмент кода был темой моей записи в блоге "Anonymous, Anonymous, Anonymous" от сентября 2008 года, которая, как вы видите, вызвала некоторые комментарии:
http://blog.marcocantu.com/blog/anonymous_3.html.

Помимо того, что графика работает только в том случае, если вы загружаете файл того же формата, что и в компоненте Image, код и его результат впечатляют. Обратите внимание, в частности, на нумерационную последовательность, основанную на захвате локальной переменной `NHit`. Что произойдет, если

дважды нажать на кнопку в быстрой последовательности? Каждый из анонимных методов получит свою копию счетчика `NHit`, и потенциально они могут быть выведены из списка, при этом второй поток начнет выдавать свой результат до первого. Результат использования этой последней кнопки показан на рисунке 15.1.

Рисунок 15.1: Вывод тройного вложенного анонимного вызова метода для получения изображения с веб-страницы.

16: Отражение и атрибуты

Традиционно компиляторы языков с сильным, статическим типом, таких как Pascal, предоставляли мало или вообще не предоставляли информации о доступных типах во время исполнения. Вся информация о типах данных была видна только на этапе компиляции.

Первая версия Object Pascal нарушила эту традицию, предоставив информацию о времени выполнения для свойств и других членов класса, помеченных специальной директивой компилятора, published. Эта возможность была включена для классов, скомпилированных со специфической настройкой {$M+} и является основой механизма потоковой передачи данных для DFM-файлов VCL (и FMX-файлов библиотеки FireMonkey), а также способа работы с формой и другими визуальными дизайнерами. Когда она впервые появилась в Delphi 1, эта возможность была совершенно новой идеей, которую позже переняли другие средства разработки и расширили ее несколькими способами.

Во-первых, были расширения системы типов (доступные только в Object Pascal) для учета обнаружения методов и динамического вызова в COM. Это до сих пор поддерживается в

Object Pascal dispatch ID, применением методов к вариантам и другими COM-функциями. В конце концов, поддержка COM в Object Pascal была расширена собственным вариантом информации о типах времени выполнения, но эта тема выходит далеко за рамки книги про язык.

Появление управляемых сред, таких как Java и .NET, привело к появлению формы очень обширной информации о типе времени выполнения, с подробным RTTI, привязанным компилятором к исполняемым модулям и доступным для обнаружения программами, использующими эти модули. Это имеет недостаток, связанный с открытием некоторых внутренних частей программы и увеличением размеров модулей, но это приводит к появлению новых моделей программирования, которые сочетают в себе некоторую гибкость динамических языков с прочной структурой и скоростью работы сильнотипных.

Нравится вам это или нет (и это действительно было предметом интенсивных дебатов в то время, когда эта функция была введена) Объект Pascal медленно движется в том же направлении, и принятие обширной формы RTTI знаменует собой очень значительный шаг в этом направлении. Как мы увидим, вы можете отказаться от RTTI, но если вы этого не сделаете, вы сможете использовать некоторые дополнительные возможности в своих приложениях.

Тема далеко не простая, поэтому я буду действовать по шагам. Сначала мы остановимся на новом расширенном RTTI, встроенном в компилятор, и новых классах модуля `Rtti`, которые можно использовать для его изучения. Во-вторых, познакомимся с новой структурой `TValue` и динамическим вызовом. В-третьих, я представлю пользовательские атрибуты, функцию, которая параллельна аналогу в .NET, и позволяет расширить информацию RTTI, генерируемую компилятором.

Только в последней части главы я попытаюсь вернуться к причинам, лежащим в основе расширенного RTTI и посмотреть на практические примеры его использования.

Расширенная RTTI

Компилятор Object Pascal по умолчанию генерирует большое количество расширенной информации RTTI. Эта информация о времени выполнения включает в себя все типы, в том числе классы и все другие определяемые пользователем типы, а также основные типы данных, предопределенные компилятором, и охватывает опубликованные поля, а также публичные, даже защищенные и приватные элементы. Это необходимо для того, чтобы иметь возможность углубиться во внутреннюю структуру любого объекта.

Первый пример

Прежде чем мы рассмотрим информацию, генерируемую компилятором, и различные техники доступа к ним, позвольте мне перепрыгнуть к выводам и показать, что можно сделать с помощью RTTI. Конкретный пример весьма минимален и мог бы быть написан со старым RTTI, но он должен дать вам представление о том, о чем я говорю (также учитывая, что не все разработчики Object Pascal явно использовали традиционную RTTI).

Предположим, у вас есть форма с кнопкой, как в проекте приложения RttiIntro. Вы можете написать следующий код, чтобы прочитать значение свойства `Text` элемента управления:

```
uses
```

```
  Rtti;

procedure TFormRttiIntro.BtnInfoClick(Sender: TObject);
var
  Context: TRttiContext;
begin
  Show (Context.
    GetType(TButton).
    GetProperty('Text').
    GetValue(Sender).ToString);
end;
```

Код использует запись `TRttiContext` для ссылки на информацию о типе `TButton`, от информации этого типа до данных RTTI о свойстве, и эти данные свойства используются для ссылки на фактическое значение свойства, которое преобразуется в строку. Если вам интересно, как это работает, продолжайте читать. Я хочу сказать, что теперь этот подход можно использовать не только для динамического доступа к свойству, но и для чтения значений полей, в том числе приватных.

Мы также можем изменить значение свойства, как показывает вторая кнопка проекта приложения RttiIntro:

```
procedure TFormRttiIntro.BtnChangeClick(Sender: TObject);
var
  Context: TRttiContext;
  AProp: TRttiProperty;
begin
  AProp := Context.GetType(TButton).GetProperty('Text');
  AProp.SetValue(BtnChange,
    StringOfChar ('*', Random (10) + 1));
end;
```

Этот код заменяет `Текст` случайным числом `*`. Отличие от приведенного кода в том, что он имеет временную локальную переменную, ссылающуюся на информацию RTTI для свойства. Теперь, когда у вас есть представление о том, чем мы занимаемся, давайте начнем с проверки расширенной информации RTTI, генерируемой компилятором.

Генерируемая компилятором

информация

Вам ничего не нужно делать, чтобы позволить компилятору добавить эту дополнительную информацию в вашу исполняемую программу (независимо от ее вида: приложение, библиотека, пакет...). Просто откройте проект и скомпилируйте его. По умолчанию компилятор генерирует Extended RTTI для всех полей (в том числе private), а также для публичных и опубликованных методов и свойств. Вы можете быть удивлены тем, что получаете RTTI информацию для приватных полей, но это необходимо для динамических операций, таких как сериализация двоичных объектов и трассировка объектов на куче.

Вы можете управлять расширенной генерацией RTTI в соответствии с матрицей настроек: на одной оси вы имеете видимость, а на другой - вид элемента. В следующей таблице показана система по умолчанию:

	Field	Method	Property
Private	x		
Protected	x		
Public	x	x	x
Published	x	x	x

Технически, четыре настройки видимости обозначаются с помощью следующего установленного типа, объявленного в модуле `System`:

```
type
  TVisibilityClasses = set of (vcPrivate,
    vcProtected, vcPublic, vcPublished);
```

Есть несколько готовых к использованию постоянных значений для этого набора, указывающих на настройки видимости RTTI по умолчанию, применяемые к `TObject` и унаследованные всеми другими классами по умолчанию:

```
const
  DefaultMethodRttiVisibility = [vcPublic, vcPublished];
```

```
DefaultFieldRttiVisibility = [vcPrivate..vcPublished];
DefaultPropertyRttiVisibility = [vcPublic, vcPublished];
```

Информация, генерируемая компилятором, управляется новой директивой $RTTI, которая имеет статус, указывающий, является ли установка только для данного типа или также для его потомков (`EXPLICIT` или `INHERITED`), за которым следуют три спецификатора, задающие видимость для методов, полей и свойств. По умолчанию применяется в модуле `System` :

```
{$RTTI INHERIT
  METHODS(DefaultMethodRttiVisibility)
  FIELDS(DefaultFieldRttiVisibility)
  PROPERTIES(DefaultPropertyRttiVisibility)}
```

Чтобы полностью отключить генерацию расширенного RTTI для всех членов ваших классов, можно воспользоваться следующей директивой:

```
{$RTTI EXPLICIT METHODS([]) FIELDS([]) PROPERTIES([])}
```

примечание Нельзя размещать директиву RTTI перед объявлением модуля, как это происходит с другими директивами компилятора, так как она зависит от настроек, определенных в `System` модуле. При этом вы получите внутреннее сообщение об ошибке, которое не очень интуитивно понятно. В любом случае, просто переместите его ниже объявления модуля.

При использовании этой настройки учитывайте, что она будет применяться только к вашему коду, и полное удаление невозможно, так как информация о RTTI для RTL и других библиотечных классах уже скомпилирована в соответствующих DCU и пакетах. Также следует помнить, что директива $RTTI не вносит никаких изменений в традиционный RTTI, генерируемый для опубликованных типов: Она все равно генерируется независимо от директивы `$RTTI`.

примечание Классы обработки RTTI, доступные в `System.Rtti` и рассматриваемые в следующем разделе, подключаются к традиционной RTTI и ее структуре `PTypeInfo`.

С помощью этой директивы вы можете остановить генерацию Extended RTTI для ваших собственных классов. На

противоположном конце шкалы вы также можете увеличить количество генерируемой RTTI, включая частные и защищенные методы и свойства, если хотите (хотя это не имеет большого смысла).

Очевидный эффект от добавления расширенной информации RTTI в исполняемый файл заключается в том, что файл будет становиться больше (основной недостаток большого файла – размер для распространения, так как дополнительное время загрузки и объем памяти не так важны). Теперь вы можете удалить RTTI из модулей вашей программы, и это может иметь положительный эффект... если вы решите, что не хотите использовать RTTI в своем коде. RTTI - это мощная техника, как вы увидите в этой главе, и в большинстве случаев она стоит дополнительного размера исполняемого файла.

Слабые и сильные связи типов

Что еще можно сделать, чтобы уменьшить размер программы? На самом деле есть кое-что, что вы можете сделать. Даже если эффект не будет большим, он будет заметен.

При оценке RTTI информации, доступной в исполняемом файле, учитывайте, что то, что компилятор добавляет, компоновщик может удалить. По умолчанию классы и методы, не скомпилированные в программе, не получат Extended RTTI (что было бы совершенно бесполезно), так как они не получают и основного RTTI. На противоположном конце шкалы, если вы хотите, чтобы все Extended RTTI было включено и работало, то вам необходимо компоновать даже те классы и методы, на которые в вашем коде нет явных ссылок.

Есть две директивы компилятора, которые вы можете использовать для управления информацией, связанной с

исполняемым файлом. Первая, которая полностью документирована, это директива $WeakLinkRTTI. При ее включении, для типов, не используемых в программе, из конечного исполняемого файла будет удален как сам тип, так и его RTTI информация.

В качестве альтернативы можно принудительно включить все типы и их Расширенный RTTI с помощью директивы $StrongLinkTypes. Влияние их на многие программы весьма драматично, с почти двукратным увеличением размера программы.

Модуль RTTI

Если генерация расширенного RTTI для всех типов является первым столпом для рефлексии в Object Pascal, то вторым столпом является возможность легкой и высокоуровневой навигации по этой информации, благодаря модулю System.Rtti. Третий столп, как мы увидим позже, это поддержка пользовательских атрибутов. Но позвольте мне делать по одному шагу за раз.

Традиционно приложения Object Pascal могли (и могут) использовать функции модуля System.TypInfo для доступа к "опубликованной" информации о типе времени выполнения. Этот модуль определяет несколько низкоуровневых структур и функций данных (все они основаны на указателях и записях) с парой процедур более высокого уровня, чтобы сделать работу немного проще.

Напротив, модуль `Rtti` позволяет очень легко работать с расширенной RTTI, предоставляя набор классов с подходящими методами и свойствами. Точкой входа для

доступа к различным объектам является структура записи TRttiContext, которая имеет четыре метода для поиска доступных типов:

```
function GetType (ATypeInfo: Pointer): TRttiType; overload;
function GetType (AClass: TClass): TRttiType; overload;
function GetTypes: TArray<TRttiType>;
function FindType (const AQualifiedName: string): TRttiType;
```

Как видите, можно передать класс, указатель `PTypeInfo`, полученный от типа, квалифицированное имя (имя типа, украшенного именем модуля, как в *"System. TObject"*), или получить весь список типов, определенных как массив типов RTTI, а точнее как `TArray<TRttiType>`.

Этот последний вызов я и использовал в следующем листинге, упрощенная версия кода из проекте приложения TypesList:

```
procedure TFormTypesList.BtnTypesListClick(Sender: TObject);
var
  AContext: TRttiContext;
  TheTypes: TArray<TRttiType>;
  AType: TRttiType;
begin
  TheTypes := AContext.GetTypes;
  for AType in TheTypes do
    if AType.IsInstance then
      Show(AType.QualifiedName);
end;
```

Метод `GetTypes` возвращает полный список типов данных, но программа фильтрует только типы, представляющие классы. В модуле представлено около десятка других классов, представляющих типы.

примечание Юнит Rtti относится к типам классов как к "экземплярам" или "типам экземпляров" (как в `TRttiInstanceType`). Это немного сбивает с толку, так как мы обычно используем термины *instance* для обозначения реального объекта.

Отдельные объекты в списке типов — это классы, которые наследуются от базового класса TRttiType. В частности, мы можем искать тип класса `TRttiInstanceType`, переписывая код выше, как в следующем модифицированном фрагменте:

```
  for AType in TheTypes do
```

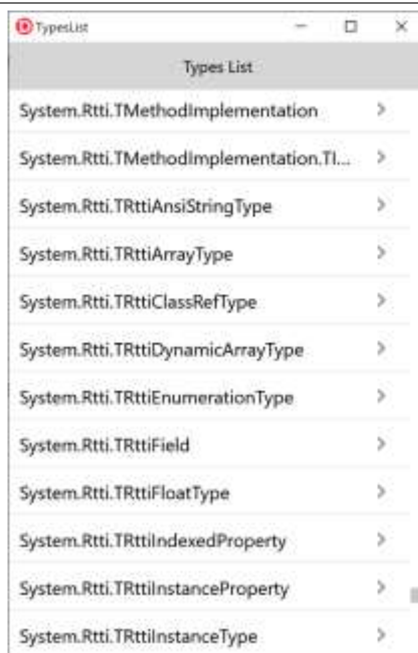
```
    if AType is TRttiInstanceType then
      Show(AType.QualifiedName);
```

Фактический код демо немного сложнее, так как он сначала заполняет список строк, сортирует элементы, а затем заполняет элемент управления ListView, используя `BeginUdpate` и `EndUdpate` для оптимизации (и попробуйте, наконец, заблокировать их, чтобы убедиться, что конечная операция всегда выполняется):

```
var
  AContext: TRttiContext;
  TheTypes: TArray<TRttiType>;
  SList: TStringList;
  AType: TRttiType;
  STypeName: string;
begin
  ListView1.ClearItems;
  SList := TStringList.Create;
  try
    TheTypes := AContext.GetTypes;
    for AType in TheTypes do
      if AType.IsInstance then
        SList.Add(AType.QualifiedName);
    SList.Sort;
    ListView1.BeginUpdate;
    try
      for STypeName in SList do
        (ListView1.Items.Add).Text := STypeName;
    finally
      ListView1.EndUpdate;
    end;
  finally
    SList.Free;
  end;
end;
```

Данный код выдает достаточно длинный список с сотнями типов данных, фактическое количество которых зависит от платформы и версии компилятора, как показано на рисунке 16.1. Обратите внимание, что в изображении перечислены типы из блока RTTI, рассмотренного в следующем разделе.

Рисунок 16.1: Вывод проекта приложения TypesList



Классы RTTI в модуле Rtti

В следующем списке можно увидеть весь граф наследования для классов, которые происходят из абстрактного класса `TRttiObject` и определены в модуле `System.Rtti`:

```
TRttiObject (abstract)
  TRttiNamedObject
    TRttiType
      TRttiStructuredType (abstract)
        TRttiRecordType
        TRttiInstanceType
        TRttiInterfaceType
      TRttiOrdinalType
        TRttiEnumerationType
      TRttiInt64Type
      TRttiInvokableType
        TRttiMethodType
        TRttiProcedureType
      TRttiClassRefType
      TRttiEnumerationType
      TRttiSetType
```

```
    TRttiStringType
      TRttiAnsiStringType
    TRttiFloatType
    TRttiArrayType
    TRttiDynamicArrayType
    TRttiPointerType
  TRttiMember
    TRttiField
    TRttiProperty
      TRttiInstanceProperty
    TRttiIndexedProperty
    TRttiMethod
  TRttiParameter
  TRttiPackage
TRttiManagedField
```

Каждый из этих классов предоставляет конкретную информацию о данном типе. В качестве примера, только в `TRttiInterfaceType` предлагается способ доступа к GUID интерфейса.

примечание В первой реализации модуля `Rtti` не было объекта RTTI для доступа к индексируемым свойствам (например, `Strings[]` из `TStringList'a`). Позже он был добавлен и теперь доступен, что делает информацию о типе исполнения действительно полной.

Жизненный цикл объектов RTTI и запись TRttiContext

Если вы посмотрите на исходные тексты метода `BtnTypesListClick`, приведенного ранее, то увидите что-то, что выглядит совершенно неправильно. Вызов `GetTypes` возвращает массив типов, но код не освобождает эти внутренние объекты.

Причина в том, что структура записи `TRttiContext` становится эффективной владельцем для всех создаваемых объектов RTTI. При утилизации записи (т.е. когда она выходит за пределы области действия) внутренний интерфейс очищается, вызывая собственный деструктор, который освобождает все объекты RTTI, созданные с его помощью.

Запись `TRttiContext` на самом деле играет двойную роль. С одной стороны, она контролирует время жизни объектов RTTI (как я только что объяснил), с другой стороны, она кэширует информацию RTTI, которую довольно дорого воссоздать с помощью поиска. Поэтому вам, возможно, захочется сохранить ссылку на запись `TRttiContext в` течение длительного периода времени, что позволит вам продолжать доступ к принадлежащим ему объектам RTTI без необходимости их воссоздания (опять же, это затратная операция).

Внутри записи `TRttiContext` используется глобальный пул типа `TRttiPool`, который использует критическую секцию для обеспечения безопасности потока доступа.

примечание Существуют исключения из потокобезопасного механизма пулинга RTTI, подробно описанные в комментариях, доступных в самом модуле `Rtti`.

Таким образом, если быть более точным, пул RTTI совместно используется с записями `TRttiContext`, поэтому объединенные объекты RTTI хранятся рядом, в то время как хотя бы одна запись `TRttiContext` находится в памяти. Цитирую комментарий в модуле:

{... работа с объектами RTTI без хотя бы одного живого контекста – это ошибка. Сохранение хотя бы одного контекста должно поддерживать переменную Pool в актуальном состоянии}.

Другими словами, необходимо избегать кэширования и хранения объектов RTTI после освобождения контекста RTTI. Это пример, который приводит к нарушению доступа к памяти (опять же, часть проекта приложения TypesList):

```
function GetThisType (AClass: TClass): TRttiType;
var
  AContext: TRttiContext;
begin
  Result := AContext.GetType(AClass);
end;

procedure TFormTypesList.Button1Click(Sender: TObject);
var
  AType: TRttiType;
```

```
begin
  AType := GetThisType (TForm);
  Show (AType.QualifiedName);
end;
```

Подводя итог, можно сказать, что объекты RTTI управляются по контексту, и их не следует держать рядом. Контекст, в свою очередь, является записью, поэтому он утилизируется автоматически. Вы можете увидеть код, который использует `TRttiContext` следующим образом:

```
AContext := TRttiContext.Create;
try
  // use the context
finally
  AContext.Free;
end;
```

Псевдо-конструктор и псевдо-деструктор устанавливают внутренний интерфейс, который управляет фактическими структурами данных, используемыми за кулисами, в `nil` для очистки механизма объединения. Однако, так как эта операция автоматическая для локального типа, такого как запись, то в этом нет необходимости, если только где-нибудь вы не ссылаетесь на контекстную запись с помощью указателя.

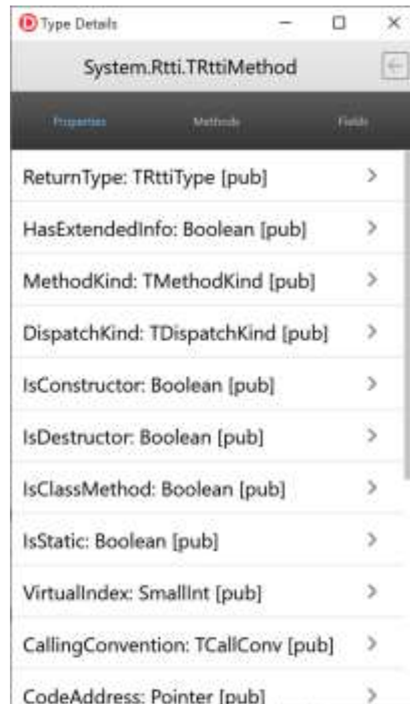
Отображение информации о классе

Наиболее релевантными типами, которые вы, возможно, захотите проверить во время выполнения, являются так называемые структурированные типы, то есть экземпляры, интерфейсы и записи. Сосредоточившись на экземплярах, мы можем обратиться к отношениям между классами, следуя информации `BaseType`, доступной, например, для типов.

Доступ к типам, безусловно, является интересной отправной точкой, но что является актуальным и особенно новым, так это

возможность узнать о дальнейших деталях этих типов, в том числе и об их членах. При нажатии на один из типов (здесь класс компонента `TPopup`) программа отображает список свойств, методов и полей этого типа на трех страницах элемента управления вкладкой, как показано на рисунке 16.2.

Рисунок 16.2: Подробная информация о типе, отображаемая в прикладном проекте TypesList для класса `TRttiMethod`



В модуле этой вторичной формы, которая, вероятно, может быть адаптирована и расширена для использования в качестве браузера общего типа в других приложениях, есть метод под названием `ShowTypeInformation`, который проходит через каждое свойство, метод и поле данного типа, добавляя их в три отдельных поля списка с указанием их видимости (*pri* для частного, *pro* для защищенного, *pub* для общественного, *pbl* для опубликованного, что возвращается простым оператором case в функции `VisibilityToken`):

```
var
```

```
  AContext: TRttiContext;
  AType: TRttiType;
  AProperty: TRttiProperty;
  AMethod: TRttiMethod;
  AField: TRttiField;
begin
  AType := AContext.FindType(TypeName);
  if not Assigned(AType) then
    Exit;

  LabelType.Text := AType.QualifiedName;
  for AProperty in Atype.GetProperties do
    FormTypeInfo.LVProperties.Items.Add.Text := AProperty.Name +
      ': ' + AProperty.PropertyType.Name + ' ' +
      VisibilityToken (AProperty.Visibility);
  for AMethod in Atype.GetMethods do
    LVMethods.Items.Add.Text := AMethod.Name + ' ' +
      VisibilityToken (AMethod.Visibility);
  for AField in AType.GetFields do
    LVFields.Items.Add.Text := AField.Name + ': ' +
      AField.FieldType.Name + ' ' +
      VisibilityToken (AField.Visibility);
end;
```

Вы можете продолжить и извлечь дополнительную информацию из типов этих свойств, получить списки параметров методов и проверить тип возврата, и многое другое. Здесь я не хочу строить полный RTTI браузер, а только дать вам представление о том, чего можно достичь.

RTTI для пакетов

Помимо методов, которые можно использовать для доступа к типу или списку типов, запись TRttiContext имеет еще один очень интересный метод - GetPackages, который возвращает список исполняемых пакетов, используемых текущим приложением. Если вы выполняете этот метод в приложении, скомпилированном без пакетов времени исполнения, все, что вы получите – это сам исполняемый файл. Но если вы выполняете его в приложении, скомпилированном с пакетами времени исполнения, вы получите список этих пакетов. С этого момента вы можете изучить типы, доступные для каждого из

пакетов. Обратите внимание, что в этом случае список типов намного длиннее, так как RTL и визуальные типы библиотек, не используемые приложением, не удаляются умным компоновщиком.

Если вы используете пакеты времени выполнения, вы также можете получить список типов для каждого из пакетов (и самого исполняемого файла), используя код:

```
var
  AContext: TRttiContext;
  APackage: TRttiPackage;
  AType: TRttiType;
begin
  for APackage in AContext.GetPackages do
  begin
    ListBox1.Items.Add('PACKAGE  ' + APackage.Name);
    for AType in APackage.GetTypes do
      if AType.IsInstance then
      begin
        ListBox1.Items.Add('   - ' + AType.QualifiedName);
      end;
  end;
```

примечание Пакеты в Object Pascal можно использовать для добавления компонентов в среду разработки, как мы видели в Главе 11. Однако, пакеты также могут использоваться во время выполнения, устанавливая основной исполняемый файл с несколькими исполняемыми пакетами, а не один, более крупный исполняемый файл. Если вы знакомы с разработкой Windows, то пакеты имеют роль, похожую на DLL (а технически это и есть DLL), а точнее, на .NET сборки. Хотя пакеты играют очень важную роль в Windows, в настоящее время они не поддерживаются на мобильных платформах (также из-за ограничений на установку приложений в операционных системах, например, в iOS).

Структура TValue

Новая расширенная RTTI позволяет не только просматривать внутреннюю структуру программы, но и предоставляет конкретную информацию, включая свойства и значения полей. В то время как устройство `TypInfo` предоставило функцию `GetPropValue` для доступа к generic свойству и получения типа

варианта со значением, новый модуль `Rtti` использует другую структуру для хранения нетипизированного элемента - записи `TValue`.

Эта запись может хранить практически любой возможный тип данных Object Pascal и делает это, отслеживая оригинальное представление данных, сохраняя как данные, так и их тип. Что она может сделать, так это прочитать и записать данные в заданном формате. Если вы записываете Integer число в `TValue`, то вы можете читать из него только Integer число. Если Вы пишете строку, Вы можете прочитать ее обратно.

Что он не может сделать, так это конвертировать из одного формата в другой. Поэтому, даже если `TValue` имеет метод `AsString` и `AsInteger`, первый можно использовать только в том случае, если данные действительно представляют собой строку, второй - только в том случае, если вы изначально присвоили ему целое число. Например, в этом случае можно использовать метод `AsInteger`, и при вызове метода `IsOrdinal` он вернет True:

```
var
  V1: TValue;
begin
  V1 := 100;
  if V1.IsOrdinal then
    Log (IntToStr (V1.AsInteger));
```

Однако вы не можете использовать метод `AsString,` который вызовет исключение недействительного приведения типов:

```
var
  V1: TValue;
begin
  V1 := 100;
  Log (V1.AsString);
```

Если же вам нужно строковое представление, то вы можете использовать метод `ToString`, который имеет большое `case` представление, пытаясь приспособиться к большинству типов данных:

```
var
  V1: TValue;
```

```
begin
  V1 := 100;
  Log (V1.ToString);
```

Вероятно, вы сможете лучше понять, прочитав слова Барри Келли (Barry Kelly), бывшего члена команды исследователей Embarcadero, работавшего над RTTI:

> *TValue - это тип, используемый для сопоставления значений методам, основанным на RTTI, и от них, а также для чтения и записи полей и свойств.*
> *Это немного похоже на Variant, но гораздо более приспособлено к системе типов Object Pascal; например, экземпляры могут храниться как напрямую, так и в виде наборов, ссылок на классы и т.д. Также он более строго типизирован и не делает (например) молчаливых приведений строк к числам.*

Теперь, когда вы лучше понимаете его роль, давайте посмотрим на реальные возможности записи `TValue`. Она имеет набор методов более высокого уровня для присвоения и извлечения фактических значений, а также набор низкоуровневых указателей. Я сконцентрируюсь на первой группе. Для назначения значений `TValue` определяет несколько операторов Implicit, что позволяет выполнять прямое присвоение, как в приведённых выше фрагментах кода:

```
оператор класса Implicit(const Value: string): Значение:;
оператор класса Implicit(Value: Integer): Значение;
оператор класса Implicit(Value: Extended): Значение;
оператор класса Implicit(Value: Int64): Значение;
оператор класса Implicit(Value: TObject): Значение;
оператор класса Implicit(Value: TClass): Значение;
оператор класса Implicit(Value: Boolean): Значение;
```

Все эти операторы вызывают generic метод класса `From`:

```
class function From<T>(const Value: T): TValue; static;
```

При вызове этих функций класса необходимо указать тип данных, а также передать значение этого типа, например,

следующий код, заменяющий присваивание значения 100 предыдущего фрагмента кода:

```
V1 := TValue.From<Integer>(100);
```

Это своего рода универсальная техника для переноса любого типа данных в `TValue`. После назначения данных можно использовать несколько методов для проверки их типа:

```
  property Kind: TTypeKind read GetTypeKind;
  function IsObject: Boolean;
  function IsClass: Boolean;
  function IsOrdinal: Boolean;
  function IsType<T>: Boolean; overload;
  function IsArray: Boolean;
```

Обратите внимание, что универсальный `IsType` может использоваться практически для любого типа данных.

Существует соответствующий метод извлечения данных, но опять же вы можете использовать только метод, совместимый с фактическими данными, хранящимися в `TValue`, так как преобразование не выполняется:

```
  function AsObject: TObject;
  function AsClass: TClass;
  function AsOrdinal: Int64;
  function AsType<T>: T;
  function AsInteger: Integer;
  function AsBoolean: Boolean;
  function AsExtended: Extended;
  function AsInt64: Int64;
  function AsInterface: IInterface;
  function AsString: string;
  function AsVariant: Variant;
  function AsCurrency: Currency;
```

Некоторые из этих методов дублируются с версией *Try*, которая возвращает False вместо того, чтобы поднимать исключение, в случае несовместимого типа данных. Существуют также некоторые ограниченные методы преобразования, наиболее актуальными из которых являются универсальный `Cast` и функция `ToString`, которую я уже использовал в коде:

```
    function Cast<T>: TValue; overload;
    function ToString: string;
```

Чтение свойств с TValue

Важность `TValue` заключается в том, что это структура, используемая при доступе к свойствам и значениям полей с помощью расширенного RTTI и модуля `Rtti`. В качестве реального примера использования `TValue` можно использовать этот тип записи для доступа как к опубликованному свойству, так и к приватному полю объекта `TButton`, как в следующем коде (часть прикладного проекта RttiAccess):

```
var
  Context: TRttiContext;
  AType: TRttiType;
  AProperty: TRttiProperty;
  AValue: TValue;
  AField: TRttiField;
begin
  AType := Context.GetType(TButton);
  AProperty := AType.GetProperty('Text');
  AValue := AProperty.GetValue(Sender);
  Show (AValue.AsString);

  AField := AType.GetField('FDesignInfo');
  AValue := AField.GetValue(Sender);
  Show (AValue.AsInteger.ToString);
end;
```

Вызов методов

Новая расширенная RTTI не только позволяет получить доступ к значениям и полям, но и обеспечивает упрощенный способ вызова методов. В этом случае необходимо определить элемент `TValue` для каждого параметра метода. Существует глобальная функция Invoke, которую можно вызвать для выполнения метода:

```
функция Invoke(CodeAddress: Указатель; const Args: TArray<TValue>;
 CallingConvention: TCallConv; AResultType: PTypeInfo): TValue;
```

В качестве лучшей альтернативы в классе `TRttiMethod существует` упрощенный перегруженный метод `Invoke`:

```
function Invoke(CodeAddress: Pointer; const Args: TArray<TValue>;
  CallingConvention: TCallConv; AResultType: PTypeInfo): TValue;
```

Два примера обращения к методам с использованием этой второй упрощенной формы (один возвращает значение, а второй требует параметра) являются частью проекта приложения RttiAccess и перечислены ниже:

```
var
  Context: TRttiContext;
  AType: TRttiType;
  AMethod: TRttiMethod;
  TheValues: array of TValue;
  AValue: TValue;
begin
  AType := context.GetType(TButton);
  AMethod := AType.GetMethod('ToString');
  TheValues := [];
  AValue := AMethod.Invoke(Sender, TheValues);
  Show(AValue.AsString);

  AType := Context.GetType(TForm1);
  AMethod := AType.GetMethod('Show');
  SetLength (TheValues, 1);
  TheValues[0] := AValue;
  AMethod.Invoke(self, TheValues);
end;
```

Использование атрибутов

Первая часть этой главы дала вам хорошую основу и представление о расширенной RTTI, сгенерированной компилятором Object Pascal, а также о возможностях доступа к RTTI, введенных новым модулем `Rtti`. Во второй части главы мы, наконец, можем сосредоточиться на одной из ключевых причин, по которой была представлена вся эта архитектура: возможности определения пользовательских атрибутов и расширения RTTI, генерируемого компилятором, специфическими способами. Мы рассмотрим эту технологию с довольно абстрактной точки зрения, а затем остановимся на

причинах, по которым это является важным шагом вперед для Object Pascal, на практических примерах.

Что такое "Атрибут"?

Атрибут (в терминах Object Pascal или C#) или аннотация (на жаргоне Java) — это комментарий или указание на то, что вы можете добавить в исходный код, применив его к типу, полю, методу или свойству), и что компилятор будет встроит в программу. Обычно это обозначается квадратными скобками:

```
type
  [MyAttribute]
  TMyClass = class
    ...
```

Прочитав эту информацию во время проектирования в инструменте разработки или во время выполнения в приложении, программа может изменить свое поведение в зависимости от найденных значений.

Как правило, атрибуты используются не для того, чтобы изменить реальные основные возможности класса объектов, а для того, чтобы позволить этим классам указывать дальнейшие механизмы, в которых они могут участвовать. Объявление класса *сериализуемым никак* не влияет на его код, но позволяет коду сериализации знать, что он может работать с этим классом и как (в случае, если вы предоставите дополнительную информацию вместе с атрибутом, или дополнительные атрибуты, маркирующие поля или свойства класса).

Именно так использовался внутри Object Pascal оригинальный и ограниченный RTTI. Свойства, помеченные как опубликованные, могут быть отображены в инспекторе объектов, переданы в DFM-файл и доступны во время выполнения. Атрибуты открывают этот механизм для того,

чтобы сделать его гораздо более гибким и мощным. Они намного сложнее в использовании и также намного проще использовать их неправильно, как и любые мощные функции языка. Я подразумеваю не стоит выбрасывать все хорошее, что вы знаете об Объектно-ориентированном программировании, чтобы принять эту новую модель, а лучше дополнять одну модель другой.

Например, класс сотрудника все равно будет представлен в иерархии в виде производного класса от класса человека; у объекта сотрудника все равно будет идентификатор его бейджа; но класс сотрудника можно "пометить" или "аннотировать" как класс, который может быть отображен в таблице базы данных или отображен с помощью специальной формы выполнения. Таким образом, мы имеем наследование (is-a), право собственности (has-a) и аннотации (marked-as) в качестве трех отдельных механизмов, которые можно использовать при проектировании приложения.

После того, как вы увидели возможности компилятора, поддерживающего пользовательские атрибуты в Object Pascal, и посмотрели несколько практических примеров, абстрактная идея, о которой я только что упомянул, должна стать более понятной, по крайней мере, я на это надеюсь!

Классы атрибутов и декларации атрибутов

Как определить новый класс атрибута (или категорию атрибута)? Вам необходимо наследовать от нового класса TCustomAttribute, доступного в `Системном` блоке:

```
type
  SimpleAttribute = class(TCustomAttribute)
  end;
```

Имя класса, которое вы даете классу атрибута, станет символом, который будет использоваться в исходном коде, с необязательным исключением постфикса *Attribute*. Таким образом, если вы назовёте свой класс `SimpleAttribute`, то сможете использовать в коде атрибут `Simple` или `SimpleAttribute`. По этой причине классический начальный `T` для классов Object Pascal обычно не используется в случае атрибутов.

Теперь, когда мы определили пользовательский атрибут, мы можем применить его к большинству символов нашей программы: типам, методам, свойствам, полям и параметрам. Синтаксис, используемый для применения атрибутов — это имя атрибута в квадратных скобках:

```
type
  [Simple]
  TMyClass = class(TObject)
  public
    [Simple]
    procedure One;
```

В данном случае я применил атрибут `Simple` к классу в целом и к методу в частности. Помимо имени, атрибут может поддерживать один или несколько параметров. Параметры, передаваемые атрибуту, должны совпадать с параметрами, указанными в конструкторе класса атрибута, если таковой имеется.

```
type
  ValueAttribute = class(TCustomAttribute)
  private
    FValue: Integer;
  public
    constructor Create(N: Integer);
    property Value: Integer read FValue;
  end;
```

Так можно применить этот атрибут с одним параметром:

```
type
  [Value(22)]
  TMyClass = class(TObject)
  public
    [Value(0)]
    procedure Two;
```

Значения атрибута, передаваемые его конструктору, должны быть константными выражениями, так как они разрешаются во время компиляции. Поэтому вы ограничены лишь несколькими типами данных: порядковыми значениями, строками, множествами и ссылками на класс. Положительным моментом является то, что можно иметь несколько перегруженных конструкторов с различными параметрами.

Обратите внимание, что к одному и тому же символу можно применить несколько атрибутов, как это было сделано в проекте приложения RttiAttrib, в котором обобщены фрагменты кода этого раздела:

```
type
  [Simple][Value(22)]
  TMyClass = class(TObject)
  public
    [Simple]
    procedure One;
    [Value(0)]
    procedure Two;
  end;
```

Что если вы попытаетесь использовать атрибут, который не определен (может быть, из-за отсутствующего утверждения uses)? К сожалению, вы получите предупреждение вводящее в заблуждение:

```
[DCC Warning] RttiAttribMainForm.pas(44): W1025
  Unsupported language feature: 'custom attribute'
```

Фактически, это предупреждение означает, что атрибут будет проигнорирован, поэтому вы должны следить за этими предупреждениями или даже лучше относиться к "неподдерживаемой функции языка" как к ошибке (вы можете сделать настройку в диалоге "Параметры проекта" на странице "Подсказки и предупреждения"):

```
[DCC Error] RttiAttribMainForm.pas(38):
  E1025 Unsupported language feature: 'custom attribute'
```

Наконец, по сравнению с другими реализациями этой же концепции, в настоящее время нет способа ограничить область

применения атрибутов, например, объявить, что атрибут может быть применен к типу, но не к методу. Вместо этого в редакторе доступна полная поддержка атрибутов в рефакторинге переименования. Не только можно изменить имя класса атрибутов, но и система подхватит, когда атрибут будет использоваться как в полном имени, так и без конечной части "*attribute*".

примечание Рефакторинг атрибутов впервые был упомянут Малкольмом Гровсом в его блоге: `http://www.malcolmgroves.com/blog/?p=554`

Атрибуты просмотра

Этот код был бы совершенно бесполезным, если бы не было способа узнать, какие атрибуты определены, и, возможно, привнести в объект другое поведение из-за этих атрибутов. Позвольте мне начать с первой части. Классы модуля `Rtti` позволяют выяснить, какие символы имеют ассоциированные с ними атрибуты. Вот код, извлеченный из прикладного проекта RttiAttrib, который показывает список атрибутов для текущего класса:

```
procedure TMyClass.One;
var
  Context: TRttiContext;
  Attributes: TArray<TCustomAttribute>;
  Attrib: TCustomAttribute;
begin
  Attributes := Context.GetType(ClassType).GetAttributes;
  for Attrib in Attributes do
    Form39.Log(Attrib.ClassName);
```

Выполнение этого кода даст выдачу:

```
SimpleAttribute
ValueAttribute
```

Его можно расширить, добавив в код `for-in` следующий код для извлечения конкретного значения данного типа атрибутов:

```
    if Attrib is ValueAttribute then
```

```
    Form39.Show (' -' + IntToStr
      (ValueAttribute(Attrib).Value));
```

Как насчет получения методов с заданным атрибутом, или с любым атрибутом? Вы не можете фильтровать методы заранее, но должны пройтись по каждому из них, проверить их атрибуты и убедиться, что они вам подходят. Чтобы помочь в этом процессе, я написал функцию, которая проверяет, поддерживает ли метод заданный атрибут:

```
type
  TCustomAttributeClass = class of TCustomAttribute;

function HasAttribute (AMethod: TRttiMethod;
  AttribClass: TCustomAttributeClass): Boolean;
var
  Attributes: TArray<TCustomAttribute>;
  Attrib: TCustomAttribute;
begin
  Result := False;
  Attributes := AMethod.GetAttributes;
  for Attrib in Attributes do
     if Attrib.InheritsFrom (AttribClass) then
       Exit (True);
end;
```

Функция `HasAttribute` вызывается программой `RttiAttrib` при проверке заданного атрибута или любого из них:

```
var
  Context: TRttiContext;
  AType: TRttiType;
  AMethod: TRttiMethod;
begin
  AType := Context.GetType(TMyClass);

  for AMethod in AType.GetMethods do
    if HasAttribute (AMethod, SimpleAttribute) then
      Show (AMethod.Name);

  for AMethod in AType.GetMethods do
    if HasAttribute (AMethod, TCustomAttribute) then
      Show (AMethod.Name);
```

Эффект заключается в перечислении методов, помеченных заданными атрибутами, как описано в последующих вызовах `Log`, которые я пропустил из кода выше:

```
Methods marked with [Simple] attribute
One
```

```
Methods marked with any attribute
One
Two
```

Вместо того, чтобы просто описывать атрибуты, как правило, добавляется некий независимое поведение, определяемое атрибутами класса, а не его реальным кодом. В качестве примера, я могу вставить специфическое поведение в предыдущий код: Целью может быть вызов всех методов класса, помеченных определенным атрибутом, рассматривая их как безпараметрические методы:

```
procedure TForm39.BtnInvokeIfZeroClick(Sender: TObject);
var
  Context: TRttiContext;
  AType: TRttiType;
  AMethod: TRttiMethod;
  ATarget: TMyClass;
  ZeroParams: array of TValue;
begin
  ATarget := TMyClass.Create;
  try
    AType := Context.GetType(ATarget.ClassType);
    for AMethod in AType.GetMethods do
      if HasAttribute (AMethod, SimpleAttribute) then
        AMethod.Invoke(ATarget, ZeroParams);
  finally
    ATarget.Free;
  end;
end;
```

Что делает этот фрагмент кода, так это создает объект, узнает его тип, проверяет на наличие заданного атрибута и вызывает каждый метод, имеющий атрибут `Simple`. Вместо того, чтобы наследовать от базового класса, реализовывать интерфейс или писать специфический код для выполнения запроса, все, что нам нужно сделать, чтобы получить новое поведение, это пометить еще один метод с заданным атрибутом. Не то чтобы этот пример делает использование атрибутов суперочевидным: для некоторых распространенных шаблонов использования атрибутов и некоторых реальных примеров можно обратиться к заключительной части этой главы.

Перехват виртуальных методов

> *В этом разделе рассматривается очень продвинутая функция Object Pascal, и вы, возможно, захотите пропустить ее прочтение, если вы только начинаете изучать язык Delphi. Он предназначен для более опытных читателей.*

Есть еще одна важная особенность, которая была добавлена после введения расширенного RTTI, а именно возможность *перехвата* выполнения виртуальных методов существующего класса, путем создания прокси-класса для существующего объекта. Другими словами, можно взять существующий объект и изменить его виртуальные методы (конкретный или все сразу).

Зачем это? В стандартном приложении Object Pascal вы, вероятно, не будете использовать эту функцию. Если вам нужен объект с другим поведением, просто измените его или создайте подкласс. Для библиотек все по-другому, потому что библиотеки должны быть написаны очень общим образом, мало зная об объектах, которыми они смогут манипулировать, и накладывая как можно меньше нагрузки на сами объекты. Именно для такого сценария был добавлен в Object Pascal перехват виртуальных методов.

примечание Очень подробный пост Барри Келли в блоге о виртуальных перехватчиках (которым я многим обязан) доступен по адресу http://blog.barrkel.com/2010/09/virtual-method-interception.html.

Прежде чем сосредоточиться на возможных сценариях, позвольте мне обсудить саму технологию. Предположим, что у

вас есть существующий класс по крайней мере с виртуальным методом, как показано ниже:

```
type
  TPerson = class
  ...
  public
    property Name: string read FName write SetName;
    property BirthDate: TDate read FBirthDate write SetBirthDate;
    function Age: Integer; virtual;
    function ToString: string; override;
  end;

function TPerson.Age: Integer;
begin
  Result := YearsBetween (Date, FBirthDate);
end;

function TPerson.ToString: string;
begin
  Result := FName + ' is ' + IntToStr (Age) + ' years old';
end;
```

Теперь можно создать объект TVirtualMethodInterceptor (новый класс, определенный в модуле RTTI), привязанный к подклассу класса объекта, изменив статический класс объекта на динамический:

```
var
  Vmi: TVirtualMethodInterceptor;
begin
  Vmi := TVirtualMethodInterceptor.Create(TPerson);
  Vmi.Proxify(Person1);
```

После того как у вас есть объект `vmi`, вы можете *установить* специальные обработчики его событий (`OnBefore`, `OnAfter` и `OnException`) анонимными методами. Они будут срабатывать перед любым вызовом виртуального метода, после любого вызова виртуального метода, и в случае возникновения исключения в виртуальном методе. Это сигнатуры трех анонимных методов:

```
 TInterceptBeforeNotify = reference to procedure(
    Instance: TObject; Method: TRttiMethod;
    const Args: TArray<TValue>; out DoInvoke: Boolean;
    out Result: TValue);
  TInterceptAfterNotify = reference to procedure(
    Instance: TObject; Method: TRttiMethod;
```

```
    const Args: TArray<TValue>; var Result: TValue);
  TInterceptExceptionNotify = reference to procedure(
    Instance: TObject; Method: TRttiMethod;
    const Args: TArray<TValue>; out RaiseException: Boolean;
    TheException: Exception; out Result: TValue);
```

В каждом случае вы получаете объект, ссылку на метод, параметры и результат (который может быть уже установлен или нет). В событии `OnBefore` вы можете установить параметр `DoInvoke`, чтобы отключить стандартное исполнение. В событии `OnExcept` вы получаете информацию об исключении.

В прикладном проекте InterceptBaseClass, использующем вышеприведенный класс `TPerson`, я перехватил виртуальные методы класса с этим кодом логгирования:

```
procedure TFormIntercept.BtnInterceptClick(Sender: TObject);
begin
  Vmi := TVirtualMethodInterceptor.Create(TPerson);
  Vmi.OnBefore := procedure(Instance: TObject; Method: TRttiMethod;
      const Args: TArray<TValue>; out DoInvoke: Boolean;
      out Result: TValue)
    begin
      Show('Before calling ' + Method.Name);
    end;
  Vmi.OnAfter := procedure(Instance: TObject; Method: TRttiMethod;
      const Args: TArray<TValue>; var Result: TValue)
    begin
      Show('After calling ' + Method.Name);
    end;
  Vmi.Proxify(Person1);
```

Обратите внимание, что объект `vmi` необходимо хранить, по крайней мере, до тех пор, пока объект `Person1` будет использоваться, или же вы будете использовать динамический класс, который больше недоступен, и будете вызывать анонимные методы, которые уже были утилизированы. В демонстрационном примере я сохранил его в виде поля формы, точно так же, как и объект, на который он ссылается.

Программа использует объект, вызывая его методы и проверяя имя базового класса:

```
   Show ('Age: ' + IntToStr (Person1.Age));
  Show ('Person: ' + Person1.ToString);
  Show ('Class: ' + Person1.ClassName);
```

```
  Show ('Base Class: ' + Person1.ClassParent.ClassName);
```

Перед установкой перехватчика выводится:

```
Age: 26
Person: Mark is 26 years old
Class: TPerson
Base Class: TObject
```

После установки перехватчика выход становится:

```
Before calling Age
After calling Age
Age: 26
Before calling ToString
Before calling Age
After calling Age
After calling ToString
Person: Mark is 26 years old
Class: TPerson
Base Class: TPerson
```

Обратите внимание, что класс имеет одно и то же имя базового класса, но на самом деле это другой, динамический класс, созданный виртуальным перехватчиком методов. Хотя официального способа восстановления класса целевого объекта до исходного нет, сам класс доступен в объекте Перехватчик виртуального метода, а также в качестве базового класса объекта. Тем не менее, можно использовать *грубую силу*, чтобы присвоить данным класса объекта (в его начальные четыре байта) правильную ссылку на класс:

```
  PPointer(Person1)^ := Vmi.OriginalClass;
```

В качестве еще одного примера, я модифицировал код `OnBefore` так, что в случае вызова `Age` он возвращает заданное значение и пропускает выполнение фактического метода:

```
  Vmi.OnBefore := procedure(Instance: TObject; Method: TRttiMethod;
      const Args: TArray<TValue>; out DoInvoke: Boolean;
      out Result: TValue)
    begin
      Show ('Before calling ' + Method.Name);
      if Method.Name = 'Age' then
      begin
        Result := 33;
        DoInvoke := False;
      end;
    end;
```

Вывод изменяется по сравнению с вышеуказанной версией следующим образом (обратите внимание, что пропускаются вызовы `Age` и относительные события `OnAfter`):

```
Before calling Age
Age: 33
Before calling ToString
Before calling Age
After calling ToString
Person: Mark is 33 years old
Class: TPerson
Base Class: TPerson
```

Теперь, когда мы рассмотрели технические подробности перехватчика виртуальных методов, мы можем вернуться к размышлениям о том, в каких сценариях вы хотите использовать эту функцию. Опять же, в стандартном приложении для нас практически нет причин для этого. Вместо этого, внимание в основном сосредоточено на тех, кто разрабатывает продвинутые библиотеки и нуждается в реализации пользовательского поведения для тестирования или обработки объектов.

Например, это могло бы помочь построить библиотеку юнит-тестирования, хотя она была бы ограничена только виртуальными методами. Вы также могли бы использовать это вместе с пользовательскими атрибутами для реализации стиля кодирования, похожего на Aspect Oriented Programming.

Сценарии использования RTTI

Теперь, когда я рассказал об основах RTTI и использовании атрибутов, стоит взглянуть на некоторые реальные ситуации, в которых использование этой техники окажется полезным.

Существует множество сценариев, в которых более гибкая RTTI и возможность ее настройки с помощью атрибутов является актуальной, но у меня нет места для длинного списка ситуаций. Вместо этого я введу вас в пошаговую разработку двух простых, но важных примеров.

Первая демонстрационная программа прояснит использование атрибутов для идентификации конкретной информации внутри класса. В частности, мы хотим иметь возможность инспектировать объект класса, который декларирует себя частью архитектуры и имеет описание и уникальный идентификатор, относящийся к самому объекту. Это может пригодиться в нескольких ситуациях, например, при описании объектов, хранящихся в коллекции (generic или традиционной).

Вторая демонстрация будет примером потоковой передачи, а именно передачи класса в XML-файл. Я начну с классического подхода к использованию опубликованного RTTI, перейду к новому расширенному RTTI и, наконец, покажу, как можно использовать атрибуты для настройки кода и сделать его более гибким.

Атрибуты для ID и описания

Если вы хотите, чтобы несколько методов были общими для многих объектов, то самым классическим подходом было определить базовый класс с виртуальными методами и наследовать различные объекты от базового класса, переопределяя виртуальные методы. Это красиво, но накладывает массу ограничений на классы, которые могут участвовать в архитектуре, так как у вас есть фиксированный базовый класс.

Стандартной методикой преодоления этой ситуации является использование интерфейса, а не общего базового класса. Многочисленные классы, реализующие интерфейс (но без общего класса-предка), могут обеспечить реализацию методов интерфейса, которые действуют очень похоже на виртуальные методы.

Совершенно другим стилем (как с преимуществами, так и с недостатками) является использование атрибутов для обозначения классов-участников и заданных методов (или свойств). Это открывает больше гибкости, не требует использования интерфейсов, но основано на сравнительно медленном и подверженном ошибкам поиске информации о времени выполнения, а не на разрешении во время компиляции. Это означает, что я не выступаю за такой стиль кодирования как подход лучше интерфейсов, но только как подход, который может быть оценен и интересен для использования в некоторых обстоятельствах.

Класс описания атрибута

Для этой демонстрации я определил атрибут с настройкой, указывающей на то, к какому элементу он применяется. Я мог бы использовать три различных атрибута, но предпочитаю не загрязнять пространство имён атрибутов. Это определение класса атрибута:

```
type
  TDescriptionAttrKind = (dakClass, dakDescription, dakId);

  DescriptionAttribute = class (TCustomAttribute)
  private
    FDak: TDescriptionAttrKind;
  public
    constructor Create (ADak: TDescriptionAttrKind = dakClass);
    property Kind: TDescriptionAttrKind read FDak;
  end;
```

Обратите внимание на использование конструктора со значением по умолчанию для его единственного параметра, чтобы можно было использовать атрибут без параметров.

Примеры Классов

Далее я написал два примера классов, использующих атрибут. Каждый класс помечен атрибутом и имеет два метода, помеченных одним и тем же атрибутом, настроенных с разными *типами*. Первый (`TPerson`) имеет описание, сопоставленное с функцией `GetName` и использует ее `TObject.`GetHashCode для предоставления (временного) ID, переименовывая метод для применения к нему атрибута (код метода - это просто вызов `унаследованной` версии):

```
type
  [Description]
  TPerson = class
  private
    FBirthDate: TDate;
    FName: string;
    FCountry: string;
    procedure SetBirthDate(const Value: TDate);
    procedure SetCountry(const Value: string);
    procedure SetName(const Value: string);
  public
    [Description (dakDescription)]
    function GetName: string;
    [Description (dakID)]
    function GetStringCode: Integer;
  published
    property Name: string read GetName write SetName;
    property BirthDate: TDate
      read FBirthDate write SetBirthDate;
    property Country: string read FCountry write SetCountry;
  end;
```

Второй класс (`TCompany`) еще проще, так как имеет собственные значения для идентификатора и описания:

```
type
  [Description]
  TCompany = class
  private
    FName: string;
    FCountry: string;
```

```
    FID: string;
    procedure SetName(const Value: string);
    procedure SetID(const Value: string);
  public
    [Description (dakDescription)]
    function GetName: string;
    [Description (dakID)]
    function GetID: string;
  published
    property Name: string read GetName write SetName;
    property Country: string read FCountry write FCountry;
    property ID: string read FID write SetID;
  end;
```

Даже если между двумя классами есть сходство, они совершенно не связаны между собой с точки зрения иерархии, общего интерфейса или чего-то подобного. Что их объединяет, так это использование одного и того же атрибута.

Проект для примера и навигация по атрибутам

Совместное использование атрибута используется для отображения информации об объектах, добавленных в список, объявленных в основной форме программы как:

```
  FObjectsList: TObjectList<TObject>;
```

Этот список создается и инициализируется при запуске программы:

```
procedure TFormDescrAttr.FormCreate(Sender: TObject);
var
  APerson: TPerson;
  ACompany: TCompany;
begin
  FObjectsList := TObjectList<TObject>.Create;

  // add a person
  APerson := TPerson.Create;
  APerson.Name := 'Wiley';
  APerson.Country := 'Desert';
  APerson.BirthDate := Date - 1000;
  fObjectsList.Add(APerson);

  // add a company
  ACompany := TCompany.Create;
  ACompany.Name := 'ACME Inc.';
```

```
  ACompany.ID := IntToStr (GetTickCount);
  ACompany.Country := 'Worldwide';
  FObjectsList.Add(ACompany);

  // add an unrelated object
  FObjectsList.Add(TStringList.Create);
```

Для отображения информации об объектах (а именно ID и описание, если доступно) программа использует обнаружение атрибутов через RTTI. Сначала она использует вспомогательную функцию для определения, помечен ли класс определенным атрибутом:

```
function TypeHasDescription (aType: TRttiType): Boolean;
var
  Attrib: TCustomAttribute;
begin
  for Attrib in AType.GetAttributes do
  begin
    if (Attrib is DescriptionAttribute) then
      Exit (True);
  end;
  Result := False;
end;
```

примечание В этом случае необходимо проверить полное имя класса, `DescriptionAttribute`, а не только " Description ", которое является символом, который можно использовать при применении атрибута.

Если это так, то программа переходит к получению каждого атрибута каждого метода, с вложенным циклом, и проверяет, не является ли это тот атрибут, который мы ищем:

```
  if TypeHasDescription (AType) then
  begin
    for AMethod in AType.GetMethods do
      for Attrib in AMethod.GetAttributes do
        if Attrib is DescriptionAttribute then
          ...
```

В ядре цикла вызываются методы, помеченные атрибутами, для чтения результатов в двух временных строках (позже добавленных в пользовательский интерфейс):

```
  if Attrib is DescriptionAttribute then
    case DescriptionAttribute(Attrib).Kind of
      dakClass: ; // ignore
      dakDescription:
        strDescr := AMethod.Invoke(AnObject, []).ToString;
      dakId:
```

```
        strID := AMethod.Invoke(AnObject, []).ToString;
```

Что программа не может сделать, так это проверить, дублируется ли атрибут (то есть, если несколько методов помечены одним и тем же атрибутом, ситуация, в которой вы можете захотеть поднять исключение).

Подводя итог всем фрагментам предыдущей страницы, вот полный код метода `UpdateList`:

```
procedure TFormDescrAttr.UpdateList;
var
  AnObject: TObject;
  Context: TRttiContext;
  AType: TRttiType;
  Attrib: TCustomAttribute;
  AMethod: TRttiMethod;
  StrDescr, StrID: string;
begin
  for AnObject in FObjectsList do
  begin
    aType := context.GetType(AnObject.ClassInfo);
    if TypeHasDescription (AType) then
    begin
      for AMethod in AType.GetMethods do
        for Attrib in AMethod.GetAttributes do
          if Attrib is DescriptionAttribute then
            case DescriptionAttribute(Attrib).Kind of
              dakClass: ; // ignore
              dakDescription:
                // should check if duplicate attribute
                StrDescr := aMethod.Invoke(
                  AnObject, []).ToString;
              dakId:
                StrID := AMethod.Invoke(
                  AnObject, []).ToString;
            end;
      // done looking for attributes
      // should check if we found anything
      with ListView1.Items.Add do
      begin
        Text := STypeName;
        Detail := StrDescr;
      end;
    end;
  end;
  // else ignore the object, could raise an exception
end;
```

Если данная программа производит довольно неинтересный вывод, то способ, которым это делается, актуален, так как я

обозначил некоторые классы и два метода этих классов атрибутом, и смог обработать эти классы внешним алгоритмом.

Другими словами, сами классы не нуждаются ни в специфическом базовом классе, ни в реализации интерфейса, ни во внутреннем коде, который должен быть частью архитектуры, а только в том, чтобы *объявить, что* они хотят быть вовлеченными в процесс с помощью атрибутов. Полная ответственность за управление классами лежит в некотором внешнем коде.

Потоковая передача XML

Одним из интересных и очень полезных случаев использования RTTI является создание *"внешнего"* изображения объекта, сохранение его статуса в файл или отправка по линии связи в другое приложение. Традиционно, подход Object Pascal к этой проблеме заключается в потоковой передаче опубликованных свойств объекта, тот же подход используется при создании файлов DFM. Теперь RTTI позволяет сохранять фактические данные объекта, его поля, а не внешний интерфейс. Это более мощный способ, хотя и может привести к дополнительным сложностям, например, в управлении данными внутренних объектов. Пример опять-таки действует, как простая демонстрация техники и не углубляется во все ее смыслы.

Эти примеры поставляются в трех версиях, скомпилированных в одном проекте для простоты. Первая — это традиционный подход Object Pascal, основанный на опубликованных свойствах, вторая - использует расширенные RTTI и поля, третья - атрибуты для настройки отображения данных.

Тривиальный класс XML Writer

Чтобы помочь с генерацией XML, я сделал проект приложения XmlPersist на расширенной версии класса `TTrivialXmlWriter`, который я изначально написал для своей книги Delphi 2009 Handbook, чтобы продемонстрировать использование класса TTextWriter. Здесь я не собираюсь снова об этом рассказывать. Достаточно сказать, что благодаря стеку строк класс может отслеживать открываемые им XML-узлы и закрывать XML-узлы в порядке LIFO (Last In, First Out).

примечание Исходный код класса `TTrivialXmlWriter` Delphi 2009 Handbook можно найти по адресу `http://github.com/MarcoDelphiBooks/Delphi2009Handbook/tree/master/07/ReaderWriter`.

В оригинальный класс я добавил немного ограниченного кода форматирования и три метода сохранения объекта, основанных на трех различных подходах, которые я рассмотрю в этом разделе. Это полное объявление класса:

```
type
  TTrivialXmlWriter = class
  private
    FWriter: TTextWriter;
    FNodes: TStack<string>;
    FOwnsTextWriter: Boolean;
  public
    constructor Create (AWriter: TTextWriter); overload;
    constructor Create (AStream: TStream); overload;
    destructor Destroy; override;
    procedure WriteStartElement (const SName: string);
    procedure WriteEndElement (Indent: Boolean = False);
    procedure WriteString (const SValue: string);
    procedure WriteObjectPublished (AnObj: TObject);
    procedure WriteObjectRtti (AnObj: TObject);
    procedure WriteObjectAttrib (AnObj: TObject);
    function Indentation: string;
  end;
```

Для получения представления о коде используется метод `WriteStartElement`, который использует функцию `Indentation` для того, чтобы оставить вдвое больше пробелов, чем текущее количество узлов на внутреннем стеке:

```
procedure TTrivialXmlWriter.WriteStartElement(
  const SName: string);
begin
  FWriter.Write (Indentation + '<' + SName + '>');
  FNodes.Push (SName);
end;
```

Полный код класса вы найдете в исходных текстах проекта.

Классический стримминг на основе RTTI

После этого введения, охватывающего класс поддержки, позвольте мне начать с самого начала, то есть сохранить объект в формате XML, используя классический RTTI для опубликованных свойств.

Код метода `WriteObjectPublished` достаточно сложен и требует небольшого пояснения. Он основан на модуле `TypInfo` и использует низкоуровневую версию старого RTTI, чтобы иметь возможность получить список опубликованных свойств для данного объекта (параметр `AnObj`), с кодом типа:

```
  NProps := GetTypeData(AnObj.ClassInfo)^.PropCount;
  GetMem(PropList, NProps * SizeOf(Pointer));
  GetPropInfos(AnObj.ClassInfo, PropList);
  for I := 0 to NProps - 1 do
    ...
```

Для этого нужно запросить количество свойств, выделить структуру данных нужного размера и заполнить структуру данных информацией об опубликованных свойствах. Вам интересно, смогли бы вы написать этот низкоуровневый код? Ну, вы только что нашли очень хорошую причину, по которой был введен новый RTTI. Для каждого свойства программа извлекает значение свойств числового и строкового типов, а для любого подобъекта - рекурсивно:

```
StrPropName := UTF8ToString (PropList[i].Name);
case PropList[i].PropType^.Kind of
  tkInteger, tkEnumeration, tkString, tkUString, ...:
  begin
    WriteStartElement (StrPropName);
    WriteString (GetPropValue(AnObj, StrPropName));
    WriteEndElement;
```

```
  end;
  tkClass:
  begin
    InternalObject := GetObjectProp(AnObj, StrPropName);
    // recurse in subclass
    WriteStartElement (StrPropName);
    WriteObjectPublished (InternalObject as TPersistent);
      WriteEndElement (True);
  end;
end;
```

Есть некоторая дополнительная сложность, но для примера и для того, чтобы дать вам представление о традиционном подходе, этого должно быть достаточно.

Для демонстрации эффекта программы я написал два класса (`TCompany` и `TPerson`), адаптированных из предыдущего примера. На этот раз, однако, в компании может быть человек, присвоенный дополнительному свойству, называемому `Boss`. В реальном мире это было бы сложнее, но для данного примера это разумное предположение. Вот опубликованные свойства двух классов:

```
type
  TPerson = class (TPersistent)
    ...
  published
    property Name: string read FName write FName;
    property Country: string read FCountry write FCountry;
  end;

  TCompany = class (TPersistent)
    ...
  published
    property Name: string read FName write FName;
    property Country: string read FCountry write FCountry;
    property ID: string read FID write FID;
    property Boss: TPerson read FPerson write FPerson;
  end;
```

Основная форма программы имеет кнопку, с помощью которой можно создать и соединить два объекта этих двух классов и сохранить их в поток XML, который в дальнейшем будет отображаться. Раздел streaming имеет следующий код:

```
  SS := TStringStream.Create;
  XmlWri := TTrivialXmlWriter.Create (SS);
  XmlWri.WriteStartElement('company');
```

```
  XmlWri.WriteObjectPublished(aCompany);
  XmlWri.WriteEndElement;
```

В результате получается XML-файл типа:

```
<company>
  <Name>ACME Inc.</Name>
  <Country>Worldwide</Country>
  <ID>29088851</ID>
  <Boss>
    <Name>Wiley</Name>
    <Country>Desert</Country>
  </Boss>
</company>
```

Стримминг полей с расширенной RTTI

С RTTI высокого уровня, доступным в Object Pascal, я мог бы преобразовать эту старую программу для использования расширенного RTTI для доступа к опубликованным свойствам. Вместо этого я собираюсь использовать его для сохранения внутреннего представления объекта, то есть его приватных полей данных. Я не только делаю что-то более сложное, но и делаю это с помощью гораздо более высокоуровневого кода. Полный код метода `WriteObjectRtti выглядит` следующим образом:

```
procedure TTrivialXmlWriter.WriteObjectRtti(AnObj: TObject);
var
  AContext: TRttiContext;
  AType: TRttiType;
  AField: TRttiField;
begin
  AType := AContext.GetType (AnObj.ClassType);
  for AField in aType.GetFields do
  begin
    if AField.FieldType.IsInstance then
    begin
      WriteStartElement (AField.Name);
      WriteObjectRtti (AField.GetValue(AnObj).AsObject);
      WriteEndElement (True);
    end
    else
    begin
      WriteStartElement (AField.Name);
      WriteString (AField.GetValue(AnObj).ToString);
      WriteEndElement;
    end;
  end;
end;
```

Получившийся XML немного похож, но несколько менее чистый, так как имена полей, как правило, менее читабельны, чем имена свойств:

```
<company>
  <FName>ACME Inc.</FName>
  <FCountry>Worldwide</FCountry>
  <FID>29470148</FID>
  <FPerson>
    <FName>Wiley</FName>
    <FCountry>Desert</FCountry>
  </FPerson>
</company>
```

Другое большое отличие, однако, в том, что в этом случае классы не нужно было наследовать от класса `TPersistent` или компилировать с помощью специальной опции.

Использование атрибутов для настройки потоковой передачи

Кроме проблемы с именами тегов, есть еще одна проблема, о которой я не упоминал. Использование имен XML-тегов, которые на самом деле являются скомпилированными символами, далеко не самая лучшая идея. Кроме того, в коде нет возможности исключить некоторые свойства или поля из XML-потоков.

примечание Потоком свойств Object Pascal можно управлять с помощью директивы `stored`, которую можно прочитать с помощью блока `TypInfo`. Тем не менее, это решение далеко не простое и чистое, даже если механизм потоковой передачи DFM использует его эффективно.

Эти проблемы мы можем решить, используя атрибуты, хотя недостаток будет заключаться в том, что они будут довольно сильно использоваться в объявлении наших классов, стиль кодирования, который мне не очень нравится. Для новой версии кода я определил конструктор атрибутов с необязательным параметром:

```
type
```

```
  XmlAttribute = class (TCustomAttribute)
  private
    FTag: string;
  public
    constructor Create (StrTag: string = '');
    property TagName: string read FTag;
  end;
```

Потоковый код, основанный на атрибутах, является разновидностью последней версии, основанной на расширенном RTTI. Единственное отличие состоит в том, что теперь программа вызывает функцию помощника `CheckXmlAttr` для проверки, имеет ли поле атрибут `xml` и (необязательно) декорирование имени тега:

```
procedure TTrivialXmlWriter.WriteObjectAttrib(AnObj: TObject);
var
  AContext: TRttiContext;
  AType: TRttiType;
  AField: TRttiField;
  StrTagName: string;
begin
  AType := AContext.GetType (AnObj.ClassType);
  for AField in AType.GetFields do
  begin
    if CheckXmlAttr (AField, StrTagName) then
    begin
      if AField.FieldType.IsInstance then
      begin
        WriteStartElement (StrTagName);
        WriteObjectAttrib (AField.GetValue(AnObj).AsObject);
        WriteEndElement (True);
      end
      else
      begin
        WriteStartElement (StrTagName);
        WriteString (AField.GetValue(AnObj).ToString);
        WriteEndElement;
      end;
    end;
  end;
end;
```

Наиболее актуальный код находится в справочной функции `CheckXmlAttr`:

```
function CheckXmlAttr (AField: TRttiField;
  var StrTag: string): Boolean;
var
  Attrib: TCustomAttribute;
begin
```

```
  Result := False;
  for Attrib in AField.GetAttributes do
    if Attrib is XmlAttribute then
    begin
      StrTag := XmlAttribute(Attrib).TagName;
      if StrTag = '' then // default value
        StrTag := AField.Name;
      Exit (True);
    end;
end;
```

Поля без атрибута XML игнорируются, а тег, используемый в выводе XML, настраивается. Для демонстрации этого в программе есть следующие классы (на этот раз я опустил опубликованные свойства из листинга, так как они не являются релевантными):

```
type
  TAttrPerson = class
  private
    [xml ('Name')]
    FName: string;
    [xml]
    FCountry: string;
    ...

  TAttrCompany = class
  private
    [xml ('CompanyName')]
    FName: string;
    [xml ('Country')]
    FCountry: string;
    FID: string; // omitted
    [xml ('TheBoss')]
    FPerson: TAttrPerson;
    ...
```

С этими декларациями XML-вывод будет выглядеть следующим образом (обратите внимание на имя тега, тот факт, что ID опущен, и (плохо выглядящее) имя по умолчанию для поля `FCountry`):

```
<company>
  <CompanyName>ACME Inc.</CompanyName>
  <Country>Worldwide</Country>
  <TheBoss>
    <Name>Wiley</Name>
    <FCountry>Desert</FCountry>
  </TheBoss>
</company>
```

Разница здесь в том, что мы можем быть очень гибкими в отношении того, какие поля включать и как их называть в XML, что не позволялось предыдущими версиями.

Даже если это просто очень схематическая реализация, я думаю, что возможность увидеть создаваемую финальную версию шаг за шагом, начиная с классической RTTI, дало вам хорошее ощущение различий между разнообразными техниками. На самом деле, важно помнить, что не всегда использование атрибутов будет лучшим решением! С другой стороны, должно быть понятно, что RTTI и атрибуты добавляют много мощности и гибкости в любом сценарии, в котором вам нужно осмотреть структуру неизвестного объекта во время выполнения.

Другие библиотеки на базе RTTI

В заключение этой главы я хотел бы отметить тот факт, что есть много библиотек, как часть продукта, так и от третьих фирм, которые начали использовать расширенную RTTI. Одним из таких примеров является механизм связывания выражений, который работает в *visual live bindings*. Вы можете создать выражение связывания, присвоить ему выражение (то есть строку текста с такими операциями, как конкатенация или сложение), и получить выражение, ссылающееся на внешний объект и его поле.

Хотя я не собираюсь слишком углубляться в эту тему, которая на самом деле является специфической библиотекой, а не частью языка или ядра системы, я думаю, что короткий фрагмент может дать вам идею:

```
var
  BindExpr: TBindingExpression;
  Pers: TPerson;
begin
```

```
  Pers := TPerson.Create;
  Pers.Name := 'John';
  Pers.City := 'San Francisco';

  BindExpr := TBindingExpressionDefault.Create;
  BindExpr.Source := 'person.name + " lives in " + person.city');
  BindExpr.Compile([
    TBindingAssociation.Create(Pers, 'person')]);
  Show (BindExpr.Evaluate.GetValue.ToString);

  Pers.Free;
  BindExpr.Free;
end;
```

Обратите внимание, что преимущество здесь заключается в том, что вы можете изменять выражение во время выполнения (хотя в конкретном фрагменте выше это константная строка). Выражение может прийти из поля, или может быть выбрано динамически из нескольких возможных выражений. Сначала оно присваивается объекту `TBindingExpression`, а затем анализируется и *компилируется* (то есть трансформируется в символическую форму, а не в ассемблерный код) во время выполнения с помощью вызова `Compile`. Затем он будет использовать RTTI при выполнении доступа к объекту `TPerson`.

Недостатком такого подхода является то, что вычисление выражений происходит значительно медленнее, чем выполнение аналогичного скомпилированного кода Object Pascal. Другими словами, приходится балансировать между сниженной производительностью и повышенной гибкостью. Кроме того, модель Visual Live Binding, построенная поверх этой техники, делает очень приятным и простым опыт разработки приложений

17: TObject и модуль System

В основе любого приложения на языке Object Pascal лежит иерархия классов. Каждый класс в системе, в конечном счете, является подклассом класса TObject, поэтому вся иерархия имеет один корень. Это позволяет использовать тип данных `TObject в` качестве замены данных любого типа класса в системе.

Класс `TObject` определен в базовом RTL модуле System, который имеет настолько важную роль, что автоматически включается в каждую компиляцию. Хотя я не буду рассказывать обо всех классах и других функциях модуля `System`, есть несколько заслуживающих внимания, и `TObject`, безусловно, самый важный из них.

примечание Это может быть предметом продолжительных обсуждений, является ли класс ядра системы, такой как `TObject`, частью языка, или он является частью библиотеки Run Time Library (RTL). То же самое относится и к другим возможностям `системного` модуля - модуля, настолько критичного, что он автоматически включается в компиляцию любого другого модуля. (На самом деле, нельзя самим добавлять его в uses.) Однако подобные дебаты были бы довольно тщетными, поэтому я просто оставлю это на другой раз.

Класс TObject.

Как я только что упомянул, класс `TObject` является самым особенным, так как все другие классы наследуют от него. Когда вы объявляете новый класс, на самом деле, если вы не указываете базовый класс, то класс автоматически наследует от `TObject`. В терминах языка программирования этот тип сценария называется иерархией классов, имеющей один корень. Это свойство Object Pascal, имеется в C#, Java и многих других современных языках программирования. Заметным исключением является C++, который не имеет понятия одного базового класса и позволяет определить несколько абсолютно отдельных иерархий классов.

Этот базовый класс не является классом, экземпляры которого вы бы создавали непосредственно. Тем не менее, это класс, который вы легко можете использовать. Каждый раз, когда Вам нужна переменная, которая может содержать объект любого типа, Вы объявляете его типом `TObject`. Хорошим примером такого использования являются обработчики событий компонентных библиотек, которые очень часто имеют `TObject в` качестве типа первого параметра, обычно называемого `Sender`. Это означает *любой* объект *любого* реального класса. Многие generic коллекции также являются коллекциями объектов, и существует несколько сценариев, в которых тип `TObject` используется непосредственно. В следующих разделах я коснусь некоторых функций этого класса, которые доступны всем классам в системе.

Конструкторы и деструкторы

Хотя напрямую создавать `TObject` не имеет смысла, конструктор и деструктор этого класса важны, так как они автоматически наследуются всеми другими классами. Если вы определяете класс без конструктора, то вы все равно можете вызвать на нем `Create`, вызвав конструктор `TObject`, который является пустым методом (так как в этом базовом классе инициализировать нечего). Этот конструктор Create не является виртуальным, и вы полностью заменяете его в своих классах, если только этот ничего не делающий конструктор не устраивает вас. Вызов конструктора базового класса является хорошей практикой для любого подкласса, даже если прямой вызов `TObject.Create` не особенно полезен.

примечание Я подчеркнул, что это не виртуальный конструктор, потому что вместо него есть другой класс основной библиотеки, `TComponent`, который определяет виртуальный конструктор. Виртуальный конструктор класса `TComponent` играет ключевую роль в потоковой системе, рассмотренной в следующей главе.

Для уничтожения объекта в классе `TObject` имеется метод `Free` (который в конечном итоге вызывает деструктор `Destroy`). Я подробно рассказал об этом в Гл. 13, вместе со многими предложениями по правильному использованию памяти, так что нет необходимости повторять их здесь.

Знания об объекте

Интересной группой методов класса `TObject` являются методы, возвращающие информацию о типе. Наиболее часто используются методы ClassType и ClassName. Метод `ClassName` возвращает строку с именем класса. Поскольку это метод класса (как и большое количество методов класса `TObject`), его можно применять как к объекту, так и к классу. Предположим,

что вы определили класс `TButton` и объект `Button1` этого класса. Тогда следующие утверждения будут иметь одинаковый эффект:

```
Text := Button1.ClassName;
Text := TButton.ClassName;
```

Конечно, вы также можете применить их к generic объекту `TObject`, и вы не получите информацию об этом объекте, а информацию об определенном классе объекта, который в данный момент присваивается переменной. Например, в обработчике события `OnClick` кнопки, вызывающей:

```
Text := Sender.ClassName;
```

скорее всего, вернет то же самое, что и вышеприведенные строки, т.е. строку '*TButton*'. Это связано с тем, что имя класса определяется во время выполнения (самим конкретным объектом), а не компилятором (который будет думать только, что это объект `TObject`).

Хотя получение имени класса может быть удобно при отладке, протоколировании и отображении информации о классе в целом, часто более важно получить доступ к ссылке на класс. В качестве примера, лучше сравнить две ссылки на класс, чем строки с именами классов. Ссылки на классы можно получить с помощью метода `ClassType`, а метод `ClassParent` возвращает ссылку на класс базового класса текущего, позволяя перейти к списку базовых классов. Единственным исключением является то, что метод возвращает `nil` для `TObject` (так как у него нет класса-родителя). Имея ссылку на класс, можно использовать его для вызова любого метода класса, в том числе и метода `ClassName`.

Еще одним очень интересным методом, возвращающим информацию о классе, является InstanceSize, который возвращает размер объекта времени выполнения, т.е. объем памяти, необходимый для его полей (и тех, которые

унаследованы от базовых классов). Эта функция используется внутри системы, когда системе необходимо выделить новый экземпляр класса.

примечание Хотя вы можете подумать, что глобальная функция `SizeOf` также предоставляет эту информацию, на самом деле эта функция возвращает размер ссылки на объект - указатель, который неизменно составляет четыре или восемь байт, в зависимости от целевой платформы – вместо размера самого объекта.

Дополнительные методы класса TObject

Есть и другие методы класса `TObject`, которые можно применить к любому объекту (а также к любому классу или ссылке на класс, поскольку они являются методами класса). Вот неполный список, с кратким описанием:

- ClassName возвращает для отображения строку с именем класса.
- ClassNameIs проверяет имя класса на соответствие значению.
- ClassParent возвращает ссылку на родительский класс текущего класса или класса объекта. Вы можете перейти от `ClassParent` к `ClassParent`, пока не достигнете самого класса `TObject`, в котором этот метод возвращает `nil`.
- ClassInfo возвращает указатель на внутреннюю, низкоуровневую информацию о типе запуска (RTTI) класса. Он использовался еще в ранние времена модуля `TypInfo`, но теперь он заменен возможностями модуля `RTTI`, о чем говорилось в главе 16. Внутри класса RTTI все еще используется именно так.
- ClassType возвращает ссылку на класс объекта (это не может быть применено непосредственно к классу, только к объекту).

- InheritsFrom проверяет, наследует ли класс (прямо или косвенно) от данного базового класса (это очень похоже на оператор `is`, и, в конечном итоге, так реализуется оператор `is` на самом деле).
- InstanceSize возвращает размер данных объекта в байтах. Это сумма полей плюс некоторые дополнительные специальные зарезервированные байты (включая, например, ссылку на класс). Заметьте, еще раз, это размер экземпляра, в то время как ссылка на экземпляр имеет только длину указателя (4 или 8 байт, в зависимости от платформы).
- UnitName возвращает имя модуля, в котором определен класс, что может быть полезно при описании класса. Имя класса, по сути, не является уникальным в системе. Как мы видели в предыдущей главе, только квалифицированное имя класса (состоящее из имени юнита и имени класса, разделенного точкой) является уникальным в приложении.
- QualifiedClassName возвращает эту комбинацию имени модуля и класса, значение, которое действительно уникально в работающей системе.

Эти методы `TObject` доступны для объектов каждого класса, так как `TObject` является общим классом-предком каждого класса. Вот как мы можем использовать эти методы для доступа к информации о классе:

```
procedure TSenderForm.ShowSender(Sender: TObject);
begin
  Memo1.Lines.Add ('Class Name: ' + Sender.ClassName);

  if Sender.ClassParent <> nil then
    Memo1.Lines.Add ('Parent Class: ' + Sender.ClassParent.ClassName);

  Memo1.Lines.Add ('Instance Size: ' + IntToStr (Sender.InstanceSize));
```

Код проверяет, является ли `ClassParent nil` в случае, если вы на самом деле используете экземпляр типа `TObject`, который не имеет базового типа. Вы можете использовать другие методы

для выполнения проверок. Например, следующим кодом можно проверить, имеет ли объект `Sender` конкретный тип:

```
if Sender.ClassType = TButton then ...
```

С помощью этого теста можно также проверить, соответствует ли параметр `Sender` заданному объекту:

```
if Sender = Button1 then...
```

Вместо того, чтобы проверять на наличие определенного класса или объекта, вам, как правило, придется проверять совместимость типа объекта с данным классом; то есть, вам придется проверять, является ли класс объекта данным классом или одним из его подклассов. Это позволит узнать, можно ли работать с объектом с помощью методов, определенных для данного класса. Этот тест можно выполнить с помощью метода InheritsFrom, который также вызывается при использовании оператора `is`. Следующие два теста эквивалентны:

```
if Sender.InheritsFrom (TButton) then ...
if Sender is TButton then ...
```

Показ информации о классе

Имея ссылку на класс, вы можете добавить к его описанию (или отобразить информацию) список всех его базовых классов. В следующих фрагментах кода базовые классы `MyClass` добавляются в элемент управления ListBox:

```
ListParent.Items.Clear;
while MyClass.ClassParent <> nil do
begin
  MyClass := MyClass.ClassParent;
  ListParent.Items Add (MyClass.ClassName);
end;
```

Обратите внимание, что в основе цикла `while` лежит ссылка на класс, который проверяет отсутствие родительского класса (в этом случае текущий класс - `TObject`). В качестве альтернативы,

мы могли бы написать оператор `while` любым из следующих способов:

```
while not MyClass.ClassNameIs ('TObject') do... // slow, error prone
while MyClass <> TObject do... // fast, and readable
```

Виртуальные методы TObject

Хотя структура класса `TObject` оставалась достаточно стабильной с первых дней существования языка Object Pascal, в какой-то момент к нему добавились три чрезвычайно полезных виртуальных метода. Это методы, которые можно вызывать на любом объекте, как и любой другой метод `TObject`, но актуальность заключается в том, что это методы, которые вы должны переопределять и переопределять в ваших собственных классах.

примечание Если вы использовали .NET-фреймворк, то сразу узнаете, что эти методы являются частью класса `System.Object` библиотеки базовых классов C#. Аналогичные методы используются для базовых классов, доступных на Java, широко распространены в JavaScript и других языках. Происхождение некоторых из них, как и у `ToString`, можно проследить до Smalltalk, который считается первым ООП-языком.

Метод ToString

Виртуальная функция `ToString` является стандартом для возврата текстового представления (описания или даже сериализации) данного объекта. Реализация метода по умолчанию в классе `TObject` возвращает имя класса:

```
function TObject.ToString: string;
begin
  Result := ClassName;
end;
```

Конечно, в этом немного пользы. Теоретически, каждый класс должен предоставлять пользователю способ описания себя, например, когда объект добавляется в визуальный список.

Некоторые классы библиотеки времени исполнения переопределяют виртуальную функцию `ToString`, такие как `TStringBuilder`, `TStringWriter`, и класс `Exception` для возврата сообщений в списке исключений (как описано в разделе "Механизм InnerException" Главы 9).

Наличие стандартного способа возврата строкового представления любого объекта - довольно интересная идея, и я рекомендую вам воспользоваться этой основной функцией класса `TObject`, рассматривая ее как функцию языка.

примечание Обратите внимание, что метод `ToString` "семантически перегружает" символ "parse token String" или `toString`, определенный в модуле `Classes`. По этой причине этот символ обозначается как `Classes.toString`.

Метод Equals

Виртуальная функция `Equals` является обозначением для проверки, не имеют ли два объекта одно и то же логическое значение, это отличается от проверки, не ссылаются ли две переменные на один и тот же объект в памяти, чего можно добиться с помощью оператора =. Однако, и это часто смущает, реализация по умолчанию делает именно это, за неимением лучшего способа:

```
function TObject.Equals(Obj: TObject): Boolean;
begin
  Result := Obj = Self;
end;
```

Пример использования данного метода (с правильным переопределением) приведен в классе `TStrings`, в котором метод `Equals` сравнивает по очереди количество строк в списке и содержание реальных строк.

Раздел библиотеки, в котором эта техника в значительной степени используется, — это поддержка дженериков, в частности, в модулях `Generics.Default` и `Generics.Collections`. Для библиотеки или фреймворка всегда важно определить понятие

"эквивалентности значения" объекта отдельно от идентичности объекта. Наличие стандартного механизма сравнения объектов "по значению" является большим преимуществом.

Метод GetHashCode

Виртуальная функция `GetHashCode` — это еще один метод, заимствованный из .NET-фреймворка, чтобы каждый класс мог вычислить хэш-код для своих объектов. Код по умолчанию возвращает кажущееся случайным значение, адрес самого объекта:

```
function TObject.GetHashCode: Integer;
begin
  Result := Integer(self);
end;
```

примечание Поскольку адреса создаваемых объектов, как правило, берутся из ограниченного набора областей кучи, распределение этих чисел неравномерно, и это может отрицательно сказаться на алгоритме хеширования. Настоятельно рекомендуется настроить этот метод создания хэша на основе логических значений с хорошим распределением хэшей на основе данных внутри объекта, а не по его адресу. Словари и другие структуры данных полагаются на хэш-значения, улучшение хэш-распределения может привести к значительному повышению производительности.

Виртуальная функция `GetHashCode` используется некоторыми классами коллекции, которые поддерживают хэш-таблицы и как способ оптимизации некоторого кода, например, `TDictionary <T>`.

Использование виртуальных методов TObject

Вот пример, основанный на некоторых виртуальных методах `TObject`. В примере есть класс, который переопределяет два из этих методов:

```
type
  TAnyObject = class
  private
    FValue: Integer;
```

```
    FName: string;
  public
    constructor Create (AName: string; AValue: Integer);
    function Equals(Obj: TObject): Boolean; override;
    function ToString: string; override;
  end;
```

В реализации трех методов мне просто пришлось изменить вызов на `GetType` на `ClassType`:

```
constructor TAnyObject.Create(AName: string; AValue: Integer);
begin
  inherited Create;
  FName := AName;
  FValue := AValue;
end;

function TAnyObject.Equals(Obj: TObject): Boolean;
begin
  Result := (Obj.ClassType = self.ClassType) and
    ((Obj as TAnyObject).Value = self.Value);
end;

function TAnyObject.ToString: string;
begin
  Result := Name;
end;
```

Обратите внимание, что объекты считаются равными, если они относятся к одному и тому же точному классу и их значения совпадают, в то время как их строковое представление включает в себя только поле `имени`. В начале программы создаются некоторые объекты этого класса:

```
procedure TFormSystemObject.FormCreate(Sender: TObject);
begin
  Ao1 := TAnyObject.Create ('Ao1', 10);
  Ao2 := TAnyObject.Create ('Ao2 or Ao3', 20);
  Ao3 := ao2;
  Ao4 := TAnyObject.Create ('Ao4', 20);
  ...
```

Обратите внимание, что две ссылки (`Ao2` и `Ao3`) указывают на один и тот же объект в памяти, и что последний объект (`Ao4`) имеет одно и то же числовое значение. Программа имеет пользовательский интерфейс, который позволяет пользователю выбрать любые два элемента и сравнить

выделенные объекты, как с помощью `Equals`, так и путем прямого сравнения ссылок.

Вот некоторые результаты:

```
Comparing Ao1 and Ao4
Equals: False
Reference = False

Comparing Ao2 and Ao3
Equals: True
Reference = True

Comparing Ao3 and Ao4
Equals: True
Reference = False
```

В программе есть еще одна кнопка, с помощью которой можно протестировать некоторые из этих методов для самой кнопки:

```
var
  Btn2: TButton;
begin
  Btn2 := BtnTest;
  Log ('Equals: ' +
    BoolToStr (BtnTest.Equals (Btn2), True));
  Log ('Reference = ' +
    BoolToStr (BtnTest = Btn2, True));
  Log ('GetHashCode: ' +
    IntToStr (BtnTest.GetHashCode));
  Log ('ToString: ' + BtnTest.ToString);
end;
```

На выходе получается следующий результат (со значением хэша, которое меняется при выполнении):

```
Equals: True
Reference = True
GetHashCode: 28253904
ToString: TButton
```

Краткое описание класса

Резюмируя, привожу полный интерфейс класса `TObject` в последней версии компилятора (при этом пропущено большинство `IFDEF` и низкоуровневых перегрузок, а также приватные и защищенные секции):

```
type
  TObject = class
  public
    constructor Create;
    procedure Free;
    procedure DisposeOf;
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    function ClassType: TClass; inline;
    class function ClassName: string;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer; inline;
    class function InstanceSize: Integer; inline;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: string): Pointer;
    class function MethodName(Address: Pointer): string;
    class function QualifiedClassName: string;
    function FieldAddress(const Name: string): Pointer;
    function GetInterface(const IID: TGUID; out Obj): Boolean;
    class function GetInterfaceEntry(
      const IID: TGUID): PInterfaceEntry;
    class function GetInterfaceTable: PInterfaceTable;
    class function UnitName: string;
    class function UnitScope: string;
    function Equals(Obj: TObject): Boolean; virtual;
    function GetHashCode: Integer; virtual;
    function ToString: string; virtual;
    function SafeCallException(ExceptObject: TObject;
      ExceptAddr: Pointer): HResult; virtual;
    procedure AfterConstruction; virtual;
    procedure BeforeDestruction; virtual;
    procedure Dispatch(var Message); virtual;
    procedure DefaultHandler(var Message); virtual;
    class function NewInstance: TObject; virtual;
    procedure FreeInstance; virtual;
    destructor Destroy; virtual;
  public
    property Disposed: Boolean read GetDisposed;
  end;
```

Юникод и имена классов

Перегруженные методы, такие как `MethodAddress` и `FieldAddress`, могут принимать как UnicodeString (UTF-16, как обычно), так и `ShortString`, который обрабатывается как строка UTF-8. На самом деле, версии, берущие обычную Unicode строку, преобразуют их вызовом функции `UTF8EncodeToShortString`:

```
function TObject.FieldAddress(const Name: string): Pointer;
```

```
begin
  Result := FieldAddress(UTF8EncodeToShortString(Name));
end;
```

Так как поддержка Unicode была введена в язык, имена классов в Object Pascal внутренне используют представление `ShortString` (массив однобайтовых символов), но в кодировке UTF-8, а не в традиционной ANSI кодировке типа `ShortString`. Это происходит как на уровне `TObject`, так и на уровне RTTI.

Например, метод `ClassName` реализован с некоторым действительно низкоуровневым кодом следующим образом:

```
class function TObject.ClassName: string;
begin
  Result := UTF8ToString (
    PShortString (PPointer (
      Integer(Self) + vmtClassName)^)^);
end;
```

Аналогично в модуле `TypInfo` все функции, обращающиеся к именам классов, преобразуют внутренние представления UTF-8 ShortString в UnicodeString. Что-то подобное происходит и с именами свойств.

Unit System

В то время как класс `TObject` явно играет фундаментальную роль для языка, из-за чего очень трудно сказать, является ли он частью языка или библиотеки исполнения, в модуле `System` есть другие классы низкого уровня, которые составляют фундаментальную и интегрированную часть поддержки компилятора. Большая часть содержимого этого модуля, однако, состоит из низкоуровневых структур данных, простых структур записей, функций и процедур, а также нескольких классов.

Здесь я сосредоточусь в основном на классах, но неоспоримо, что многие другие функции в модуле `System` являются ключевыми для языка. Например, системный модуль

определяет так называемые "intrinsic" функции, которые не имеют реального кода, но разрешаются непосредственно компилятором. Примером является `SizeOf`, который компилятор напрямую заменяет на фактический размер структуры данных, переданной в качестве параметра.

Вы можете получить представление об особой роли модуля `System`, прочитав комментарий, добавленный к началу (в основном для того, чтобы объяснить, почему просмотр системных символов приводит к этому юниту... но не к тому символу, который вы искали):

```
{ Predefined constants, types, procedures, }
{ and functions (such as True, Integer, or }
{ Writeln) do not have actual declarations.}
{ Instead they are built into the compiler }
{ and are treated as if they were declared }
{ at the beginning of the System unit.     }
```

Чтение исходного кода этого модуля может быть довольно утомительным, в том числе и потому, что здесь можно найти некоторый код нижнего уровня всей библиотеки времени исполнения. Поэтому я решил описать только очень ограниченный выбор его содержания.

Избранные системные типы

Как упоминалось выше, `модуль System` определяет основные типы данных и множество типов псевдонимов для различных числовых типов, других порядковых типов и строк. Существуют и другие базовые типы данных (состоящие из перечислений, записей и псевдонимов строгих типов), используемые системой на низком уровне, на которые стоит обратить внимание:

- TVisibilityClasses - это перечисление, используемое для настройки видимости RTTI (см. главу 16 для более подробной информации).

- TGUID - это запись, используемая для представления GUID на Windows, а также на всех других поддерживаемых операционных системах.
- TMethod - это основная запись, представляющая структуру, используемую для обработчика событий, с указателем на адрес метода и один на текущий объект (кратко упоминается в главе 10).
- TMonitor - это запись, реализующая механизм синхронизации потоков (называемый "монитор"), придуманный C.A.R Hoare и Per Brinch Hansen и подробно описанный в Википедии под голосом "Monitor synchronization". Это основная функция поддержки синхронизации потоков в самом языке, так как информация `TMonitor` прикрепляется к любому объекту в системе.
- TDateTime - это сильно типизированный псевдоним типа Double, используемый для хранения информации о дате (в интегральной части значения) и времени (в десятичной части). Другие псевдонимы включают в себя типы `TDate` и `TTime`. Эти типы были рассмотрены в Главе 2.
- THandle - это псевдоним числовых типов, используемый для представления ссылки на объект операционной системы, обычно называемый "handle" (как минимум на жаргоне Windows API).
- TMemoryManagerEx - это запись, содержащая операции с памятью ядра, которая позволяет заменить системный менеджер памяти на пользовательский (это более новая версия `TMemoryManager`), который все еще доступен для обратной совместимости.
- THeapStatus - это запись с информацией о состоянии памяти кучи, кратко упомянутая в Главе 13.
- TTextLineBreakStyle - это перечисление, указывающее на стиль разрыва строки для текстовых файлов в данной операционной системе. Глобальная переменная этого типа DefaultTextLineBreakStyle хранит текущую информацию, используемую многими системными библиотеками. Аналогично

константа sLineBreak выражает ту же информацию, что и строковое значение.

Интерфейсы в System Unit

Существует несколько типов интерфейсов (и несколько классов, реализующих интерфейсы на уровне ядра), которые являются частью модуля `System`, на которые стоит обратить внимание. Интерфейсы были рассмотрены в главе 11. Вот наиболее релевантные типы интерфейсов в модуле `System`:

- IInterface является базовым типом интерфейса, от которого наследуют все остальные интерфейсы, и имеет ту же фундаментальную роль, что и `TObject` для классов.
- IInvokable и IDispatch - интерфейсы, поддерживающие формы динамического вызова (частично связанные с реализацией Windows COM).
- Поддержка счетчиков и операции сравнения определяются следующими интерфейсами: `IEnumerator`, `IEnumerable`, `IEnumerator<T>`, `IEnumerable<T>, IComparable`, `IComparable<T> и IEquatable`<T>.

Существует также несколько основных классов, которые предлагают базовую реализацию интерфейсов. Вы часто наследуете от этих классов при реализации интерфейса, о чем также говорилось в Гл. 11:

- TInterfacedObject, класс, который имеет базовую реализацию подсчета ссылок и проверки идентификатора интерфейса
- TAggregatedObject и TContainedObject, два класса, которые предлагают специальную реализацию для агрегированного объекта и синтаксиса реализаций.

Избранные системные подпрограммы

Количество внутренних и стандартных процедур и функций в модуле `System` достаточно велико, но большинство из них используется нечасто. Здесь очень ограниченный выбор основных функций и процедур, о которых должен знать каждый разработчик Object Pascal:

- Move - это основная операция копирования памяти в системе, просто копирование заданного количества байт из ячейки памяти в другую (очень мощная, очень быстрая, но немного опасная).
- Функции ParamCount и ParamStr могут использоваться для обработки параметров командной строки приложения (и фактически работают и на GUI-системах, таких как Windows и Mac).
- Random и Randomize - это две классические функции (вероятно, исходящие из BASIC), предоставляющие случайные значения (но псевдослучайные только в том случае, если вы не забыли вызвать `Randomize`, в противном случае вы получаете одну и ту же последовательность при каждом выполнении).
- Значительное количество основных математических функций, полностью пропущенных здесь.
- Многие функции обработки и преобразования строк (между UTF-16 Unicode, UTF-8, ANSI и другими форматами строк), некоторые из которых специфичны для платформы

примечание Некоторые из этих функций имеют косвенное определение. Другими словами, функция на самом деле является указателем на реальную функцию, так что исходное поведение системы может быть динамически заменено в коде во время выполнения. (Если вы, конечно, знаете, что делаете, так как это может быть хорошим способом замусорить память вашего приложения).

Предопределенные атрибуты RTTI

Последняя группа типов данных, которую я хочу упомянуть в этой главе, относится к атрибутам, дополнительная информация RTTI, которую вы можете прикрепить к любому символу языка. Эта тема была рассмотрена в Главе 16, но там я не упомянул о предопределенных атрибутах в системе.

Вот классы атрибутов, определенные в модуле `System`:

- TCustomAttribute является базовым классом для всех пользовательских атрибутов. Это базовый класс, от которого необходимо наследовать атрибуты (и это единственный способ, которым компилятор идентифицирует класс как атрибут, так как нет специального синтаксиса декларации).
- WeakAttribute используется для указания слабых ссылок для ссылок на интерфейс (см. главу 13).
- UnsafeAttribute используется для отключения подсчета ссылок для ссылок интерфейса (также рассматривается в главе 13).
- Очевидно, что для ссылочных значений используется RefAttribute.
- VolatileAttribute указывает на *волатильные* переменные, которые могут быть модифицированы извне и не должны быть оптимизированы компилятором
- StoredAttribute - это альтернативный способ выражения `stored` флага свойств.
- HPPGENAttribute управляет генерацией файла интерфейса C++ (HPP)
- HFAAttribute может использоваться для тонкой настройки передачи параметров 64-битного процессора ARM, управляя гомогенными агрегатами с плавающей запятой (HFA).

В модуле `System` есть кое-что еще, но это для разработчиков-экспертов. Я лучше перейду к последней главе, где я коснусь модуля `Classes` и некоторых возможностей RTL.

18: Другие базовые классы RTL

Если класс `TObject` и модуль `System` можно рассматривать как структурную часть языка, как необходимое самому компилятору для сборки любого приложения, то всё остальное в библиотеке runtime можно считать необязательными расширениями к ядру системы.

RTL имеет очень большую коллекцию системных функций, включающую наиболее распространенные стандартные операции и частично восходящую к дням Turbo Pascal, предшествовавшим языку Object Pascal. Многие модули RTL представляют собой коллекции функций и процедур, включая основные утилиты (`SysUtils`), математические функции (`Math`), строковые операции (`StringUtils`), обработку даты и времени (`DateUtils`) и многие другие.

В этой книге я не планирую углубляться в эту более традиционную часть RTL, а сосредоточусь на базовых классах, которые являются основой библиотек визуальных

компонентов, используемых в Object Pascal (VCL и FireMonkey), а также в других подсистемах. Класс `TComponent`, например, определяет понятие "компонентно-ориентированной" архитектуры. Он также является фундаментальным для управления памятью и другими базовыми возможностями. Класс `TPersistent` является ключевым для потокового представления компонентов.

Есть много других классов, на которые мы могли бы обратить внимание, так как RTL чрезвычайно велика и включает в себя файловую систему, поддержку параллельного программирования, библиотеку параллельного программирования, построение строк, много различных типов коллекций и классов контейнеров, основные геометрические структуры (например, точки и прямоугольники), основные математические структуры (например, векторы и матрицы), и многое, многое другое.

Учитывая, что в центре внимания книги действительно находится язык Object Pascal, а не руководство по библиотекам, здесь я остановлюсь только на нескольких избранных классах, выбранных либо из-за их ключевой роли, либо потому, что они были введены в последние годы и в значительной степени игнорируются разработчиками.

Unit Classes

Модуль в основе библиотеки классов Object Pascal RTL (а также визуальных библиотек) называется соответственно: System.Classes. Этот модуль содержит большую коллекцию классов, большей частью неупорядоченную, без специфической направленности. Стоит вкратце взглянуть на наиболее важные

из них, а затем провести углубленный анализ наиболее важных.

Классы в модуле Classes

Итак, вот краткий список (примерно половина классов, фактически определенных в модуле):

- TList - это основной список указателей, который часто приспосабливается как нетипизированный список. В целом, рекомендуется использовать `TList`<T>, как описано в Гл. 14.
- TInterfaceList - это потокобезопасный список интерфейсов, реализующий `IInterfaceList`, на который стоит взглянуть (но здесь он не рассматривается).
- TBits – это очень простой класс для манипулирования отдельными битами в числе или каком-либо другом значении. Это гораздо более высокий уровень, чем манипулирование битами со сдвигами и двоичными операторами `or` и `and`.
- TPersistent - это фундаментальный класс (базовый класс `TComponent`), подробно рассмотренный в следующем разделе.
- `TCollectionItem` и TCollection - это классы, используемые для определения свойств коллекции, то есть свойств с массивом значений. Это важные классы для разработчиков компонентов (и косвенно при использовании компонентов), в меньшей степени для generic кода конечного пользователя.
- TStrings - это абстрактный список строк, в то время как TStringList - это реальная реализация базового класса `TStrings`, обеспечивающая хранение реальных строк. К каждому элементу также прикреплен объект, и это стандартный способ использования списков строк для пар строк имя/значение. Дополнительная информация об этом

классе приведена в разделе "Использование строковых списков" в конце этой главы.

- TStream - это абстрактный класс, представляющий любую последовательность байт с последовательным доступом, который может включать в себя множество различных вариантов хранения (память, файлы, строки, сокеты, BLOB-поля и многие другие). В модуле `Classes` определены многие специфические классы потоков, в том числе `THandleStream`, `TFileStream`, `TCustomMemoryStream`, `TMemoryStream`, `TBytesStream`, `TStringStream` и `TResourceStream`. Остальные специфические потоки объявляются в других модулях RTL. Введение в потоки можно прочитать в разделе "Введение в потоки" этой главы.
- Классы для низкоуровневого потокового воспроизведения компонентов, такие как `TFiler`, `TReader`, `TWriter` и `TParser`, в основном используются авторами компонентов... и даже ими не так часто.
- Класс TThread, определяющий поддержку платформонезависимых многопоточных приложений. Также это класс для асинхронных операций, называемый `TBaseAsyncResult`.
- Классы для реализации шаблона Наблюдатель (используются, например, в visual live bindings), включая `TObservers`, `TLinkObservers` и `TObserverMapping`.
- Классы для определенных пользовательских атрибутов, таких как `DefaultAttribute`, `NoDefaultAttribute`, `StoredAttribute` и `ObservableMemberAttribute`.
- Фундаментальный класс TComponent, базовый класс всех визуальных и невизуальных компонентов как в VCL, так и в FireMonkey, подробно рассматривается далее в этой главе.
- Классы поддержки действий и списков действий (действия — это абстракция "команд", внутренними или выполняемых элементами пользовательского интерфейса), в том числе TBasicAction и `TBasicActionLink`.

- Класс, представляющий собой невизуальный контейнер компонентов, TDataModule.
- Высокоуровневые интерфейсы для файловых и потоковых операций, включая `TTextReader` и `TTextWriter`, `TBinaryReader` и `TBinaryWriter`, `TStringReader` и `TStringWriter`, `TStreamReader` и `TStreamWriter`. Эти классы также рассмотрены в данной главе.

Класс TPersistent

Класс `TObject` имеет очень важный подкласс, одну из основ всей библиотеки, называемую `TPersistent`. Если посмотреть на методы класса, то его важность может вызвать удивление... так как класс делает очень мало. Одним из ключевых элементов класса `TPersistent` является то, что он определяется с помощью специальной опции компилятора {M+}, роль которой заключается в включении ключевого слова `published`, описанного в Главе 10.

Это ключевое слово имеет фундаментальную значение для потоковых свойств, и это объясняет название класса. Первоначально в качестве типа данных опубликованных свойств могли использоваться только классы, наследующие от `TPersistent`. Расширение RTTI в более поздних версиях компилятора Object Pascal немного изменило картину, но роль ключевого слова `published` и опции компилятора {$M+} все еще есть.

примечание Используя сегодняшний компилятор, если добавить ключевое слово `published` к классу, который не наследуется от `TPersistent` и не имеет флага компилятора {$M+}, то система все равно добавит соответствующую поддержку, сообщив об этом с предупреждением.

Какова специфическая роль класса `TPersistent` в иерархии? Во-первых, он служит базовым классом для класса `TComponent`,

который я представлю в следующем разделе. Во-вторых, он используется в качестве базового класса для типов данных, используемых для значений свойств, чтобы эти свойства и их внутренняя структура могли корректно передаваться через потоки. Примерами являются классы, представляющие собой список строк, растровых изображений, шрифтов и других объектов.

Если наиболее актуальной особенностью класса `TPersistent` является "активация" ключевого слова `published`, то в нем еще есть пара интересных методов, заслуживающих внимания. Первый - метод Assign, который используется для копирования данных объекта из одного экземпляра в другой (глубокая копия, а не копия ссылок). Это особенность, которую каждый персистентный класс, используемый для значений свойств, должен реализовывать вручную (так как в языке нет автоматической операции глубокой копии). Вторая - обратная операция, `AssignTo`, которая защищена protected. Эти два метода, а также несколько других, доступных в классе, используются в основном авторами компонентов, а не разработчиками приложений.

Класс TComponent

Класс `TComponent` является краеугольным камнем библиотек компонентов, которые чаще всего используются совместно с компиляторами Object Pascal. Концепция компонента в основном состоит в том, что он имеет некоторое дополнительное поведение во время проектирования, специфические возможности потоковой передачи (для того, чтобы конфигурация времени проектирования могла быть сохранена и восстановлена в работающем приложении), а

также модель PME (property-method-event), которую мы обсуждали в главе 10.

Данный класс определяет значительное количество стандартных моделей поведения и функций, вводит собственную модель памяти, основанную на концепции владения объектами, уведомления о кросс-компонентах и многое другое. Не делая полного анализа всех свойств и методов, безусловно, стоит сосредоточиться на некоторых ключевых особенностях класса `TComponent` из-за его центральной роли в RTL.

Еще одной критичной особенностью класса `TComponent` является введение виртуального конструктора `Create`, что критично для возможности создания объекта из ссылки на класс при одновременном обращении к конкретному коду конструктора класса. Мы коснулись этого в Главе 12, но это особенность языка Object Pascal, достойная понимания.

Владение компонентом

Механизм собственности (владения) является ключевым элементом класса `TComponent`. Если компонент создается с компонентом-обладателем (переданным в качестве параметра в его виртуальный конструктор), то этот компонент-обладатель становится ответственным за уничтожение компонента-собственности. Короче говоря, каждый компонент имеет не только ссылку на своего владельца (свойство Owner), но и список компонентов, которыми он владеет (свойство Components array), и их число (свойство ComponentCount).

По умолчанию, когда вы бросаете компонент в конструктор (форму, рамку или модуль данных), конструктор считается владельцем компонента. При создании компонента в коде, необходимо указать его владельца или передать nil (в этом

случае вы сами будете отвечать за освобождение компонента из памяти).

Вы можете использовать свойства `Components` и `ComponentCount`, чтобы перечислить компоненты, принадлежащие компоненту (в данном случае `aComp`), с таким же кодом:

```
var
  I: Integer;
begin
  for I := 0 to aComp.ComponentCount - 1 do
    aComp.Components[I].DoSomething;
```

Или используйте поддержку встроенного перечисления, написав:

```
var
  childComp: TComponent;
begin
  for childComp in aComp do
    childComp.DoSomething;
```

Когда компонент уничтожен, он удаляет себя из списка владельцев (если таковой имеется) и уничтожает все компоненты, которыми он владеет. Этот механизм очень важен для управления памятью в Object Pascal: поскольку сбор мусора отсутствует, владение может решить большинство проблем управления памятью, как мы частично видели в Гл. 13.

Как я уже упоминал, как правило, все компоненты в форме или модуле данных имеют форму или модуль данных в качестве владельца. Как только вы освобождаете форму или модуль данных, компоненты, которые они содержат, также уничтожаются. Это то, что происходит, когда компоненты создаются из потока.

Свойства компонентов

Помимо важнейшего механизма владения (который также включает в себя уведомления и другие функции, не описанные здесь) любой компонент имеет два опубликованных свойства:

- Name — это строка с именем компонента. Она используется для динамического поиска компонента (вызов метода FindComponent владельца) и для соединения компонента с полем формы, ссылающимся на него. Все компоненты, принадлежащие одному владельцу, должны иметь разные имена, но их имя также может быть пустым. Два коротких правила здесь: установите правильные имена компонентов для улучшения читабельности кода и никогда не меняйте имя компонента во время выполнения (если только вы действительно не знаете о возможных неприятных побочных эффектах).
- Tag - это значение NativeInt (в прошлом это был Integer), не используемое библиотекой, но доступное для подключения дополнительной информации к компоненту. Этот тип совместим с указателями и ссылками на объекты, которые часто хранятся в `Tag` компонента.

Потоковая передача компонентов

Механизм потоковой передачи, используемый как FireMonkey, так и VCL для создания файлов FMX или DFM, основывается на классе `TComponent`. Потоковый механизм Delphi сохраняет опубликованные свойства и события компонента и его подкомпонентов. Это представление, которое вы получаете в DFM или FMX файле, а также то, что вы получаете, если копируете и вставляете компонент из дизайнера в текстовый редактор.

Существуют методы получения той же информации во время выполнения, в том числе методы `WriteComponent` и `ReadComponent` класса `TStream`, а также методы `ReadComponentRes` и `WriteComponentRes` того же класса, плюс `ReadRootComponent` и `WriteRootComponent` специальных классов `TReader` и `TWriter`, помогающие работать с потоком компонентов. Эти операции обычно используют двоичное представление потоков форм: для преобразования

бинарного представления формы в текстовое можно использовать глобальную процедуру `ObjectResourceToText, a` для обратного преобразования - `ObjectTextToResource`.

Ключевым элементом является то, что поток не является полным набором опубликованных свойств компонента. Потоковый обмен включает в себя:

- Публикуемые свойства компонента со значением, отличным от их значения по умолчанию (другими словами, значения по умолчанию не сохраняются, чтобы уменьшить размер)
- Только опубликованные свойства, помеченные как сохраненные (по умолчанию). Свойство с сохраненным значением false (или функция, возвращающая false), не будет сохранено.
- Дополнительные записи, не соответствующие свойствам компонента, добавляемые во время выполнения путем переопределения метода `DefineProperties`.

При создании компонента из потокового файла происходит следующая последовательность:

- Вызывается виртуальный конструктор компонента Create (выполнение соответствующего кода инициализации).
- Свойства и события загружаются из потока (в случае возникновения событий происходит перенаправление имени метода на фактический адрес метода в памяти)
- Виртуальный метод Loaded вызывается для завершения загрузки (а компоненты могут выполнять дополнительную пользовательскую обработку, на этот раз со значениями свойств, уже загруженными из потока)

Современный доступ к

файлам

Object Pascal все еще имеет ключевые слова и основные механизмы языка для обработки файлов, которые были позаимствованы у своего предка - языка Pascal. В основном они были устаревшими, когда был представлен Object Pascal, и я не собираюсь затрагивать их в этой книге. Вместо этого в этом разделе я расскажу о паре современных техник обработки файлов, познакомив с модулем IOUtils, классами потоков, а также с классами читателей и писателей.

Модуль управления вводом/выводом

Модуль System.IOUtils является относительно недавним дополнением к библиотеке Run Time Library. Он определяет три записи для большинства методов класса: TDirectory, TPath и TFile. Хотя вполне очевидно, что `TDirectory` предназначена для просмотра папок и поиска своих файлов и подпапок, может быть не совсем понятно, в чем разница между `TPath` и `TFile`. Первый, `TPath`, используется для манипулирования именами файлов и каталогов, с методами извлечения диска, именами файлов без путей, расширений и т.п., а также для манипулирования UNC-путями. Запись `TFile` вместо этого позволяет проверять временные метки и атрибуты файла, а также манипулировать файлом, записывать в него или копировать его. Как обычно, проще посмотреть на примере. Прикладной проект IoFilesInFolder может извлечь все вложенные папки данной папки и захватить все файлы с заданным расширением, доступные в этой папке.

Извлечение Подкаталогов

Программа может заполнить поле списка списком папок под каталогом, используя метод GetDirectories записи `TDirectory`, передав в качестве параметра значение `TSearchOption.soAllDirectories`. В результате получится строковый массив, который можно перечислить:

```
procedure TFormIoFiles.BtnSubfoldersClick(Sender: TObject);
var
  PathList: TStringDynArray;
  StrPath: string;
begin
  if TDirectory.Exists (EdBaseFolder.Text) then
  begin
    ListBox1.Items.Clear;
    PathList := TDirectory.GetDirectories(EdBaseFolder.Text,
      TSearchOption.soAllDirectories, nil);
    for StrPath in PathList do
      ListBox1.Items.Add (StrPath);
  end;
end;
```

Поиск файлов

Вторая кнопка программы позволяет получить все файлы этих папок, сканируя каждую директорию вызовом GetFiles по заданной маске. Более сложную фильтрацию можно получить, передав анонимный метод типа TFilterPredicate перегруженной версии `GetFiles`.

В этом примере используется более простая фильтрация на основе масок и заполняется внутренний список строк. Элементы этого списка строк затем копируются в пользовательский интерфейс после удаления полного пути, сохраняя только имя файла. При вызове метода `GetDirectories` вы получаете только вложенные папки, но не текущую. Поэтому программа сначала ищет в текущей папке, а затем просматривает каждую вложенную папку:

```
procedure TFormIoFiles.BtnPasFilesClick(Sender: TObject);
var
```

```
  PathList, FilesList: TStringDynArray;
  StrPath, StrFile: string;
begin
  if TDirectory.Exists (EdBaseFolder.Text) then
  begin
    // clean up
    ListBox1.Items.Clear;

    // search in the given folder
    FilesList := TDirectory.GetFiles (EdBaseFolder.Text, '*.pas');
    for StrFile in FilesList do
      SFilesList.Add(StrFile);

    // search in all subfolders
    PathList := TDirectory.GetDirectories(EdBaseFolder.Text,
      TSearchOption.soAllDirectories, nil);
    for StrPath in PathList do
    begin
      FilesList := TDirectory.GetFiles (StrPath, '*.pas');
      for StrFile in FilesList do
        SFilesList.Add(StrFile);
    end;

    // now copy the file names only (no path) to a listbox
    for StrFile in SFilesList do
      ListBox1.Items.Add (TPath.GetFileName(StrFile));
  end;
end;
```

В последних строках функция `GetFileName` в `TPath` используется для извлечения имени файла из полного пути к нему. Запись TPath имеет несколько других интересных методов, в том числе `GetTempFileName`, `GetRandomFileName`, метод объединения путей, несколько методов для проверки того, являются ли они правильными или содержат ли они незаконные символы, и многое другое.

Введение в потоки

Если модуль `IOUtils` предназначен для поиска и манипулирования файлами, то при чтении или записи файла (или любой другой подобной структуры последовательного доступа к данным) можно использовать класс TStream и его многочисленные классы-потомки. Абстрактный класс `TStream`

имеет всего несколько свойств (Size и Position) вместе с базовым интерфейсом, к которому имеют общий доступ все классы потока, с основными методами Read и Write. Понятие, выраженное этим классом, - последовательный доступ. Каждый раз, когда вы читаете и записываете некоторое количество байт, текущая позиция увеличивается на это число. Для большинства потоков можно перемещать позицию назад, но могут быть и однонаправленные потоки.

Общие классы потоков

Как я упоминал ранее, модуль `Classes` определяет несколько конкретных классов потоков и включает в себя следующие:

- `THandleStream` определяет поток файлов диска, на который ссылаются через файловый хендлер.
- TFileStream определяет поток файлов на диске, на который ссылаются по имени файла.
- TBufferedFileStream - это оптимизированный поток дисковых файлов, который использует буфер памяти для дополнительной производительности. Этот класс потоков был представлен в Delphi 10.1 Berlin.
- TMemoryStream определяет поток данных в памяти, к которому также можно получить доступ с помощью указателя.
- TBytesStream представляет собой поток байт в памяти, к которому также можно получить доступ как к массиву байт
- TStringStream связывает поток со строкой в памяти.
- TResourceStream определяет поток, который может читать данные ресурса, связанные с исполняемым файлом приложения.

Использование Потоков

Создание и использование потока может быть таким же простым, как создание переменной определенного типа и вызов методов компонента для загрузки содержимого из файла. Например, при наличии потока и компонента memo можно написать:

```
AStream := TFileStream.Create (FileName, fmOpenRead);
Memo1.Lines.LoadFromStream (AStream);
```

Как видно из этого кода, метод `Create` для файловых потоков имеет два параметра: имя файла и некоторый флаг, указывающий на запрашиваемый режим доступа. Как я уже упоминал, потоки поддерживают операции чтения и записи, но это достаточно низкий уровень, поэтому я бы лучше рекомендовал использовать классы чтения и записи, о которых мы поговорим в следующем разделе. Прямое использование потока — это комплексные операции, такие как загрузка всего потока в приведенный выше фрагмент кода, или копирование одного потока в другой:

```
procedure CopyFile (SourceName, TargetName: String);
var
  Stream1, Stream2: TFileStream;
begin
  Stream1 := TFileStream.Create (SourceName, fmOpenRead);
  try
    Stream2 := TFileStream.Create (TargetName,
      fmOpenWrite or fmCreate);
    try
      Stream2.CopyFrom (Stream1, Stream1.Size);
    finally
      Stream2.Free;
    end
  finally
    Stream1.Free;
  end
end;
```

Использование Reader и Writer

Очень хороший подход для записи и чтения из потоков — это использование классов чтения и записи, которые являются частью RTL. Существует шесть классов чтения и записи, определенных в модуле Classes:

- TStringReader и TStringWriter работают со строкой в памяти (напрямую или с помощью `TStringBuilder`)
- TStreamReader и TStreamWriter работают с общим потоком (файловый поток, поток памяти и т.д.).
- `TBinaryReader` и `TBinaryWriter` работают с двоичными данными, а не с текстом.

Каждый из *читателей (readers)* текста реализует несколько основных приемов чтения:

```
function Read: Integer; overload;
function ReadLine: string;
function ReadToEnd: string;
```

Каждый из *писателей (writers)* текстов имеет два набора перегруженных операций без (`Write`) и с (`WriteLine`) разделителем конечных строк. Вот первый набор:

```
procedure Write(Value: Boolean); overload;
procedure Write(Value: Char); overload;
procedure Write(const Value: TCharArray); overload;
procedure Write(Value: Double); overload;
procedure Write(Value: Integer); overload;
procedure Write(Value: Int64); overload;
procedure Write(Value: TObject); overload;
procedure Write(Value: Single); overload;
procedure Write(const Value: string); overload;
procedure Write(Value: Cardinal); overload;
procedure Write(Value: UInt64); overload;
procedure Write(const Format: string; Args: array of const); overload;
procedure Write(Value: TCharArray; Index, Count: Integer); overload;
```

Читатели и писатели текста

Для записи в поток класс TStreamWriter использует поток или создает его, используя имя файла, атрибут append/create и кодировку Unicode, переданную в качестве параметров.

Так что мы можем писать, как я делал это в проекте приложения ReaderWriter:

```
var
  Sw: TStreamWriter;
begin
  Sw := TStreamWriter.Create('test.txt',
    False, TEncoding.UTF8);
  try
    Sw.WriteLine ('Hello, world');
    Sw.WriteLine ('Have a nice day');
    Sw.WriteLine (Left);
  finally
    Sw.Free;
  end;
```

Для чтения TStreamReader можно опять поработать с потоком или файлом (в этом случае он сможет определить кодировку из UTF BOM-маркера):

```
var
  SR: TStreamReader;
begin
  SR := TStreamReader.Create('test.txt', True);
  try
    while not SR.EndOfStream do
      Memo1.Lines.Add (SR.ReadLine);
  finally
    SR.Free;
  end;
```

Обратите внимание, как можно проверить статус EndOfStream. По сравнению с прямым использованием текстовых потоков (или даже строк), эти классы особенно удобны в использовании и обеспечивают хорошую производительность.

Двоичный Reader и Writer

Классы TBinaryReader и TBinaryWriter предназначены для управления двоичными данными, а не текстовыми файлами.

Эти классы обычно инкапсулируют поток (поток файлов или любой поток внутри памяти, включая сокеты и BLOB-поля таблиц базы данных) и имеют перегруженные методы чтения и записи.

В качестве (довольно простого) примера я написал проект приложения BinaryFiles. В своей первой части эта программа записывает в файл пару двоичных элементов (значение свойства и текущее время) и считывает их обратно, присваивая значение свойству:

```
procedure TFormBinary.BtnWriteClick(Sender: TObject);
var
  BW: TBinaryWriter;
begin
  BW := TBinaryWriter.Create('test.data', False);
  try
    BW.Write(Left);
    BW.Write(Now);
    Log ('File size: ' + IntToStr (BW .BaseStream.Size));
  finally
    BW.Free;
  end;
end;

procedure TFormBinary.BtnReadClick(Sender: TObject);
var
  br: TBinaryReader;
  time: TDateTime;
begin
  br := TBinaryReader.Create('test.data');
  try
    Left := br.ReadInt32;
    Log ('Left read: ' + IntToStr (Left));
    time := br.ReadDouble;
    Log ('Time read: ' + TimeToStr (time));
  finally
    br.Free;
  end;
end;
```

Ключевое правило при использовании этих классов чтения и записи заключается в том, что вы должны читать данные в том же порядке, в котором вы их записали, иначе вы полностью испортите данные. Фактически, сохраняются только двоичные данные отдельных полей, без информации о самом поле. Ничто

не мешает вам упорядочить данные и метаданные в файле, например, сохраняя размер следующей структуры данных перед ее фактическим значением или токена, ссылающегося на поле.

Построение строк и списков строк

После просмотра файлов и потоков, я хочу потратить немного времени, сосредоточившись на способах манипулирования строками и списками строк. Это очень распространенные операции, и на них сконцентрирован богатый набор функций RTL. Здесь я представлю только некоторые из них.

Класс TStringBuilder

Я уже упоминал в главе 6, что в отличие от других языков, Object Pascal имеет полную поддержку прямого конкатенирования строк, что на самом деле является довольно быстрой операцией. RTL языка, однако, также включает в себя специальный класс для сборки строки из фрагментов различных типов данных, называемый TStringBuilder.

В качестве простого примера использования класса `TStringBuilder` рассмотрим следующий фрагмент кода:

```
var
  SBuilder: TStringBuilder;
  Str1: string;
begin
  SBuilder := TStringBuilder.Create;
  SBuilder.Append(12);
  SBuilder.Append('hello');
  Str1 := SBuilder.ToString;
```

```
  SBuilder.Free;
end;
```

Обратите внимание, что мы должны создать и уничтожить этот объект `TStringBuilder`. Еще один момент, который вы можете заметить выше, это то, что существует много различных типов данных, которые вы можете передавать в качестве параметров в функцию `Append`.

Другие интересные методы класса TStringBuilder включают `AppendFormat` (с внутренним вызовом `Format`) и `AppendLine`, который добавляет значение sLineBreak. Наряду с Append существует соответствующий ряд методов Insert перегруженных, а также метод Remove и несколько методов Replace.

примечание Класс `TStringBuilder` имеет красивый интерфейс и предлагает хорошее удобство использования. С точки зрения производительности, однако, использование стандартных функций конкатенации и форматирования строк может дать лучшие результаты, в отличие от других языков программирования, которые определяют неизменяемые строки и имеют очень плохую производительность в случае чистой конкатенации строк.

Цепочки методов в StringBuilder

Особенностью класса TStringBuilder является то, что большинство методов — это функции, возвращающие текущий объект, к которому они были применены.

Эта идиома кодирования открывает возможность цепочки методов, то есть вызова метода на объекте, возвращенном предыдущим. Вместо того, чтобы писать:

```
  SBuilder.Append(12);
  SBuilder.AppendLine;
  SBuilder.Append('hello');
```

можно написать строку:

```
  SBuilder.Append(12).AppendLine.Append('hello');
```

которая также может быть отформатирована как

```
SBuilder.
    Append(12).
```

```
AppendLine.
Append('hello');
```

Мне больше нравится этот синтаксис, чем оригинальный, но я знаю, что это просто синтаксический сахар, и некоторые люди предпочитают оригинальную версию с объектом, прописанным в каждой строке. В любом случае, имейте в виду, что различные вызовы `Append` не возвращают новые объекты (так что никаких потенциальных утечек памяти), а точно тот же объект, к которому вы применяете методы.

Использование списков строк

Списки строк являются очень распространенной абстракцией, используемой многими визуальными компонентами, но также используются как способ манипулирования текстом, состоящим из отдельных строк. Существует два основных класса для обработки списков строк:

- TStrings - это абстрактный класс для представления всех форм строковых списков, независимо от их реализации в хранилище. Данный класс определяет абстрактный список строк. По этой причине объекты `TStrings` используются только в качестве свойств компонентов, способных хранить сами строки.
- TStringList, подкласс `TStrings`, определяет список строк со своим хранилищем. С помощью этого класса можно определить список строк в программе.

Два класса списков строк также имеют готовые методы хранения или загрузки их содержимого в или из текстового файла, SaveToFile и LoadFromFile (которые полностью поддерживают Юникод). Для перебора списка можно использовать простой оператор `for`, основанный на его индексе, как если бы список был массивом, или `for-in c` перечислителем.

Библиотека RTL на самом деле большая.

В RTL есть гораздо больше возможностей, которые можно использовать вместе с компиляторами Object Pascal, охватывая множество основных возможностей для разработки на нескольких операционных системах. Подробное описание всей библиотеки Run-Time Library легко заполнит еще одну книгу того же размера, что и эта.

Если рассматривать только основную часть библиотеки, то есть пространство имён " *System* ", то оно включает в себя следующие модули (из которых я удалил несколько редко используемых):

- System.Actions включает в себя основную поддержку архитектуры действий, которая обеспечивает способ представления пользовательских команд, подключенных, но абстрагированных, с уровня пользовательского интерфейса.
- System.AnsiStrings имеет старые функции для обработки строк Ansi (только в Windows), рассмотренные в Главе 6.
- System.Character имеет встроенные помощники типа для символов Юникода (типа char), уже рассмотренные в Главе 3.
- System.Classes предоставляет основные классы системы и является единицей, о которой я подробно рассказал в первой части этой главы.
- System.Contnrs включает в себя старые, не generic, контейнерные классы, такие как список объектов, словарь, очередь и стек. Я рекомендую использовать generic версию тех же классов, когда это возможно.

- System.ConvUtils содержит библиотеку утилит преобразования для различных единиц измерения.
- System.DateUtils включает функции для обработки значений даты и времени.
- System.Devices взаимодействует с системными устройствами (например, GPS, акселерометр и т.д.).
- System.Diagnostics определяет структуру записи для точного измерения истекшего времени в тестовом коде, которое я иногда использую в книге.
- `System.Generics` имеет фактически два отдельных модуля, один для generic коллекций и один для generic типов. Эти модули описаны в Главе 14.
- System.Hash имеет основную поддержку для определения значений хэша.
- System.ImageList включает в себя абстрактную, библиотечно-независимую реализацию для управления списками изображений и порциями одного изображения как коллекцией элементов.
- System.IniFiles определяет интерфейс для обработки конфигурационных файлов INI, часто встречающихся в Windows.
- System.IOUtils определяет записи для доступа к файловой системе (файлы, папки, пути), которые были рассмотрены ранее в этой главе.
- System.JSON включает в себя те же самые основные классы для обработки данных в широко используемой Нотации объектов JavaScript, или JSON.
- System.Math определяет функции для математических операций, включая тригонометрические и финансовые функции. В его пространстве имен также имеются другие юниты для векторов и матриц.

- System.Messaging имеет общий код для обработки сообщений на разных операционных системах.
- System.NetEncoding включает в себя обработку некоторых распространенных Интернет-кодировок, таких как base64, HTML и URL.
- System.RegularExpressions определяет поддержку регулярных выражений (*regex*).
- System.Rtti имеет полный набор классов RTTI, как объяснено в главе 16.
- System.StrUtils имеет основные и традиционные функции обработки строк.
- System.SyncObjs определяет несколько классов для синхронизации многопоточных приложений.
- System.SysUtils содержит основную коллекцию системных утилит, некоторые из которых являются самыми традиционными, восходящими к раннему периоду существования компилятора.
- System.Threading включает в себя интерфейсы, записи и классы сравнительно недавно созданной библиотеки параллельного программирования.
- System.Types имеет несколько базовых дополнительных типов данных, таких как `TPoint`, `TRectangle` и записи `TSize`, класс `TBitConverter` и многие другие базовые типы данных, используемые в RTL.
- System.TypInfo определяет старый интерфейс RTTI, также представленный в главе 16, в основном замененный интерфейсом в `System.RTTI`.
- System.Variants и `System.VarUtils` имеют функции для работы с вариантами (функция языка описана в главе 5).
- System.Zip подключает библиотеку сжатия и распаковки файлов.

Есть также несколько других частей RTL, которые являются подразделами пространства имен *System*, каждый раздел включает в себя несколько модулей (иногда большое количество, как пространства имен `System.Win`), включая HTTP клиентов (`System.Net`), и поддержки Интернет вещей (`System.Beacon`, `System.Bluetooth`, `System.Sensors`, и `System.Tether`). Есть также, конечно, переведенные API и заголовочные файлы для взаимодействия со всеми поддерживаемыми операционными системами.

Опять же, существует множество готовых к использованию RTL функций, типов, записей, интерфейсов и классов, которые вы можете изучить, чтобы использовать всю силу Object Pascal. Не торопитесь просматривать системную документацию, чтобы узнать больше.

В заключение

Глава 18 отмечает конец книги, за исключением следующих трех приложений. Изначально это была моя первая книга, сосредоточенная исключительно на языке Object Pascal, и я делаю все возможное, чтобы продолжать обновлять ее и поддерживать текст книги и примеры во времени. Я опубликовал обновление для Delphi 10.1 Berlin (только в формате PDF), а сейчас вы читаете эту новую версию для Delphi 10.4 Sydney.

Вернитесь к Введению, чтобы получить свежие исходные тексты книг на GitHub и посетите книжный сайт или мой блог для получения информации и обновлений в будущем.

Надеюсь, вам понравилось читать книгу так же, как мне понравилось писать ее и писать о Дельфи в течение последних 25 лет. Счастливого программирования на Дельфи!

end.

В этом заключительном разделе книги находятся несколько приложений, которые посвящены конкретным побочным вопросам, которые стоит рассмотреть, но не в тексте. Имеется краткая история языков Pascal и Object Pascal, а также глоссарий.

Резюме раздела приложений

Приложение A: Эволюция Object Pascal

Приложение B: Глоссарий терминов

Приложение C: Индекс

A: Эволюция Object Pascal

Object Pascal - это язык, созданный для растущего спектра современных цифровых устройств, от смартфонов и планшетов до настольных компьютеров и серверов. Он не просто появился из воздуха. Он был тщательно разработан на прочном фундаменте, чтобы стать инструментом выбора для современных программистов. Он обеспечивает почти идеальный баланс между скоростью программирования и скоростью результирующих программ, ясностью синтаксиса и выразительностью.

Основой, на которой построен Object Pascal, является семейство языков программирования Pascal. Точно так же, как язык Google Go или язык Apple Objective-C уходит корнями в C, Object Pascal уходит корнями в Pascal. Без сомнения, вы догадались бы об этом по названию.

Это приложение включает в себя краткую историю семейства языков и актуальных инструментов, связанных с Pascal, Turbo Pascal, Delphi's Pascal и Object Pascal. Хотя на самом деле не обязательно читать это приложение для изучения языка, безусловно, стоит понимать эволюцию языка и то, где он находится сегодня.

Язык программирования Object Pascal, который мы используем сегодня в инструментах разработки Embarcadero, был изобретен в 1995 году, когда Borland представил Delphi, который в то время был новой визуальной средой разработки. Первый язык Object Pascal был расширен от языка, уже используемого в продуктах Turbo Pascal, где этот язык обычно назывался Turbo Pascal. Borland не изобрел Pascal, он только помог сделать его очень популярным, и расширил его основы, чтобы преодолеть некоторые его ограничения по сравнению с языком C.

В следующих разделах рассматривается история языка от Wirth's Pascal до последнего компилятора Object Pascal Delphi на базе LLVM для чипов ARM и мобильных устройств.

Паскаль Вирта

Изначально язык Паскаль (Pascal) был разработан в 1971 году Никлаусом Виртом, профессором Цюрихского политехнического института (Швейцария). Наиболее полную биографию Wirth можно найти на `сайте` `http://www.cs.inf.ethz.ch/~wirth`.

Паскаль был разработан в качестве упрощенной версии языка Алгол для образовательных целей. Сам Алгол был создан в 1960 году. Когда был изобретен Паскаль, существовало много языков программирования, но лишь немногие из них были широко распространены: FORTRAN, Assembler, COBOL и BASIC. Ключевой идеей нового языка был порядок, управляемый с помощью концепции сильных типов данных, объявления переменных и структурированных программных элементов управления. Язык также был задуман как средство

обучения, то есть обучения программированию с использованием передового опыта.

Нет необходимости говорить о том, что основные принципы Паскаля Вирта оказали огромное влияние на историю всех языков программирования, далеко выходящее за пределы и превышающее множество тех, которые до сих пор базируются на синтаксисе Паскаля. Что касается преподавания языков, то слишком часто школы и университеты следовали другим критериям (таким как запросы о приеме на работу или пожертвования от поставщиков инструментов) вместо того, чтобы смотреть, какой язык помогает лучше изучить ключевые концепции программирования. Но это уже другая история.

Турбо-Паскаль

Всемирно известный компилятор Pascal компании Borland, называемый Turbo Pascal, был представлен в 1983 году, реализовавший "Руководство и отчет пользователя Pascal" Дженсена и Вирта. Компилятор Turbo Pascal был одним из самых продаваемых компиляторов всех времён, и сделал язык особенно популярным на платформе ПК, благодаря балансу простоты, мощности и цены. Первоначальным автором компилятора был Андерс Хейлсберг, впоследствии отец очень популярных в Microsoft языков программирования C# и TypeScript.

Turbo Pascal представила интегрированную среду разработки (IDE), в которой можно редактировать код (в редакторе, совместимом с WordStar), запускать компилятор, просматривать ошибки и переходить к строкам, содержащим эти ошибки. Сейчас это звучит банально, но раньше

приходилось выходить из редактора, возвращаться в DOS; запускать компилятор командной строки, записывать строки с ошибками, открывать редактор и перепрыгивать на строки с ошибками.

Более того, Borland продавал Turbo Pascal за 49 долларов, а компилятор Microsoft Pascal продавался за несколько сотен. Многолетний успех Turbo Pascal способствовал тому, что в конечном итоге компания Microsoft отказалась от своего компилятора Pascal.

На самом деле вы можете скачать копию оригинальной версии Borland's Turbo Pascal из раздела *"Музей"* сети разработчиков Embarcadero:

`http://edn.embarcadero.com/museum`

история После оригинального языка Pascal Никлаус Вирт (Nicklaus Wirth) разработал язык Modula-2, расширение синтаксиса Pascal, которое теперь почти забыто, и которое ввело концепцию модуляризации, очень похожую на концепцию модулей в ранней версии Turbo Pascal и сегодняшней версии Object Pascal.

Еще одним расширением Modula-2 стала Modula-3, обладающая объектно-ориентированными особенностями, схожими с Object Pascal. Modula-3 использовался еще меньше, чем Modula-2, при этом большинство коммерческих разработок языка Pascal двигались в сторону компиляторов Borland и Apple, пока Apple не отказалось от Object Pascal для Objective-C, оставив Borland практически монополию на язык.

Ранние дни Object Pascal в Delphi

После 9 версий компиляторов Turbo и Borland Pascal, которые постепенно расширили язык в область объектно-ориентированного программирования (ООП), в 1995 году Borland выпустила Delphi, превратив Pascal в язык визуального программирования. Delphi расширила язык Pascal

несколькими путями, включая многие объектно-ориентированные расширения, которые отличаются от других разновидностей Object Pascal, в том числе и в компиляторе *Borland Pascal с* компилятором *Objects* (последнее воплощение Turbo Pascal).

история 1995 год был действительно особенным годом для языков программирования, так как в нем состоялся дебют Delphi Object Pascal, Java, JavaScript и PHP. Это одни из самых популярных языков программирования, которые используются до сих пор. Фактически, большинство других популярных языков (C, C++, Objective-C и COBOL) гораздо старше, в то время как единственным более новым популярным языком является C#. Историю языков программирования можно посмотреть на сайте http://en.wikipedia.org/wiki/History_of_programming_languages.

С помощью Delphi 2 компания Borland перенесла компилятор Pascal в 32-битный мир, фактически переделав его для создания генератора кода, общего с компилятором C++. Это принесло много оптимизаций, ранее встречавшихся только в Си/Си++ компиляторах, в язык Pascal. В Delphi 3 Borland добавила к языку концепцию интерфейсов, сделав скачок вперед в выразительности классов и их отношениях.

С выходом версии 7 Delphi компания Borland формально начала называть язык Object Pascal языком Delphi, но на самом деле на тот момент в этом языке ничего не изменилось. В то время Borland также создал Kylix, версию Delphi для Linux, а позже создал компилятор Delphi для Microsoft .NET framework (продуктом был Delphi 8). Оба проекта были позже заброшены, но в Delphi 8 (выпущенной в конце 2003 года) был внесен очень большой набор изменений в язык, изменения, которые впоследствии были приняты в компиляторе Win32 Delphi и во всех последующих компиляторах.

Object Pascal из CodeGear в

Embarcadero

Поскольку компания Borland была не уверена в своих инвестициях в инструменты разработки, более поздние версии, такие как Delphi 2007, были произведены компанией CodeGear, ее дочерней компанией. Эта дочерняя компания (или подразделение) была позже продана Embarcadero Technologies. После этого релиза Embarcadero переориентировалась на развитие и расширение языка Object Pascal, добавив такие долгожданные функции, как поддержка Unicode (в Delphi 2009), дженерики, анонимные методы или замыкания, расширенная информация о типе исполнения или отражения, а также многие другие важные языковые функции (в основном описанные в Части III этой книги).

В то же время, наряду с компилятором Win32, компания представила компилятор Win64 (в Delphi XE2) и компилятор macOS, вернувшись к мультиплатформенной стратегии после попытки, сделанной ранее на Linux с недолговечным продуктом Kylix. Однако на этот раз идея состояла в том, чтобы иметь в Windows единую среду разработки и кросс-компилировать код на другие платформы. Поддержка Mac была только началом стратегии компании по поддержке мульти-устройств, охватывающей настольные и мобильные платформы, такие как iOS и Android. Эта стратегия стала возможной благодаря принятию нового фреймворка GUI под названием FireMonkey.

Переход к мобильности

Переход на мобильный и первый Object Pascal компилятор для чипов ARM (все предыдущие платформы, поддерживаемые Delphi, были только на чипах Intel x86) был связан с общей переработкой компиляторов и связанных с ними инструментов (или "инструментальной цепочки компилятора"), основанных на открытой архитектуре компилятора LLVM.

примечание LLVM - это короткое название инфраструктуры компилятора LLVM или *"собрание модульных и многократно используемых компиляторов и технологий цепочки инструментов"*, как вы можете прочитать на сайте `https://llvm.org/`.

Компилятор ARM для iOS, выпущенный в Delphi XE4, был первым компилятором Object Pascal, основанным на LLVM, а также первым, который ввел некоторые новые функции, такие как автоматический подсчёт ссылок (или ARC, теперь удалён из языка).

Позже в том же году (2013) Delphi XE5 добавил поддержку платформы Android, со вторым компилятором ARM на базе LLVM. В итоге, Delphi XE5, поставляется с 6 компиляторами для языка Object Pascal (для Win32, Win64, macOS, iOS Simulator on Mac, iOS ARM, и поддержка Android ARM). Все эти компиляторы поддерживают в основном общее определение языка, с несколькими существенными отличиями, которые я подробно осветил в книге.

В первые несколько месяцев 2014 года Embarcadero выпустила новый инструмент разработки, основанный на тех же базовых мобильных технологиях и получивший название Appmethod. В апреле 2014 года компания также выпустила версию Delphi XE6, а в сентябре 2014 года вышла третья версия AppMethod и Delphi XE7, за которой весной 2015 года последовал Delphi XE8,

включающий в себя первый ARM 64-битный компилятор, нацеленный на iOS.

Эпоха Delphi 10.x

После Delphi 10 Seattle, после вхождения в компанию Idera Corp., Embarcadero создала серию релизов 10.x: Delphi 10.1 Berlin, Delphi 10.2 Tokyo, Delphi 10.3 Rio и Delphi 10.4 Sydney. В этих версиях Embarcadero добавила поддержку новых целевых платформ и операционных систем: Linux 64-бит, Android 64-бит и MacOS 64-бит. Компания также сосредоточилась на библиотеке Windows VCL, добавив специальную поддержку операционной системы Windows 10.

В течение выпуска 10.x серии Embarcadero продолжалось развитие языка Object Pascal с введением таких функций, как объявления встроенных переменных и пользовательских управляемых записей, а также многих других небольших улучшений, все из которых описаны в этой книге.

B: Глоссарий

A

Abstract Class	класс, который не до конца определен и предоставляет только интерфейс метода, который должны реализовать подклассы.
Ambiguous call	- это сообщение об ошибке, которое вы получаете в случае, если компилятор имеет два или более вариантов разрешения вызова функции и не имеет возможности автоматически определить, какой из них вы пытаетесь вызвать.
Android	Название операционной системы Google для телефонов и планшетов, принятое сотнями поставщиков оборудования (кроме Google), учитывая его открытый характер. В настоящее время Android является самой используемой операционной системой в мире, обогнав Microsoft Windows.
Anonymous Method	Анонимный метод или анонимная функция - это функция, которая не связана с именем функции и может быть присвоена переменной или передана в качестве аргумента другой функции, которая позже может выполнить свой код. Вы можете подумать, что анонимные методы являются немного магическими по сравнению с обычными функциями. Так и есть! Настоящая магия заключается в том, что они могут получать доступ к переменным из блока, в котором они объявлены, даже если в конечном итоге они

	выполняются в другом блоке. Анонимная функция и переменные, к которым она может получить доступ, известны как *замыкание*, которое является другим именем, используемым для той же самой функции.
API	Интерфейс прикладного программирования (API) предоставляется программным обеспечением (например, операционными системами), чтобы с ним могли работать прикладные программы. Например, когда приложение выводит на экран строку текста, обычно оно вызывает функцию в графическом интерфейсе компьютера . Набор функций, предоставляемых графическим интерфейсом компьютера, известен как API GUI. Обычно, когда программное обеспечение предоставляет API для языка, оно написано на нем. Например, Microsoft Windows предоставляет API, ориентированный на языки Си и Си++. *Примечание*: Устройство Object Pascal `WinAPI.Windows` предоставляет Object Pascal API для Microsoft Windows, устраняя трудности прямого вызова функций, записанных для вызова с C или C++.

B

Boolean Expression	Булевские выражения - это выражения, которые оценивают либо как истинные, либо как ложные. Простой пример - 1 = 2, которое оказывается ложным. Булевское выражение не обязательно должно быть традиционным математическим выражением, оно может быть просто переменной булевого типа или даже вызовом функции, которая возвращает булевское значение.

C

Cardinal	Cardinal - один из натуральных номеров. Проще говоря, это означает число, которое может быть использовано для подсчета вещей и которое всегда больше или равно нулю.
Class	Класс - это определение свойств, методов и полей данных, которые будут иметь объект (этого класса) при его создании. *Примечание:* не все языки, ориентированные на объект, требуют классов для определения объектов. Объекты в JavaScript, IO и Rebol могут быть определены непосредственно без предварительного определения класса. *Примечание:* определение записи очень похоже на определение класса в Object Pascal. Запись имеет члены, которые выполняют те же функции, что и свойства для класса, а также процедуры и функции, которые делают то же, что и методы для класса.
Code Point	Числовое значение элемента набора символов Юникода. Каждая буква, число, пунктуация каждого алфавита мира имеет кодовую точку Юникода, представляющую ее.
Compiler Directive	Директива компилятора - это специальная инструкция компилятору, которая изменяет его стандартное поведение. Директивы компилятора присваиваются специальными словами, префиксом которых служит знак $, или могут быть заданы в Параметрах проекта.
Components	Компоненты - это предварительно собранные, готовые к использованию объекты кода, которые можно легко комбинировать как с кодом приложения, так и с другими компонентами, что значительно сокращает время, затрачиваемое на разработку приложений. Библиотека VCL и платформа FireMonkey

	являются двумя большими коллекциями таких компонентов, поставляемыми вместе с Delphi.
COM	Объектная модель COMComponent является основной частью архитектуры Microsoft Windows.
Control	- это элемент графического интерфейса пользователя, такой как кнопка, поле ввода текста, контейнер с изображениями и т.д. Элементы управления часто обозначаются как визуальные компоненты.
CPU	CPU или центральный процессор - это ядро любого компьютера и то, что на самом деле выполняет код. Объектные выражения языка Object Pascal должны быть транслированы в ассемблерный двоичный код, чтобы быть понятными ЦПУ. Обратите внимание, что в отладчике у вас есть вид ЦП, а не что-то для новичков. Центральный процессор часто работает параллельно с FPU.

D

Data Type	Тип данных указывает на потребность в хранении и операции, которые можно выполнить с переменной этого типа. В Object Pascal каждая переменная имеет свой тип данных, как это происходит в языках программирования с сильным типом.
Design Patterns	Шаблоны проектирования. Узнав о программных архитектурах, которые используются разными разработчиками для решения различных задач, вы можете заметить сходства и общие элементы. Шаблон проектирования - это признание такой общей конструкции, выраженное стандартным образом и достаточно абстрактное, чтобы быть применимым в ряде различных ситуаций.

Движение шаблонов дизайна в мире программного обеспечения началось в 1994 году, когда Эрих Гамма, Ричард Хельм, Ральф Джонсон и Джон Влиссидес написали книгу "Шаблоны дизайна, элементы многоразового объектно-ориентированного программного обеспечения" (Addison-Wesley, 1994, ISBN: 0-201-633612). Авторов часто называют "Гамма и др.", но чаще - "Банда четырех" или просто " GoF ". Книга часто упоминается в разговорной форме как "Книга GoF ".
В книге GoF авторы описывают понятие паттернов программного обеспечения, указывают точный способ их описания и приводят каталог из 23 паттернов, разделенных на три группы: креативные, структурные и поведенческие.

DLL — Dynamic Link Library - это библиотека функций, которые не входят в исполняемый код приложения, но хранятся в отдельном файле. Когда приложение запускается, оно загружает библиотеку в память и затем может вызывать функции, содержащиеся в библиотеке. Эти библиотеки обычно предназначены для использования многими приложениями. На платформах, отличных от Windows, тот же тип библиотеки называется Shared Object (или SO).

E

Event (Событие) — Событие - это действие или операция, происходящие в приложении, например, щелчок мыши или изменение размера формы. Delphi реализует события через специальное свойство класса, позволяющее объекту делегировать некоторое "поведение" внешнему методу. События являются частью модели разработки RAD.

F

FireMonkey	FireMonkey или FMX - это библиотека визуальных и невизуальных компонентов, поставляемая вместе с Delphi. Компоненты кроссплатформенны, поэтому они одинаково хорошо работают на Windows, MacOS, iOS, Android и даже Linux (через дополнительную библиотеку FMXLinux).
Form	форма - термин, используемый для обозначения окна в библиотеках VCL и FireMonkey.
Файловая система	Файловая система является частью операционной системы компьютера, которая организует хранение данных на компьютере и управляет их хранением и извлечением.
FPU	FPU или модуль с плавающей запятой является дополнением к процессору, ориентированному на выполнение сложных вычислений чисел с плавающей запятой чрезвычайно быстро.
Function	Функция представляет собой блок кода, который выполняет какое-либо действие (или вычисление) и возвращает результат. Она может принимать заранее заданное количество параметров для изменения вычисления.
Function Overloading	Перегрузка функций — это особенность языков программирования со строгими соглашениями о типах переменных, которая позволяет программисту объявлять различные версии функций, которые могут принимать различные типы параметров.

G

Global Memory	Глобальная память - это статическая область памяти для глобальных переменных ваших

приложений. Эта память используется на протяжении всего срока службы приложения, и она не может расти (см. раздел Heap memory (Куча памяти) для динамически выделяемой области памяти). Глобальная память редко используется в приложениях Object Pascal.

GUI — Графический интерфейс пользователя, позволяющий пользователям взаимодействовать с компьютерами, планшетами и телефонами с помощью графических иконок и других визуальных индикаторов. В большинстве случаев взаимодействие пользователя с графическим интерфейсом осуществляется путем наведения, прикосновения, нажатия, пролистывания и других жестов с помощью мыши (или аналогичного указательного устройства) или пальцев.

H

Heap Memory (куча памяти) - это область памяти для динамически распределяемых блоков памяти. Как следует из названия, в выделении кучи памяти нет никакой структуры или последовательности. Всякий раз, когда блок необходим, он берется из свободной области. Жизненный цикл отдельных блоков разный, и порядок выделения и де-выделения не связан. Память кучи используется для данных объектов, строк, динамических массивов и других типов ссылок (см. Ссылки), а также для блоков, выделенных вручную (см. Указатели). Куча большая, но не бесконечная, и если вы не освободите из памяти неиспользуемые объекты, то в конечном итоге ваше приложение исчерпает память.

I

IDE	*Интегрированная среда разработки* - это единое приложение, которое предоставляет разработчику широкий спектр инструментов для обеспечения высокой производительности. Как минимум IDE предоставляет редактор исходного кода, инструменты автоматизации сборки и отладчик. Современная идея IDE была придумана вместе с первыми компиляторами Turbo Pascal, которые пришли из Borland, предшественников современных Object Pascal IDE от Embarcadero Technologies. Object Pascal IDE, поставляемая вместе с Delphi, очень сложна и включает в себя, например, дизайн графического интерфейса, шаблоны кода, рефакторинг кода и тестирование встроенных блоков.
(Type) Inheritance	Наследование типа является одним из основных принципов объектно-ориентированного программирования (ООП). Идея заключается в том, что тип данных может расширять существующий тип данных, добавляя к нему новые возможности. Это расширение типа, если оно известно как наследование типа, наряду с такими терминами, как базовый класс и класс-потомок, или родительский класс и класс-потомок.
Interface	Интерфейс обычно относится к абстрактному объяснению того, что может сделать программный модуль. В Object Pascal интерфейс представляет собой чисто абстрактное определение класса (сделанное только из методов, и без данных), как в C# или Java. Полное описание см. в Гл. 11. Однако язык также имеет концепцию интерфейса для модуля, в этом случае это раздел модуля, который декларирует то, что он видит для других

	модулей. В обоих случаях используется одно и то же ключевое слово `interface`.
iOS	Название операционной системы, питающей iPhone, iPad и подобные устройства Apple.

M

Method	Метод - это функция или процедура, которая привязана к объекту. Методы имеют доступ ко всем данным, хранящимся в объекте.

O

Object	Объект представляет собой комбинацию некоторых элементов данных (свойств и полей) и кода (методов). Объект является экземпляром класса, который определяет семейство (или тип) объектов.
OOP	объектно-ориентированное программирование - это концептуальная структура, лежащая в основе Object Pascal, основанная на таких понятиях, как классы, наследование и полиморфизм. Современный Object Pascal поддерживает и другие парадигмы программирования, благодаря таким возможностям, как дженерики, анонимные методы и рефлексия.
Ordinal Type	- тип данных, состоящий из элементов, которые могут быть подсчитаны и имеют последовательность. Вы можете думать о целых числах, но символы также имеют последовательность и даже пользовательские перечисляемые типы.

macOS

Старое название операционной системы компьютеров Apple Mac теперь заменяется на Mac OS.

P

Pointer

Указатель - это переменная, содержащая непосредственно адрес памяти. Указатель может ссылаться на месторасположение некоторых данных или функции в памяти. Указатели используются нечасто, в то время как ссылки (см. Ссылку) — это непрозрачные и управляемые указатели, которые чрезвычайно распространены, но при этом значительно проще в использовании.

Полиморфизм

- это способность вызова метода принимать "различные формы" (то есть в конечном итоге выполнять различные операции) в зависимости от объекта. Это стандартная черта ООП-языков.

Procedure

Процедура - это блок кода (или подпрограммы), который может быть вызван из других частей программы. Процедура может принимать параметры для изменения того, что она делает. В отличие от функции, процедура не возвращает значение.

Project Options

набор опций конфигурации, которые влияют не только на общую структуру проекта приложения, но и на поведение компилятора и компоновщика.

Property

Свойство определяет состояние объекта, абстрагируясь от фактической реализации, данное свойство может быть сопоставлено с данными или использовать методы для чтения и записи значения.

R

RAD	Rapid Application Development - это характеристика среды разработки, позволяющая легко и быстро создавать приложения. Инструменты RAD, как правило, основаны на визуальных проектировщиках, хотя сегодня это довольно старое определение используется редко.
Record	Простая запись представляет собой набор элементов данных, которые хранятся в структурированном виде. Запись определяется в определении типа, показывающем порядок и тип отдельных элементов данных в записи. Object Pascal также включает в себя расширенные записи, которые могут иметь методы, аналогичные объекту.
Recursion	Рекурсия или рекурсивный вызов - это способ описания функции, которая продолжает вызывать себя до тех пор, пока не будет выполнено заданное условие. Рекурсивный вызов часто является лучшей альтернативой циклу или циклу. Примером рекурсивной реализации умножения может быть взятие значения первого числа и сложение к нему одного и того же числа, умноженного на другое минус одно, до тех пор, пока остальные числа не станут равны нулю.
Reference	Ссылка - это переменная, которая ссылается на некоторые данные в другом месте в памяти, а не хранит их напрямую. В Object Pascal переменная типов, таких как классы и строки, а также интерфейсы и динамические массивы, являются ссылкой. В отличие от указателей (см. Pointers) ссылки, как правило, управляются компилятором и библиотекой времени исполнения и не требуют от разработчика глубоких знаний и прямых манипуляций с памятью.

RTTI (или Reflection)	Сокращение Run Time Type Information (Информация о типе выполняемого приложения) - это возможность доступа к информации о типе (традиционно доступной только компиляторам) в реальном приложении во время выполнения. Другие среды программирования называют эту возможность отражением.
Run-Time Library (RTL)	Библиотека времени выполнения - это набор заранее написанных процедур, которые компилятор автоматически включает в код приложения для сборки исполняемого приложения. Она включает поддержку многих фундаментальных операций, особенно тех, которые требуют взаимодействия с операционной системой при запуске приложения (например, выделение памяти, чтение и запись данных, взаимодействие с файловой системой).

S

SDK	Software Development Kit - это набор программных средств, с помощью которых можно создавать программное обеспечение для конкретной среды. Каждая операционная система предоставляет SDK, включающий библиотеки API (Application Programming Interfaces) и инструменты разработчика, необходимые для сборки, тестирования и отладки приложений для платформы.
Search Path	набор папок, которые компилятор будет искать при поиске внешнего модуля, на который делается ссылка в операторе uses
Stack (память)	Стек, если динамически и упорядоченно выделенная область памяти. Каждый вызов метода, процедуры или функции резервирует

свою собственную область памяти (для локальных переменных, в том числе временных, и параметров). Как только метод возвращает управление, память очищается очень упорядоченно. Единственный реальный сценарий исчерпания памяти стека — это когда метод входит в бесконечный рекурсивный вызов (см. Recursion).

Замечание: В большинстве случаев локальные переменные, выделенные на стеке, не инициализируются до нуля: перед их использованием необходимо установить их значение.

U

ЮникодUnicode - это стандартный способ записи отдельных текстовых символов в виде двоичных данных (последовательность 0s и 1s). Текстом можно надежно обмениваться между программами, обрабатывать и отображать, если он соответствует стандарту Unicode. Стандарт очень большой и охватывает более 110 000 различных символов из примерно 100 различных алфавитов и шрифтов написания.

V

VCL Библиотека Визуальных Компонентов представляет собой огромный набор Визуальных Компонентов, поставляемый вместе с Delphi. GUI-компоненты VCL являются родными Windows GUI-компонентами.

Virtual Methods	Виртуальный метод - это функция или процедура, объявленная в определении типа класса, которая может быть переопределена классами, являющимися его подклассами. Так называемый базовый класс может также включать определение метода, который может использоваться подклассами. Если базовый класс не определяет версию метода по умолчанию, то любой подкласс должен дать определение Виртуального метода.

W

Window	Окно - это область экрана, содержащая элементы GUI, с которыми пользователь может взаимодействовать. GUI-приложение может отображать несколько окон. В VCL и FireMonkey окна определяются с помощью объекта Form.
Windows	Название повсеместно распространенной операционной системы Microsoft, которая стала пионером (наряду с другими операционными системами того времени, такими как операционная система Apple Mac) концепции графических окон (см. запись выше).

C: Index

.... 260
_AddRef 458
_Release 458
€ 247, 260
1252 code page 247
1983 743
1995 744
64-bit 239, 571, 748
abstract 358
abstract classes 453
Abstract Classes 750
Access violations 549
ActionList 443
Adapter Pattern 476
address of 216, 238
AfterDestruction 391
AJAX 639
Algol 742
alignment 208
Alignment 328
Allen Bauer 184
Alphabet 245
Ambiguous calls 750
Ambiguous Calls 168
Anders Hejlsberg 297, 352, 743
Android 747, 750
Android, ... 755
angle brackets 557, 562
anonymous delegate 620
Anonymous Event Handlers 632
Anonymous methods 620
Anonymous Methods 750
ANSI 289
AnsiChar 283
AnsiString 291
ANSIString 292
API 751, 761
Append 733
Apple 744, 758, 759, 763
Apple's Instruments tool 542
Application 389
Appmethod 747
ArcExperiments Example 530
ArcExperiments example 530
ARM chips 747
Array Properties 413, 447
Arrays 187
Arrays of Records 205
as 363, 457, 474, 548
ASCII 245
Assembler 742
Assign 331, 521, 719
Assigned 240, 550
Attribute Classes 667
Attributes 665
AutoSize 328
Barry Kelly 662, 673
BASIC 742
Basic Multilingual Plane 250
BeforeConstruction 391
Bertrand Meyer 335
BOM marker 730
Boolean 86

Boolean Expression 751
Boolean type..751
Borland 743, 744
Borland C++ 4.0 Object-Oriented Programming 555
Break...161
Buffer Overruns 539
Byte Order Mark 251
ByteLength 285
C 92, 151, 152, 162, 179, 188, 212
C#.....151, 210, 297, 300, 304, 317, 333, 340, 348, 352, 361, 364, 374, 402, 404, 413, 417, 423, 425, 426, 452, 479, 555, 579, 620
C++ 151, 162, 188, 216, 297, 311, 317, 326, 348, 374, 452, 555, 614, 745
callback functions 486
calling convention..................................517
Calling Conventions................................177
captured variable.................................. 625
Cardinal 752
caret 238
catch... 374
cdecl 178
Char 86, 256
Chars[] 269, 270, 275
Checking Memory.................................. 534
Chris Bensen 414
Class...752
Class Completion 299
Class Constraints................................... 576
Class Constructors 489, 573
Class Data 482
Class Helpers..501
Class Methods 482
class of 494
class operator 219
Class XE "Properties"Properties 487
Class References.................................... 493
class var 483
Classes 297
Classes unit.. 729
ClassInfo 698
ClassName............................ 495, 696, 698
ClassNameIs...698
ClassParent 698
ClassType......................................696, 698
Clear... 546
ClientDataSet...598
closures 423, 620, 751
COBOL...742
Code 24
code completion 428
Code Completion................... 406, 408, 413
Code Parameters 159, 167
Code Point 752
Code Points... 247
CodeGear 745
COM.. 164, 231, 293, 414, 453, 454, 474, 753
Comments..31
common ancestor class...........................340
Compare 274
Compiler directive
  $ALIGN 208, 209
  $DEFINE...539
  $HIGHCHARUNICODE 261
  $IF 67
  $IFDEF......................................68, 182
  $IFEND 67
  $INLINE..173
  $M..............................419, 420, 644, 718
  $RTTI 649
  $SCOPEDENUMS 110
  $StrongLinkTypes............................ 651
  $VARPROPSETTER 415
  $WeakLinkRTTI 651
  $ZEROBASEDSTRING 270, 271
  Example..385
  RTTI..649
Compiler Directives 752
ComponentCount...................................720
Components 720
Concatenating Strings.............................271
const 522
const reference 165
Constant Parameters.............................. 164
Constructors 320, 354, 390, 444, 544
Contains... 274
Controls 753
ControlsEnum
  Example...509
ControlsEnum application project.........509
ConvertFromUtf32..................................258
Copy 195, 196, 274
copy-on-write.............................264, 265

CountChars....275
Covariant Return Types....614
CPU....753, 755
Create....340, 545, 696, 723
Creating a Component....434
Current....437
Data Types....753
debugger....375
Dec....256
default....414
Default....570
Default Constructor Constraint....583
Default Parameters....170
DefaultTextLineBreakStyle....709
DefaultUnicodeCodePage....285
Delayed Loading....183
delegation....423
Delegation....426
Delete....197
Delphi....744, 753, 757
DeQuoted....275
Design Patterns....753
design-time....418
Destroy....322, 340, 525
destructor....322, 525
Destructors....544
Dictionary....595
Dispose....239
DisposeOf....527
DLL....754
do....374
DoCompare....593
DupeString....278
dynamic....355, 432
Dynamic Arrays....192, 197
dynamic binding....348
Dynamic Link Library....754
early binding....347
EDivByZero....377
EExternalException....184
EInvalidCast....364
Elixir....157
Embarcadero Technologies....746, 757
Empty....263
encapsulation....402, 420
Encapsulation....213, 310, 340
EndOfStream....730
EndsWith....274
Enumeration....437
EProgrammerNotFound....378
Equals....274, 702
Erich Gamma....754
Erik van Bilsen....387, 614
Erlang....157
EurekaLog....383
Event-Driven Programming....422
events....422, 424
Events....426, 427, 429, 442, 754
Example
- AdvancedExcept....394, 396, 399
- AlignTest....209
- Animals1....346, 347, 348
- Animals2....348, 350
- Animals3....359
- AnonAjax....638, 640, 641
- AnonButton....632
- AnonLargeStrings....635
- AnonymFirst....621, 624, 627
- ArraysTest....188, 190
- AutoRTTI....421
- BinaryFiles....731
- CharTest....258, 259
- ClassConstraint....576
- ClassCtor....490
- ClassHelperDemo....501
- ClassRef....497, 499
- ClassStatic....485, 487
- ClicksCount....315
- CodePoints....247
- ControlHelper....504
- CountObj....488
- CreateComps....318, 320
- CustomerDictionary....595
- Date3....322, 324
- DateComp....435
- DateComponent....436
- DateCompTest....437
- DateEvent....431, 436
- DatePackage....436, 437
- DateProperties....410
- Dates1....303
- Dates2....312
- Dates3....325
- Dates4....327

DerivedDates....337, 338
DynamicEvents....427
DynArray....194, 196
DynArrayConcat....197
EncodingsTest....289
ErrorLog....390
ExceptFinally....383, 384
ExceptionFlow....382
ExceptionsTest....375
FlowTest....161
FormatString....282
FormProperties....408
FunctionsTest....153
FunctionTest....151, 155, 157
GenericClassCtor....574
GenericCodeGen....568
GenericInterface....601, 603
GenericMethod....566
GenericTypeFunc....571, 572
InliningTest....174
InterceptBaseClass....675
Intf101....455, 457
IntfConstraint....579, 582, 584, 602
IntfContraints....601
IntfDemo....466, 467, 473
IntfError....461
IoFilesInFolder....724
KeyValueClassic....556
KeyValueGeneric....559, 560
LargeString....272, 634
LeakTest....537
ListDemoMd2005....588, 590
NestedClass....333
NestedTypes....332
NumbersEnumerator....438
ObjFromIntf....474
OpenArray....199, 202
OperatorsOver....221, 223
OverloadTest....166, 169, 170
ParamsTest....161, 163, 164
PointersTest....238, 240
ProcType....180
Protection....342, 344
ReaderWriter....730
RecordMethods....213, 217
RecordsDemo....203
RecordsTest....206, 208
ReintroduceTest....353
RttiAccess....664, 665
RttiAttrib....669, 670
RttiIntro....646, 647
SafeCode....545, 547, 552
ShowMemory....535
ShowUnicode....252
SmartPointers....610
SmartPointersMR....609
StaticCallBack....486
StringHelperTest....276
StringListVsDictionary....599
StringMetaTest....284, 285
Strings101....265
TypeAliasHelper....513
TypeCompRules....561, 564
TypesList....652, 656
VariantTest....233, 236
VarProp....415
ViewDate....328, 337, 410
VisualInheritTest....368
WebFind....636, 638, 640
XmlPersist....685
except....374, 376, 379, 384
Exception....395, 397, 491
exception handling....373
Exceptions....544
Exceptions Hierarchy....377
Exit....160, 161
Explicit....220, 221
expression context....621
Extended RTTI....648
External Functions....182
fastcall....178
FastMM4....535, 539
Fields Alignments....208
File Access....723
File System....755
File Types....242
Files
DFM....417, 449
FMX....417, 449
INI....736
Final Methods....361
finalization....491
finally....374, 384, 523, 545
Finally....383, 546

FindComponent ... 722
FindHInstance ... 550
FindType ... 652
FireMonkey ... 746, 752, 755
FMXLinux ... 755
for ... 437
Form ... 441, 443, 755, 763
Format ... 200, 274, 279
FormatDateTime ... 236
Forms ... 407
FORTRAN ... 742
forward ... 154
Forward Declarations ... 154
FPU ... 753, 755
Free ... 306, 323, 340, 392, 524, 525
FreeAndNil ... 307, 527
FreeInstance ... 543
FreeMem ... 239, 540
friend classes ... 311
From ... 662
function ... 152
Function ... 755
function pointer ... 179
Gang of Four ... 754
garbage collection ... 305
Garbage Collection ... 514
Generic Constraints ... 575
Generic Containers ... 587
Generic Dictionary ... 595
Generic Methods ... 565
generic type declaration ... 563
GetDirectories ... 725
GetEnumerator ... 437, 439
GetFiles ... 725
GetHashCode ... 275, 680, 703
GetMem ... 239, 519, 540
GetMemoryManagerState ... 535
GetMemoryMap ... 535
GetMinimumBlockAlignment ... 537
GetNumericValue ... 257
GetPackages ... 659
GetPropValue ... 421
GetType ... 652
GetTypeKind ... 571
GetUnicodeCategory ... 258
GetUserName ... 183
GetWindowText ... 540
Global Memory ... 516, 755
Global variables ... 516
Global Variables ... 444
Google ... 750
Graphemes ... 247
GUI ... 751, 753, 756
GUID ... 455, 473, 601
Haskell ... 157
HasWeakRef ... 571
heap ... 263
Heap ... 518, 756
HFAAttribute ... 712
High ... 189, 190, 194, 256, 271
HPPGENAttribute ... 712
IComparer ... 590, 592, 604
IDE ... 167, 214, 365, 406, 419, 435, 743, 757
IDispatch ... 710
IEqualityComparer ... 604
IfThen ... 254, 279
IInterface ... 454, 455, 456, 468, 473, 710
IInvokable ... 710
implementation ... 54
implements ... 468, 710
Implicit ... 220, 221, 224, 612, 662
Inc ... 256, 406
indexers ... 413
IndexOf ... 274
IndexOfAny ... 274
information hiding ... 310
Inheritance ... 335, 345, 354, 757
inherited ... 324, 354, 370
InheritsFrom ... 495, 549, 699, 700
initialization ... 491
inline ... 173
InnerException ... 394, 395, 398
Insert ... 197, 274, 733
InstanceSize ... 495, 577, 697, 699
interface ... 54
Interface Constraints ... 579
Interface Delegation ... 467
Interface ID ... 601
Interface Properties ... 466
interfaces ... 453
Interfaces ... 452, 458, 476, 757
Internet of Things ... 738
interposer class ... 477, 501
invalid typecast ... 661

Invoke ... 664
IO ... 294
iOS ... 747, 758
iOS, ... 755
is ... 362, 364, 472, 474, 548
IsControl ... 258
IsDelimiter ... 274
IsEmpty ... 275
IsInArray ... 258
IsLetterOrDigit ... 258
IsLower ... 258
IsManagedType ... 571
IsNullOrWhiteSpace ... 275
IsNumber ... 258
ISO Encodings ... 245
IsPointerToObject ... 551
IsSurrogate ... 258
IsUpper ... 258
IsWhiteSpace ... 258
IUnknown ... 454
Java .. 151, 188, 216, 300, 304, 317, 333, 340, 348, 361, 374, 402, 404, 417, 425, 452, 479, 745
JavaScript . 151, 188, 210, 232, 294, 317, 423, 620, 745
JclDebug ... 383
Join ... 274
JSON ... 736
key-value pair ... 556
Label ... 328
LastIndexOf ... 274
Late Binding ... 347
lazy initialization ... 524, 596
leak detection ... 534
Length ... 194, 263
Lifetime of Local Variables ... 624
LIFO ... 517
Linux ... 755
ListBox ... 589
ListView ... 252, 598
Literals ... 260
Little Endian ... 288
LLVM ... 747
Loaded ... 723
LoadFromFile ... 734
Low ... 189, 190, 194, 256, 271
LowerCase ... 274
Lua ... 294
macOS ... 755
madExcept ... 383
Malcolm Groves ... 670
malloc ... 542
Marco Cantu ... 9
Max ... 174
MediaPlayer ... 350
memory leak ... 305
Message Handlers ... 356
MessageDlg ... 155
metaclasses ... 495
Method ... 158
Method Chaining ... 733
Method Pointers ... 424
Methods ... 212, 302, 758
Methods Aliases ... 469
Microsoft ... 744, 763
Modula-2 ... 744
Monitor synchronization ... 709
More on Weak References ... 528
Move ... 711
Multi-Dimensional Static Arrays ... 190
Multiple inheritance ... 452
Name ... 442, 722
named constructors ... 321
NativeUInt ... 239
Nested Exceptions ... 394
Nested Types ... 331
NeverSleepOnMMThreadContention ... 537
New ... 239
NewInstance ... 543
Nicklaus Wirth ... 744
Niklaus Wirth ... 742
nil ... 235, 307, 525
Notification ... 594
NULL ... 235
Object ... 758
Object Inspector ... 427, 431, 433, 436
Object Reference Model ... 304, 519
Objective-C ... 216, 232
object-oriented programming ... 294, 297
Objects ... 297, 302
Objects as Parameters ... 520
of object ... 425, 430
on ... 374
OnChange ... 497

OnClick 426, 427, 430, 442, 633
OnCreate 253, 486
OnException 380, 389
OnMouseDown 497
OOP 757, 758
Open Array Parameters 198
open-closed principle 335, 500
operator overloading 219
Ord 256
out 164
overload 165, 434
Overloaded Methods 325
Overloading 755
override 324, 349, 350, 434
owner 523
Owner 720
Ownership 720
PadLeft 274
Parallel Programming Library 638, 737
ParamCount 711
Parameters 158
parametric type 558
ParamStr 711
Parent 319, 498
ParentClass 495
Parse 274
pascal 178
Pascal 741, 742
PHP 232, 745
Piacenza 2
Pierre La Riche 535
Pointers 237, 759
Polymorphism 344, 347, 471, 759
Polytechnic of Zurich 742
Position 727
private 215, 308, 342
Private 309, 311
procedural types 179
procedure 151
Procedure 759
Project Manager 435
Project Options 759
Properties 402, 429, 448, 487, 759
Properties by Reference 414
protected 310, 342
Protected 309, 340
Protected Hack 341
public 310, 418
Public 309
published 416, 418, 419, 644, 718
Python 232
QualifiedClassName 699
QueryInterface 473
QuotedString 275
RAD 760
RAD and OOP 441
raise 374, 381
RaiseOuterException 395
Random 711
Randomize 711
Range checking 188
RawByteString 293
read 403
Read 727, 729, 731
Readers 729
read-only property 403
Rebol 294
Record 760
record helper 510
Record Helpers 509
record type 203
Records vs. Classes 307
Records with XE "Methods"Methods 212
recursion 156
Recursion 760
RefAttribute 712
Reference 760
Reference Counting 458
Reference Parameters 162
Reference Types 630
reference-counting 264
reflection 417
Reflection 644, 761
register 178
Register 435
RegisterComponents 435
RegisterExpectedMemoryLeak 537
reintroduce 353, 434, 444
Remove 274, 733
Replace 274, 277, 733
ReportMemoryLeaksOnShutdown 536
ResemblesText 278
RestoreCursor example 385
Result 152, 160

return type 152
Return Values 158
ReverseString 279
Robust Applications 543
RTTI 416, 419, 420, 421, 449, 645, 761
RTTI Classes 654
Ruby 232
run time type information 645
Run-Time Library 761
SaveToFile 734
Screen 445
SDK 761
Sealed Classes 360
Search Path 761
self 216, 445, 484
Self 215, 316, 317, 318, 424, 484, 525
Sender 429, 633
SetLength 194
SetMinimumBlockAlignment 537
SetTimer 486
Shared Object 754
ShortCut Key
  Ctrl+C 299
  Ctrl+Shift+C 214, 407, 408, 409
  Ctrl+Shift+C 406
  Ctrl+Shift+G 455
  Ctrl+Shift+Up 214
ShortString 283, 292
ShowMemory 535
ShowMessage 28
singleton pattern 459
singly-rooted class hierarchy 695
Size 727
SizeOf 204, 209, 571, 577
Slice 199
sLineBreak 710, 733
Smalltalk 232, 298
Smart Pointers 605
Sort 590
Split 275
square brackets 268, 666
Stack 381, 517, 761
stack overflow 155
Standard Template Library 587
StartsWith 274
Static 187
Static Class Methods 486
stdcall 178
Steve Tendon 555
StoredAttribute 712
Streaming 722
Streams 726
strict private 309
strict protected 310
String 262
string concatenation 264
String Helper 273
StringRefCount 284
subclassing 337
SubString 274
Support 472
Swift 352
Synchronize 636
TabControl 252
Tag 548, 722
TAggregatedObject 467, 710
TBasicAction 717
TBinaryReader 730
TBinaryWriter 730
TBits 716
TBufferedFileStream 727
TButton 497, 538, 588
TBytesStream 727
TCharacter 257
TCharHelper 257
TClass 496
TCollection 716
TComparer 590, 592
TComponent 434, 444, 459, 717, 719
TContainedObject 710
TCustomAttribute 667, 712
TDataModule 718
TDateTime 312, 709
TDictionary 587
TDirectory 724
TEdit 449, 497
template classes 555
TEncoding 287, 288, 293
TextFile 242
TFile 724
TFileStream 727
TFilterPredicate 725
TForm 365
TFunc 631

TGUID 505, 709
THandle 709
The Delphi Magazine 441, 477
THeapStatus 709
this 216
Threads Synchronization 636
throw 374
TInterfacedObject ... 456, 458, 459, 468, 710
TInterfaceList 716
TList 545, 548, 587, 588, 716
TMemoryManagerEx 709
TMemoryStream 727
TMethod 425, 709
TMonitor 709
TObject ...305, 322, 340, 354, 364, 391, 454, 495, 525, 694, 695
TObjectDictionary 587, 594, 595
TObjectList 549, 587, 589, 594
TObjectQueue 587, 594
TObjectStack 587, 594
ToCharArray 274
ToInteger 274
ToLower 258, 274
Tools Palette 435, 436
ToString 397, 701
ToUpper 258, 274
TPath 724, 726
TPersistent 418, 419, 420, 521, 716, 718
TProc 630
TQueue 587
TResourceStream 727
Trim 274
TRttiContext 652, 655, 659
TRttiObject 654
TRttiType 652
try 374, 383, 384
TSingletonImplementation 459, 604
TStack 587
TStopWatch 174
TStream 717, 726
TStreamReader 729, 730
TStreamWriter 729, 730
TStringBuilder 271, 732, 733
TStringList 521, 599, 716, 734
TStringReader 729
TStrings 413, 716, 734
TStringStream 727
TStringWriter 729
TTextLineBreakStyle 709
TTextWriter 685
TThread 636, 717
TUnicodeBreak 257
TUnicodeCategory 257
Turbo Pascal 743, 744, 757
TValue 660, 664
TVarData 201, 234
TVarRec 201
TVirtualMethodInterceptor 674
TVisibilityClasses 648, 708
Type Aliases 512
Type Compatibility 345
type compatibility rules 564
type derivation 337
TypeInfo 570
TypeScript 297
Type-Variant Open Array Parameters 200
UCS4Char 249, 256
UCS4String 292
UIntPtr 238, 239
Unicode 245, 252, 287, 762
Unicode Transformation Formats 249
unit
- Generics.Collections 587, 588, 594
- Generics.Defaults 459, 590, 604
- System 234, 242, 694, 707
- System.Actions 735
- System.AnsiStrings 735
- System.Character 257, 735
- System.Classes 715, 729, 735
- System.Contnrs 587, 735
- System.ConvUtils 736
- System.DateUtils 736
- System.Devices 736
- System.Diagnostics 174, 736
- System.Hash 736
- System.ImageList 736
- System.IniFiles 736
- System.IOUtils 724, 726, 736
- System.JSON 736
- System.Math 165, 174, 736
- System.Messaging 737
- System.NetEncoding 737
- System.RegularExpressions 737
- System.Rtti 651, 737

System.StrUtils 255, 278, 737
System.SyncObjs................................ 737
System.SysUtils..255, 278, 287, 377, 492, 511, 630, 737
System.Threading.............................. 737
System.Types 737
System.TypInfo...........421, 572, 651, 737
System.Variants................................. 737
System.Zip.. 737
Winapi.Windows 182
UnitName..699
Unsafe... 533
Unsafe references....................................462
UnsafeAttribute..712
UpCase... 258
UpperCase 274
UTF-16..250, 287
UTF-32..250
UTF32Char...292
UTF-8 ....250, 251, 287, 290
UTF8String..................................... 291, 293
var 71, 162
Variant 201
Variant Records 207
Variants 231, 234, 235
VCL 752, 762
virtual....................................... 349, 351, 355
Virtual Class Methods..............................484
virtual method table 355
Virtual Methods 763
Virtual Methods Interceptors.................. 673
Visual Basic.. 210
Visual Form Inheritance..........................365
vmtInstanceSize..551
vmtSelfPtr..551
VolatileAttribute..712
VType..234
Weak references.......................................462
Weak References 532
WeakAttribute...712
WideString..293
Windows 750, 754, 755, 763
Windows API...........................182, 357, 486
with 211, 212
WM_USER.. 357
write 403
Write 727, 729, 731
Writers.. 729
XML Streaming..684